中国科学技术大学 计算机科学与技术系
University of Science and Technology of China
Department of Computer Science and Technology

算法基础
Foundation of Algorithms

主讲人　徐云

Fall 2019, USTC

Part 6 Graph Algorithms

22 Elementary Graph Algorithms

23 Minimum Spanning Trees

24 Single-Source Shortest Paths

25 All-Pairs Shortest Paths

23 Minimum Spanning Trees

- Definition and Example
- History and Some Algorithms
- Growing an MST
- Two Greedy Algorithms
- Kruskal versus Prim

Thank prof. meng@binghamton for his slides.

Minimum Spanning Trees (MST)

- *Input*

  A connected, undirected graph $G = (V, E)$ with weight function $w: E \rightarrow R$.

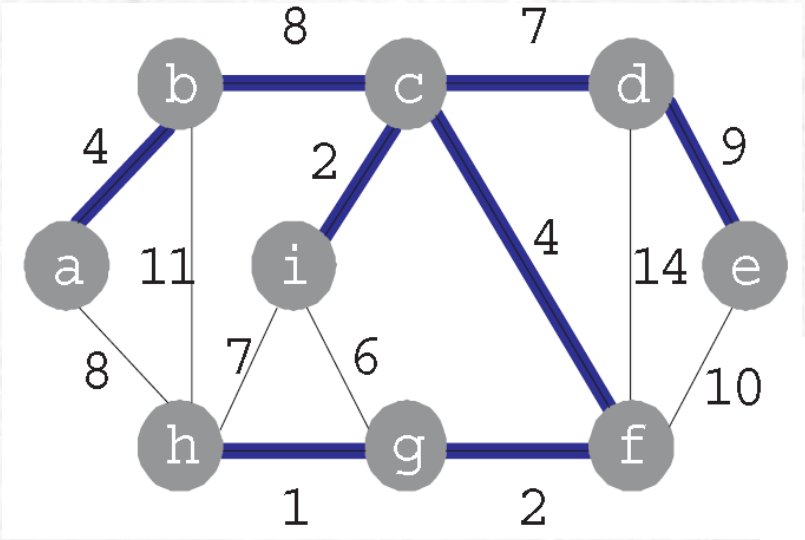
- *Output*

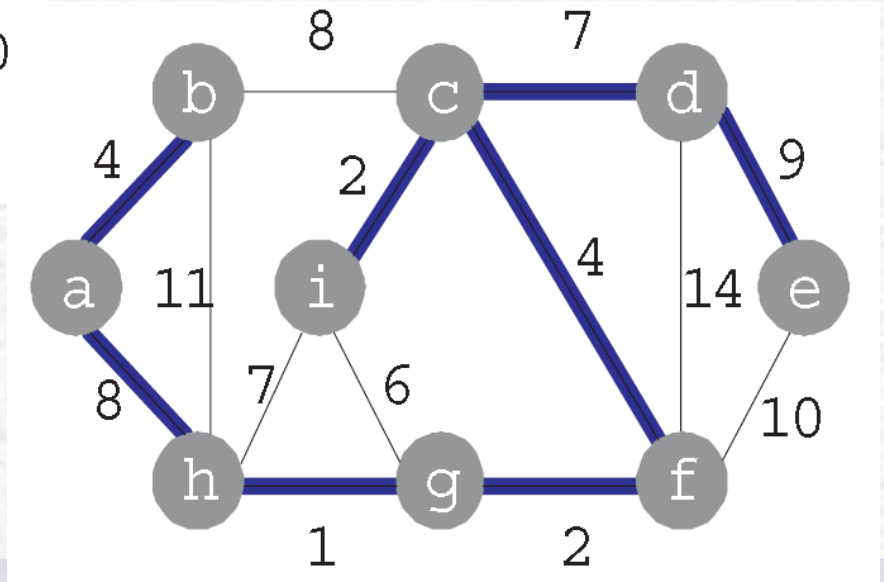
  A spanning tree T — a tree that connects all vertices — of minimum weight:

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

MST: an Example

- There is an MST, but it's *not unique*.





History and Some Algorithms

- *History*
  - In 1926, O. Boruvka firstly proposed an MST alg..
  - In 1956, J. B. Kruskal reported his method.
  - In 1957, R. C. Prim invented another alg., but it was also invented by V. Jarnik in 1930.
  - The problem of *spanning tree verification* can be solved in linear time (V. King, 1997).

- *Greedy methods*
  - Kruskal's and Prim's greedy methods for MST, whose time cost is $O(|E|\log|V|)$ or $O(|E|\log|E|)$.
  - By using Fibonacci heaps, Prim's alg. runs in time $O(|E| + |V|\log|V|)$, which improves over the binary-heap implementation if $|V|$ is much smaller than $|E|$.

Growing an MST and Idea

- *Growing an MST*
  - Grow an MST by adding one edge at a time.
- *Idea*
  - The algorithm *manages a set of edges A*, maintaining the following *loop invariant* : for each iteration, *A is a subset of some MST.*
- *Safe edge*
  - At each step we determine *an edge (u, v) such that* $A \cup \{(u, v)\}$ *is still a subset of an MST* and (u, v) is called a safe edge for A.

Generic MST Algorithm

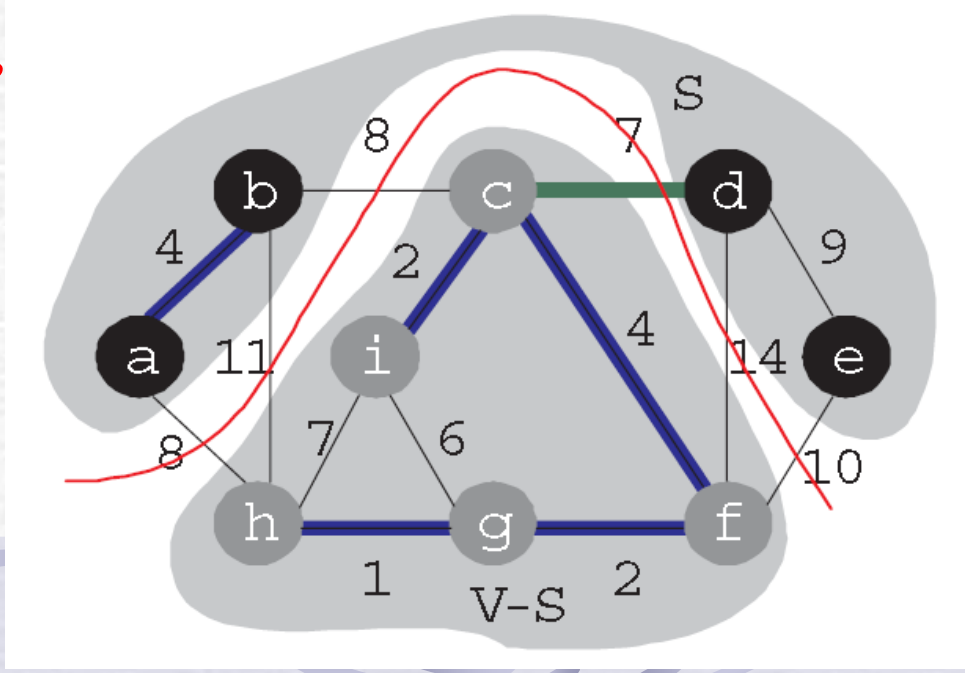
- Generic algorithm:

```
GENERIC-MST(G, w)
1  A = ∅
2  while A does not form a spanning tree
3      find an edge (u, v) that is safe for A
4      A = A ∪ {(u, v)}
5  return A
```

- In the followings, we will provide a rule for *recognizing safe edges*, and the two algorithms use the rule *efficiently*.

Cut and Light Edge

- *Definitions*
  - A *cut* $(S, V-S)$ of an undirected graph $G = (V, E)$ is *a partition of* V. An *edge* $(u, v) \in E$ *crosses the cut* iff one of its endpoints is in S and the other is in $V-S$.
  - An edge is a *light edge* crossing a cut if its weight is the minimum of any edge crossing the cut.
- *Example*



Recognizing Safe Edge

- *Theorem 23.1*
  - $G = (V, E)$ be a connected, undirected, and weighted graph,
  - $A \subseteq E$ be included in some MST,
  - $(S, V-S)$ be any cut of G that respects (针对) A, and
  - (u, v) be a *light edge* crossing $(S, V-S)$.
  - Then (u, v) is safe for A.

