

Algorithms 1

Notes taken by Yeheli Fomberg <yp@campus.technion.ac.il>
Based on lectures by Roy Schwartz and Seffi Naor and notes by Shlomo Moran

Spring 2026

These notes are not endorsed by the lecturers. I have revised them outside lectures to incorporate supplementary explanations, clarifications, and material for fun. While I have strived for accuracy, any errors or misinterpretations are most likely mine.

Contents

1	Preliminaries	3
2	Topological Sorting	5
3	BFS	10

1 Preliminaries

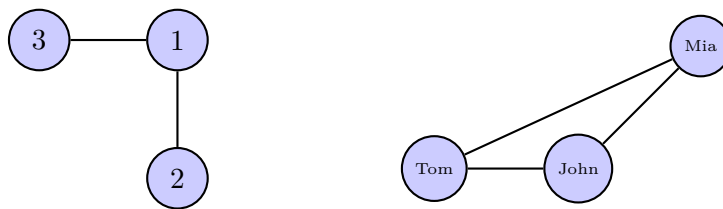
This is a course about algorithms, and since many problems we are going to solve will require some knowledge about graphs we will write here some basic definitions so that we always have a reference if we forget something.

Definition 1.1 (Graph). An ordered pair $G = (V, E)$ such that $E \subseteq \{\{v_1, v_2\} \mid v_1, v_2 \in V\}$ is called a graph.

Remark 1.1. The elements in V are called vertices, and the elements in E are called edges.

Remark 1.2. In these notes we denote $|V| = n$ and $|E| = m$, but in complexity calculations we may also abuse notation and write expressions like $\mathcal{O}(V + E)$.

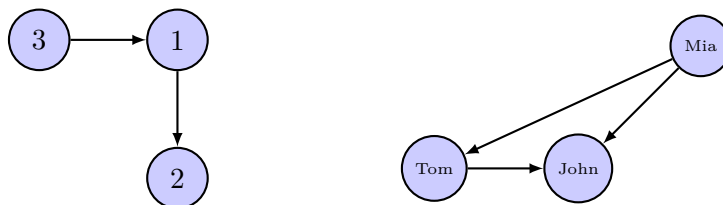
Example 1.1. We can imagine graphs as nodes with some key value, and edges between them as two pointers that connect them. Here are some examples of graphs:



In the left graph, we have $V = \{1, 2, 3\}$ and $E = \{\{1, 3\}, \{2, 3\}\}$.

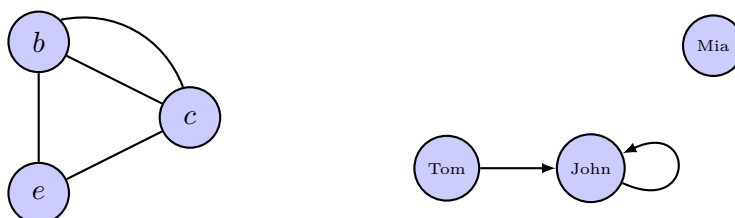
Definition 1.2 (Directed graph). An ordered pair $G = (V, E)$ such that $E \subseteq V \times V$ is called a directed graph.

Example 1.2. Here are some examples of directed graphs:



Definition 1.3 (Simple graph). A simple graph is a graph in which there are no edges from a vertex to itself, or multiple edges from one vertex to another.

Example 1.3. The graphs below are nonexamples of (they are not) simple graphs:



Remark 1.3. Note that in Example 1.3 we stated that there are two graphs, but the node of Mia could be viewed as a separate third graph by itself. We will try to minimize this ambiguity in the future.

Remark 1.4. The following definitions (Definition 1.4 to Definition 1.11) are mostly technical and formal and you probably won't use them in this course. It is important to understand how all of them are distinct, but you don't have to learn the names by heart.

Definition 1.4 (Walk). A walk is a finite or infinite sequence of edges which joins a sequence of vertices.

Definition 1.5 (Trail). A trail is a walk in which all edges are distinct.

Definition 1.6 (Path). A path is a trail in which all vertices (and therefore also all edges) are distinct.

