**Shortest Paths Algorithms**

**Variations of Shortest Path Algorithms:**

* Single source shortest path

Shortest path computed for one source to multiple destinations.

* Single Destination shortest path

Shortest path computed for multiple sources to a single destination

* Single Pair

(Single Source and Single Destination)

* All source shortest path

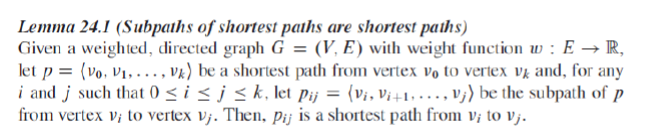
Shortest path computed from multiple sources to multiple destinations/

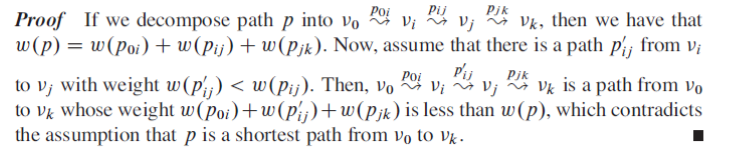
**Optimal Substructure:**

Shortest Path algorithms rely on the property that the shortest path between two vertices contains other shortest paths within it.

Example: A -> B -> C -> D

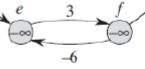
If the shortest path from A to D is through B and C, then the shortest path from B to C is B->C





**Negative Weight Cycles:**

If the resultant weight of a cycle is negative, then this cycle is said to be Negative Weight Cycle.

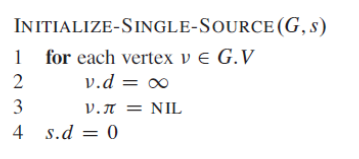
Example  3-6=-3, thus it is a negative weight cycle.

**Single Source Algorithms**

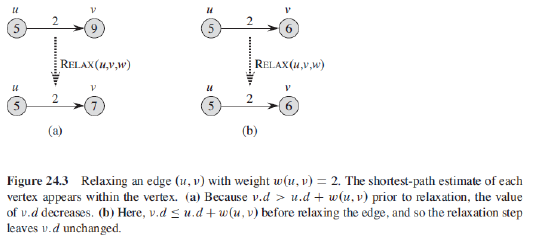
**Common Functions:**

In each of these algorithms we keep track of the current shortest path with each vertex using an array with size equals the number of vertices of the graph. Each index represents a vertex. Ex: arr[1] = 9 , then the shortest path from the source to vertex 1 is 9 and so on.

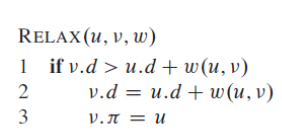
Initially, the path from the source to all the vertices is infinity except for the path from the source to the source is zero.



***Edge relaxation****.* To relax an edge v->w means to test whether the best-known way from s to w is to go from s to v, then take the edge from v to w, and, if so, update our data structures.

Example: 

**Implementation:**



Bellman-Ford Algorithm

**Idea:**

Input: Graph and a source vertex src

Output: Shortest distance to all vertices from src. If there is a negative weight cycle, then shortest distances are not calculated, negative weight cycle is reported.

**Steps:**

1) This step initializes distances from source to all vertices as infinite and distance to source itself as 0. Create an array dist[] of size |V| with all values as infinite except dist[src] where src is source vertex.

2) This step calculates shortest distances. Do following |V|-1 times where |V| is the number of vertices in given graph.

…..a) Do following for each edge u-v

………………If dist[v] > dist[u] + weight of edge uv, then update dist[v]

………………….dist[v] = dist[u] + weight of edge uv

3) This step reports if there is a negative weight cycle in graph. Do following for each edge u-v

……If dist[v] > dist[u] + weight of edge uv, then “Graph contains negative weight cycle”

The idea of step 3 is, step 2 guarantees shortest distances if graph doesn’t contain negative weight cycle. If we iterate through all edges one more time and get a shorter path for any vertex, then there is a negative weight cycle

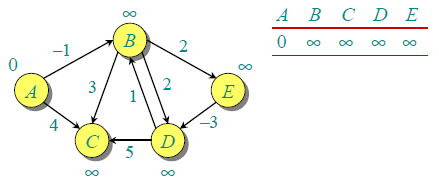
***How does this work?*** Like other Dynamic Programming Problems, the algorithm calculates shortest paths in bottom-up manner. It first calculates the shortest distances which have at-most one edge in the path. Then, it calculates shortest paths with at-most 2 edges, and so on. After the i-th iteration of outer loop, the shortest paths with at most i edges are calculated. There can be maximum |V| – 1 edges in any simple path, that is why the outer loop runs |v| – 1 times. The idea is, assuming that there is no negative weight cycle, if we have calculated shortest paths with at most i edges, then an iteration over all edges guarantees to give shortest path with at-most (i+1) edges

Note: The maximum number of edges in a path between two vertices does not exceed V-1, that is why we loop V-1 times.

