Chapter 4

Directed Acyclic Graphs

4.1 Directed Acyclic Graphs (DAG)

4.1.1 Definition

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.

Source: A source is a node that has no incoming edges.
Sink: A sink is a node that has no outgoing edges.

4.1.2 Properties

A DAG has at least one source and one sink. 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.

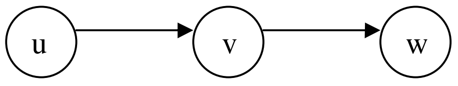
4.2 Topological Sorting

Suppose we have a set of tasks to do, but certain tasks have to be performed before other tasks. Only one task can be performed at a time and each task must be completed before the next task begins. Such a relationship can be represented as a directed graph and topological sorting can be used to schedule tasks under precedence constraints.We are going to determine if an order exists such that this set of tasks can be completed under the given constraints. Such an order is called a *topological sort* of graph *G*, where *G* is the graph containing all tasks, the vertices are tasks and the edges are constraints. This kind of ordering exists if and only if the graph does not contain any cycle (that is, no self-contradict constraints). These precedence constraints form a directed acyclic graph, and any topological sort (also known as a *linear extension*) defines an order to do these tasks such that each is performed only after all of its constraints are satisfied.

4.2.1 Definition

The topological sort of a DAG $G(V,E)$ is a linear ordering of all its vertices such that if G contains an edge (u,v) then u appears before v in the ordering. Mathematically, this ordering is described by a function t that maps each vertex to a number

$t : V \rightarrow \{1,2,\ldots,n\} : t(v) = i$ such that $t(u) < t(v) < t(w)$ for the vertices u, v, w of the following diagram



4.2.2 Algorithm

The following algorithms perform topological sorting of a DAG:

Topological sorting using DFS

1. perform DFS to compute finishing times *f*[*v*] for each vertex *v*
2. as each vertex is finished, insert it to the front of a linked list
3. return the linked list of vertices

Topological sorting using bucket sorting

This algorithm computes a topological sorting of a DAG beginning from a source and using bucket sorting to arrange the nodes of *G* by their in-degree (#incoming edges).

The bucket structure is formed by an array of lists of nodes. Each node also maintains links to the vertices (nodes) to which its outgoing edges lead (the red arrows in Figure 4.1). The lists are sorted so that all vertices with *n* incoming edges lay on the list at the *n*-th position of the table, as shown in Figure 4.1:

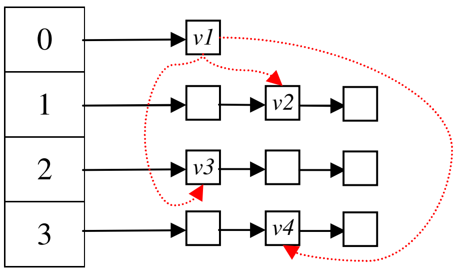


Figure 4.1 Bucket Structure Example

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 *v1* 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 *v1*) and re-allocating them in the bucket. This means that node *v2* is moved to the list at the 0th index, and thus becoming a source. *v3* is moved to the list at the 1st index, and *v4* 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. 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.

Figure 4.2 displays an example of the algorithm execution sequence.

4.2.3 Time Complexity

DFS takes $\Theta(m + n)$ time and the insertion of each of the vertices to the linked list takes $O(1)$ time. Thus topological sorting using DFS takes time $\Theta(m + n)$.

4.2.4 Discussion

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.