Definition 1.7 (Circuit). A circuit is a nonempty trail in which the first and last vertices are equal (a closed trail).

Definition 1.8 (Cycle). A cycle (or simple circuit) is a circuit in which only the first and last vertices are equal.

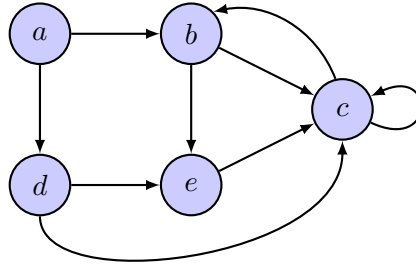
Remark 1.5. In math, a continuous path that does not intersect itself is called simple. Compare this with the terminology “simple circuit” for a cycle.

Definition 1.9 (Underlying graph). Given a directed graph $G = (V, E)$, its induced underlying graph is the undirected graph obtained by replacing every directed edge with an undirected edge.

Definition 1.10 (Source). A source is a vertex that has no incoming edges.

Definition 1.11 (Sink). A sink is a vertex that has no outgoing edges.

Example 1.4. In the following graph:



the vertex a is a source, and there are no sinks. Also note that $[(b, e), (e, c), (c, b)]$ forms a cycle, and that there is no cycle in this graph based at the node d .

Definition 1.12 (Distance between vertices). Let $G = (V, E)$ be a (directed) graph, and let $v, u \in V$. The distance between v and u is the length of a shortest (directed) path between them. We usually denote the distance by $\delta(u, v)$. If there does not exist such a path, we write $\delta(u, v) = \infty$.

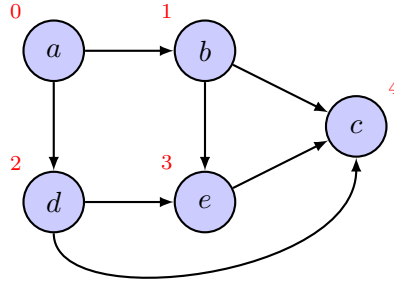
Definition 1.13 (Subgraph). Let $G = (V, E)$ be a graph, and let $U \subseteq V$. A subgraph $G[U]$ is the graph $(U, E[U])$, where $E[U] = E \cap (U \times U)$ denotes all edges in G of vertices in U .

2 Topological Sorting

Definition 2.1 (Topological sorting). A topological sorting of a directed graph (V, E) is a numbering of the vertices $N: V \rightarrow \mathbb{N}$ given by $N: v \rightarrow N(v)$, such that $(v, u) \in E$ implies $N(v) < N(u)$. Equivalently, a topological sorting is a graph traversal in which each node v is visited only after all nodes u such that $(u, v) \in E$ have already been visited.

Remark 2.1. The function $N: V \rightarrow \mathbb{N}$ can be implemented as an array $N[1..n]$.

Example 2.1. The following is an example of a topological sorting:



where the red numbers represent the value of the vertex: $N(v)$.

Remark 2.2. A topological sorting does not always exist! It is clear that if a directed graph contains a cycle¹ then there cannot exist a topological sorting. We will soon show that if a directed graph is acyclic (such graphs are called DAGs, for **D**irected **A**cylic **G**raphs) then there exists an algorithm which produces a topological sorting in linear time $\mathcal{O}(|V| + |E|)$. Beyond topological sorting, the family of DAGs is one of the most important families of graphs in computer science.

Remark 2.3. Recall that a source is a vertex that has no incoming edges. We get that every DAG has at least one source, otherwise we could choose a random vertex and start going backwards ad infinitum without visiting the same vertex twice, which is a contradiction.²

Definition 2.2 (Topological sorting algorithm). The process is as follows:

Algorithm 1: Topological Sorting

Input: A directed graph $G = (V, E)$

Output: A topological numbering $N(v)$ for all $v \in V$, or report that G has a cycle.