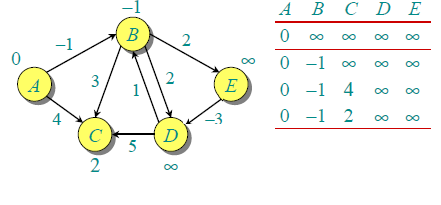
**Example:**

Let all edges are processed in following order: (B,E), (D,B), (B,D), (A,B), (A,C), (D,C), (B,C), (E,D).

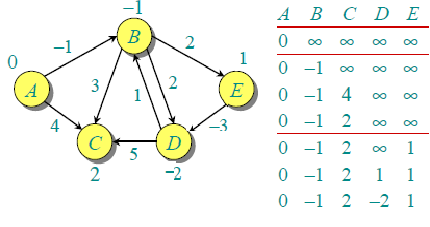
The initial Graph



After 1st iteration

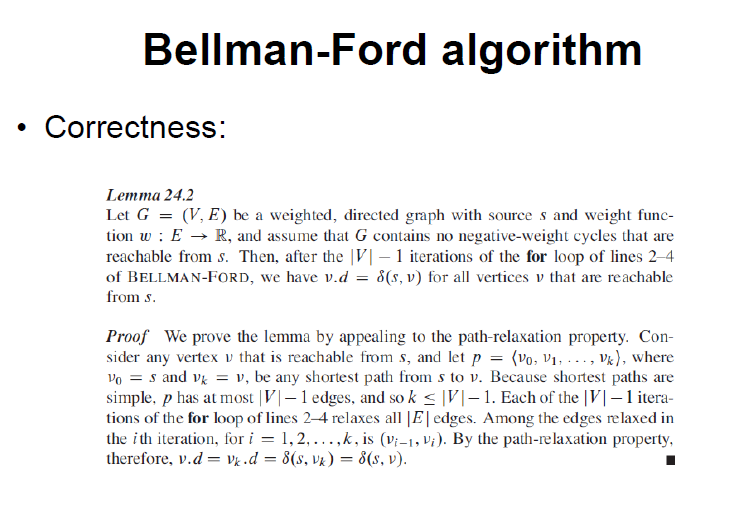
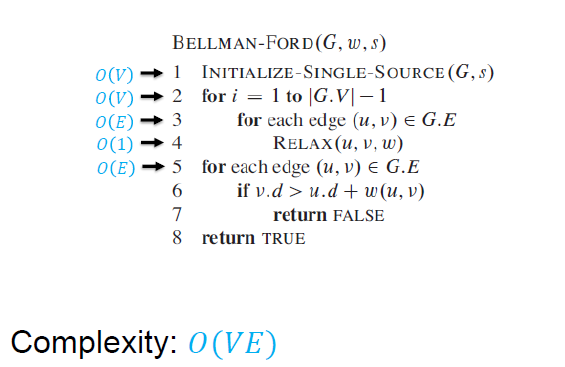


After 2nd iteration



The third and fourth iterations will not change anything further

**Code:**



Dijkstra’s Algorithm

**Idea:**

We generate a SPT (shortest path tree) with given source as root. We maintain two sets, one set contains vertices included in shortest path tree, other set includes vertices not yet included in shortest path tree. At every step of the algorithm, we find a vertex which is in the other set (set of not yet included) and has minimum distance from source.

Below are the detailed steps used in Dijkstra’s algorithm to find the shortest path from a single source vertex to all other vertices in the given graph.

**Steps:**

1) Create a set sptSet (shortest path tree set) that keeps track of vertices included in shortest path tree, i.e., whose minimum distance from source is calculated and finalized. Initially, this set is empty.

