Depth First Search
&
Directed Acyclic Graphs

Charalampos Papamanthou
cpap@csd.uoc.gr

Department of Computer Science
University of Crete

A Review for the Course **Graph Algorithms**

Winter 2004
LaTeX

1

1 Depth First Search

1.1 General

Depth First Search (DFS) is a systematic way of visiting the nodes of either a directed or an undirected graph. As with breadth first search, DFS has a lot of applications in many problems in Graph Theory. It comprises the main part of many graph algorithms. DFS visits the vertices of a graph in the following manner. It selects a starting vertex v. Then it chooses an incident edge (v, w) and searches recursively deeper in the graph whenever possible. It does not examine all the incident edges one by one at the same time, but on the other hand, it goes deeper and deeper in the graph till no other such path exists. When all edges incident to v have been explored, the algorithm *backtracks* to explore edges leaving the vertex from which v was discovered.

1.2 The Algorithm

DFS has two main differences form BFS. First of all, it does not discover all vertices at distance k before discovering vertices at distance $k + 1$, but discovers one vertex at a specific depth each time. Additionally, the *predecessor* subgraph produced by DFS may be composed of several trees, because the search may be repeated from several sources. This predecessor subgraph forms a depth first forest E composed of several *depth first* trees and the edges in E are called tree edges. The predecessor subgraph in DFS is the graph $G = (V, E_\pi)$ where

$$E_\pi = \{(\pi[v], v) : v \in V \land \pi[v] \neq nil\}$$

On the other hand, the predecessor subgraph of BFS forms a tree.

The DFS procedure takes as input a graph G, and outputs its predecessor subgraph in the form of a depth-first forest. In addition, it assigns two timestamps to each vertex: discovery and finishing time. The algorithm initializes each vertex to *white* to indicate that they are not discovered yet. It also sets each vertex's parent to null. The procedure begins by selecting one vertex u from the graph, setting its color to *gray* to indicate that the vertex is now discovered (but not finished) and assigning to it discovery time 0. For each vertex v that belongs to the set $Adj[u]$, and is still marked as *white*, DFS-Visit is called recursively, assigning to each vertex the appropriate discovery time $d[v]$ (the time variable is incremented at each step). If no white descendant of v exists, then v becomes black and is assigned the appropriate finishing time, and the algorithm returns to the exploration of v's ancestor $p[v]$.

If all of u's descendants are black, u becomes black and if there are no other white vertices in the graph, the algorithm reaches a finishing state, otherwise a new "source" vertex is selected, from the remaining white vertices, and the procedure continues as before.

As referred above, the DFS procedure computes two timestamps for each vertex v, the *dis-*

covery time, denoted with $d[v]$ and the finishing time denoted with $f[v]$. These timestamps are integers between 1 and $2n$, since there is one discovery event and one finishing event for each of the n vertices. Next, we present the pseudocode of the algorithm.

Algorithm DFS(G)
1 **for** *each vertex $u \in V(G)$*
2      $color[u] \leftarrow white$
3      $\pi[u] \leftarrow nil$
4 $time \leftarrow 0$
5 **for** *each vertex $u \in V(G)$*
6   **if** $color[u] == white$
7   **DFS-visit**(u)

DFS-visit(u)
1 $color[u] \leftarrow gray$ *{white vertex u has just been discovered}*
2 $d[u] \leftarrow time \leftarrow time + 1$
3 **for** *each vertex $v \in Adj[u]$*   *{explore edge (u, v)}*
4   **if** $color[v] == white$
5          $\pi[v] \leftarrow u$
6   **DFS-visit**(v)
7 $color[u] \leftarrow black$ *{Blacken u;it is finished}*
8 $f[u] \leftarrow time \leftarrow time + 1$

Procedure **DFS** works as follows. Lines 1-3 paint all vertices white and initialize their π field to *nil*. Line 4 resets the global time counter. Lines 5-7 check each vertex in V in turn and, when a white vertex is found, visit it using **DFS-visit**. Every time **DFS-visit**(u) is called in line 7, vertex u becomes the root of a new tree in the depth-first forest. When **DFS** returns, every vertex u has been assigned a discovery time $d[u]$ and a finishing time $f[u]$.

In each **DFS-visit**(u) call, vertex u is initially white. Line 1 paints u gray, and line 2 records the discovery time $d[u]$ by incrementing and saving the global variable *time*. Lines 3-6 examine each vertex v adjacent to u and recursively visit v if it is white. As each vertex $v \in Adj[u]$ is considered in line 3, we say that edge (u, v) is *explored* by the depth-first search. Finally, after every edge leaving u has been explored, lines 7-8 paint u black and record the finishing time in $f[u]$.

The initialization part of DFS has time complexity $O(n)$, as every vertex must be visited once so as to mark it as "white". The main (recursive) part of the algorithm has time complexity $O(m)$, as every edge must be crossed (twice) during the examination of the adjacent vertices of every vertex. In total, the algorithm's time complexity is $O(m + n)$. An example of the BFS execution can be seen in Figure 1.
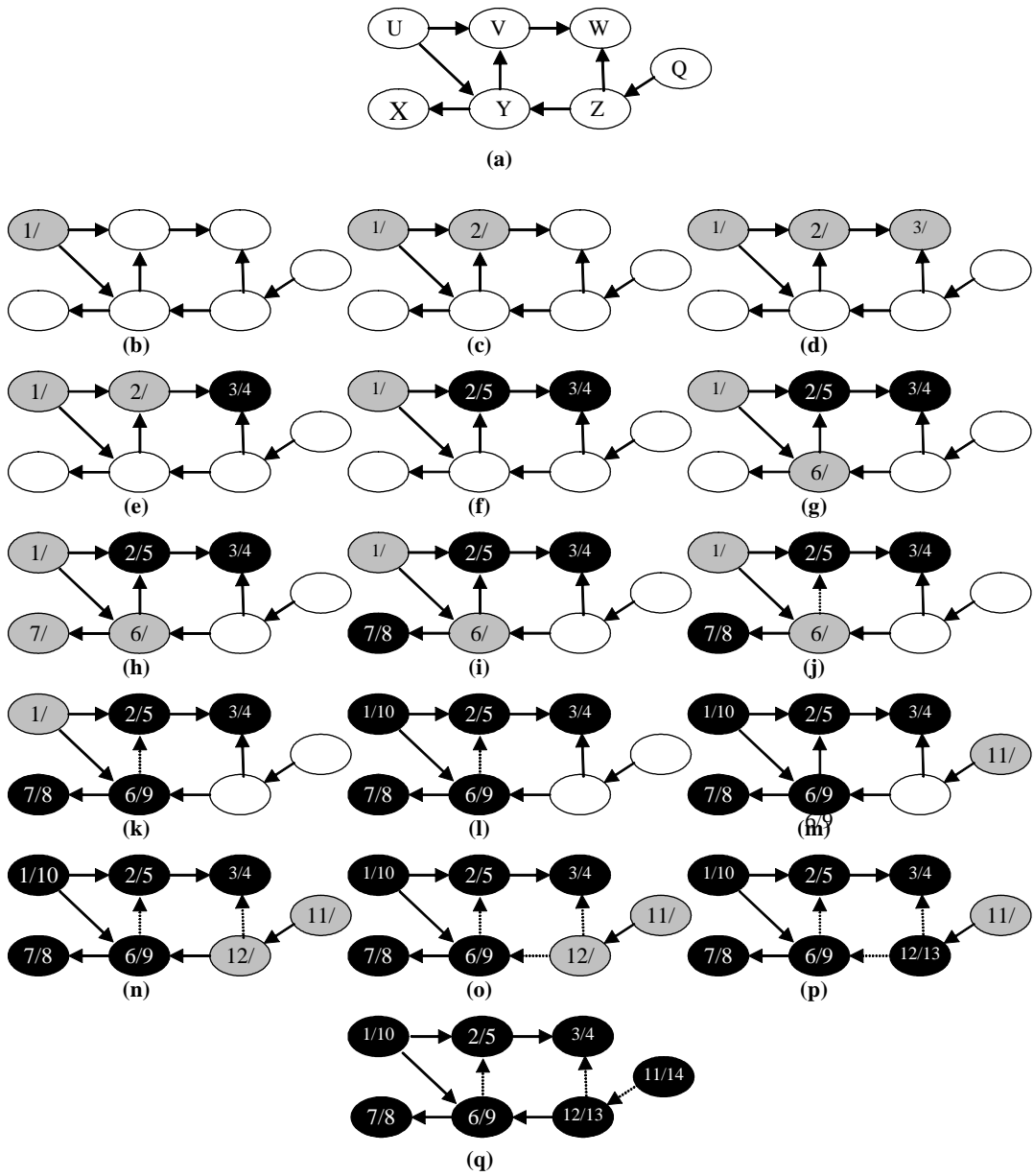
Figure 1: *(b) shows the progress of the DFS algorithm for the graph of (a). Starting from node U we can either discover node V or Y. Suppose that we discover node V, which has a single outgoing edge to node W. W has no outgoing edges, so this node is finished, and we return to V. From V there is no other choice, so this node is also finished and we return to U. From node U we can continue to discover Y and its descendants, and the procedure continues similarly. At stage (l) we have discovered and finished nodes U, V, W, X, Y. Selecting node Q as a new starting node, we can discover the remaining nodes (in this case Z).*