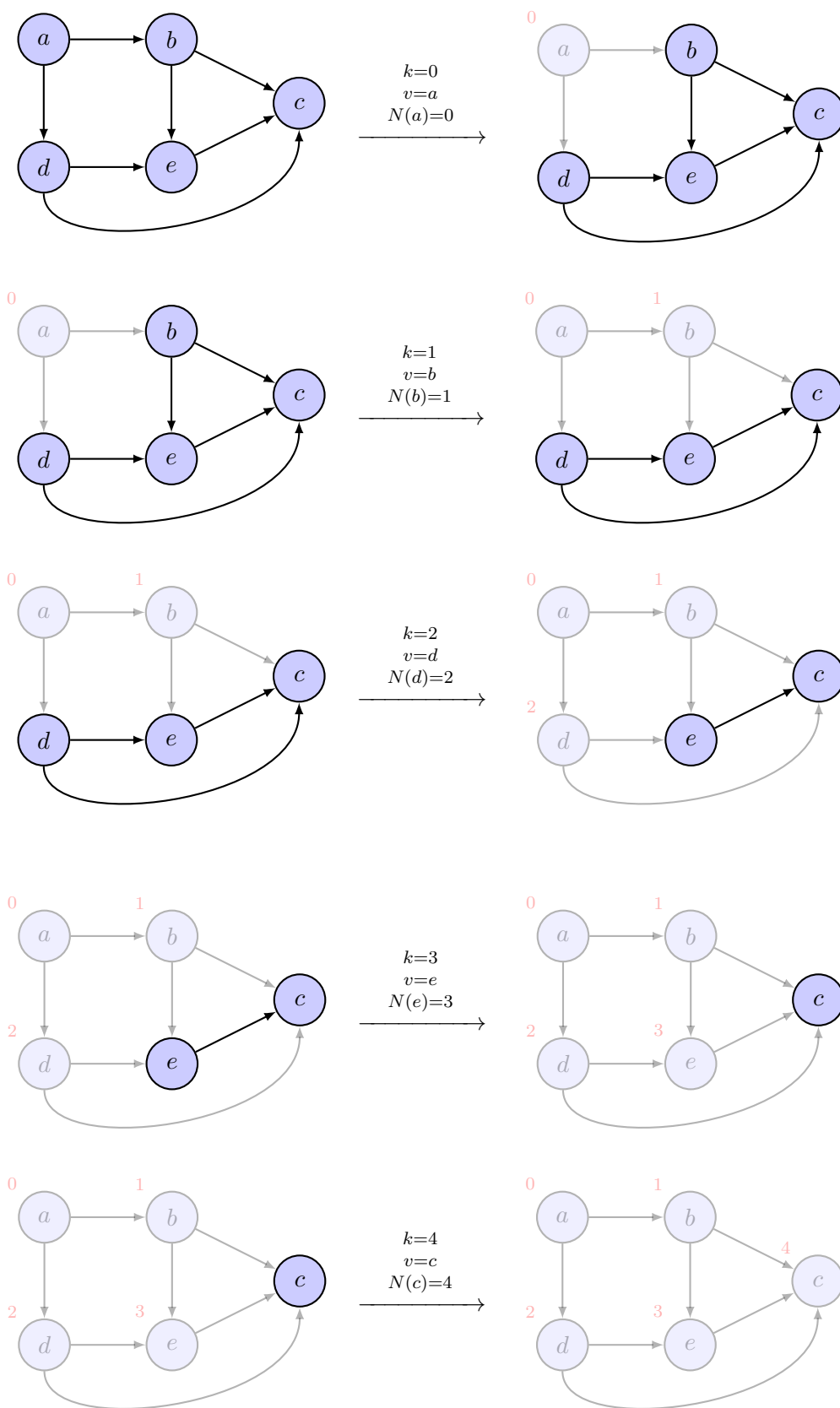
```

1  $k \leftarrow 0$ ;
2 while a source exists in  $G$  do
3    $v \leftarrow \text{FindSource}(G)$ ;
4    $N(v) \leftarrow k$ ;
5    $\text{RemoveSource}(v)$  ;           // Removes  $v$  and all its outgoing edges from  $G$ 
6    $k \leftarrow k + 1$ ;
7 if  $k = |V|$  then
8   return  $N(v)$  for all  $v \in V$  (a topological order)
9 else
10  return " $G$  contains a cycle"
```

¹In a directed graph, unless stated otherwise, a cycle is always a directed cycle.