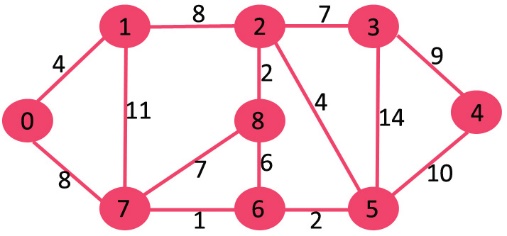
2) Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE. Assign distance value as 0 for the source vertex so that it is picked first.

3) While sptSet doesn’t include all vertices

….a) Pick a vertex u which is not there in sptSetand has minimum distance value.

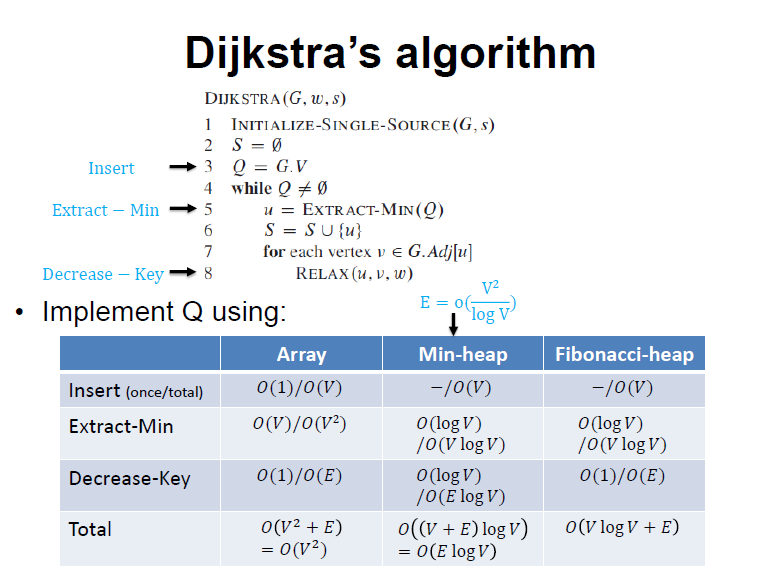
….b) Include u to sptSet.

….c) Update distance value of all adjacent vertices of u. To update the distance values, iterate through all adjacent vertices. For every adjacent vertex v, if sum of distance value of u (from source) and weight of edge u-v, is less than the distance value of v, then update the distance value of v.

**Example:** The initial Graph 

Finally, 



DAG Shortest Path Algorithm

**Idea:**

This algorithm is used for Directed Acyclic Graphs because it has better complexity than Dijkstra and Bellman-Ford.

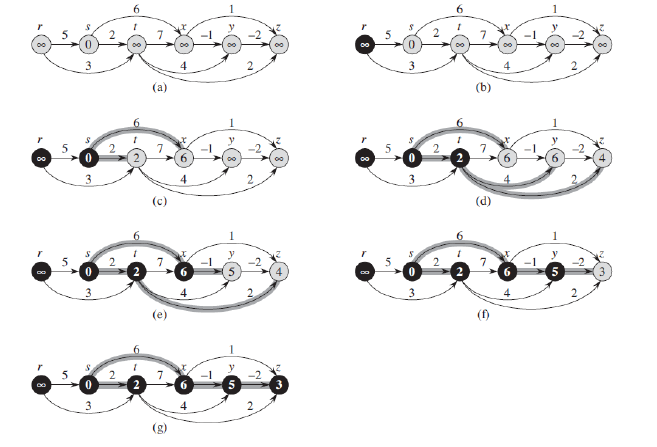
We initialize distances to all vertices as infinite and distance to source as 0, then we find a topological sorting of the graph. [Topological Sorting](https://www.geeksforgeeks.org/topological-sorting/) of a graph represents a linear ordering of the graph. Once we have topological order (or linear representation), we one by one process all vertices in topological order. For every vertex being processed, we update distances of its adjacent using distance of current vertex.