3

1.3 Some Properties

Depth-first search provides us with much information about the structure of the graph. Perhaps the most basic property of depth-first search is that the predecessor subgraph G_π, as it has been defined in the previous section, does indeed form a forest of trees, since the structure of the depth-first trees exactly mirrors the structure of recursive calls of **DFS-visit**. This means that the algorithm sets $u = \pi[v]$ if and only if **DFS-visit**(v) was called during a search of u's adjacency list.

Another important property of DFS is that discovery and finishing times have a *parenthesis structure*. If we represent the discovery of a vertex u with a left parenthesis "$(u$" and represent its finishing by a right parenthesis "$u)$", then the history of discoveries and finishings makes a well-formed expression in the sense that the parentheses are properly nested. The parenthesis structure p corresponding to the depth-first traversal of figure1 is the following

$$p = (s \ (z \ (\ y \ (x \ x) \ y) \ (w \ w) \ z) \ s) \ (t \ (v \ v) \ (u \ u) \ t)$$

Another way of stating the condition of the parenthesis structure is given in the following
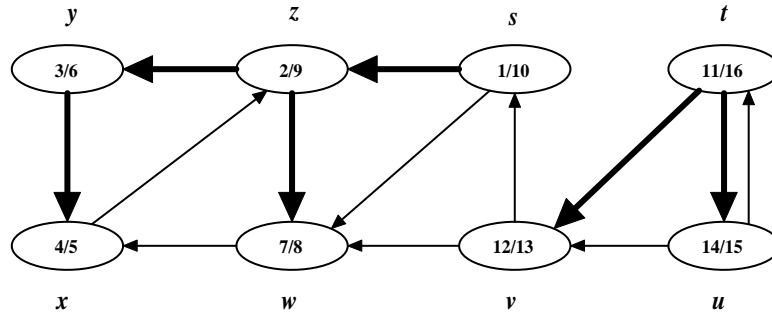

Figure 2: *A depth first traversal with the corresponding discovery and finishing times.*

theorem.

Theorem 1.1 (Parenthesis Theorem). *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 $[d[u], f[u]]$ and $[d[v], f[v]]$ are entirely disjoint,*

- *the interval $[d[u], f[u]]$ is contained entirely within the interval $[d[v], f[v]]$, and u is a descendant of v in the depth-first tree, or*

- *the interval $[d[v], f[v]]$ is contained entirely within the interval $[d[u], f[u]]$, and v is a descendant of u in the depth-first tree.*

4

Proof

We begin with the case in which $d[u] < d[v]$. There two subcases to consider, according to whether $d[v] < f[u]$ or not. In the first subcase, $d[v] < f[u]$, so v was discovered while u was still gray. This implies that v is a descendant of u. Moreover, since v was discovered more recently than u, all of its outgoing edges are explored, and v is finished, before the search returns to and finishes u. In this case, therefore, the interval $[d[v], f[v]]$ is entirely contained within the interval $[d[u], f[u]]$. In the other subcase, $f[u] < d[v]$, and $d[u] < f[u]$, implies that the intervals $[d[u], f[u]]$ and $[d[v], f[v]]$ are disjoint. The case in which $d[v] < d[u]$ is similar, with the roles of u and v reverse in the above argument.□

Corollary 1.2 (Nesting of descendants' intervals). *Vertex v is a proper descendant of vertex u in the depth-first forest for a (directed or undirected) graph G if and only if $d[u] < d[v] < f[v] < f[u]$.*

Proof

Immediate from the Parenthesis Theorem.□

Theorem 1.3 (White Path Theorem). *In a depth-first forest of a graph $G = (V, E)$, vertex v is a descendant of vertex u if and only if at the time $d[u]$ that the search discovers u, vertex v can be reached from u along a path consisting entirely of white vertices.*

Proof

To prove the direct, assume that v is a descendant of u. Let w be any vertex on the path between u and v in the depth-first tree, so that w is a descendant of u. By Corollary 1.2, $d[u] < d[w]$, and so w is white at time $d[u]$.

For the reverse case, suppose that vertex v is reachable from u along at path of white vertices at time $d[u]$, but v does not become a descendant of u in the depth-first tree. Without loss of generality, assume that every other vertex along the path becomes a descendant of u. (Otherwise, let v be the closest vertex to u along the path that doesn't become a descendant of u.) Let w be the predecessor of v in the path, so that w is a descendant of u (w and u may in fact be the same vertex) and, by corollary 1.2, $f[w] \leq f[u]$. Note that v must be discovered after u is discovered, but before w is finished. Therefore, $d[u] < d[v] < f[w] \leq f[u]$. Theorem 1.1 then implies that the interval $[d[v], f[v]]$ is contained entirely within the interval $[d[u], f[u]]$. By corollary 1.2, v must after all be a descendant of u.□

1.4   Edge Classification

One of the most important properties of depth first search is that the search can be used to clarify the edges of the input graph $G = (V, E)$. During DFS, the edges are separated into certain groups that reveal important information about the graph itself. For example, we will see that a directed graph is acyclic if and only if a depth first search yields no *back* edges. We can define four edge types in terms of the depth-first forest G_π produced by a depth-first search on a 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). A tree edge always describes a relation between a node and one of its direct descendants. This indicates that $d[u] < d[v]$, (u's discovery time is less than v's discovery time), so a tree edge points from a "low" to a "high" node.

2. **Back** edges are those edges (u, v) connecting a vertex u to an ancestor v in a depth-first tree. Self-loops are considered to be back edges. Back edges describe descendant-to-ancestor relations, as they lead from "high" to "low" nodes.

3. **Forward** edges are those non-tree edges (u, v) connecting a vertex u to a descendant v in a depth-first tree. Forward edges describe ancestor-to-descendant relations, as they lead from "low" to "high" nodes.

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. Cross edges link nodes with no ancestor-descendant relation and point from "high" to "low" nodes.

The DFS algorithm can be modified to classify edges as it encounters them. The key idea is that each edge (u, v) can be classified by the color of the vertex v that is reached when the edge is first explored (except that forward and cross edges are not distinguished):

- *White* indicates a tree edge,

- *Gray* indicates a back edge, and

- *Black* indicates a forward or cross edge

It is important to point out that, however, in an undirected graph there may be some ambiguity in the edge classification, since (u, v) and (v,u) is really the same edge. In such a case, the edge is classified as the *first* type in the classification list that applies. This means that the edge is classified according to whichever (u, v) or (v, u) is encountered first during the execution of the algorithm. We will now prove that forward and cross edges never occur in a depth first search of an undirected graph.

