Lecture #10:

0.0.1 Graph Algorithms: Shortest Path (Chapter 24-25)

Problem 1 *Given a directed graph $G = [V, E]$, and weight function $w : E \longrightarrow R$ mapping edges to real valued weights. The weight of a path $p = \langle v_o, v_1, ..., v_k \rangle$ is the sum of the weights of edges along the path and is given by $w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$. A shortest path between v_0 and v_k is a path whose weight is minimum among all paths from v_0 to v_k and we denote this shortest distance as $\delta(v_0, v_k)$.*

Some variants of this problem: (i)from a single origin to all other nodes; (ii)from all other nodes to a single destination; (iii)between all pairs of nodes. Some algorithms require all weights to be nonnegative and others permit some to be negative. If there is a negative cycle, then we may get into difficulties.

Representing Shortest Paths:

For each pair of nodes (u, v) we keep track of the node that occurs just before the destination v as its predecessor, $\pi(u, v)$. In the case of a single origin, we get a tree using these predecessors in the form a predecessor subgraph $G_\pi = [V_\pi, E_\pi]$. Here, the set of nodes correspond to nodes that can be reached from the origin s. Thus, $V_\pi = \{v \in V : \pi[s, v] \neq NIL\} \cup \{s\}; E_\pi = \{(\pi[s, v], v) \in E : v \in \{V_\pi - \{s\}\}$. For the single origin case, G_π is a tree rooted at s and is known as a shortest path tree.

Single source shortest path algorithms use a technique called *relaxation.*

INITIALIZE-SINGLE-SOURCE(G, s)
for each vertex $v \in V[G]$
  do $d[v] \longleftarrow \infty$
    $\pi[v] \longleftarrow NIL$
$d[s] \longleftarrow 0$

RELAX(u, v, w)
if $d[v] > d[u] + w(u, v)$
  then $d[v] \longleftarrow d[u] + w(u, v)$
    $\pi[v] \longleftarrow u$

Let $\delta(s, v)$ denote the length of a shortest path from s to v. In all this, $d[v]$ should be thought of as the current estimate of $\delta(s, v)$.

Lemma 2 *Let $p = \langle v_o, v_1, ..., v_k \rangle$ be a shortest path from v_0 to v_k in $G = [V, E]$. For any i and j such that $1 \leq i \leq j \leq k$, let $p_{i,j} = \langle v_i, ..., v_j \rangle$ be a sub-path from v_i to v_j. Then $p_{i,j}$ is a shortest path between these vertices.*

Corollary 3 *Let a shortest path p between s and v be broken into $p_{s,u}$ and the edge (u, v). Then $\delta(s, v) = \delta(s, u) + w(u, v)$.*

Lemma 4 *Immediately after relaxing edge (u, v) by executing RELAX(u, v, w), we have $d[v] \leq d[u] + w(u, v)$.*

Lemma 5 *Assume that we initialize as shown above. Then, the relaxation algorithm maintains the invariant that $d[v] \geq \delta(s, v)$ for all $v \in V$. Moreover, if equality is achieved at any step, it does not change.*

Bellman's Equations:
This is for a single source case.

$$\begin{aligned} \delta(s, s) &= 0 \\ \delta(s, j) &= \min_{k \neq j}[\delta(s, k) + w(k, j)] \quad \text{for } j \neq s \end{aligned}$$

If there are no negative cycles reachable from s, then it is clear that $\delta(s, s) = 0$, and $\delta(s, j)$ are well defined for nodes j reachable from s. For each such node j, there is a last node on the shortest path from s to j other than j. If we denote this node by k, then the above equations are clear.