**Steps:**

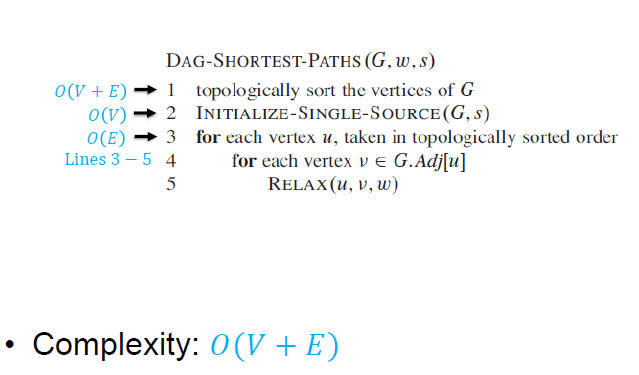
**1)** Initialize dist[] = {INF, INF, ….} and dist[s] = 0 where s is the source vertex.  
**2)** Create a topological order of all vertices.  
**3)**Do following for every vertex u in topological order.  
………..Do following for every adjacent vertex v of u  
………………if (dist[v] > dist[u] + weight(u, v))  
………………………dist[v] = dist[u] + weight(u, v)

**Example:**

The following figure shows the execution of the algorithm after the topological sort.

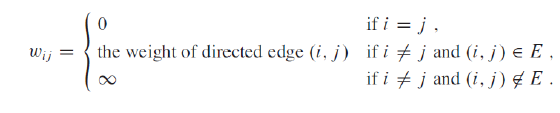


**Code:**

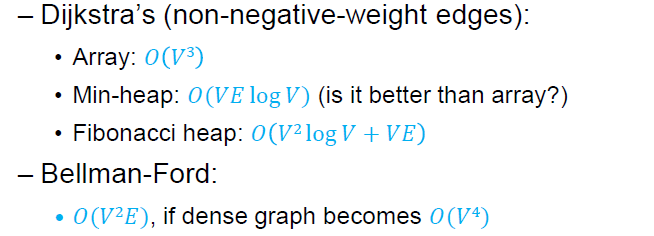


**All Pairs Shortest Paths**

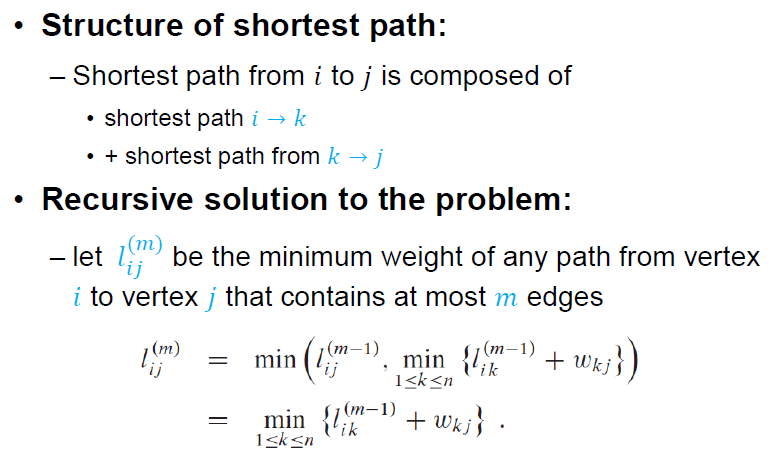
The input to these algorithms is a Directed Graph G(V,E).

The output is nxn matrix W = wij which is the shortest path from I to j

We can get this matrix simply running one of the single source algorithms once on each vertex, the coplexties would be as follows:



But we can do better than this , we can use Dynamic Programming to calculate all pairs shortest paths.



Slow All Pairs Shortest Path

**Idea:**

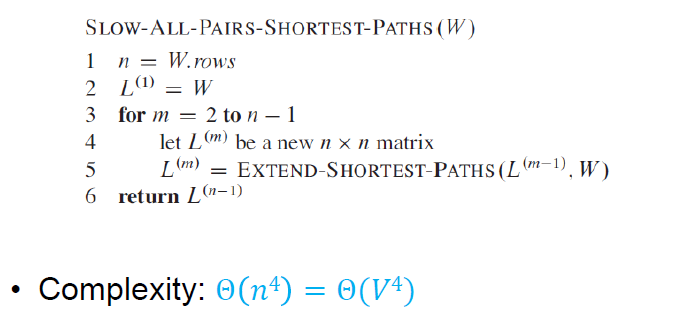
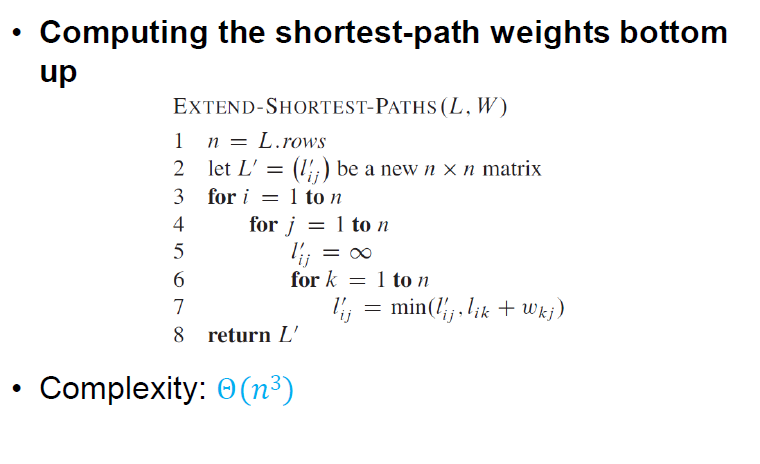
Find all vertices reachable in two edges, L(2) , save the matrix, and use it to