Theorem 1.4. *In a depth-first search of an undirected graph G, every edge of G is either a tree edge or a back edge.*

Proof
Let (u, v) be an arbitrary edge of G, and suppose without loss of generality that $d[u] < d[v]$. Then, v must be discovered and finished before we finish u, since v is on u's adjacency list. If the edge (u, v) is explored first in the direction from u to v, then (u, v) becomes a tree edge. If (u, v) is explored first in the direction from v to u, then (u, v) is a back edge, since

u is still gray at the time the edge is first explored.□

If we want to distinguish between back and cross edges, there is an easy way to do this. For every non-tree edge (u, v), that leads from a *high* node to a *low* node, we must determine if v is an ancestor of u: starting from u me must traverse the depth-first tree to visit u's ancestors. If v is found then (u, v) is a back edge, else -if we reach the root without having found v- (u, v) is a cross edge. This can easily be done in $O(n)$ time, as we have to traverse at most $n - 1$ edges to identify connection between two certain nodes of a tree. Following, we present a relative lemma.
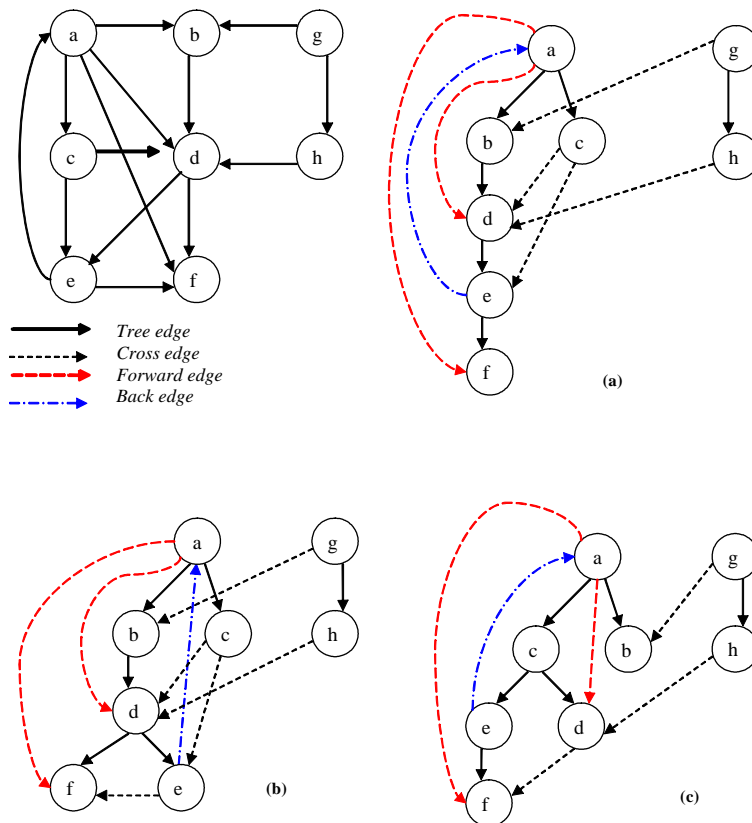

Figure 3: 3 possible scenarios of edge classification with DFS

Lemma 1.5. *A directed graph G contains no cycles if and only if a depth-first search of G yields no back edges.*

Proof

For the direct, assume that there is a back edge (u, v). Then vertex v is an ancestor of vertex u in the depth-first forest. There is thus a path from v to u in G, and the back edge (u, v) completes a cycle.

In reverse, suppose that G contains a cycle c. We show that a depth-first search of G yields

a back edge. Let v be the first vertex to be discovered in c, and let (u, v) be the preceding edge in c. At time $d[v]$, there is a path of white vertices from v to u. By the white-path theorem, vertex u becomes a descendant of v in the depth-first forest. Therefore (u, v) is a back edge.□

In figure 3, you can see three possible scenario of edge classification produced by a depth-first traversal of a directed graph.

At this point, we must add that DFS can be implemented non-recursively, by *simulating* the system stack of the recursion with a classic higher-level implementation of a stack. In this way, one can see the similarities between the two kinds of traversals, (i.e. in BFS we use a queue, whereas in DFS we use a stack).

2 Directed Acyclic Graphs

2.1 General

The Directed Acyclic Graphs (DAGs) comprise a very important class of graphs that play a major role in graph applications. DAG is a directed graph where no path starts and ends at the same vertex (i.e. it has no cycles). It is also called an oriented acyclic graph. A DAG has at least one source (i.e. a node that has no incoming edges) and one sink (i.e. a node that has no outgoing edges). If it has more that one source, we can create a DAG with a single source, by adding a new vertex, which only has outgoing edges to the sources, thus becoming the new single (super-) source. The same method can be applied in order to create a single (super-) sink, which only has incoming edges from the initials sinks.

2.2 Topological Numberings and Sortings

Let $G = (V, E)$ be a directed acyclic graph. A *topological numbering* is a function $x : V \rightarrow \{1, \ldots, n\}$ such that

$$x(v) \geq x(u) \quad \forall (u, v) \in E \tag{2.1}$$

Respectively, we can talk about topological sorting of a directed acyclic graph when equation 2.1 is satisfied as a strict inequality. Therefore, a *topological sorting* is a function $y : V \rightarrow \{1, \ldots, n\}$ such that

$$y(v) > y(u) \quad \forall (u, v) \in E \tag{2.2}$$

It is easy to see that equation 2.1 defines a linear ordering of all its vertices such that if G contains an edge (u, v) then u appears before v in the ordering. Note that topological sorting assigns distinct integers to every vertex of G. Thus, a topological numbering can easily be derived from a topological sorting.

Additionally, the numberings defined above can be redefined as *weighted* numberings if we assign positive weights $w(e)$ for every edge $e \in E$ and add to the right part of the inequalities 2.1, 2.2 the quantity $w(e)$, $e = (u, v)$. Thus a weighted topological numbering of a directed acyclic graph $G = (V, E)$ with a positive weight function w is a function $r : V \rightarrow \{1, \ldots, n\}$ such that

$$r(v) \geq r(u) + w(e) \quad \forall e = (u, v) \in E \tag{2.3}$$

Note that the normal numberings can be derived form the weighted numberings if we set all the weights of the graph equal to zero. Additionally, we say that a weighted topological is optimal if the quantity $\max_{u,v} |x(v) - x(u)|$ is minimized.

In figure 4, we can see a directed acyclic graph G. The reader can verify that vector $x = [\, 5 \quad 4 \quad 4 \quad 3 \quad 3 \quad 2 \quad 1\,]$ is a topological numbering and vector $y = [\, 7 \quad 6 \quad 5 \quad 4 \quad 3 \quad 2 \quad 1\,]$ is a topological sorting of graph G. Note that the labels beside the nodes of figure 4 are the identity numbers of the nodes of the graph, and not a topological numbering or sorting.
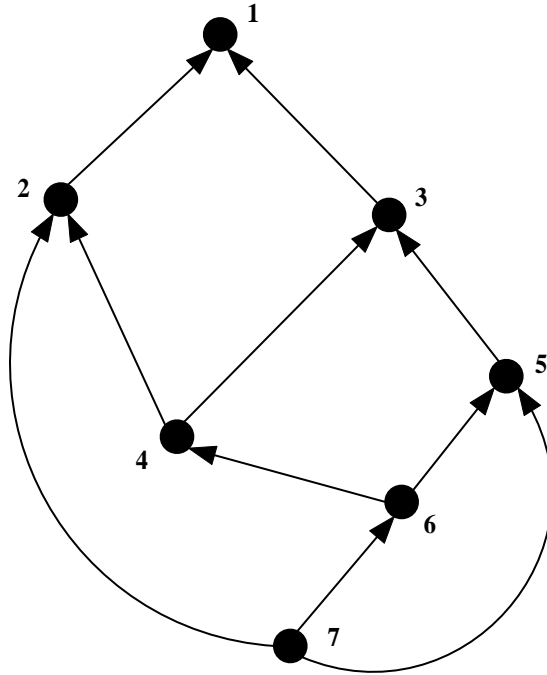
Figure 4: *A directed acyclic graph.*

2.2.1 Computing a Topological Sorting

As the topological numbering can easily be derived from a topological sorting we will present two algorithms for the computation of a topological sorting.

One very clever algorithm for the computation of the topological sorting of a directed acyclic graph $G = (V, E)$ is based on bucket sorting. The algorithm works as follows. Our graph has for sure a source with in-degree equal to zero. We build up a list with $n = \max_{i \in V} \{deg^-(i)\} + 1$ entries. Each entry $0 \leq i \leq n$ is a linked list of nodes k with $deg^-(k) = i$. Each node in this list has pointers to its reachable nodes (i.e. nodes t that can be reached following the outgoing edges (i, t)). The algorithm subtracts one by one nodes k such that $deg^-(k) = 0$, giving them an increasing topological sorting index and simultaneously decreasing the in-degree of their out-neighborhood by one. The algorithm goes on till all nodes get an index.

