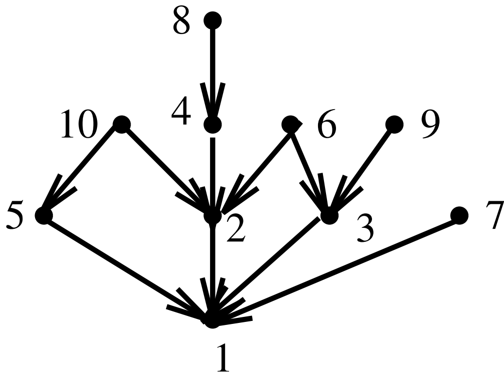


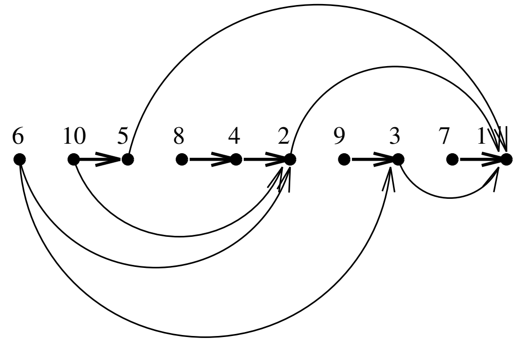


INPUT OUTPUT

## 15.2 Topological Sorting

**Input description**: A directed acyclic graph $G = (V, E)$, also known as a *partial order* or *poset*.

**Problem description**: Find a linear ordering of the vertices of $V$ such that for each edge $(i, j) \in E$, vertex $i$ is to the left of vertex $j$.

**Discussion**: Topological sorting arises as a subproblem in most algorithms on directed acyclic graphs. Topological sorting orders the vertices and edges of a DAG in a simple and consistent way and hence plays the same role for DAGs that a depth-first search does for general graphs.

Topological sorting can be used to schedule tasks under precedence constraints. Suppose we have a set of tasks to do, but certain tasks have to be performed before other tasks. 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.

Three important facts about topological sorting are

1. *Only* DAGs can be topologically sorted, since any directed cycle provides an inherent contradiction to a linear order of tasks.

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 can often be topologically sorted in many different ways, especially when there are few constraints. Consider $n$ unconstrained jobs. Any of the $n!$ permutations of the jobs constitutes a valid topological ordering.

The conceptually simplest linear-time algorithm for topological sorting performs a depth-first search of the DAG to identify the complete set of *source vertices*, where source vertices are vertices of in-degree zero. At least one such source must exist in any DAG. Source vertices can appear at the start of any schedule without violating any constraints. Deleting all the outgoing edges of these source vertices will create new source vertices, which can then sit comfortably to the immediate right of the first set. We repeat until all vertices are accounted for. A modest amount of care with data structures (adjacency lists and queues) is sufficient to make this run in $O(n + m)$ time.

An alternate algorithm makes use of the observation that ordering the vertices in terms of decreasing DFS finishing time yields a linear extension. An implementation of this algorithm with an argument for correctness is given in Section 5.10.1 (page 179).

Two special considerations with respect to topological sorting are:

- *What if I need all the linear extensions, instead of just one of them?* – In certain applications, it is important to construct *all* linear extensions of a DAG. Beware, because the number of linear extensions can grow exponentially in the size of the graph. Even the problem of counting the number of linear extensions is NP-hard.

  Algorithms for listing all linear extensions in a DAG are based on backtracking. They build all possible orderings from left to right, where each of the in-degree zero vertices are candidates for the next vertex. The outgoing edges from the selected vertex are deleted before moving on. An optimal algorithm for listing (or counting) linear extensions is discussed in the notes.

  Algorithms for constructing random linear extensions start from an arbitrary linear extension. We then repeatedly sample pairs of vertices. These are exchanged if the resulting permutation remains a topological ordering. This results in a random linear extension given enough random samples. See the Notes section for details.

- *What if your graph is not acyclic?* – When a set of constraints contains inherent contradictions, the natural problem becomes removing the smallest set of items that eliminates all inconsistencies. The sets of offending jobs (vertices) or constraints (edges) whose deletion leaves a DAG are known as the *feedback vertex set* and the *feedback arc set*, respectively. They are discussed in Section 16.11 (page 559). Unfortunately, both problems are NP-complete.

  Since the topological sorting algorithm gets stuck as soon as it identifies a vertex on a directed cycle, we can delete the offending edge or vertex and continue. This quick-and-dirty heuristic will eventually leave a DAG, but might delete more things than necessary. Section 9.10.3 (page 348) describes a better approach to the problem.

**Implementations**: Essentially all the graph data structure implementations of Section 12.4 (page 381) include implementations of topological sorting. This means the Boost Graph Library [SLL02] (*http://www.boost.org/libs/graph/doc*) and LEDA (see Section 19.1.1 (page 658)) for C++. For Java, the *Data Structures Library in Java* (JDSL) (*http://www.jdsl.org/*) includes a special routine to compute a unit-weighted topological numbering. Also check out JGraphT (*http://jgrapht.sourceforge.net/*).

The *Combinatorial Object Server* (*http://theory.cs.uvic.ca/*) provides C language programs to generate linear extensions in both lexicographic and Gray code orders, as well as count them. An interactive interface is also provided.

My (biased) preference for C language implementations of all basic graph algorithms, including topological sorting, is the library associated with this book. See Section 19.1.10 (page 661) for details.

**Notes**: Good expositions on topological sorting include [CLRS01, Man89]. Brightwell and Winkler [BW91] prove that it is #P-complete to count the number of linear extensions of a partial order. The complexity class #P includes NP, so any #P-complete problem is at least NP-hard.

Pruesse and Ruskey [PR86] give an algorithm that generates linear extensions of a DAG in constant amortized time. Further, each extension differs from its predecessor by either one or two adjacent transpositions. This algorithm can be used to count the number of linear extensions $e(G)$ of an $n$-vertex DAG $G$ in $O(n^2 + e(G))$. Alternately, the reverse search technique of Avis and Fukuda [AF96] can be employed to list linear extensions. A backtracking program to generate all linear extensions is described in [KS74].

Huber [Hub06] gives an algorithm to sample linear extensions uniformly at random from an arbitrary partial order in expected $O(n^3 \lg n)$ time, improving the result of [BD99].

**Related Problems**: Sorting (see page 436), feedback edge/vertex set (see page 559).