²Note that we have made a tacit assumption that the graph is finite. Don't panic! In this course everything is finite and discrete.

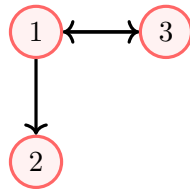
Example 2.2. Here is an example for the topological sorting process of the graph from Example 2.1:



Remark 2.4. In order to compute the complexity of Algorithm 1 we must first implement

lines 3 and 5 in the code. We will call these functions **FindSource** and **RemoveSource** respectively.

For example a topological sorting of the graph above:



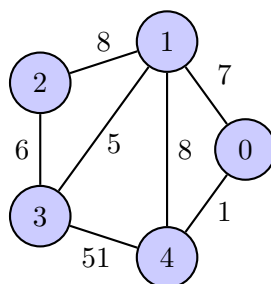
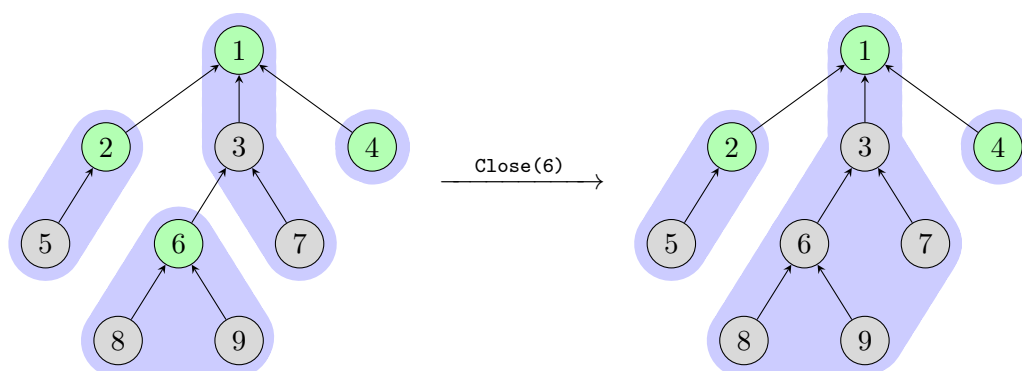
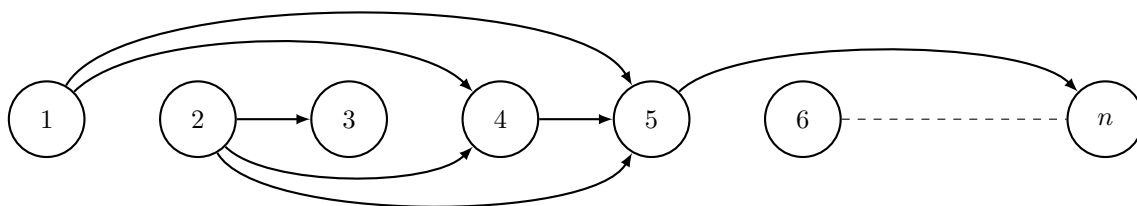
Is not possible. In fact such an ordering exists if and only if the graph we are trying to sort is a DAG.

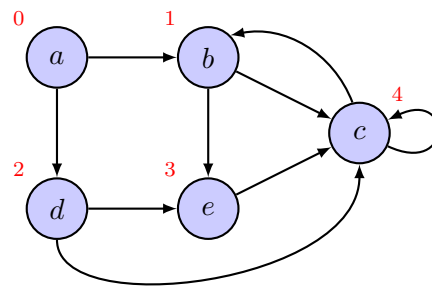
Algorithm 2: Kahn's Algorithm for Topological Sorting**Input:** Directed Acyclic Graph (DAG) $G = (V, E)$ **Output:** Topologically sorted order of vertices

```

1 Compute in-degrees of all vertices;
2 Initialize queue  $Q$  with all vertices of in-degree 0; // Also known as sources
3 Initialize empty list  $L$ ;
4 while  $Q$  is not empty do
5   Dequeue a vertex  $v$  from  $Q$ ;
6   Append  $v$  to  $L$ ;
7   for each neighbor  $u$  of  $v$  do
8     Decrement in-degree of  $u$ ;
9     if in-degree of  $u$  becomes 0 then
10      Enqueue  $u$  to  $Q$ ;
11 if  $L$  contains all vertices then
12   return  $L$ ;
13 else
14   Error: Graph has at least one cycle;