The main idea is that at each step we exclude a node which does not depend on any other. That maps to removing an entry v_1 from the 0th index of the bucket structure (which contains the sources of G) and so reducing by one the in-degree of the entry's adjacent nodes (which are easily found by following the "red arrows" from v_1) and re-allocating them in the bucket. This means that node v_2 is moved to the list at the 0th index, and thus becoming a source. v_3 is moved to the list at the 1st index, and v_4 is moved to the list at the 2nd index. After we subtract the source from the graph, new sources may appear,

because the edges of the source are excluded from the graph, or there is more than one source in the graph (see figure 5). Thus the algorithm subtracts a source at every step and inserts it to a list, until no sources (and consequently no nodes) exist. The resulting list represents the topological sorting of the graph. It easy to prove that the algorithm indeed
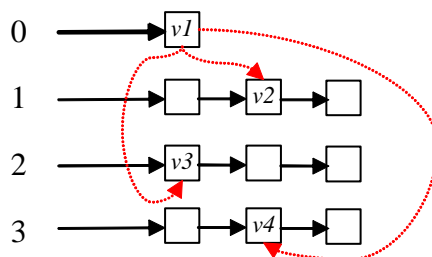

Figure 5: *Bucket structure example.*

produces a legal topological sorting as every directed arc (u, v) is processed such as its origin u is always subtracted before its destination v. Thus v will never get a lower number and the topological sorting will be legal. Figure 5 displays an example of the algorithm execution sequence.

Another algorithm used for the computation of a topological sorting is totally based on depth first traversal:

- Perform DFS to compute finishing times $f[v]$ for each vertex v.

- As each vertex is finished, insert it to the front of a linked list

- Return the linked list of vertices

As far as the time complexity of the algorithm is concerned, DFS takes $O(m + n)$ time and the insertion of each of the vertices to the linked list takes $O(1)$ time. Thus topological sorting using DFS takes time $O(m + n)$.

Three important facts about topological sorting are:

1. Only directed acyclic graphs can have linear extensions, since any directed cycle is an inherent contradiction to a linear order of tasks. This means that it is impossible to determine a proper schedule for a set of tasks if all of the tasks depend on some "previous" task of the same set in a cyclic manner.

2. Every DAG can be topologically sorted, so there must always be at least one schedule for any reasonable precedence constraints among jobs.

3. DAGs typically allow many such schedules, especially when there are few constraints. Consider n jobs without any constraints. Any of the $n!$ permutations of the jobs constitutes a valid linear extension. That is if there are no dependencies among tasks so that we can perform them in any order, then any selection is "legal".

In the figure 6, we present an execution of the topological sorting algorithm using bucket sorting. Figure 7 shows the possible topological sorting results for three different DAGs.
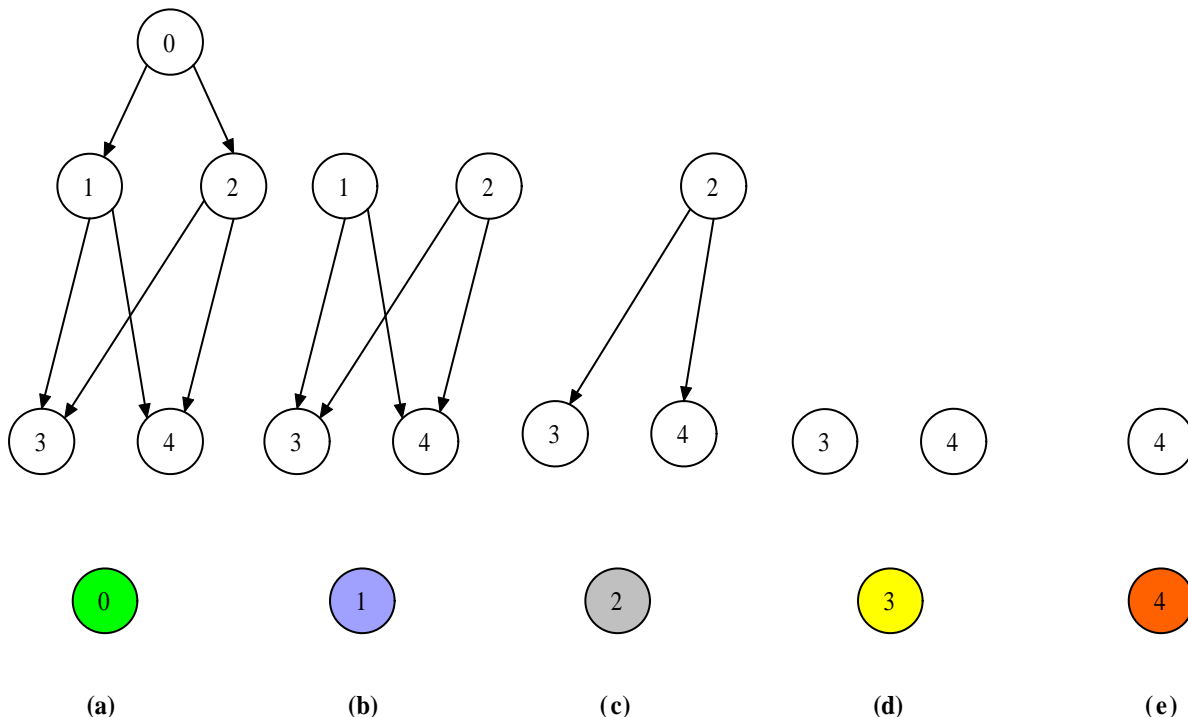

Figure 6: *Execution sequence of the topological sorting algorithm. (a) Select source 0. (b) Source 0 excluded. Resulting sources 1 and 2. Select source 1. (c) Source 1 excluded. No new sources. Select source 2. (d) Source 2 excluded. Resulting sources 3 and 4. Select source 3. (e) Source 3 excluded. No new sources. Select source 4.*

2.3 Single Source Shortest Path

The computation of the single source shortest path in a weighted directed acyclic graph $G = (V, E)$ can be achieved in time $O(m+n)$ by applying relaxation of its edges, according to a topological sort of its vertices. Before the description of the algorithm, we give some basic definitions. We define the weight of a path $p = \begin{bmatrix} v_1 & v_2 & \dots & v_n \end{bmatrix}$ as
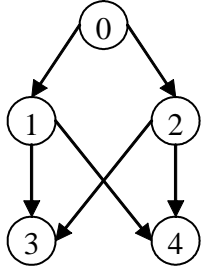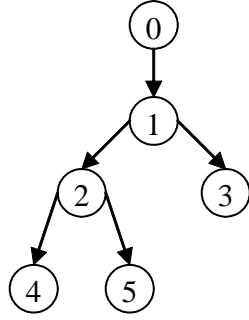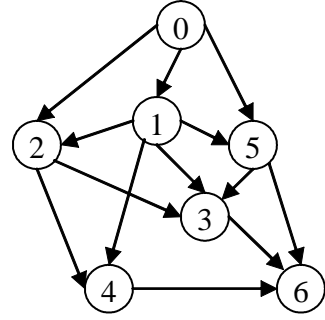
$$w(p) = \sum_{i=1}^{n-1} w(v_i, v_{i+1})$$

Figure 7: *Possible topological sorting results for three DAGS.*

We also define the shortest path from vertex u to vertex v by

$$\delta(u,v) = \begin{cases} \min\{w(p) : v \longrightarrow^p v)\} & , \exists \text{ path from } u \text{ to } v \\ \infty & , \text{ otherwise} \end{cases}$$

A shortest path from vertex u to vertex v is then defined as any path p with weight $w(p) = \delta(u,v)$. There are variants of the single-source shortest-paths problem:

1. Single-destination shortest-paths problem

2. Single-pair shortest-path problem

3. All-pairs shortest-paths problem

We can compute a single-source shortest path in a DAG, using the main idea of the topological sorting using bucket sorting algorithm. Starting from a given node u, we set the minimum path $\delta(u,v)$ from u to every node $v \in adj(u)$ as the weight $w(e)$, e being the edge (u,v), and we remove u. Then we select one of these nodes as the new source u' and we repeat the same procedure and for each node z we encounter, we check if $\delta(u,u') + w(u',z) < \delta(u,z)$ and if it holds we set $\delta(u,z) = \delta(u,u') + w(u',z)$.

We must point out that graph G cannot contain any cycles (thus it is a DAG), as this would make it impossible to compute a topological sorting, which is the algorithm's first step. Also, the initial node u is converted to a DAG source by omitting all its incoming

13

edges. The algorithm examines every node and every edge once at most (if there is a forest of trees, some nodes and some edges will not be visited), thus its complexity is $O(m + n)$. The algorithm that will be presented uses the technique of relaxation. For each vertex $v \in V$, we maintain an attribute $d[v]$, which is an upper bound on the weight of a shortest path from source s to v. We call $d[v]$ a shortest-path estimate and we initialize the shortest-path estimates and predecessors during the initialization phase. The process of relaxing an edge (u, v) consists of testing whether we can improve the shortest path to v found so far by going through u. If the shortest path can be improved then $d[v]$ and $\pi[u]$ are updated. A relaxation step may decrease the value of the shortest-path estimate $d[u]$ and update v's predecessor field $\pi[u]$. Next, we present the algorithm that computes the shortest paths of a directed acyclic graph.