Directed Acyclic Graphs (DAG):(Chapter 25.4)

If G is a driected acyclic graph, then we can do a topological sort of its vertices in time equal to $\Theta(|V| + |E|)$. This is done by DFS

(Depth First Search) on the graph G. This is shown below with an example. Notice the time stamps. The first of these is the time when a node is first discovered and the second is when the node is finished – we have considered all edges starting from this node. The final topological order is according to decreasing values of finish times. If the graph is acyclic, there will no **back** edge. Edge classification is as follows:

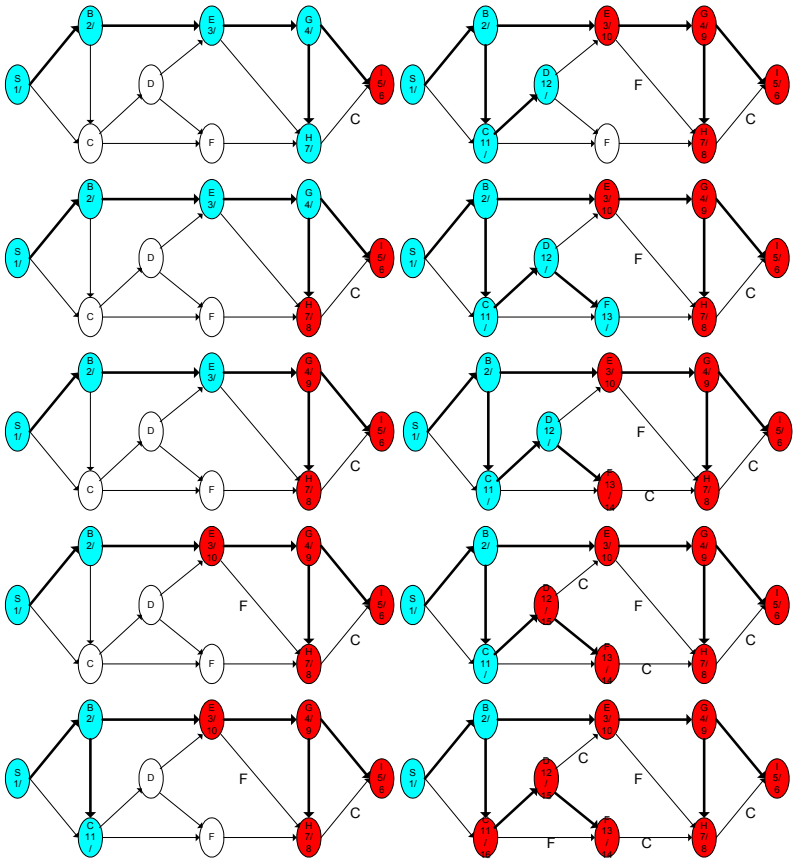
Tree edges: When a node v is first discovered, by using an edge (u, v), this edge is part of the DFS tree and the node u is the parent of the node v.

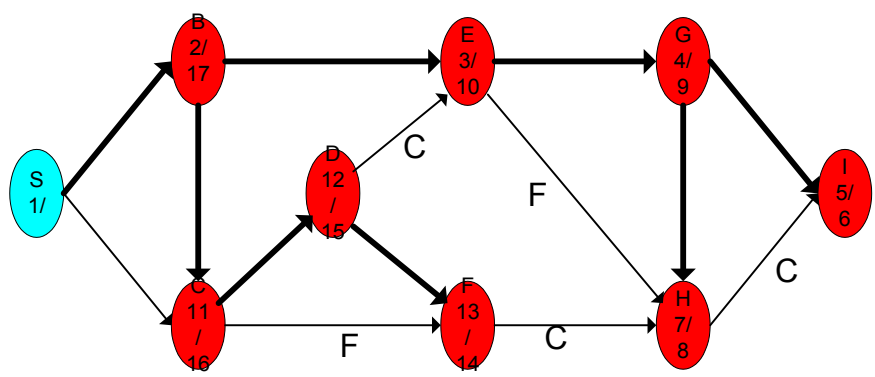
Forward edges: If an edge (u, v) has the node u as an ancestor of the node v, the this edge is a forward edge.