find all vertices reachable in three edges, L(3), save the matrix, and use it to

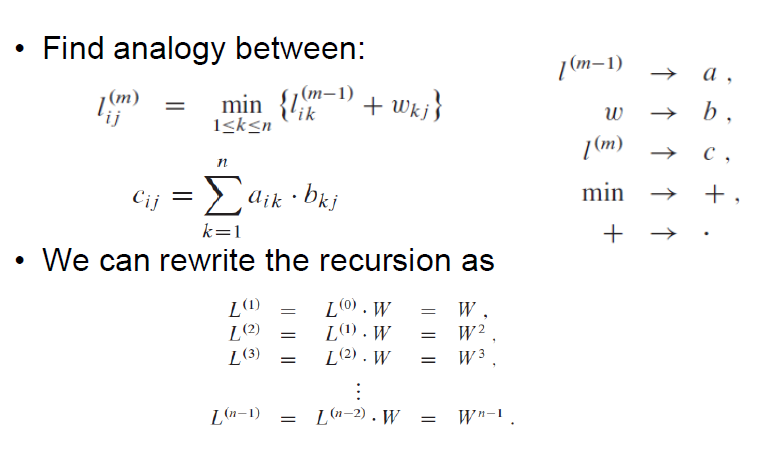
find all vertices reachable in four edges, L(4), and so on until we find L(n-1) .

This matrix will contain the shortest path between every pair of vertices in

the graph.



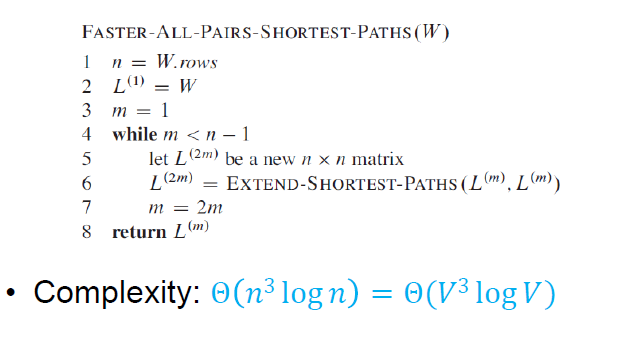
Analogy between Multiplication and Getting the minimum path:



Faster All Pairs Shortest Path

**Idea:**

We are interested in the matrix L(n-1), so we don’t need to calculate all matrices L(1),L(2),L(3) upto matrix L(n-l), instead we change our step per loop instead of m=m+1 , we do m=2m , so compute L(1),L(2),L(4),L(8).. and so on, decreasing the complexity of this loop to logn instead of n.



Floyd-Warshall Algorithm

**Idea:**

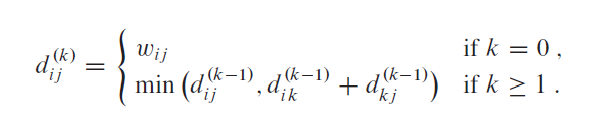
Find all vertices reachable using intermediate nodes in the range 1...1 (D(1)), save the matrix, and use it to find all vertices reachable using intermediate vertices in the range 1...2 (D(2)), save the matrix, and use it to find all vertices reachable using intermediate vertices in the range 1...3