Algorithm DAG-SHORTEST-PATHS(G, s)

1 *compute a topological sorting of G*
2 **for** *each vertex $v \in V[G]$*
3     $d[v] \leftarrow \infty$
4     $\pi[v] \leftarrow nil$
5 $d[s] \leftarrow 0$
6 **for** *each u taken in topologically sorted order*
7   **for** *each $v \in adj(u)$*
8   **if** $d[v] > d[u] + w(u, v)$
9             $d[v] \leftarrow d[u] + w(u, v)$
10             $\pi[v] \leftarrow u$

Theorem 2.1. *If a weighted, directed graph $G = (V, E)$ has a source vertex s and no cycles, then at termination of the **DAG-SHORTEST-PATHS**(G, s) algorithm, it is $d[v] = \delta(s, v)$ for all vertices $v \in V$.*

Proof
If v is not reachable from s, then it is obvious that $d[v] = \delta(s, v) = \infty$. Now, suppose that v is reachable from s, so that there is a shortest path $p = \begin{pmatrix} v_0 & v_1 & \dots & v_k \end{pmatrix}$, where $v_0 = s$ and $v_k = v$. Because we process the vertices in topologically sorted order, the edges on p are relaxed in the order $(v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)$. Hence, $d[v_i] = \delta(s, v_i)$ at termination for $i = 0, 1, 2, \dots, k$.$\square$

2.4   Single Source Longest Path

The algorithm for the computation of the shortest path in a directed acyclic graph presented in the previous section can easily be modified in order to compute longest paths in a given graph $G = (V, E)$. To achieve this, we must perform the following changes.

- We set $d[u] \leftarrow 0$ instead of $d[u] \leftarrow \infty$ during the initialization phase.

- We increase the value of the estimate $d[u]$ by changing the direction of the inequality at the relaxation phase.

These changes are necessary because in the case of longest-path we check if the path being examined is longer than the one previously discovered. In order for every comparison to work properly, we must initialize the distance to all vertices at 0. The time complexity of the algorithm is obviously the same as the one of shortest-path (i.e. $O(m+n)$). The pseudocode of the algorithm is similar with the one presented before concerning the shortest paths problem.

Algorithm DAG-LONGEST-PATHS(G, s)
1 *compute a topological sorting of G*
2 **for** *each vertex $v \in V[G]$*
3     $d[v] \leftarrow 0$
4     $\pi[v] \leftarrow nil$
5 $d[s] \leftarrow 0$
6 **for** *each u taken in topologically sorted order*
7   **for** *each $v \in adj(u)$*
8   **if** $d[v] < d[u] + w(u, v)$
9             $d[v] \leftarrow d[u] + w(u, v)$
10             $\pi[v] \leftarrow u$

At this point, we must refer to the nature of the longest path problem. This problem is generally a difficult problem, as there is a polynomial time algorithm only for the case of directed acyclic graphs. If we try to apply the algorithm in directed graphs with cycles, the algorithm will never end, as it will always discover longer and longer paths, approaching the infinity. Hence, longest paths in directed graphs with cycles, are not well defined. This case is similar to the case of the shortest paths problem, where negative weight cycles are not allowed. As a consequence, longest paths in an undirected graph is NP-complete, something we will immediately prove.

Theorem 2.2. *Longest Path Problem in undirected graphs is NP-complete.*

Proof
When we refer to the longest path problem, we are given an undirected graph $G = (V, E)$ and a positive integer $k \leq |V|$ and we want to know if G has a simple path with k or more edges. Obviously, Longest Path belongs to the class NP, as a solution of the problem can be verified in polynomial time (there a succinct positive certificate). For the NP-completeness, we reduce Hamilton Path, which is a known NP-complete problem, to it. Given an instance $G' = (V', E')$ for Hamilton Path, count the number $|V'|$ of nodes in G'

and output the instance $G = G'$, $k = |V| - 1$ for Longest Path. Obviously, G has a simple path of length $|V| - 1$ if and only if G' has a Hamilton Path.□

In figure 8, we present an example of the shortest path algorithm execution in a directed acyclic graph.

Using the single-source longest-path algorithm, the distance matrix d would initialize to 0, and in every step of the algorithm we would compare the weight of every new path we
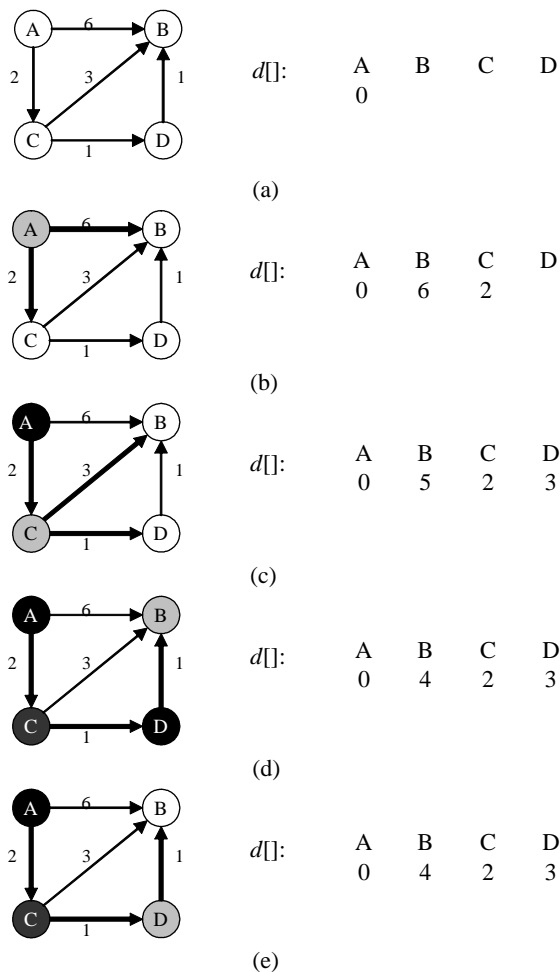

Figure 8: *Execution sequence of the shortest-path algorithm (a) Initial graph-All shortest paths are initialized to ∞ (b) Starting from node A, we set shortest paths to B and C, 6 and 2 respectively (c) From node C, we can follow the edges to B and D. The new shortest path to B is 5 because $w(AC) + w(CB) < w(AB)$. The shortest path to D is $A \rightarrow C \rightarrow D$ and its weight is the sum $w(AC) + w(CD) = 3$ (d) From node D, we follow the edge to B. The new shortest path to B is 4 because $w(ACD) + w(DB) < w(ACB)$. (e) From node B there is no edge we can follow. The algorithm is finished and the resulting shortest paths are: $A \rightarrow C$, $A \rightarrow C \rightarrow D$ and $A \rightarrow C \rightarrow D \rightarrow B$ with weights 4, 2, 3 respectively.*

discover to the content of the matrix and record the higher value. The resulting longest paths would be $A \rightarrow B$, $A \rightarrow C$ and $A \rightarrow C \rightarrow D$ with weights 6, 2 and 3 respectively.

2.5 The Bellman Ford Algorithm

The *Bellman Ford* algorithm solves the single source shortest paths problem in the general case in which edge weights can be negative. Given a weighted directed graph $G = (V, E)$ with source s and weight function $w : E \rightarrow \Re$, the algorithm returns a boolean value indicating whether or not there is a negative weight cycle that is reachable form the source. If there is such a cycle, the algorithm indicates that no solution exists. If not, the algorithm produces the shortest paths and their weights.

Like Dijkstra's algorithm, the algorithm uses the technique of relaxation. This means that it produces an estimate $d[v]$ on the weight of the shortest path from the source s to each vertex $v \in V$ until it achieves the actual shortest path weight $\delta(s, v)$.