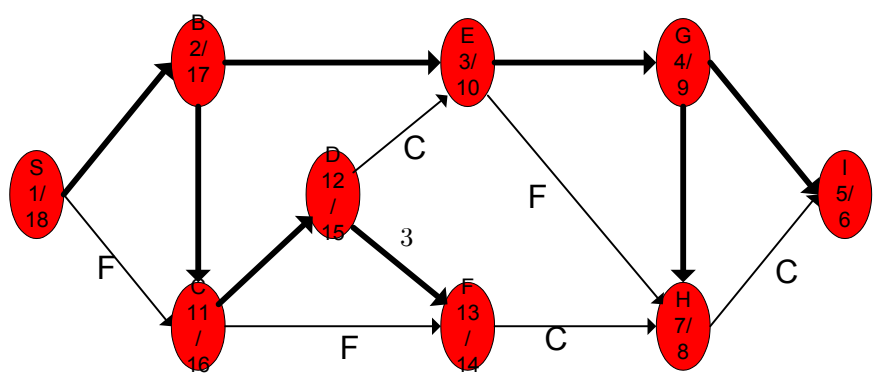
Back edges: If an edge (u, v) has the node v as an ancestor of the node u, the this edge is a back edge. Back edges indicate the presence of directed cycles.

Cross edges: All other edges are cross edges. These may be between two nodes in distinct trees or between two nodes that do not have an ancestor-descendent relationship.

In the example below, we have three kinds since there are no back edges.