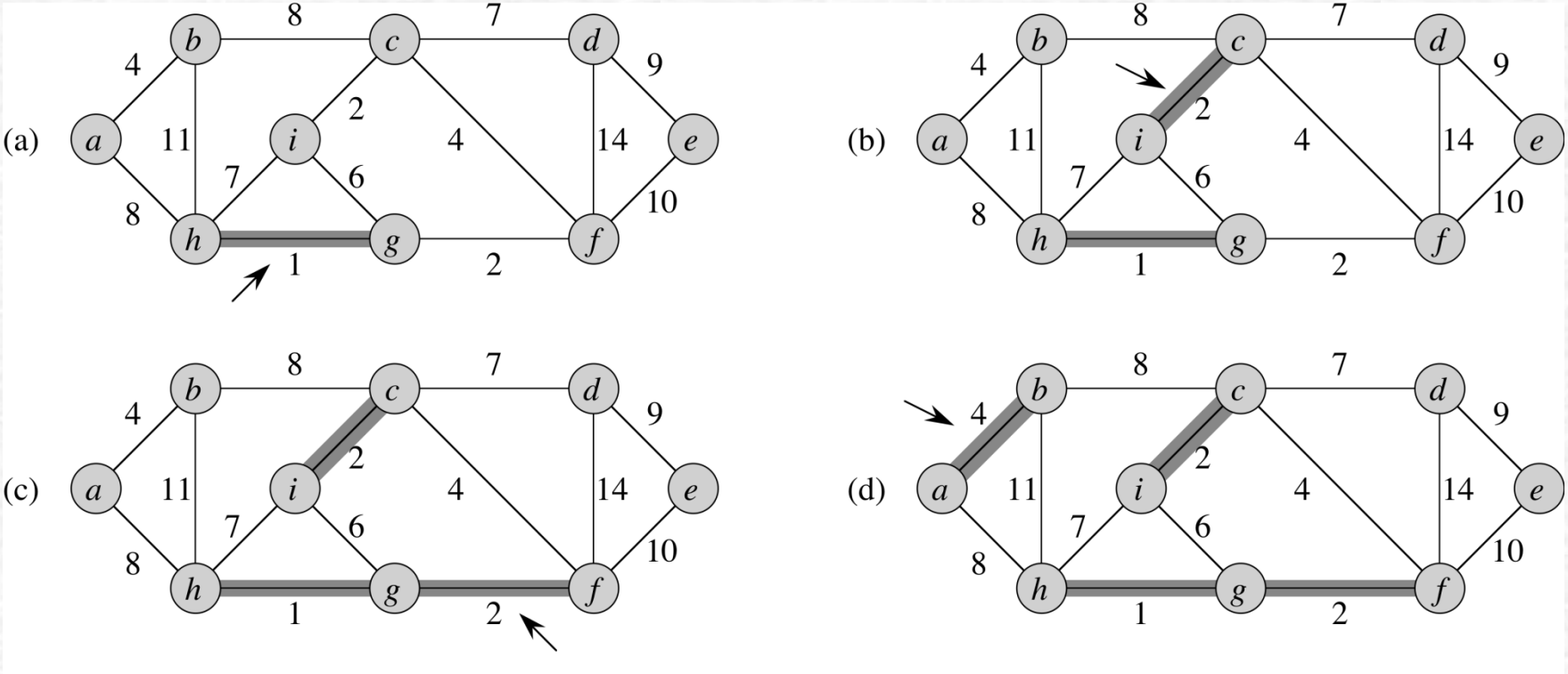
- *The proof is omitted.*

Two Greedy Algorithms

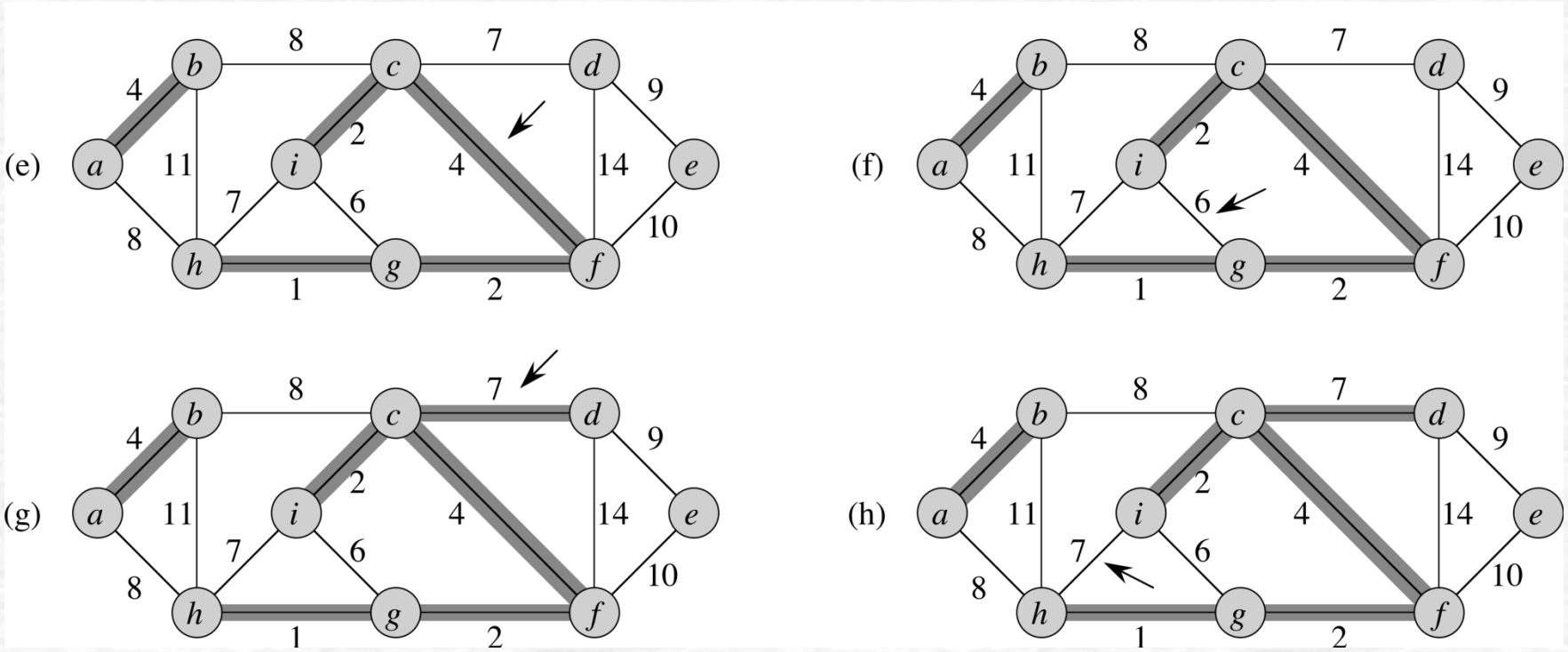
- *Kruskal's algorithm*: *Grow an MST from a forest of spanning trees* (connected components) by adding the edge that has the minimum weight and connects two spanning trees each time.

- *Prim's algorithm*: *Grow an MST from a partial MST* by adding the edge that has the minimum weight and has one end in the partial MST and one end not in each time.

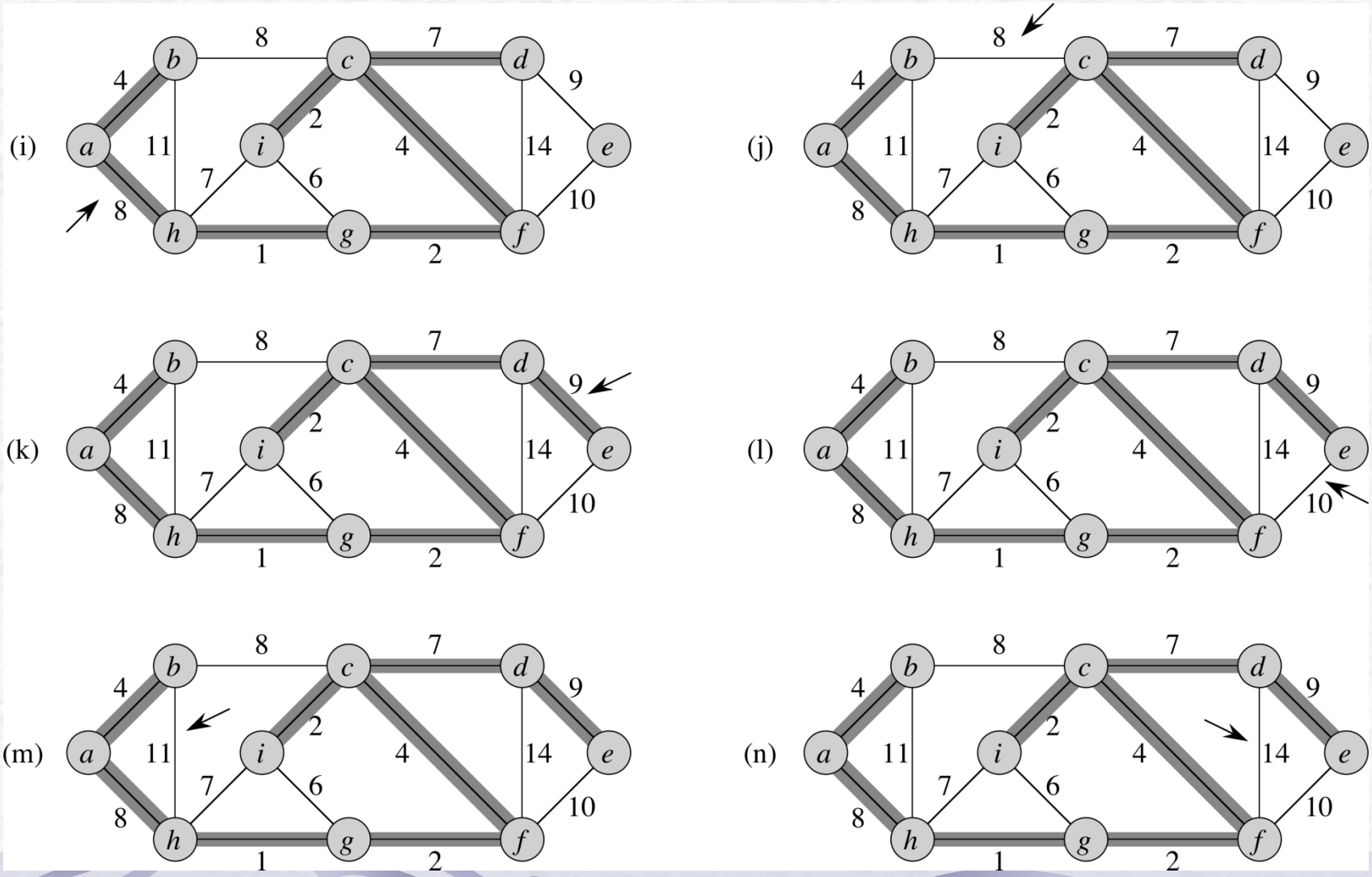
Kruskal Algorithm: Example (1)