Algorithm BELLMAN-FORD(G, W, s)

1 **for** *each* $v \in V[G]$
2      $d[v] \leftarrow \infty$
3      $\pi[v] \leftarrow nil$
4 $d[s] \leftarrow 0$
5 **for** $i \leftarrow 1$ **to** $V[G] - 1$
6   **for** *each* $(u, v) \in E[G]$
7   **if** $d[v] > d[u] + w(u, v)$
8              $d[v] \leftarrow d[u] + w(u, v)$
9              $\pi[v] \leftarrow u$
10 **for** *each* $(u, v) \in E[G]$
11   **if** $d[v] > d[u] + w(u, v)$
12   **return FALSE**
13 **return TRUE**

We will now apply the algorithm to the graph of figure 9. The execution of the algorithm is depicted in figure 10.

The Bellman Ford algorithm runs in time $O(mn)$, since the initialization takes $O(n)$ time, each of the $|V| - 1$ passes over the edges takes $O(m)$ time and the final *for* loop takes time $O(m)$. The Bellman Ford algorithm runs in time $O(mn)$, since the initialization takes $O(n)$ time, each of the $|V| - 1$ passes over the edges takes $O(m)$ time and the final *for* loop takes time $O(m)$.

The main differences between this algorithm and the Dijkstra algorithm are the following:

1. The Dijkstra algorithm works only with positive weighted graphs with no cycles, while the Bellman Ford algorithm works with graphs with positive and negative weighted

edges and non negative cycles.

2. The Dijkstra algorithm at every step discovers the shortest path to a new node and inserts this node to its special set. On the other hand, the Bellman Ford algorithm uses no special set, as at every step updates the shortest paths for the nodes that are
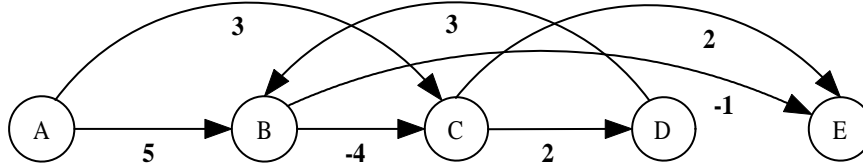

Figure 9: *A directed graph with source A containing a cycle $B \to C \to D \to B$ with positive weight $w = 1$.*
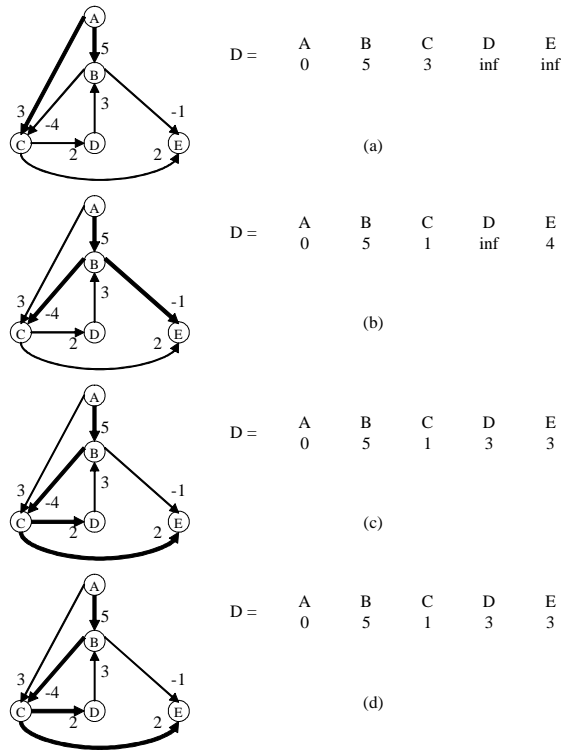

Figure 10: *The sequence of steps of the Bellman - Form algorithm for the graph of Figure 9. (a) $D[B]$ becomes 5 and $D[C]$ becomes 3. (b) $D[E]$ becomes 4 and $D[C]$ becomes 1, since $D[B] + w(B, C) < D[C]$. (c) $D[D]$ becomes 3 and $D[E]$ becomes 3, since $D[C] + w(C, E) < D[E]$. (d)Nothing changes, since no shortest path can be improved. At this point there are no more relaxation operations to be performed and the algorithm returns the distance matrix D.*

adjacent to the node being processed at the current step. Thus, the shortest path to a node is not determined at an intermediate step, because a new shortest path to this node can be discovered at a later step.

2.6 All Pairs Shortest Paths

2.6.1 A Matrix Multiplication-like Algorithm

In this section we will present an algorithm for the all pair shortest paths problem and we will see how it is connected to the classic algorithm used for matrix multiplication. In fact, the algorithm is a dynamic programming algorithm and each major loop of the dynamic program will invoke an operation that is very similar to matrix multiplication.

Firstly, we will refer to the structure of a shortest path. It is true that the subpaths contained in a shortest path are also shortest paths for the certain nodes. Suppose that our graph is represented with the adjacency matrix $W = w_{ij}$. Now, let $d_{ij}^{(m)}$ be the minimum weight of any path from vertex i to vertex j that contains at most m edges. Define $d_{ij}^{(0)} = 0$ if $i = j$ and $d_{ij}^{(0)} = \infty$ otherwise. To compute $d_{ij}^{(m)}$ as a function of $d_{ij}^{(m-1)}$, we can set

$$d_{ij}^{(m)} = \min_{1 \le k \le n} \{d_{ik}^{(m-1)} + w_{kj}\}$$

But what are the actual shortest paths δ_{ij}? If the graph contains no negative cycles, then all shortest paths are simple and thus contain at most $n-1$ edges. This means that a path form a vertex i to a vertex j cannot have less weight than a shortest path from i to j. The actual shortest path weights are therefore given by

$$\delta_{ij} = d_{ij}^{(n-1)} = d_{ij}^{(n)} = d_{ij}^{(n+1)} = \ldots$$

The algorithm works as follows. It accepts as input the matrix $W = w_{ij}$, it computes a series of matrices $D^{(1)}, D^{(2)}, \ldots, D^{(n-1)}$ where for $m = 1, 2, \ldots, n-1$, we have $D^{(m)} = d_{ij}^{(m)}$. The final matrix $D^{(n-1)}$ contains the actual shortest path weights. Observe that since $d_{ij}^{(1)} = w_{ij}$ for all vertices $i, j \in V$, we have $D^{(1)} = W$.

The heart of the algorithm is the following procedure, which, given matrices $D^{(m-1)}$ and W, returns the matrix D^m. That is, it extends the shortest paths computed so far by one more edge.

Algorithm EXTEND-SHORTEST-PATHS$(D^{(m-1)}, W)$
1 $n \leftarrow rows[D^{(m-1)}]$
2 **for** $i \leftarrow 1$ **to** n
3   **for** $j \leftarrow 1$ **to** n
4         $d_{ij}^{(m)} \leftarrow \infty$
5   **for** $k \leftarrow 1$ **to** n

6          $d_{ij}^{(m)} \leftarrow \min\{d_{ij}^{(m)}, d_{ik}^{(m-1)} + w_{kj}\}$

7 **return** $D^{(m)}$

We can now see the relation to matrix multiplication. Suppose we want to compute the matrix product $C = A \cdot B$ of two $n \times n$ matrices A and B. Then, for $i, j = 1, \ldots n$ it is

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

Observe that if we substitute $d^{(m-1)}$ with a, w with b, d^m with c, " min " with " $+$ " and " $+$ " with " $\cdot$ ", we turn line 6 of **EXTEND-SHORTEST-PATHS**$(D^{(m-1)}, W)$ to matrix multiplication. So it easy to obtain the straightforward $O(n^3)$-time algorithm for matrix multiplication.

Algorithm MATRIX-MULTIPLY(A, B)

1 $n \leftarrow rows[A]$

2 **for** $i \leftarrow 1$ **to** n

3   **for** $j \leftarrow 1$ **to** n

4          $c_{ij} \leftarrow 0$

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

6              $c_{ij} \leftarrow c_{ij} + a_{ik} \cdot b_{kj}$

7 **return** C

Returning to the all pairs shortest paths problem, we compute the shortest path weights by extending shortest paths edge by edge. We define $\odot$ to be the *e-product* between two $n \times n$ matrices A, B returned by **EXTEND-SHORTEST-PATHS**(A, B) algorithm. Similarly, we can define the n-th *e-power* A^n of a matrix A to be $A^{n-1} \odot A$. In this way, we can compute the sequence of $n - 1$ matrices in the following way

$$
\begin{aligned}
D^{(1)} &= D^{(0)} \odot W = W \\
D^{(2)} &= D^{(1)} \odot W = W^2 \\
D^{(3)} &= D^{(2)} \odot W = W^3 \\
&\vdots \\
D^{(n-1)} &= D^{(n-2)} \odot W = W^{n-1}
\end{aligned}
$$

As we argued above, the matrix $D^{(n-1)} = W^{n-1}$ contains the shortest path weights. So, we have the following algorithm

Algorithm SHOW-ALL-PAIRS-SHORTEST-PATHS(W)

1 $n \leftarrow rows[W]$

2 $D^{(1)} \leftarrow W$

3 **for** $m \leftarrow 2$ **to** $n - 1$

4     $D^{(m)} \leftarrow$ **EXTEND-SHORTEST-PATHS**$(D^{(m-1)}, W)$

5 **return** $D^{(n-1)}$

This algorithm runs in $O(n^4)$ time. We can, however, improve the running time to $O(n^3 \log n)$ by applying some matrix multiplication properties.