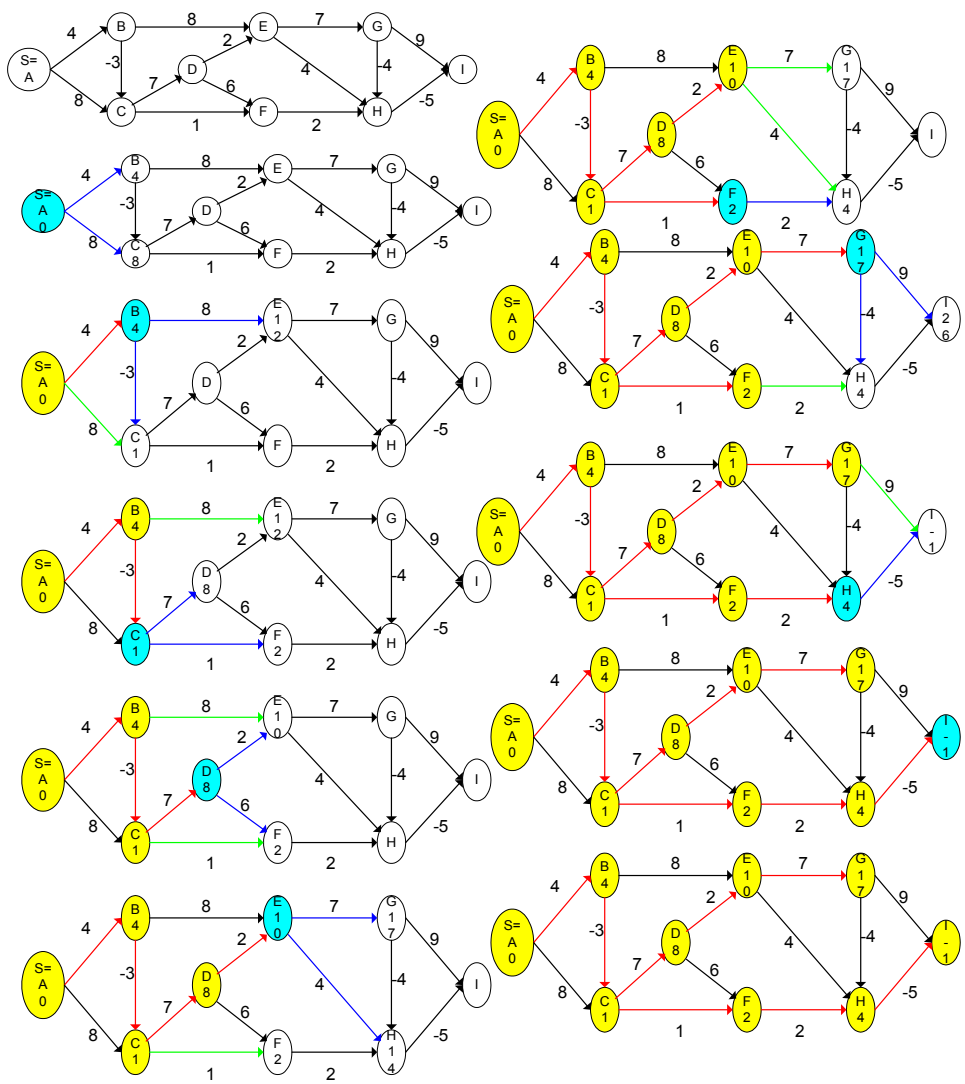






DAG-SHORTEST-PATHS(G, w, s)
topologically sort the vertices of G
INITIALIZE-SINGLE-SOURCE(G, s)
for each vertex u taken in the topological sorted order
  do for each vertex $v \in Adj[u]$
    do RELAX(u, v, w)

An example is shown below: Vertices are in topological order (=alphabetic order):