Kruskal Algorithm: Example (2)



Kruskal Algorithm: Example (3)

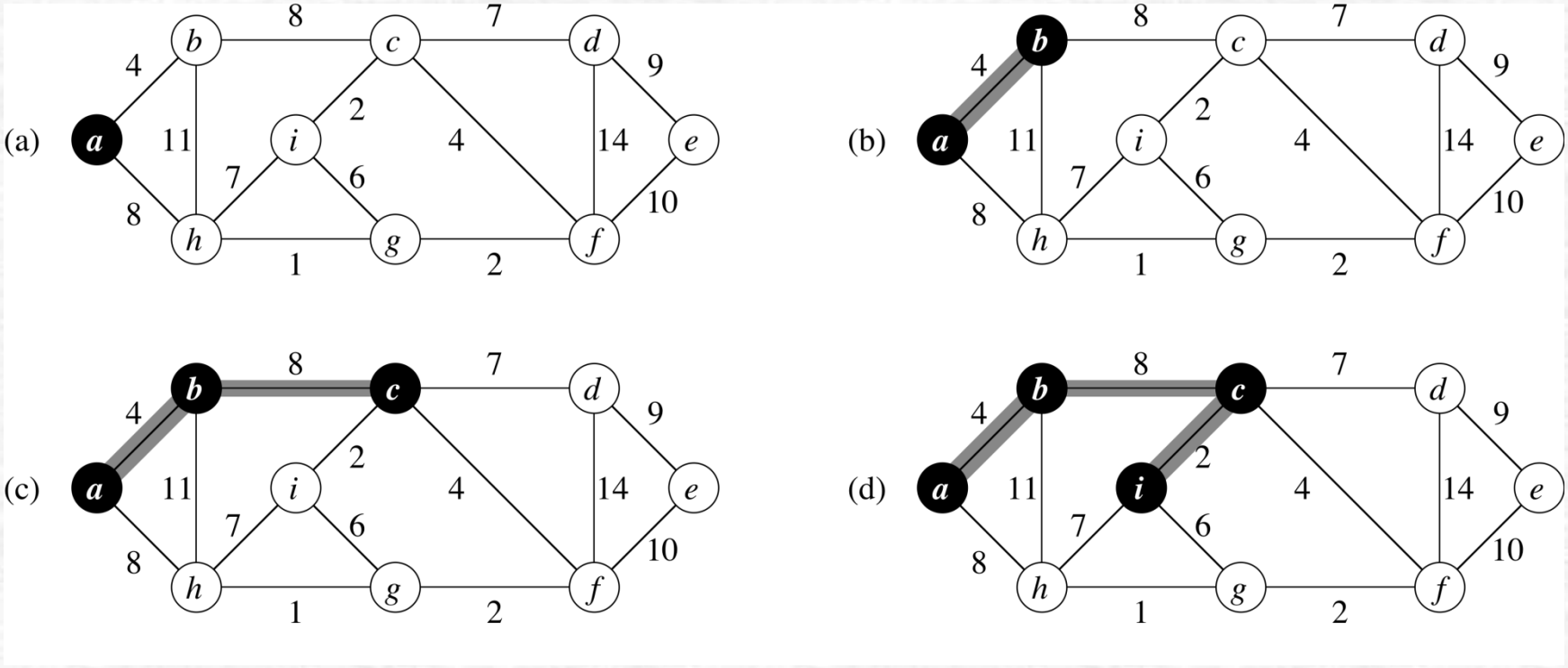


Kruskal Algorithm and Analysis

```
KRUSKAL(G, w)
  A = ∅
  for each vertex v ∈ G.V
      MAKE-SET(v)
  sort the edges of G.E into nondecreasing order by weight w
  for each (u, v) taken from the sorted list
      if FIND-SET(u) ≠ FIND-SET(v)
          A = A ∪ {(u, v)}
          UNION(u, v)
  return A
```

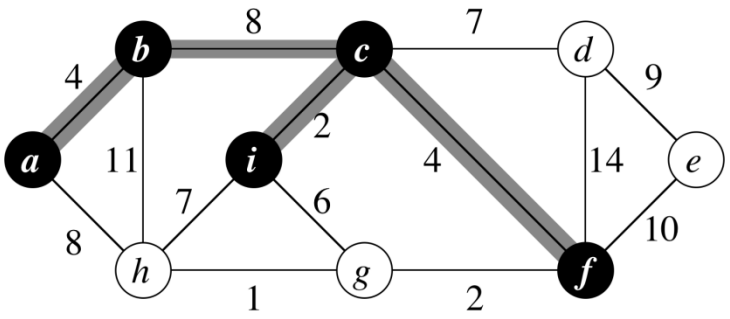
- Analysis:
  - First for loop: $|V|$ Make-Sets, $O(1)$ each, total $O(|V|)$
  - Sort E: $O(|E|\log|E|)$
  - Second for loop: $O(|E|)$ Find-Sets, $O(1)$ each
    $O(|E|)$ Unions: $O(\log|V|)$ each (*with a special method in ch21*)
  - *Total time*: $O(|E|\log|E|)$ or $O(|E|\log|V|)$

Prim Algorithm: Example (1)

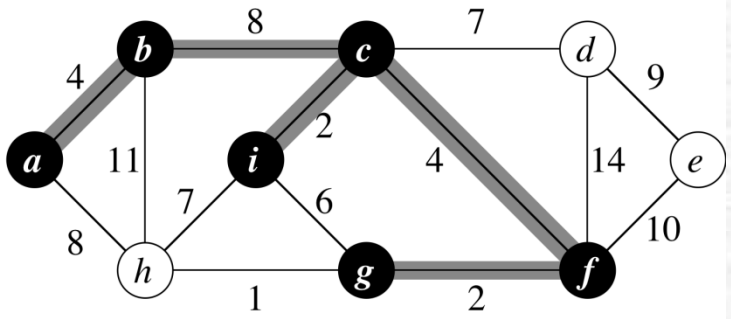


Prim Algorithm: Example (2)

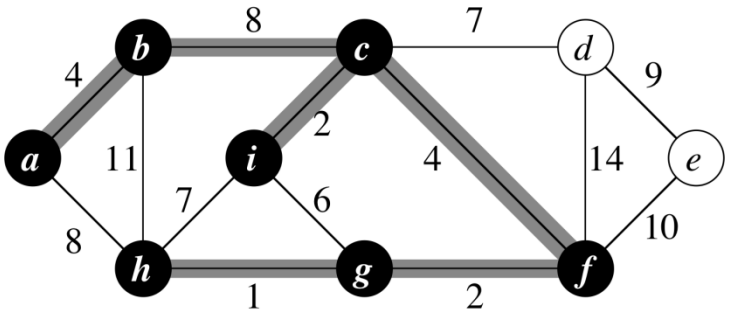
(e)



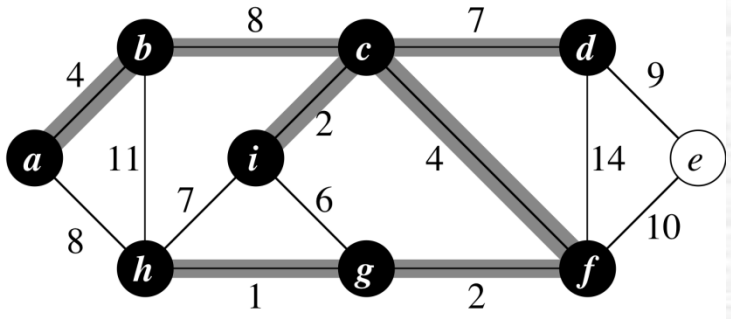
(f)



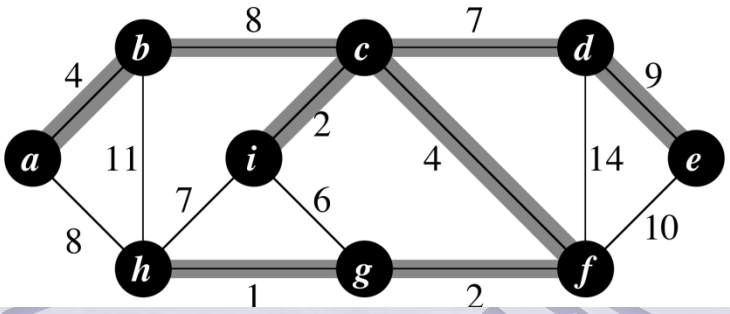
(g)



(h)



(i)



Prim Algorithm and Analysis

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

The tree starts from an arbitrary root vertex r.