2.6.2 The Floyd-Warshall Algorithm

In this section we present a very efficient, simply programmed, and widely used algorithm that finds the shortest paths between all pairs of nodes, all at once. Furthermore, it has the important advantage over Dijksta's algorithm of working when the arc weights are allowed to be negative and will in fact allow us to detect negative-cost cycles. The algorithm works with an $n \times n$ distance matrix d_{ij}, initially set to the arc weights c_{ij} of the directed graph $G = (V, E)$. For our purposes we assume that $c_{ii} = \infty$ for every i.

Something very important for the function of our algorithm is the *triangle* operation. Given an $n \times n$ distance matrix d_{ij}, a triangle operation for a fixed node j is $d_{ik} = \min\{d_{ik}, d_{ij} + d_{jk}\}$ for all $i, k = 1, \ldots, n$ but $i, k \neq j$. Note that we allow $i = k$.

This operation replaces, for all i and k, the d_{ik} entry with the distance $d_{ij} + d_{jk}$ if the latter is shorter.
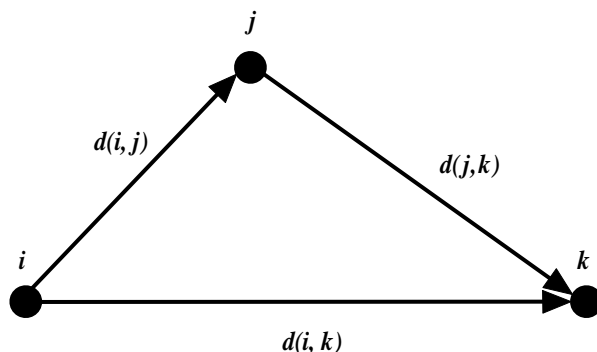

Figure 11: *The triangle operation for fixed j and all other i and k.*

Following, we present a theorem that states the correctness of the algorithm.

Theorem 2.3. *If we perform a triangle operation for successive values $j = 1, \ldots, n$, each entry d_{ik} becomes equal to the length of the shortest path from i to k, assuming the weights $c_{ij} \geq 0$.*

Proof

We shall show by induction that after the triangle operation for $j = j_0$ is executed, d_{ik} is

the length of the shortest path from i to k with intermediate nodes $v \leq j_0$, for all i and k. For the basis, we set $j_0 = 1$, which is clear. Assume then that the inductive hypothesis is true for $j = j_0 - 1$. We must prove that for $j = j_0$ it is

$$d_{ik} = \min\{d_{ik}, d_{ij_0} + d_{j_0 k}\}$$

We must see two different cases. If the shortest path from i to k with intermediate nodes $v \leq j_0$ does not pass through j_0, d_{ik} will be unchanged, as the second argument of the min operation will be ∞. So it will satisfy the inductive hypothesis. On the other hand, if the shortest path from i to k with intermediate nodes $v \leq j_0$ *does* pass through j_0, d_{ik} will be replaced by $d_{ij_0} + d_{j_0 k}$. By the inductive hypothesis, d_{ij_0} and $d_{j_0 k}$ are both optimal distances with intermediate vertices $v \leq j_0 - 1$, so $d_{ij_0} + d_{j_0 k}$ is optimal with intermediate vertices $v \leq j_0$. This completes the proof.$\square$

Following, we present the main body of the algorithm.

Algorithm FLOYD-WARSHALL(W)
1 **for** $i \leftarrow 1$ **to** n
2   **for** $j \leftarrow 1$ **to** n
3         $d_{ij} \leftarrow w_{ij}$
4         $\pi_{ij} \leftarrow null$
5 **for** $j \leftarrow 1$ **to** n
6   **for** $i \leftarrow 1$ **to** n
7   **for** $k \leftarrow 1$ **to** n
8   **if** $d_{ik} > d_{ij} + d_{jk}$
9                 $d_{ik} \leftarrow d_{ij} + d_{jk}$
10                $\pi_{ik} \leftarrow j$
11 **return** D

Note that the algorithm uses an extra matrix, where the shortest paths are stored. It is easy to see that the algorithm runs in time $O(n^3)$.
We will now apply the algorithm to a graph $G = (V, E)$ of the following matrices:

$$W = \begin{bmatrix} 0 & \infty & 1 & \infty & \infty \\ 8 & 0 & \infty & \infty & 2 \\ \infty & 3 & 0 & -4 & \infty \\ \infty & 4 & \infty & 0 & \infty \\ 7 & \infty & 5 & \infty & 0 \end{bmatrix}, \Pi = \begin{bmatrix} null & null & 1 & null & null \\ 2 & null & null & null & 2 \\ null & 3 & null & 3 & null \\ null & 4 & null & null & null \\ 5 & null & 5 & null & null \end{bmatrix}$$

By applying the algorithm, we get the following sequence of matrices:

- $j = 1$

$$W = \begin{bmatrix} 0 & \infty & 1 & \infty & \infty \\ 8 & 0 & 9 & \infty & 2 \\ \infty & 3 & 0 & -4 & \infty \\ \infty & 4 & \infty & 0 & \infty \\ 7 & \infty & 5 & \infty & 0 \end{bmatrix}, \Pi = \begin{bmatrix} null & null & 1 & null & null \\ 2 & null & 1 & null & 2 \\ null & 3 & null & 3 & null \\ null & 4 & null & null & null \\ 5 & null & 5 & null & null \end{bmatrix}$$

- $j = 2$

$$W = \begin{bmatrix} 0 & \infty & 1 & \infty & \infty \\ 8 & 0 & 9 & \infty & 2 \\ 11 & 3 & 0 & -4 & 5 \\ 12 & 4 & \infty & 0 & 6 \\ 7 & \infty & 5 & \infty & 0 \end{bmatrix}, \Pi = \begin{bmatrix} null & null & 1 & null & null \\ 2 & null & 1 & null & 2 \\ 2 & 3 & null & 3 & 2 \\ 2 & 4 & null & null & 2 \\ 5 & null & 5 & null & null \end{bmatrix}$$

- $j = 3$

$$W = \begin{bmatrix} 0 & 4 & 1 & -3 & 6 \\ 8 & 0 & 9 & 5 & 2 \\ 11 & 3 & 0 & -4 & 5 \\ 12 & 4 & \infty & 0 & 6 \\ 7 & 10 & 5 & 3 & 0 \end{bmatrix}, \Pi = \begin{bmatrix} null & 3 & 1 & 3 & 3 \\ 2 & null & 1 & 3 & 2 \\ 2 & 3 & null & 3 & 2 \\ 2 & 4 & null & null & 2 \\ 5 & 3 & 5 & 3 & null \end{bmatrix}$$

- $j = 4$

$$W = \begin{bmatrix} 0 & 1 & 1 & -3 & 3 \\ 8 & 0 & 9 & 5 & 2 \\ 8 & 0 & 0 & -4 & 2 \\ 12 & 4 & \infty & 0 & 6 \\ 7 & 7 & 5 & 3 & 0 \end{bmatrix}, \Pi = \begin{bmatrix} null & 4 & 1 & 3 & 4 \\ 2 & null & 1 & 3 & 2 \\ 4 & 4 & null & 3 & 4 \\ 2 & 4 & null & null & 2 \\ 5 & 4 & 5 & 3 & null \end{bmatrix}$$

- $j = 5$

$$W = \begin{bmatrix} 0 & 1 & 1 & -3 & 3 \\ 8 & 0 & 7 & 5 & 2 \\ 8 & 0 & 0 & -4 & 2 \\ 12 & 4 & 11 & 0 & 6 \\ 7 & 7 & 5 & 3 & 0 \end{bmatrix}, \Pi = \begin{bmatrix} null & 4 & 1 & 3 & 4 \\ 2 & null & 5 & 3 & 2 \\ 4 & 4 & null & 3 & 4 \\ 2 & 4 & 5 & null & 2 \\ 5 & 4 & 5 & 3 & null \end{bmatrix}$$

2.6.3 Transitive Closure of a Directed Graph

Using a slightly modified version of the Floyd - Warshall algorithm, we can answer the question whether there is a path that leads from node u to node v, or if node v is reachable from u. This can be done by running a DFS from node u, and see if v becomes a descendant of u in the DFS tree. This procedure though takes $O(m + n)$ time which may be too long if we want to repeat this check very often. Another method is by computing the transitive closure of the graph which can answer the reachability question in $O(1)$ time.