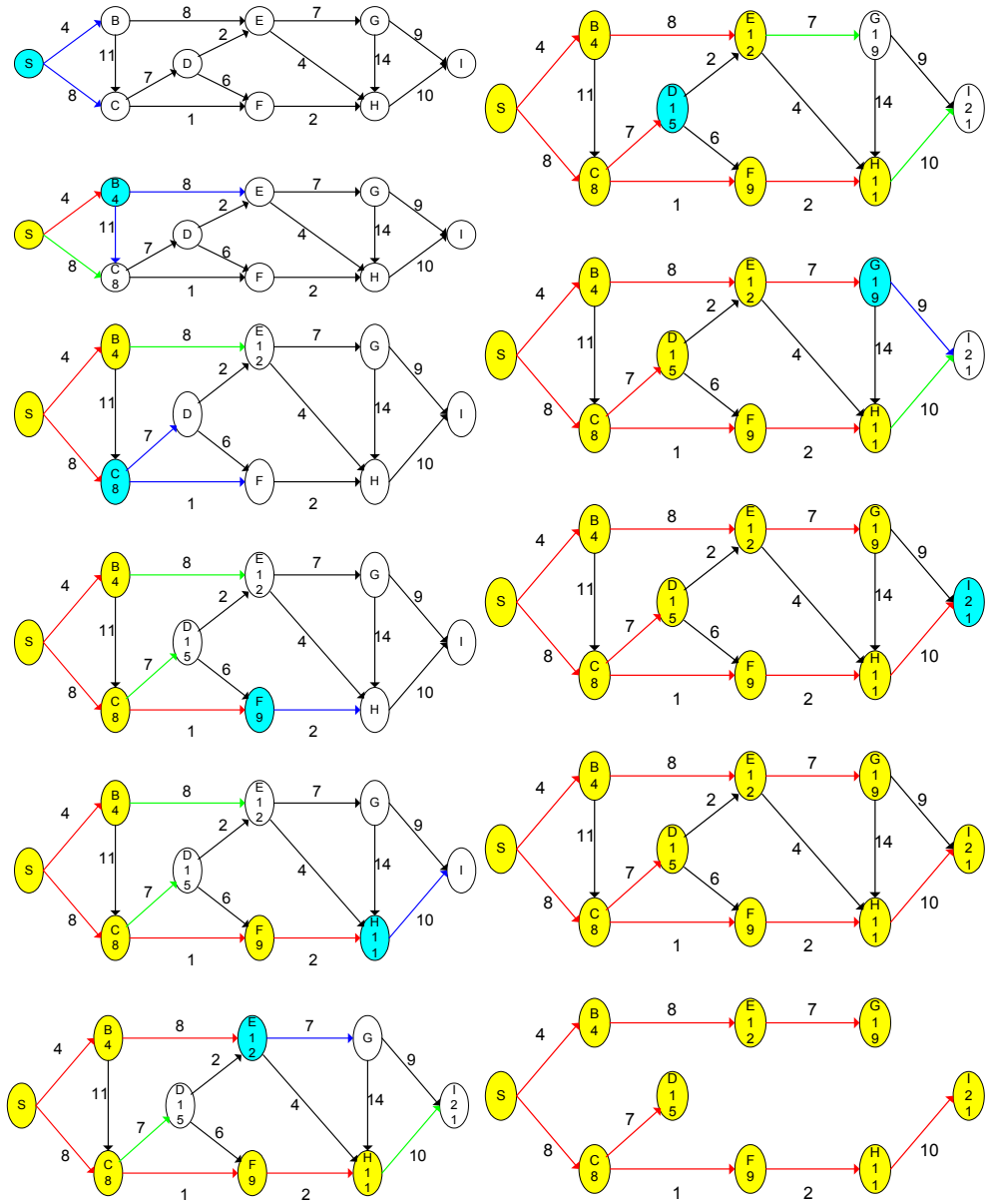


Dijkstra's Algorithm:

Here we assume that we have a single source and that $w(u, v) \geq 0$ for all edges. The algorithm maintains a set S of vertices whose final shortest path weights from the source have already been determined. It does this in increasing order of $\delta(s, j)$ values. The algorithm repeatedly selects a vertex $u \in V - S$ with minimum shortest path estimate, inserts u into S, and relaxes all edges leaving u. We maintain a priority queue Q that contains all vertices in $V - S$, keyed by their d values.

```
DIJKSTRA(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
S ⟵ φ
Q ⟵ V[G]
while Q ≠ φ
    do u ⟵EXTRACT-MIN(Q)
        S ⟵ S ∪ {u}
        for each vertex v ∈ Adj[u]
            do RELAX(u, v, w)
```

The following diagram shows the evolution of this algorithm:



Bellman Algorithm:

This algorithm attempts to solve Bellman's equations as follows:

$$\begin{aligned} d^1[s] &= 0; d^1[j] = w(s,j) \quad \text{for } j \neq s \\ d^m[j] &= \min\{d^{m-1}[j], \min_{k \neq j}[d^{m-1}[k] + w(k,j)]\} \quad \text{for all } j \end{aligned}$$

Algorithm stops when we compute $d^n[j]$ or if $d^m[j] = d^{m-1}[j]$ for all j for some $m < n$.

If you think of $d^m[j]$ as the minimum weight path from s to j **using no more than** m **edges**, you can see why this works. This is a dynamic programming algorithm. If $d^m[s] < 0$ at any step, we have a negative cycle reachable from s.

The time complexity of this algorithm is $O(|V[G]|^3)$. For graphs with fewer edges, we have a faster algorithm that is described below.