- Suppose Q is a *binary min-heap*.
- Initialize Q and first for loop: $O(|V|)$.
- while loop: $|V|$ iterations.
  - Extract-Min: $O(\log|V|)$ each.
  - for loop: $O(|E|)$ times altogether
    - $v \in Q$ in $O(1)$ (*extended heap*).
    - Last line: $O(\log|V|)$.
    - Others operations in $O(1)$.
- Total while loop: $O(|V|\log|V|) + O(|E|\log|V|) = O(|E|\log|V|)$.
- *Overall total*: $O(|E|\log|V|)$.

Question: if use Fibonacci heap, how much can the run time be improved?

Kruskal versus Prim

- *Question 1*: Which has smaller running time?
- They have the same asymptotic running time: $O(|E|\log|V|)$.
  - Kruskal: $O(|E|\log|E|)$.
  - Prim: $O(|V|\log|V|) + O(|E|\log|V|)$.
  - Kruskal is better when $|E|$ is small (sparse graph).
  - Prim is better for dense graph ($|V| << |E|$).

- *Question 2*: Which is easier to implement?
  - Kruskal algorithm is easier.

Part 6 Graph Algorithms

22 Elementary Graph Algorithms

23 Minimum Spanning Trees

24 Single-Source Shortest Paths

25 All-Pairs Shortest Paths

24 Single-Source Shortest Paths

- Definitions and History
- Bellman-Ford Algorithm
- Directed Acyclic Graph Scenario
- Dijkstra Algorithm

Thank prof. meng@binghamton and yu@pku for their slides partial.

Definitions of Shortest Path

- *Shortest path*: A shortest path from u to v is a path of minimum weight from u to v. The *shortest-path weight* from u to v is defined as

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

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

- The *weight of path* $p = v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k$ is defined to be

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

- *Single-source shortest paths*: 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.
  - If a graph G contains a negative-weight cycle, then some shortest paths may no exist.

History

- Dijkstra algorithm was proposed in 1959.
- R. Bellman and L. R. Ford found BELLMAN-FORD alg. in 1958 and 1959, respectively.
- In 2000, M. Thorup gave an algorithm with $O(|E|\log\log|V|)$. For undirected graph with integer weights, Thorup gave an $O(|V|+|E|)$ time algorithm in 1999.

Shortest Path Representation

- Given a graph $G = (V, E)$, maintain for each vertex $v \in V$ a *predecessor* $\pi[v]$ that is either another vertex or Nil.

- Let $G_\pi = (V_\pi, E_\pi)$ be the subgraph (the *predecessor subgraph*) induced by the π values, where

  $V_\pi = \{u \in V \mid \pi[u] \neq \text{Nil}\} \cup \{s\}$, and

  $E_\pi = \{(\pi[u], u) \in E \mid u \in V_\pi - \{s\}\}$.

- The algorithms will produce G_π as a *shortest-paths tree*, containing a shortest path from s to every node reachable from s.

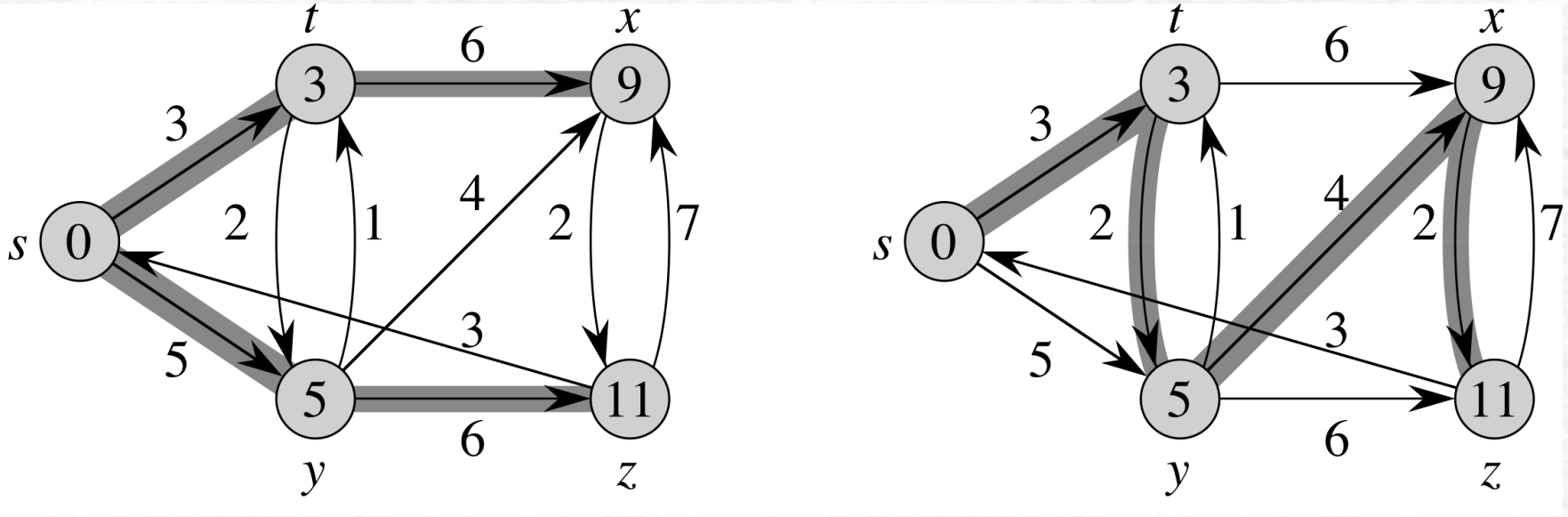
Shortest-Paths Tree

- A *shortest-paths tree* (SPT) is similar to a *breadth-first tree* (BFT) in Chapter 22.
  - *SPT* contains shortest paths from a source *in terms of edge weights*.
  - *BFT* contains shortest paths from a source *in terms of the number of edges*.

- An SPT rooted at s is a directed subgraph $G' = (V', E')$, $V' \subseteq V$ and $E' \subseteq E$, such that
  - V' is the set of vertices reachable from s,
  - G' forms a rooted tree with root s, and
  - for each $v \in V'$, the unique simple path from s to v in G' is a shortest path from s to v in G.

Shortest-Paths Tree: Example

- For a weighted, directed graph, there may be more than one *shortest-paths tree*.



Two SPTs of a weighted, directed graph.

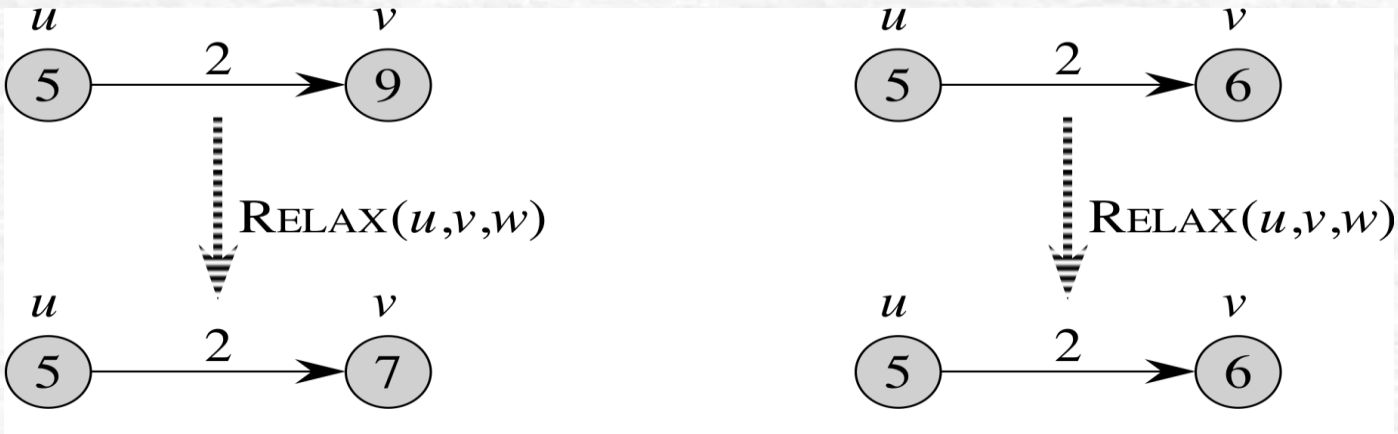
Technique of Edge Relaxation

- For *each vertex* $v \in V$, $d[v]$ is an upper bound on the weight of a shortest path from s to v, called *shortest-path estimate*.

```
INITIALIZE-SINGLE-SOURCE(G, s)
1  for each vertex v ∈ V
2       do d[v] ← ∞
3          π[v] ← NIL
4  d[s] ← 0
```

```
RELAX(u, v, w)
1  if d[v] > d[u] + w(u, v)
2     then d[v] ← d[u] + w(u, v)
3          π[v] ← u
```

Examples of Relaxation



- Each algorithm calls INITIALIZE-SINGLE-SOURCE(G, s) and then *repeatedly relaxes the edges*, differing in the order and time of relaxation of each edge.
  - In *Dijkstra algorithm* and the shortest-paths algorithm for directed acyclic graphs, each edge is *relaxed exactly once*.
  - In *Bellman-Ford algorithm*, each edge is *relaxed many times*.

