We can think of graphs as generalizations of trees: they consist of nodes and edges connecting nodes. The main difference is that graphs do not in general represent hierarchical organizations.

**Types of graphs.** Different applications require different types of graphs. The most basic type is the *simple undirected graph* that consists of a set $V$ of *vertices* and a set $E$ of *edges*. Each edge is an unordered pair (a set) of two vertices. We always assume $V$ is finite, and we write $\binom{V}{2}$ for the collection of all unordered pairs. Hence $E$ is a subset of $\binom{V}{2}$. Note that because $E$ is a set, each edge can occur only once. Similarly, because each edge is a set (of two vertices), it cannot connect to the same vertex twice. Vertices $u$ and $v$ are *adjacent* if $\{u, v\} \in E$. In this case $u$ and $v$ are called *neighbors*.

Other types of graphs are

- **directed:** $E \subseteq V \times V$.
- **weighted:** has a weighting function $w : E \to \mathbb{R}$.
- **labeled:** has a labeling function $\ell : V \to \mathbb{Z}$.
- **non-simple:** there are loops and multi-edges.

A *loop* is like an edge, except that it connects to the same vertex twice. A *multi-edge* consists of two or more edges connecting the same two vertices.
**Representation.** The two most popular data structures for graphs are direct representations of adjacency. Let \( V = \{0, 1, \ldots, n-1\} \) be the set of vertices. The adjacency matrix is the \( n \times n \) matrix \( A = (a_{ij}) \) with

\[
a_{ij} = \begin{cases} 
1 & \text{if } \{i, j\} \in E, \\
0 & \text{if } \{i, j\} \notin E.
\end{cases}
\]

For undirected graphs, we have \( a_{ij} = a_{ji} \), so \( A \) is symmetric. For weighted graphs, we encode more information than just the existence of an edge and define \( a_{ij} \) as the weight of the edge connecting \( i \) and \( j \). The adjacency matrix of the graph in Figure 1 is

\[
A = \begin{pmatrix}
0 & 1 & 0 & 1 & 0 \\
1 & 0 & 1 & 0 & 0 \\
0 & 1 & 0 & 1 & 0 \\
1 & 0 & 1 & 0 & 1 \\
0 & 0 & 0 & 1 & 0
\end{pmatrix},
\]

which is symmetric. Irrespective of the number of edges, the adjacency matrix has \( n^2 \) elements and thus requires a quadratic amount of space. Often, the number of edges is quite small, maybe not much larger than the number of vertices. In these cases, the adjacency matrix wastes memory, and a better choice is a sparse matrix representation referred to as adjacency lists, which is illustrated in Figure 2. It consists of a linear array \( V \) for the vertices and a list of neighbors for each vertex. For most algorithms, we assume that vertices and edges are stored in structures containing a small number of fields:

```
struct Vertex { int d, f, π; Edge *adj; }
struct Edge { int v; Edge *next; }
```

The \( d, f, π \) fields will be used to store auxiliary information used or created by the algorithms.

**Depth-first search.** Since graphs are generally not ordered, there are many sequences in which the vertices can be visited. In fact, it is not entirely straightforward to make
sure such that each vertex is visited once and only once. A useful method is depth-first search. The pseudocode below is valid both for directed and undirected graphs (although in the discussion we will assume for simplicity that the graph is undirected). Variable \( V[i].\pi \) stores the parent of vertex \( i \) through which we arrived at \( i \); it also indicates whether the vertex has already been visited (for avoiding repeated visits).

Optionally, for each node \( i \) we can record the time of the first and the last visits (variables \( V[i].d \) and \( V[i].f \) respectively). This information can be useful for analyzing the structure of the graph (one example is given later). For that we maintain a global variable, \( time \), which is incremented after every time-stamping.

```cpp
void VISIT(int i)
1   \( V[i].d = time; \ time++; \)  //optional
2    for all outgoing edges \( ij \) do
3       if \( V[j].\pi < 0 \) then
4          \( V[j].\pi = i; \ VISIT(j) \)
5         endif
6    endfor;
7   \( V[i].f = time; \ time++. \)  //optional
```

The test in line 2 checks whether the neighbor \( j \) of \( i \) has already been visited. The assignment in line 3 records that the vertex is visited from vertex \( i \). A vertex is first stamped in line 1 with the time at which it is encountered. A vertex is second stamped in line 4 with the time at which its visit has been completed. To prepare the search, we initialize the global time variable to 1, label all vertices as not yet visited, and call \( VISIT \) for all yet unvisited vertices.

\[ time = 1; \]

\[ \text{forall vertices} \ i \ \text{do} \ V[i].\pi = -1 \ \text{endfor}; \]

\[ \text{forall vertices} \ i \ \text{do} \]

\[ \text{if} \ V[i].\pi < 0 \ \text{then} \ V[i].\pi = 0; \ \text{VISIT}(i) \ \text{endif} \]

\[ \text{endfor}. \]

Let \( n \) be the number of vertices and \( m \) the number of edges in the graph. Depth-first search visits every vertex once and examines every edge twice, once for each endpoint. The running time is therefore \( O(n + m) \), which is proportional to the size of the graph and therefore optimal.