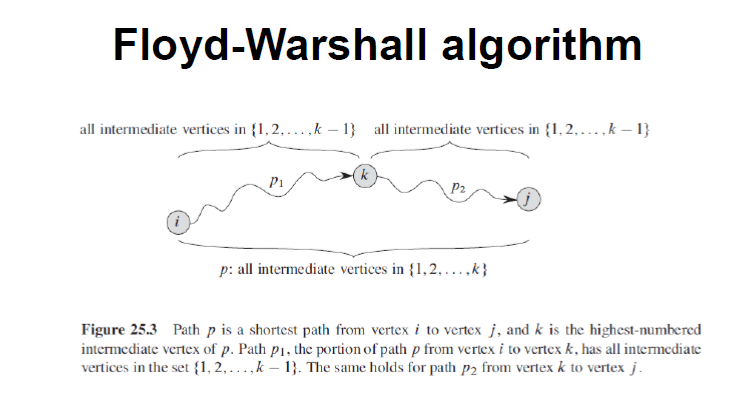
(D(3)), and so on until we find D(n). This matrix will contain the shortest path between every pair of vertices in the graph.

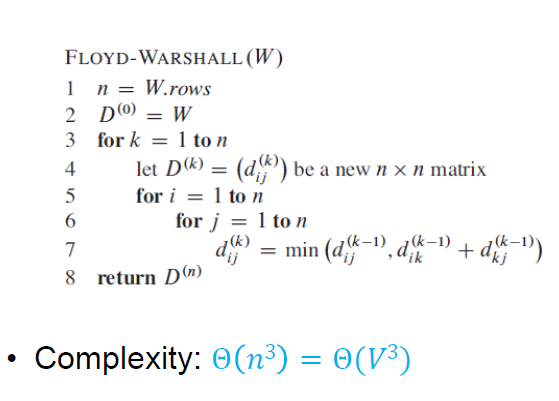
**Steps:**

In each iteration of Floyd-Warshall algorithm is this matrix recalculated, so it contains lengths of paths among all pairs of nodes using gradually enlarging set of intermediate nodes. The matrix D1, which is created by the first iteration of the procedure, contains paths among all nodes using exactly one (predefined) intermediate node. D2 contains lengths using two predefined intermediate nodes. Finally, the matrix Dn uses n intermediate nodes.

For every pair (i, j) of source and destination vertices respectively, there are two possible cases.  
**1)** k is not an intermediate vertex in shortest path from i to j. We keep the value of d[i][j] as it is.  
**2)** k is an intermediate vertex in shortest path from i to j. We update the value of d[i][j] as the min between d[i][j] and d[i][k] + d[k][j].







Transitive Closure

**Idea:**

The graph is given in the form of adjacency matrix say ‘graph[V][V]’ where graph[i][j] is 1 if there is an edge from vertex i to vertex j or i is equal to j, otherwise graph[i][j] is 0.

Floyd Warshall Algorithm can be used, we can calculate the distance matrix dist[V][V] using Floyd Warshall, if dist[i][j] is infinite, then j is not reachable from i, otherwise j is reachable and value of dist[i][j] will be less than V.

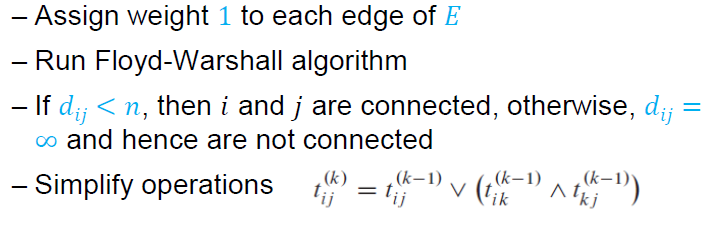
Instead of directly using Floyd Warshall, we can optimize it in terms of space and time, for this particular problem. Following are the optimizations:

**1)** Instead of integer resultant matrix (dist[V][V] in floyd warshall), we can create a boolean reach-ability matrix reach[V][V] (we save space). The value reach[i][j] will be 1 if j is reachable from i, otherwise 0.

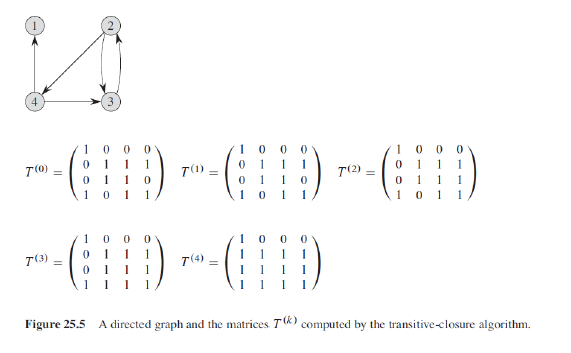
**2)** Instead of using arithmetic operations, we can use logical operations. For arithmetic operation ‘+’, logical and ‘&&’ is used, and for min, logical or ‘||’ is used. (We save time by a constant factor. Time complexity is same though)

The resultant matrix is composed of 1s an 0s.