Some Properties (1)

- *Triangle Inequality* (Lemma 24.10)
  - For any edge $(u, v) \in E$, we have $\delta(s, v) \leq \delta(s, u) + w(u, v)$.
- *Upper-bound Property* (Lemma 24.11)
  - We always have $d[v] \geq \delta(s, v)$ for all vertices $v \in V$ and once $d[v]$ achieves the value $\delta(s, v)$, it never changes.
  - Corollary: If there is no path from s to v, then we always have $d[v] = \delta(s, v) = \infty$.
- *Convergence Property* (Lemma 24.14)
  - If $s \rightsquigarrow u \rightarrow v$ is a shortest path in G for some $u, v \in V$ and if $d[u] = \delta(s, u)$ at any time prior to relaxing edge (u, v), then $d[v] = \delta(s, v)$ at all times afterward.

Some Properties (2)

- *Path-relaxation Property* (Lemma 24.15)
  - If $p = \langle v_0, v_1, \ldots, v_k \rangle$ is a shortest path from $s = v_0$ to v_k, and the edges of p are relaxed in the order $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$ then $d[v_k] = \delta(s, v_k)$
  - Once $d[v] = \delta(s, v)$ for all $v \in V$, the predecessor subgraph is a shortest-paths tree rooted at s.

Shortest-Path Algorithms

- Here, introduce *three algorithms* to find single-source shortest paths.
- They all start by calling Init-Single-Source() first and then repeatedly relax edges.
- *Bellman-Ford algorithm* relaxes each edge $|V| - 1$ times.
  - *Allows negative-weight edges*.
  - *Allows cycles*.
- The *shortest-paths algorithm for DAG* and *Dijkstra algorithm* relax each edge exactly once.
  - The shortest-paths algorithm for DAG *allows negative-weight edges but not cycle*.
  - Dijkstra algorithm *allows cycle but not negative-weight edges*.

Bellman-Ford Algorithm: Basic Idea

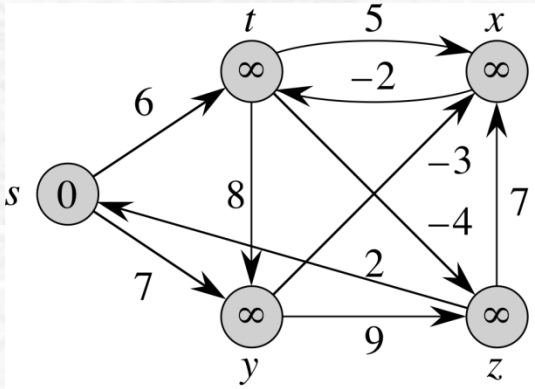
- It allows negative-weight edges.
- It computes $d[v]$ and $\pi[v]$ for each $v \in V$.
- It returns a Boolean value indicating whether there is a *negative-weight cycle* that is reachable from the source s.
  - It returns *False if a negative-weight cycle reachable from s is found*.
  - *If there is no such a cycle, it returns True* and also produces the shortest path to each vertex and the corresponding weight.

Bellman-Ford Algorithm: Pseudocode

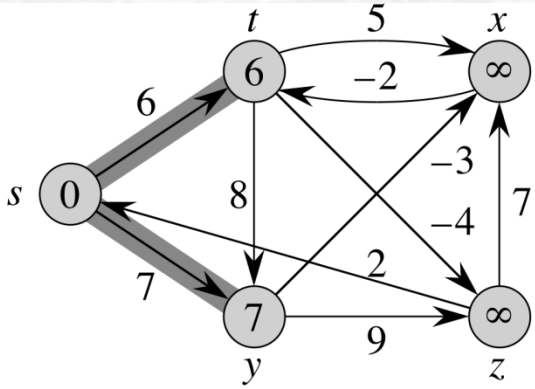
```
BELLMAN-FORD(G, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i ← 1 to |V| − 1
3       do for each edge (u, v) ∈ E
4            do RELAX(u, v, w)
5  for each edge (u, v) ∈ E
6       do if d[v] > d[u] + w(u, v)
7            then return FALSE
8  return TRUE
```

- The time cost of BELLMAN-FORD is $O(|V|\cdot|E|)$.

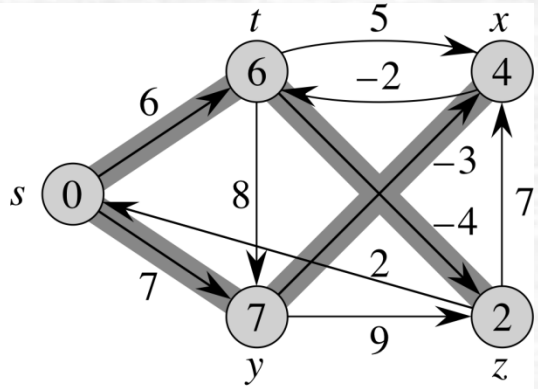
Bellman-Ford Algorithm: Example



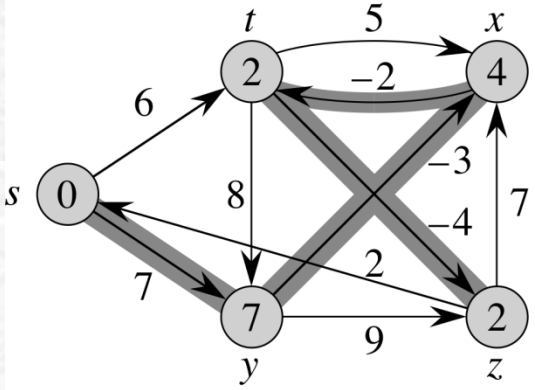
Input graph



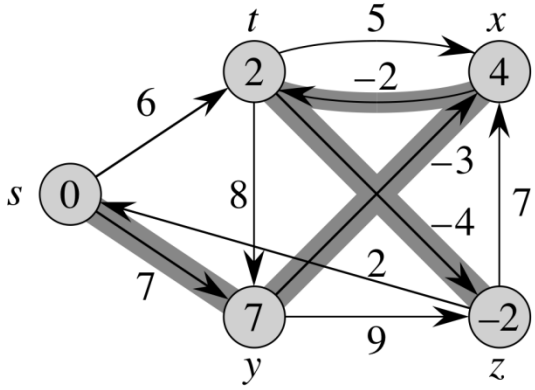
After 1st pass



After 2nd pass



After 3rd pass



After 4th pass

Based on the following order of examining edges: (t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y).

- Different orders affect speed of convergence.
- returns TRUE in this example

Why Only Need |V|-1 Iterations?

- **Lemma 24.2**(没有权重为负值环的Bellma-Ford算法正确性)

  Let $G = (V, E)$ be a weighted, directed graph. Assume that G has no negative-weight cycles that are reachable from s. Then, after $|V|-1$ iterations of lines 2-4 in BELLMAN-FORD, we have $d[v] = \delta(s, v)$ for all vertex v that is reachable from s.

- Proof: We prove the lemma by *appealing to the path-relaxation property*.

  Consider any vertex v that is reachable from s, and let $p = \langle v_0, v_1, \ldots, v_k \rangle$, where $v_0 = s$ and $v_k = v$, be any shortest path from s to v.

  Because shortest paths are simple, p has at most $|V|-1$ edges, and so $k \leqq |V|-1$.

  Each of the $|V|-1$ iterations of the for loop of lines 2-4 relaxes all $|E|$ edges. Among the edges relaxed in the i^{th} iteration, for $i = 1, 2, \ldots, k$, is (v_{i-1}, v_i).