```





3 BFS

The BFS (Breadth First Search) algorithm is an algorithm that traverses all the vertices in the graph in a way breadth first approach, meaning that if we start at vertex s we will not traverse vertices in distance i from s before visiting all vertices of distance $< i$ from s . We can also use this algorithm to efficiently find the distance of any vertex from s .

Suppose we need to find the distance between the points s, v on a graph $G = (V, E)$, we could first look at u , then consider all of its neighbors and mark their distance from u as 1 and then consider all of the neighbors unvisited neighbors and mark their distance from u as 2 and so on. The reason this is called “Breadth First Search” is because we are going to mark all vertices with distance i from u before giving the vertices and distance $i + 1$ from u . What I just described is how the algorithm is supposed to work, but we need to be more exact and write pseudo-code for it, then prove its correctness, and calculate its complexity.

Algorithm 3: BFS Algorithm

Input: Graph $G = (V, E)$, starting vertex s

Output: Visited vertices in BFS order

```

1 Initialize queue  $Q$ ;
2 Mark all vertices as unvisited;
3 Mark the source vertex  $s$  as visited and enqueue  $s$  to  $Q$ ;
4 while  $Q$  is not empty do
5     Dequeue a vertex  $v$  from  $Q$ ;
6     for each neighbor  $u$  of  $v$  do
7         if  $u$  is not visited then
8             Mark  $u$  as visited;
9             Enqueue  $u$  to  $Q$ ;

```

This is the basic BFS algorithm that just traverses the graph. If we want to also find the distances of all the vertices from s we can first initialize our approximation distance function and correct it along the way.

Algorithm 4: BFS Algorithm for finding distances

Input: Graph $G = (V, E)$, starting vertex s

Output: λ such that $\lambda = \delta$

```

1 Initialize queue  $Q$ ;
2 Initialize the  $\lambda(s, v) = \infty$  for every  $v \in V$ ;
3 Redefine  $\lambda(s, s) = 0$  and enqueue  $s$  to  $Q$ ;
4 while  $Q$  is not empty do
5     Dequeue the top vertex  $v$  from  $Q$ ;
6     for each neighbor  $u$  of  $v$  do
7         if  $\lambda(s, u) = \infty$  then
8              $\lambda(s, u) = \lambda(s, v) + 1$ ;
9             Enqueue  $u$  to  $Q$ ;

```

We need to show find its complexity and show correctness. First we can find the complexity. We know that in the initialization we go over all the vertices so the complexity there is $O(|V|)$, and then every iteration we do in $O(\deg(u))$ so the total complexity is:

$$O(|V|) + \sum_{u \in V} O(\deg(u)) = O(|V|) + O(|E|) = O(|V| + |E|)$$

To prove correctness we will show that $\delta(s, v) \leq \lambda(s, v)$ and $\lambda(s, v) \leq \delta(s, v)$. If we have $\delta(s, v) = \infty$ then the inequality is correct, otherwise we can show that by induction over the order the vertices enter the queue.

Base case

if v is the first vertex to enter the queue then we have $v = s$ and then:

$$\delta(s, s) = \lambda(s, s) = 0$$

Step

Now suppose we know that for all the previous vertices that entered the queue we have the inequality, by the algorithm definition we get:

$$\lambda(s, v) = \lambda(s, u) + 1 \underbrace{\geq}_{\text{induction}} \delta(s, v) + 1 \geq \delta(s, v)$$

Which means that for all $v \in V$ we have $\delta(s, v) \leq \lambda(s, v)$ which completes one side of the proof. Try proving the other side yourself.