In order to compute the transitive closure of graph G, we can use the Floyd - Warshall algorithm with the following modifications:

- Instead of the distance matrix D we use a transitive closure matrix T which is initialized to the values of the adjacency matrix A of G.

- We change the relaxation part by substituting lines 8-10 of the **FLOYD-WARSHALL**(W) algorithm with the statement:

$$t_{ik} \leftarrow t_{ik} \vee (t_{ij} \wedge t_{jk})$$

Algorithm TRANSITIVE-CLOSURE(G)

1 **for** $i \leftarrow 1$ **to** n
2   **for** $j \leftarrow 1$ **to** n
3         $t_{ij} \leftarrow a_{ij}$
4 **for** $j \leftarrow 1$ **to** n
5   **for** $i \leftarrow 1$ **to** n
6   **for** $k \leftarrow 1$ **to** n
7             $t_{ik} \leftarrow t_{ik} \vee (t_{ij} \wedge t_{jk})$
8 **return** T

In the above algorithm, matrix A is the adjacency matrix of graph G and is defined as follows:

$$a_{ij} = \begin{cases} 0 & \text{, if } (i \neq j) \wedge ((i,j) \notin E) \\ 1 & \text{, if } (i = j) \vee ((i,j) \in E) \end{cases}$$

It is evident that the two algorithms presented are very similar. They are both based on a type of algebraic structure called a *closed semiring*.

2.7 Program Evaluation and Review Technique (PERT)

2.7.1 General

Program evaluation and review technique (PERT) charts depict task, duration, and dependency information. Each chart starts with an initiation node from which the first task, or

tasks, originates. If multiple tasks begin at the same time, they are all started from the node or branch, or fork out from the starting point. Each task is represented by a line which states its name or other identifier, its duration, the number of people assigned to it, and in some cases the initials of the personnel assigned. The other end of the task line is terminated by another node which identifies the start of another task, or the beginning of any slack time, that is, waiting time between tasks.

Each task is connected to its successor tasks in this manner forming a network of nodes and connecting lines. The chart is complete when all final tasks come together at the completion node. When slack time exists between the end of one task and the start of another, the usual method is to draw a broken or dotted line between the end of the first task and the start of the next dependent task.

A PERT chart may have multiple parallel or interconnecting networks of tasks. If the scheduled project has milestones, checkpoints, or review points (all of which are highly recommended in any project schedule), the PERT chart will note that all tasks up to that point terminate at the review node. It should be noted at this point that the project review, approvals, user reviews, and so forth all take time. This time should never be underestimated when drawing up the project plan. It is not unusual for a review to take 1 or 2 weeks. Obtaining management and user approvals may take even longer.

When drawing up the plan, be sure to include tasks for documentation writing, documentation editing, project report writing and editing, and report reproduction. These tasks are usually time-consuming, so don't underestimate how long it will take to complete them.

PERT charts are usually drawn on ruled paper with the horizontal axis indicating time period divisions in days, weeks, months, and so on. Although it is possible to draw a PERT chart for an entire project, the usual practice is to break the plans into smaller, more meaningful parts. This is very helpful if the chart has to be redrawn for any reason, such as skipped or incorrectly estimated tasks. Many PERT charts terminate at the major review points, such as at the end of the analysis. Many organizations include funding reviews in the projects life cycle. Where this is the case, each chart terminates in the funding review node. Funding reviews can affect a project in that they may either increase funding, in which case more people have to made available, or they may decrease funding, in which case fewer people may be available. Obviously more or less people will affect the length of time it takes to complete the project.

A PERT network can be modelled as a weighted, acyclic digraph (directed graph) in which each edge represents an activity (task), and the edge weight represents the time needed to perform that activity. An acyclic graph must have (at least) one vertex with no predecessors, and (at least) one with no successors, and we will call those vertices the start and stop vertices for a project. All the activities which have arrows into node x must be completed before any activity "out of node x" can commence. At node x, we will want to compute two job times: the earliest time $et(x)$ at which all activities terminating at x can be completed, and $lt(x)$, the latest time at which activities terminating at x can be

completed so as not to delay the overall completion time of the project (the completion time of the stop node, or - if there is more than one sink - the largest of their completion times).

2.7.2   Critical Path Method

The main interest in time scheduling problems is to compute the minimum time required to complete the project based on the time constraints between tasks. This problem is analogous to finding the longest path of a PERT network (which is a DAG). This is because in order for a task X to commence, every other task Y_i on which the former may depend must be completed. To make sure this happens, X must wait for the slowest of Y_i to complete. For example in Figure 9, task X must wait for Y_1 to complete because if it starts immediately after Y_2, Y_1, which is required for X, will not have enough time to finish its execution.
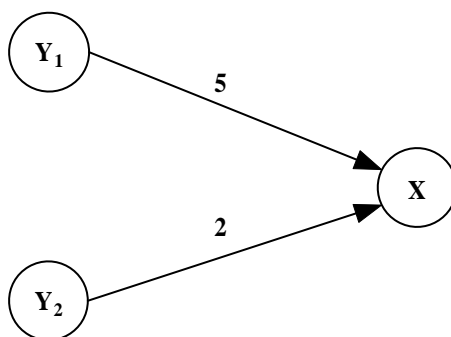

Figure 12: *X must wait for Y_1 to complete before it starts.*

The longest path is also called the critical path, and this method of time scheduling is also called critical path method. A simple example of a PERT diagram for an electronic device manufacturing is shown in Figure 9.

1. IC Design	2. IC Program ming	3. Prototyping
ma 12.1.1998 ke 14.1.1998	la 17.1.1998 su 18.1.1998	ke 21.1.1998 to 22.1.1998

4. User - Interface
ti 13.1.1998 su 18.1.1998

5. Field Tests	6. Manufacturing	7. QA Lab Tests
su 25.1.1998 ma 26.1.1998	to 29.1.1998 ma 2.2.1998	to 5.2.1998 ma 9.2.1998

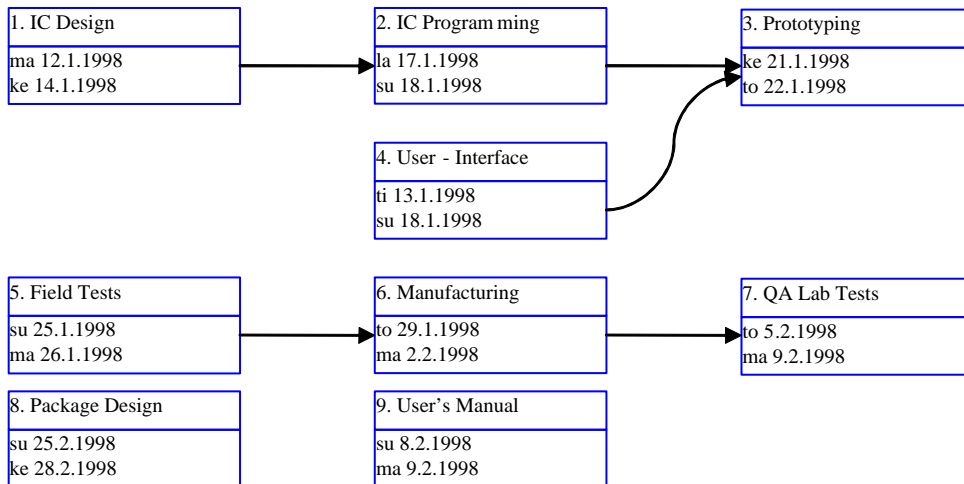| 8. Package Design	9. User's Manual
su 25.2.1998 ke 28.2.1998	su 8.2.1998 ma 9.2.1998

Figure 13: *PERT diagram for the production of an electronic device.*