**DFS forest.** Figure 3 illustrates depth-first search by showing the time-stamps \( d \) and \( f \) and the pointers \( \pi \) indicating the predecessors in the traversal. We call an edge \( \{i, j\} \in E \) a tree edge if \( i = V[j].\pi \) or \( j = V[i].\pi \) and a back edge, otherwise. The tree edges form the **DFS forest** of the graph. The forest is a tree if the graph is connected and a collection of two or more trees if it is not connected. Figure 4 shows the DFS forest of the graph in Figure 3 which, in this case, consists of a single tree. The time-stamps \( d \) are consistent with the preorder traversal of the DFS forest. The time-stamps \( f \) are consistent with the postorder traversal. The two stamps can be used to decide, in constant time, whether two nodes in the forest live in different subtrees or one is a descendent of the other.
Figure 3: The traversal starts at the vertex with time-stamp 1. Each node is stamped twice, once when it is first encountered and another time when its visit is complete.

Figure 4: Tree edges are solid and back edges are dotted.

**NESTING LEMMA.** Vertex \( j \) is a proper descendent of vertex \( i \) in the DFS forest iff \( V[i].d < V[j].d \) as well as \( V[j].f < V[i].f \).

Similarly, if you have a tree and the preorder and postorder numbers of the nodes, you can determine the relation between any two nodes in constant time.

**Directed graphs and relations.** As mentioned earlier, we have a directed graph if all edges are directed. A directed graph is a way to think and talk about a mathematical relation. A typical problem where relations arise is scheduling. Some tasks are in a definite order while others are unrelated. An example is the scheduling of undergraduate computer science courses, as illustrated in Figure 5. Abstractly, a relation is a pair \((V,E)\), where \( V = \{0, 1, \ldots, n - 1\} \) is a finite set of elements and \( E \subseteq V \times V \) is a
finite set of ordered pairs. Instead of \((i, j) \in E\) we write \(i \prec j\) and instead of \((V, E)\) we write \((V, \prec)\). If \(i \prec j\) then \(i\) is a \textit{predecessor} of \(j\) and \(j\) is a \textit{successor} of \(i\). The terms relation, directed graph, digraph, and network are all synonymous.

**Directed acyclic graphs.** A \textit{cycle} in a relation is a sequence \(i_0 \prec i_1 \prec \ldots \prec i_k \prec i_0\). Even \(i_0 \prec i_0\) is a cycle. A \textit{linear extension} of \((V, \prec)\) is an ordering \(j_0, j_1, \ldots, j_{n-1}\) of the elements that is consistent with the relation. Formally this means that \(j_k \prec j_\ell\) implies \(k < \ell\). A directed graph without cycle is a \textit{directed acyclic graph}.

**Extension Lemma.** \((V, \prec)\) has a linear extension iff it contains no cycle.

**Proof.** “\(\Rightarrow\)” is obvious. We prove “\(\Leftarrow\)” by induction. A vertex \(s \in V\) is called a \textit{source} if it has no predecessor. Assuming \((V, \prec)\) has no cycle, we can prove that \(V\) has a source by following edges against their direction. If we return to a vertex that has already been visited, we have a cycle and thus a contradiction. Otherwise we get stuck at a vertex \(s\), which can only happen because \(s\) has no predecessor, which means \(s\) is a source.

Let \(U = V - \{s\}\) and note that \((U, \prec)\) is a relation that is smaller than \((V, \prec)\). Hence \((U, \prec)\) has a linear extension by induction hypothesis. Call this extension \(X\) and note that \(s, X\) is a linear extension of \((V, \prec)\).

**Topological sorting with queue.** The problem of constructing a linear extension is called \textit{topological sorting}. A natural and fast algorithm follows the idea of the proof: find a source \(s\), print \(s\), remove \(s\), and repeat. To expedite the first step of finding a source, each vertex maintains its number of predecessors and a queue stores all sources. First, we initialize this information.

```plaintext
forall vertices \(j\) do \(V[j].d = 0\) endfor;
forall vertices \(i\) do
    forall successors \(j\) of \(i\) do \(V[j].d++\) endfor;
endfor;
forall vertices \(j\) do
    if \(V[j].d = 0\) then \texttt{ENQUEUE}(\(j\)) endif endfor.
```

Next, we compute the linear extension by repeated deletion of a source.

```plaintext
while queue is non-empty do
    \(s = \texttt{DEQUEUE}\);
    forall successors \(j\) of \(s\) do
        \(V[j].d--\);
        if \(V[j].d = 0\) then \texttt{ENQUEUE}(\(j\)) endif
    endfor
endwhile.
```

The running time is linear in the number of vertices and edges, namely \(O(n + m)\). What happens if there is a cycle in the digraph? We illustrate the above algorithm for the directed acyclic graph in Figure 6. The sequence of vertices added to the queue is
also the linear extension computed by the algorithm. If the process starts at vertex $a$ and if the successors of a vertex are ordered by name then we get $a, f, d, g, c, h, b, e,$ which we can check is indeed a linear extension of the relation.

**Topological sorting with DFS.** Another algorithm that can be used for topological sorting is depth-first search. We output a vertex when its visit has been completed, that is, when all its successors and their successors and so on have already been printed. The linear extension is therefore generated from back to front. Figure 7 shows the same digraph as Figure 6 and labels vertices with time stamps. Consider the sequence of vertices in the order of decreasing second time stamp:

$$a(16), f(14), g(13), h(12), d(9), c(8), e(7), b(5).$$

Although this sequence is different from the one computed by the earlier algorithm, it is also a linear extension of the relation.