  By the path-relaxation property, therefore,

  $$d[v] = d[v_k] = \delta(s, v_k) = \delta(s, v)$$

Correctness of Bellman-Ford

1. If G has no negative-weight cycles, then

$$\begin{aligned} d[v] &= \delta(s,v) \\ &\le \delta(s,u) + w(u,v) \\ &= d[u] + w(u,v) \end{aligned}$$

BELLMAN-FORD returns TRUE.

2. If G has a negative-weight cycle $c = \langle v_0, v_1, \cdots, v_k \rangle$, where $v_0 = v_k$. Then

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

If $d[v_i] \le d[v_{i-1}] + w(v_{i-1}, v_i)$, then

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

which implies that $\sum_{i=1}^{k} w(v_{i-1}, v_i) \ge 0$, contradiction! BELLMAN-FORD returns FALSE.

Single-Source Shortest Paths in DAG (1)

- Problem: Finding single-source shortest paths in a *directed acyclic graph (DAG)*.
- Question: Applying Bellman-Ford alg. directly would take $O(|V|\cdot|E|)$ time. Can we do better for a DAG?
- Solution: Yes, use topological sort.

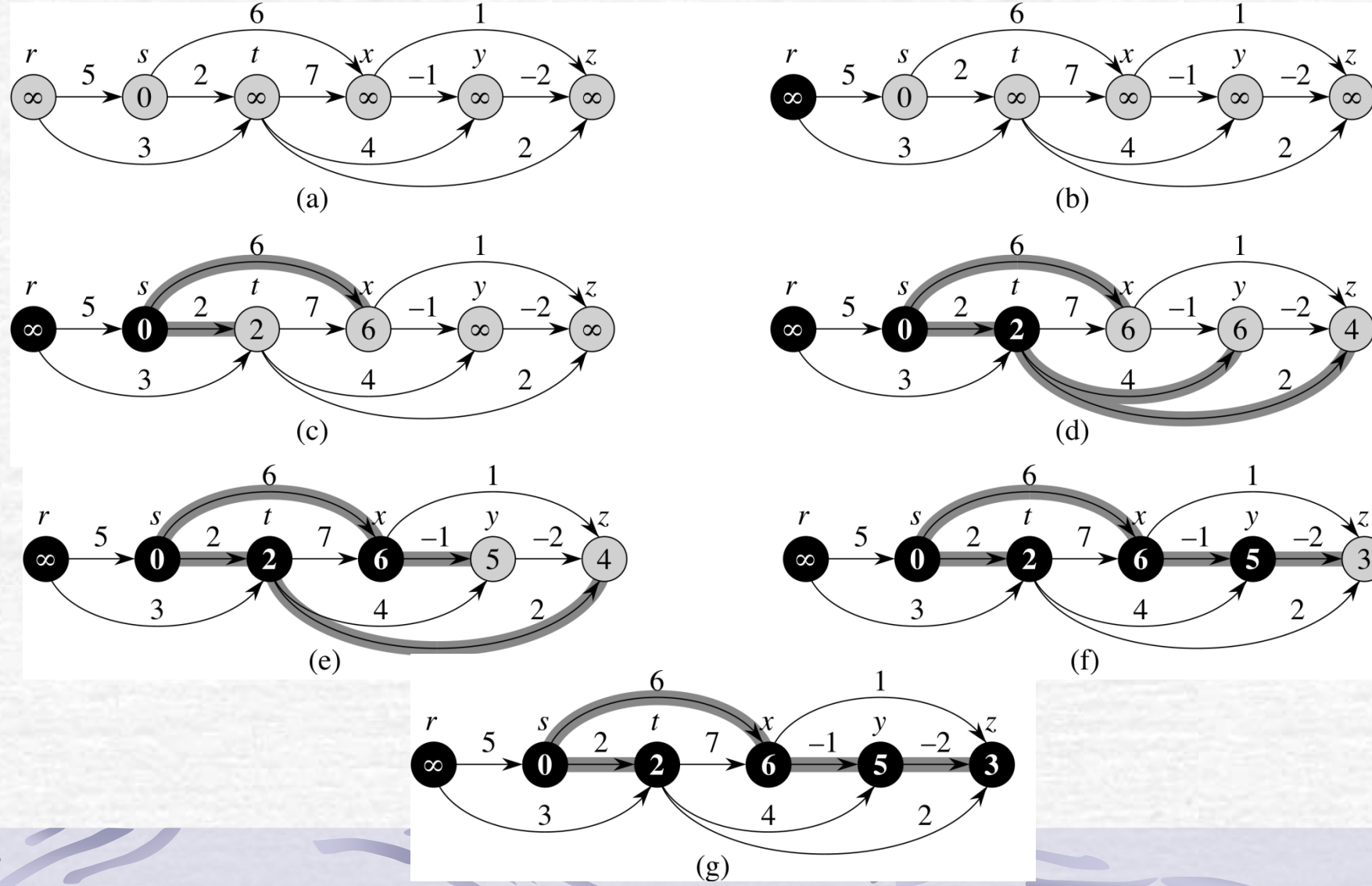
Single-Source Shortest Paths in DAG (2)

- Pseudocode of algorithm :

```
DAG-SHORTEST-PATHS(G, w, s)
    topologically sort the vertices
    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)
```

- Running time:
  - Topological sort: $\Theta(|V| + |E|)$
  - Init-Single-Source( ): $\Theta(|V|)$
  - For loop: $\Theta(|V| + |E|)$
  - Total: : $\Theta(|V| + |E|)$

Single-Source Shortest Paths in DAG (3)



Single-Source Shortest Paths in DAG (4)

Correctness of DAG-Shortest-Paths.

- Need to show $d[v] = \delta(s, v)$ for every vertex v reachable from s. If v is not reachable from s, $d[v] = \delta(s, v) = \infty$.
- If v is reachable from s, then there is a shortest path $p = \langle v_0, v_1, \ldots, v_k \rangle$ with $v_0 = s$ and $v_k = v$.
- Because we process vertices in topologically sorted order, we relax the edges on p in the order of $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$. By Lemma 3 (path relaxation property), $d[v_i] = \delta(s, v_i)$ for $i = 0, \ldots, k$.
- Therefore, DAG-Shortest-Paths is correct.

Dijkstra Algorithm: Basic Idea

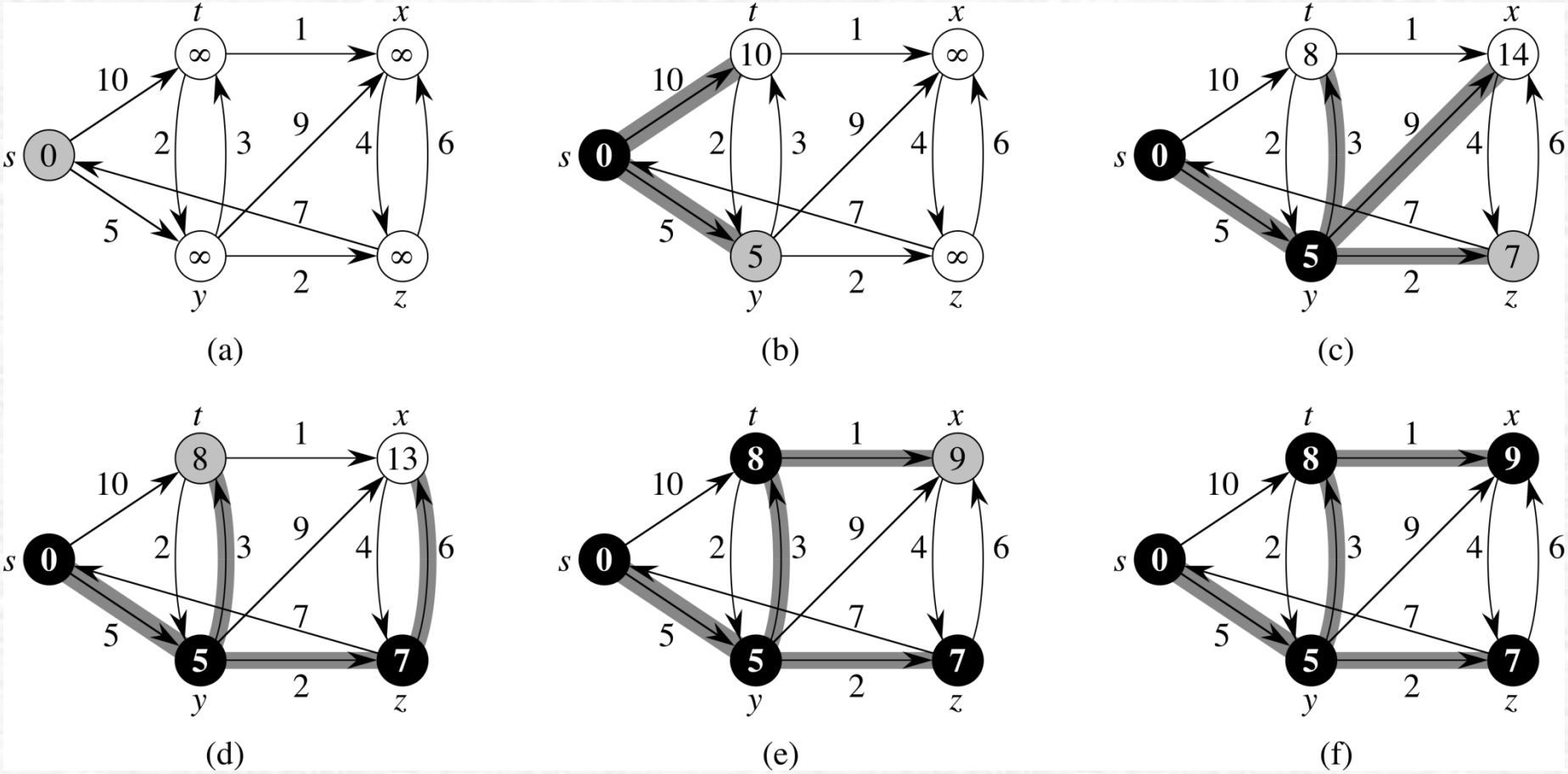
- It is for graphs with no negative-weight edges.
- *Similar to breadth-first search – a weighted version of BFS*.
  - Grow a tree gradually, advancing from vertices taken from a queue.
  - Instead of a FIFO queue, it uses a priority queue with keys being the shortest-path weights (*d[v]*).
- In process, two sets of vertices:
  - *S* = vertices whose final shortest-path weights have been determined
  - *Q* = priority queue = *V* - *S*.