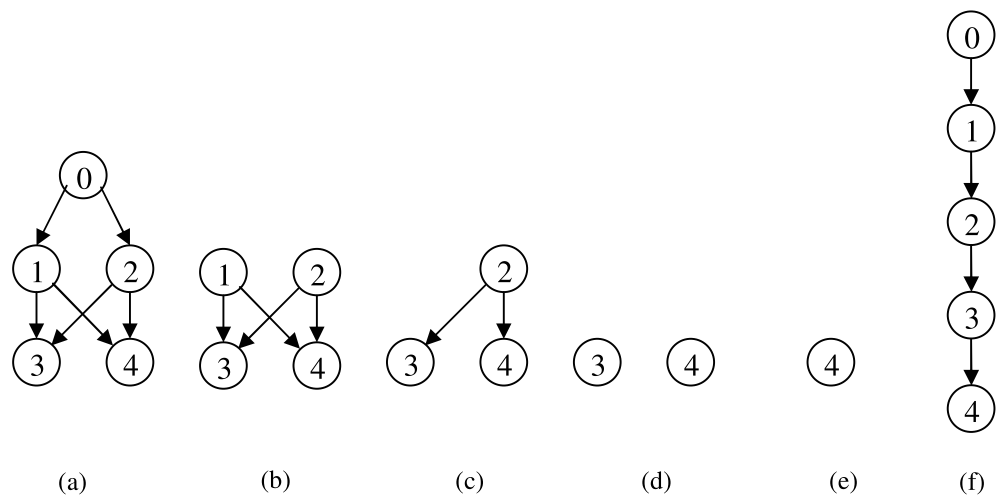
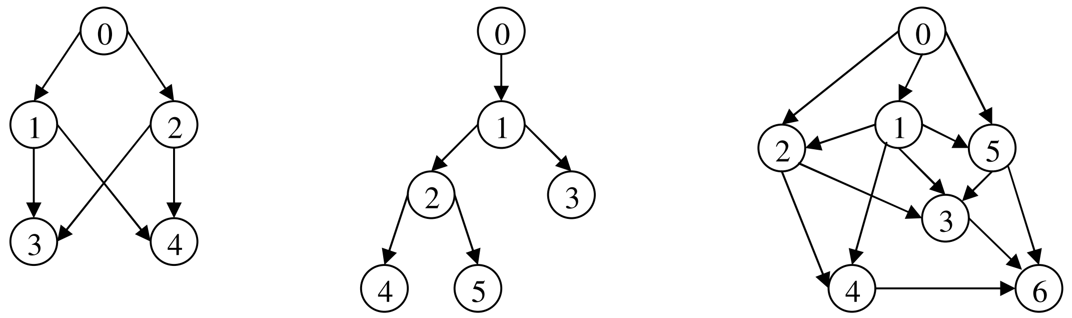


Figure 4.2 : Execution sequence of the topological sorting algorithm.
(a) Select source 0.
(b) Source 0 excluded. Resulting sources 1 and 2. Select source 1.
(c) Source 1 excluded. No new sources. Select source 2.
(d) Source 2 excluded. Resulting sources 3 and 4. Select source 3.
(e) Source 3 excluded. No new sources. Select source 4.
(f) Topological Sorting of the graph.

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".

Example

Figure 4.3 shows the possible topological sorting results for three different DAGs.



0, 1, 2, 3, 4	0, 1, 3, 2, 4, 5	0, 1, 2, 4, 5, 3, 6
0, 1, 2, 4, 3	0, 1, 3, 2, 5, 4	0, 1, 2, 5, 3, 4, 6
0, 2, 1, 3, 4	0, 1, 2, 3, 4, 5	0, 1, 2, 5, 4, 3, 6
0, 2, 1, 4, 3	0, 1, 2, 3, 5, 4	0, 1, 5, 2, 3, 4, 6
	0, 1, 2, 4, 3, 5	0, 1, 5, 2, 4, 3, 6
	0, 1, 2, 4, 5, 3	
	0, 1, 2, 5, 3, 4	
	0, 1, 2, 5, 4, 3	

Figure 4.3 : Possible topological sorting results for three DAGS.

4.3 Single-Source Shortest Path

In the shortest-path problem, we are given a weighted, directed graph $G = (V,E)$, with weight function $w: E \rightarrow R$ mapping edges to real valued weights. The weight of path $p = (v_0, v_1, \ldots, v_k)$ is the sum of the weights of each constituent edges

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

We define the shortest-path weight from u to v by

$$\delta(u,v) = \begin{cases} \min\{w(p) : u \xrightarrow{p} v\} & \text{if there is a path from } u \text{ to } v \\ \infty & \text{otherwise} \end{cases}$$

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:

3 Single-destination shortest-paths problem

4 Single-pair shortest-path problem

5 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 $\delta(u,z) = \delta(u,u') + w((u',z))$.