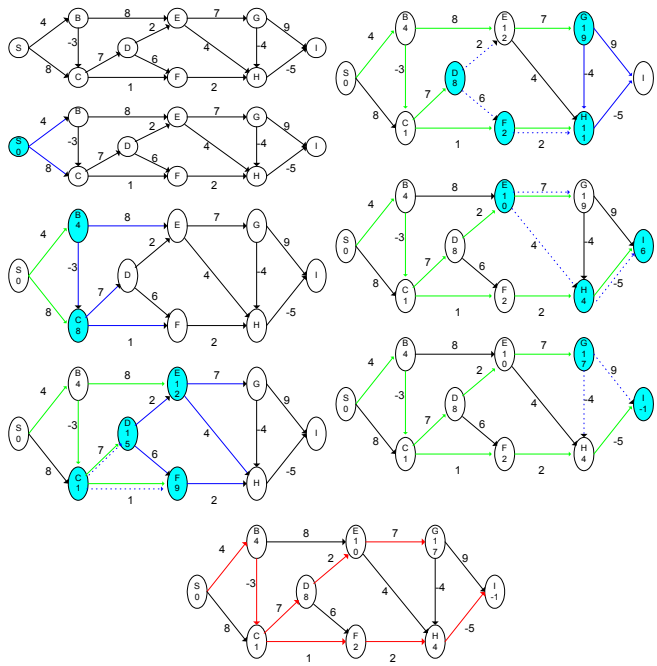
Bellman-Ford Algorithm:

This algorithm is also for a single source to all other nodes shortest path problem. But it can handle the case when there are negative weights for edges. The input is (G, w, s) – the same triple as in Dijkstra's algorithm. The output of the algorithm is an indication of the presence of a negative cycle if this is the case or a set of shortest paths if there is no negative cycle. The algorithm returns TRUE if and only if the graph contains no negative cycles that are reachable from s.

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

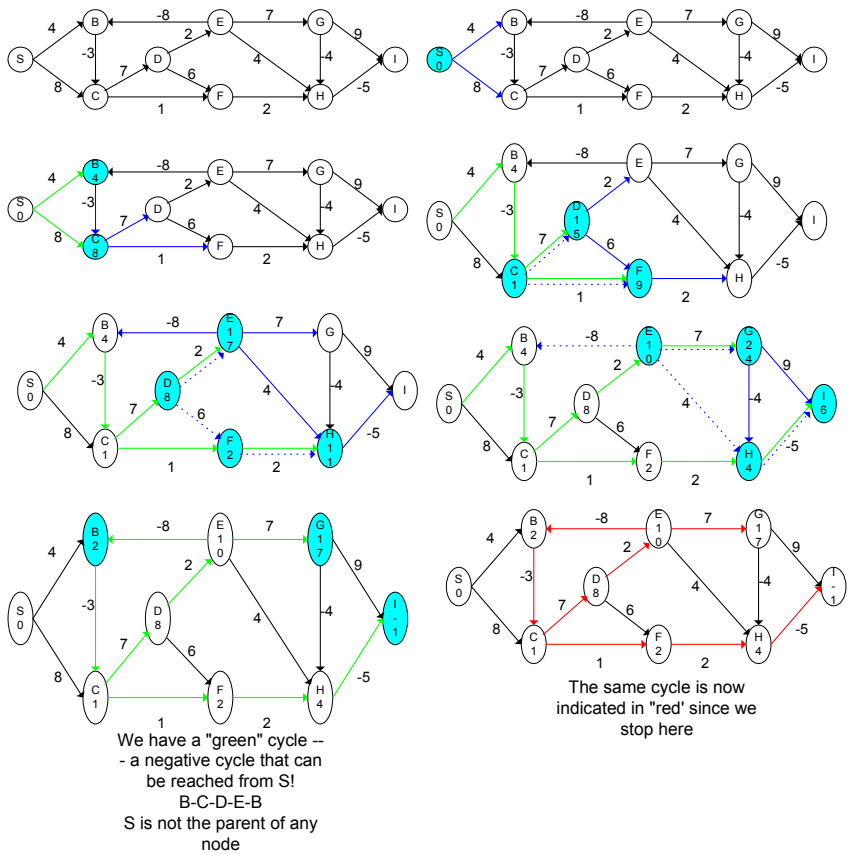
The following diagrams show the evolution of this algorithm on a problem

that has no negative cycles:



We now show a case when there is a negative cycle that can be reached from

the origin:



There are two ways to detect this in this algorithm one of which is shown above. The other is to check if the values for $d[v]$ keep changing for some node even in the $|V|$ step. That is $d^{(|V|-1)}[v] \neq d^{(|V|)}[v]$ for some node v.