Dijkstra Algorithm: Pseudocode

```
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)
      S = S ∪ {u}
      for each vertex v ∈ G.Adj[u]
          RELAX(u, v, w)
```

- Similar to Prim's algorithm, but computing *d[v]*, and using shortest-path weights as keys.
- It is a greedy algorithm: always choose the "closest" vertex in *V - S* to add to *S*.

Dijkstra Algorithm: Example



Dijkstra Algorithm: Running Time

```
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)
      S = S ∪ {u}
      for each vertex v ∈ G.Adj[u]
          RELAX(u, v, w) // Essentially Decrease-Key( )
```

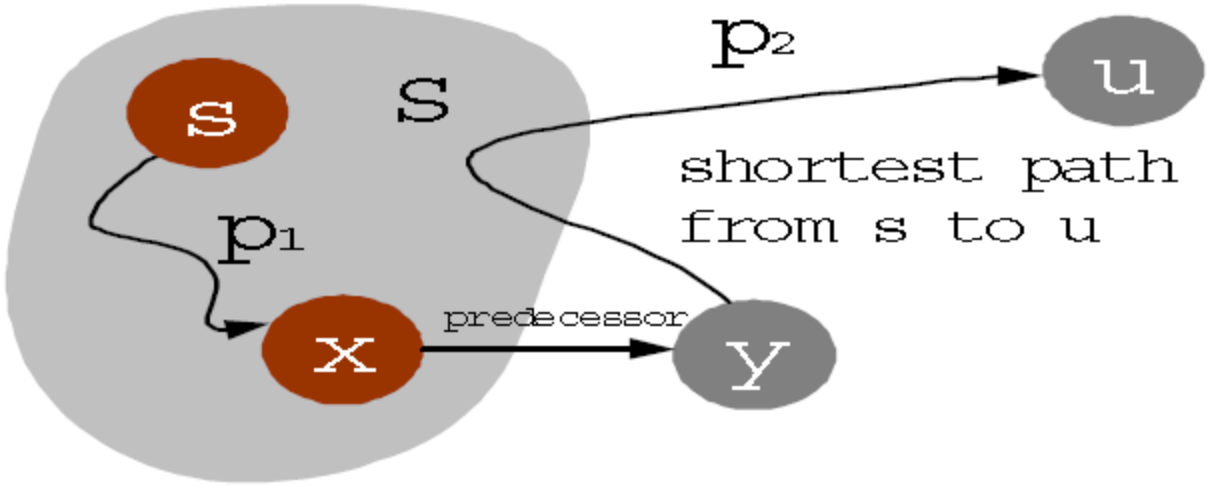
Running time: $O(|E|\log|V|)$ if binary heap is used for Q

- Initialization part: $O(|V|)$
- While loop: $O(|V|)$
- Each Extract-Min(): $O(\log|V|)$

} Together: $O(|V|\log|V|)$

- For loop (total number of iterations for entire While loop): $O(|E|)$
- Each Relax (Decrease-Key): $\log|V|$
- The total for the For loop: $O(|E|\log|V|)$

Dijkstra Algorithm: Correctness



flash of adding u to S

Let u is the first vertex that $d[u] \neq \delta(s,u)$ when u is added to S. (1) $d[y] = \delta(s,y) \leq \delta(s,u) \leq d[u]$, (2) $d[u] \leq d[y]$ when u is added to S. So, we have $d[u] = \delta(s,u)$, a contradiction!

Part 6 Graph Algorithms

22 Elementary Graph Algorithms

23 Minimum Spanning Trees

24 Single-Source Shortest Paths

25 All-Pairs Shortest Paths

25 All-Pairs Shortest Paths

- Problem and History
- Shortest paths and matrix multiplication
- Floyd-Warshall Algorithm
- Johnson Algorithm for sparse graphs

Thank prof. meng@binghamton and yu@pku for their slides partial.

Problem and History

- *Problem*: Given a weighted, directed graph $G = (V, E)$, for any $u, v \in V$, find a shortest path from u to v.
- *Remark*
  - Algorithms of single-source shortest paths can not be used directly because of the complexity.
  - We will investigate the relation of the all pairs shortest-paths problem to matrix multiplication and study its algebraic structure.
- *History*
  - FLOYD-WARSHALL is due to R. W. Floyd in 1962, based on a theorem of Warshall that describes how to compute the transitive closure of boolean matrices.
  - JOHNSON is due to D. B. Johnson in 1977.

Weight Matrix

- *Definition*

  The weighted, directed graph $G = (V, E)$ can be represented by a $|V| \times |V|$ matrix $W = (w_{ij})$, where

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

- *Assuming*
  - In following, we assume that $|V| = n$.
  - *Negative-weight edges are allowed, but* we assume for the time being that the input graph *contains no negative-weight cycles*.

Output of Problem

- The tabular output of the allpairs shortest-paths algorithms is an $n \times n$ matrix $D = (d_{ij})$, where $d_{ij} = \delta(i, j)$.

- *Predecessor matrix*, say $\Pi = (\pi_{ij})$, is used to record the shortest paths:
  - π_{ij} = NIL if either $i = j$ or there is not path from i to j.
  - π_{ij} is the predecessor of j on some shortest path from i to j.

Predecessor Subgraph

- *Definition*: For each vertex $i \in V$, we define *predecessor subgraph* of G for i as $G_{\pi,i} = (V_{\pi,i}, E_{\pi,i})$, where
  - $V_{\pi,i} = \{j \in V \mid \pi_{ij} <> \text{NIL}\} \cup \{i\}$
  - $E_{\pi,i} = \{(\pi_{ij}, j) \mid j \in V_{\pi,i} - \{i\}\}$
- If $G_{\pi,i}$ is a shortest-paths tree, then PRINTALL-PAIRS-SHORTEST-PATH will print a shortest path from i to j.

Print The Shortest Path

```
PRINT-ALL-PAIRS-SHORTEST-PATH(Π, i, j)
1  if i = j
2     then print i
3     else if π_ij = NIL
4             then print "no path from"
                       i "to" j "exists"
5             else PRINT-ALL-PAIRS-SHORTEST
                   -PATH(Π, i, π_ij)
6                  print j
```

A Recursive Solution

- Let $l^{(m)}_{ij}$ denote the minimum weight of any path from i to j that contains at most m edges.
  - When m = 0, then

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

  - When m ≥ 1, then

$$\begin{aligned} l_{ij}^{(m)} &= \min\left(l_{ij}^{(m-1)}, \min_{1 \le k \le n}\{l_{ik}^{(m-1)} + w_{kj}\}\right) \\ &= \min_{1 \le k \le n}\{l_{ik}^{(m-1)} + w_{kj}\} \end{aligned} \quad (※)$$

Because $w_{jj} = 0$ for all j, then get the unified later formula.

- If $\delta(i, j) < \infty$, then there is a shortest path from i to j that is simple and thus contains at most $n - 1$ edges. Therefore, $\delta(i, j) = l^{(n-1)}_{ij} = l^{(n)}_{ij} = l^{(n+1)}_{ij} = \cdots$.

Computing Shortest-path: Idea

- Now, we have to compute a series of matrices $L^{(1)}, L^{(2)}, \cdots, L^{(n-1)}$, where $L^{(m)} = (l^{(m)}_{ij})$.
  - Because $l^{(1)}_{ij} = w_{ij}$ for all $i, j \in V$, so $L^{(1)} = W$.
  - Given matrices $L^{(m-1)}$ and W, how to get $L^{(m)}$?
- The method extends the shortest paths computed so far by one more edge.