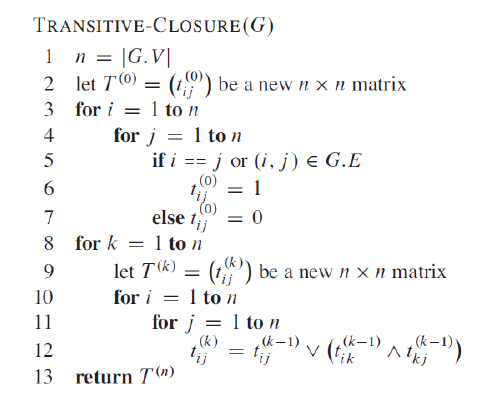
**Steps:**



**Example:**

****

**Code:**

****

Johnson’s Algorithm

**Idea:**

The problem is to find shortest paths between every pair of vertices in a given weighted directed Graph and weights may be negative. We have discussed Floyd Warshall Algorithm for this problem. Time complexity of Floyd-Warshall Algorithm is Θ(V3). Using Johnson’s algorithm, we can find all pair shortest paths in O(V2log V + VE) time. Johnson’s algorithm uses both Dijkstra and Bellman-Ford as subroutines.

If we apply Dijkstra’s Single Source shortest path algorithm for every vertex, considering every vertex as source, we can find all pair shortest paths in O(V\*VlogV) time. So, using Dijkstra’s single source shortest path seems to be a better option than Floyd-Warshall, but the problem with Dijkstra’s algorithm is, it doesn’t work for negative weight edge.

The idea of Johnson’s algorithm is to re-weight all edges and make them all positive, then apply Dijkstra’s algorithm for every vertex.

**Steps:**

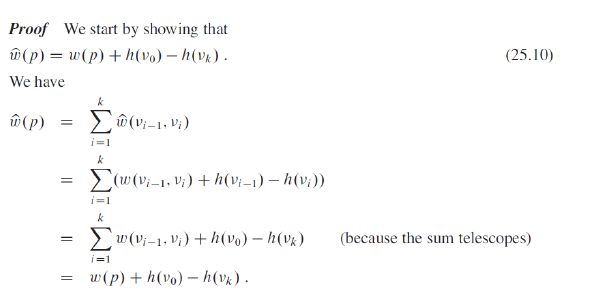
**1)** Let the given graph be G. Add a new vertex s to the graph, add edges from new vertex to all vertices of G. Let the modified graph be G’.

**2)** Run Bellman-Ford algorithm on G’ with s as source. Let the distances calculated by Bellman-Ford be h[0], h[1], .. h[V-1]. If we find a negative weight cycle, then return. Note that the negative weight cycle cannot be created by new vertex s as there is no edge to s. All edges are from s.

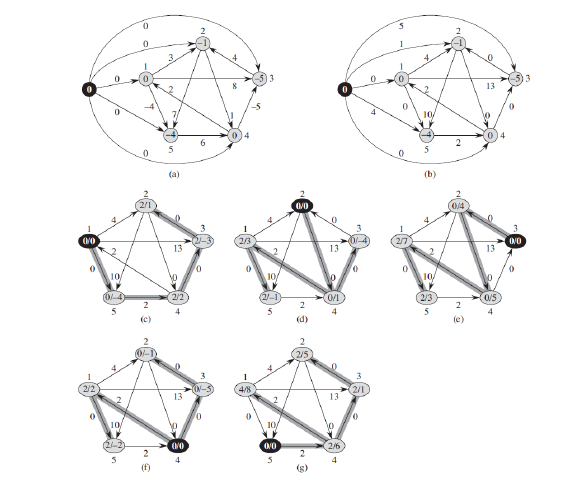
**3)** Reweight the edges of original graph. For each edge (u, v), assign the new weight as “original weight + h[u] – h[v]”.



**4)** Remove the added vertex s and run Dijkstra’s algorithm for every vertex.



**Example:**



**Code:**