All-Pairs Shortest Paths:

Given a matrix W with $w(i,j)$ representing the direct distance between i and j (or the edge weight of (i,j)). We can generalize Bellman's algorithm to the following:

$$\begin{aligned} l_{i,j}^{m} &= \min\{l_{i,j}^{m-1}, \min_{k \neq j}[l_{i,k}^{m-1} + w(k,j)]\} \\ &= \min_{k}[l_{i,k}^{m-1} + w(k,j)] \end{aligned}$$

We assume that $w(j,j) = 0$ for all j. If the graph has no negative cycle, then $\delta(i,j) = l_{i,j}^{n-1} = l_{i,j}^{n}$ for all i,j. View $l_{i,j}^{m}$ as the minimum length among all paths from i to j **using no more than** m **edges**.If we denote the matrix

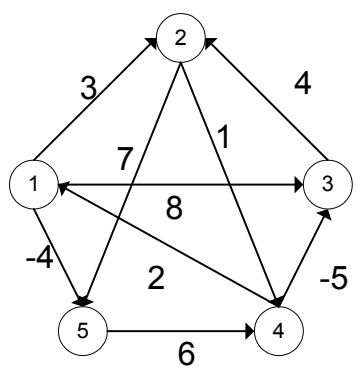
whose elements are $l_{i,j}^m$ by $L^{(m)}$, then we have the relations:

$$\begin{aligned} L^{(1)} &= W \\ L^{(m)} &= L^{(m-1)} \oplus W = W \oplus L^{(m-1)} = L^{(i)} \oplus L^{(m-i)} \end{aligned}$$

where $A \oplus B = C$ means the following: $c_{i,j} = \min_k[a_{i,k} + b_{k,j}]$. We assume that both matrices are square from now on.

MATRIX-MINADDITION(A, B)
$n \longleftarrow rows[A]$
let C be an $n \times n$ matrix
for $i \longleftarrow 1$ **to** n
  do for $j \longleftarrow 1$ **to** n
    do $c_{i,j} \longleftarrow \infty$
      for $k \longleftarrow 1$ **to** n
        do $c_{i,j} \longleftarrow \min[c_{i,j}, a_{i,k} + b_{k,j}]$
return C

This is known as **min-addition operation** on two matrices and is similar to multiplication of matrices with the sum operation replaced by minimum and the product operation replaced by addition. We can compute these in powers of 2: $L^{(1)}, L^{(2)}, L^{(4)}, ...$ This requires $\lg|V|$ such matrix operations and hence the time complexity is $O(|V|^3 \lg(|V|))$. This algorithm is shown in the following example with its W matrix



0	3	8	∞	-4
∞	0	∞	1	7
∞	4	0	∞	∞
2	∞	-5	0	∞
∞	∞	∞	6	0

$= W = L^{(1)}$

0	3	8	2	-4
3	0	-4	1	7
∞	4	0	5	11
2	-1	-5	0	-2
8	∞	1	6	0

$= L^{(2)}$

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

$= L^{(4)}$

There is a slightly faster algorithm due to R.W.Floyd.

See page 561 of your book for the same example worked out with Floyd-Warshall Algorithm.

Floyd-Warshall Algorithm:

FLOYD-WARSHALL(W)
$n \longleftarrow rows[W]$
$D^{(0)} \longleftarrow W$
for $k \longleftarrow 1$ **to** n
  do for $i \longleftarrow 1$ **to** n
    do for $j \longleftarrow 1$ **to** n
      $d_{i,j}^{(k)} \longleftarrow \min[d_{i,j}^{(k-1)}, d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)}]$
return $D^{(n)}$

Here we think of $d_{i,j}^{(k)}$ as the minimum length of all paths from i to j **using as intermediate nodes only those from the set** $\{1, 2, ..., k\}$**.**