Extend Shortest Paths

- Let $L^{(m-1)} = L$, $L^{(m)} = L'$. We have

```
EXTEND-SHORTEST-PATHS(L, W)
1  n ← rows[L]
2  let L' = (l'_ij) be an n × n matrix
3  for i ← 1 to n
4       do for j ← 1 to n
5            do l'_ij ← ∞
6               for k ← 1 to n
7                  do l'_ij ← min(l'_ij, l_ik + w_kj)
8  return L'
```

- EXTEND-SHORTEST-PATHS costs $\Theta(n^3)$ time.

Borrowing Matrix Multiplication

- Equation (※) is similar with matri... multiplication $C(c_{ij}) = A(a_{ij}) \cdot B(b_{ij}) = \left(\sum_{k=1}^{n} a_{ik} \cdot b_{kj} \right)$ where the correspondence is:

$$
\begin{aligned}
l^{(m-1)} &\rightarrow a \\
w &\rightarrow b \\
l^{(m)} &\rightarrow c \\
\min &\rightarrow \sum \\
+ &\rightarrow \cdot
\end{aligned}
$$

- And therefore

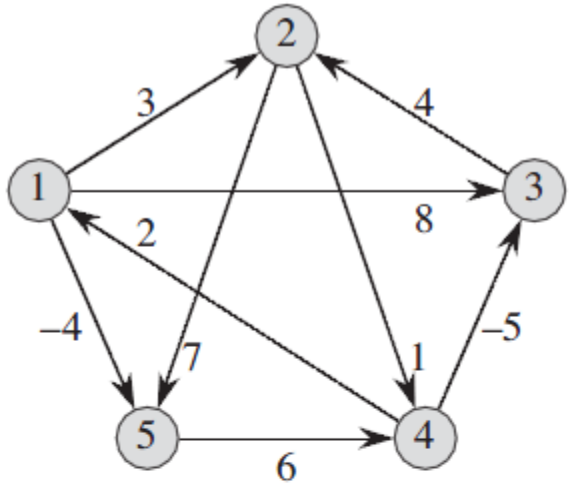
$$
\begin{aligned}
\mathbf{L}^{(1)} &= \mathbf{W} \\
\mathbf{L}^{(2)} &= \mathbf{L}^{(1)} \cdot \mathbf{W} = \mathbf{W}^2 \\
&\vdots \\
\mathbf{L}^{(n-1)} &= \mathbf{L}^{(n-2)} \cdot \mathbf{W} = \mathbf{W}^{n-1}
\end{aligned}
$$

All-pairs Shortest Paths

```
SLOW-ALL-PAIRS-SHORTEST-PATHS(W)
1  n ← rows[W]
2  L^(1) ← W
3  for m ← 2 to n − 1
4       do L^(m) ← EXTEND-SHORTEST
                   -PATHS(L^(m−1), W)
5  return L^(n−1)
```

- SLOW-ALL-PAIRS-SHORTEST-PATHS costs $\Theta(n^4)$ time.
- We are just interested in $L^{(n-1)}$, so only $L^{(1)}$, $L^{(2)}$, $L^{(4)}$, . . . , $L^{(2^{\text{ceiling}(\log(n-1))})}$ are needed, because we have $2^{\text{ceiling}(\log(n-1))} \geq n - 1$.

Computing Example



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

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

Repeated Squaring

```
FASTER-ALL-PAIRS-SHORTEST-PATHS(W)
1  n ← rows[W]
2  L^(1) ← W
3  m ← 1
4  while m < n − 1
5        do L^(2m) ← EXTEND-SHORTEST
                     -PATHS(L^(m), L^(m))
6           m ← 2m
7  return L^(m)
```

- FASTER-ALL-PAIRS-SHORTEST-PATHS costs $\Theta(n^3 \log n)$ time.

Intermediate Vertex

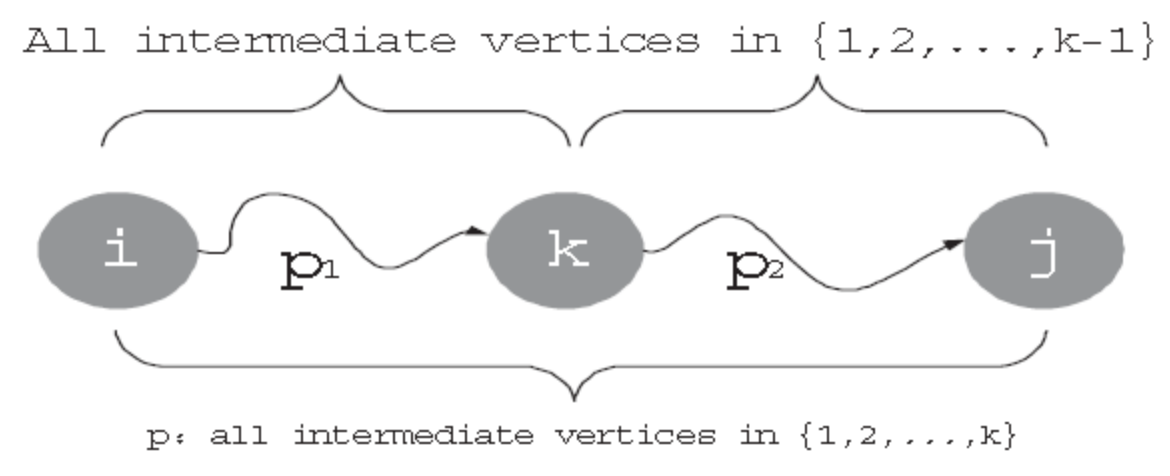
- Definition: An *intermediate vertex* of a simple path $p = \langle v_1, v_2, \cdots, v_l \rangle$ is any vertex of p other than v_1 and v_l.
- Observation: Floyd-Warshall algorithm is based on the observation of the intermediate vertices, which costs $\Theta(|V|^3)$ time.
- Let $V = \{1, 2, \cdots, n\}$. For any pair of vertices $i, j \in V$, consider all paths from i to j whose intermediate vertices are all drawn from $\{1, 2, \cdots, k\}$, and let p be a minimum-weight path from among them.

Floyd-Warshall Algorithm: Idea

Divide into two cases:

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

- If k is an intermediate vertex of path p, then we break p down into $i \overset{p_1}{\rightsquigarrow} k \overset{p_2}{\rightsquigarrow} j$.



Recurrence

- *Definition*: Let $d^{(k)}_{ij}$ be the weight of a shortest path from vertex i to vertex j for which all intermediate vertices are in the set $\{1, 2, \cdots, k\}$.

- We have the following recurrence:

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

- Because for any path, all intermediate vertices are in the set $\{1, 2, \cdots, n\}$, the matrix $D^{(n)} = (d^{(n)}_{ij})$ gives the final answer: $d^{(n)}_{ij} = \delta(i, j)$ for all $i, j \in V$.

Floyd-Warshall Algorithm

```
FLOYD-WARSHALL(W)
1  n ← rows[W]
2  D^(0) ← W
3  for k ← 1 to n
4      do for i ← 1 to n
5             do for j ← 1 to n
6                    do d_ij^(k) ← min(d_ij^(k−1),
                                       d_ik^(k−1) + d_kj^(k−1))
7  return D^(n)
```

- FLOYD-WARSHALL costs $\Theta(n^3)$ time.

Floyd-Warshall Alg.: Example (1)

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

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

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

Floyd-Warshall Alg.: Example (2)

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

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

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

Johnson Algorithm: Idea

- *It uses BELLMAN-FORD and DIJKSTRA as subroutines*. If some edges are negative, we will redefine weight function w': $E \rightarrow R$, satisfying
  - If *p* is a shortest path from *i* to *j* by *w*, then it is also a shortest path by nonnegative w'.
  - *Preserving shortest paths by reweighting*.
- *Johnson algorithm* finds shortest paths between all pairs in $O(|V|^2 \log |V| + |V||E|)$ *time for sparse graph*.
- In this course, the detail of weight transformation are not given here.

Johnson Algorithm

```
JOHNSON(G)
 1  compute G', where V[G'] = V[G] ∪ {s},
    E[G'] = E[G] ∪ {(s, v) | v ∈ V[G]}, and
    w(s, v) = 0 for all v ∈ V[G]
 2  if BELLMAN-FORD(G', w, s)=FALSE
 3     then print "negative-weight cycle"
 4     else for each vertex v ∈ V[G']
 5              do set h(v) to δ(s, v) computed
                      by BELLMAN-FORD
 6          for each edge (u, v) ∈ E[G']
 7              do ŵ(u, v) ← w(u, v) + h(u) − h(v)
 8          for each vertex u ∈ V[G]
 9              do run DIJKSTRA(G, ŵ, u) to
                     compute δ̂(u, v) for all v ∈ V[G]
10                 for each vertex v ∈ V[G]
11                     do δ(u, v) ← δ̂(u, v) + h(v) − h(u)
12          return D
```

- Appling the min-priority queue in Dijkstra algorithm by a Fibonacci heap, Johnson algorithm runs in $O(|V|^2\log|V|+|V||E|)$ time.

End of Ch22-25-part2