CS 577: Introduction to Algorithms Fall 2019

Graph Primitives

Instructor: Dieter van Melkebeek

DRAFT

(Last updated: September 9, 2019)

In this lecture we review graphs and related notions from discrete math. We revisit the basic algorithmic schema for graph traversal and illustrate their use as primitives. Along the way we recap the notion of mathematical induction.

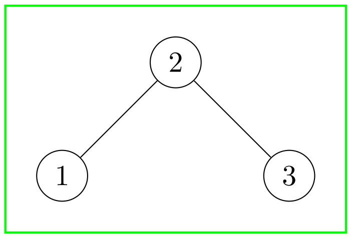
1 Graphs

Graphs are ubiquitous in computer science as they express (symmetric) relationships between discrete objects. For instance, the "is a friend of" relationship that appears in social networks has a natural expression as a graph on the set of participants in the network.

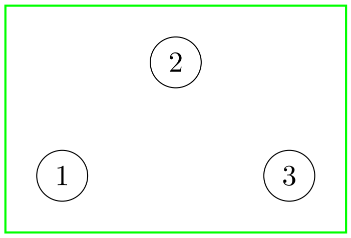
Graphs can be defined in a number of ways, some more general than others. We keep things simple and define a *graph* G as a pair (V, E), where V is some finite set, and E is a set of unordered pairs of distinct elements of V. The elements of V are referred to as *vertices* or *nodes*, and the elements of E are referred to as *edges*. The elements of an edge e are called the endpoints of e.

Note that our definition does not allow parallel edges (E is a set rather than a multiset) nor self-loops (no identical pairs). Figure 1 has a few examples of graphs and non-graphs drawn in the conventional way.

Figure 1: Graph examples and non-examples

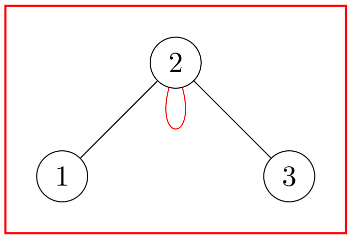


(a) 3 vertices with 2 edges

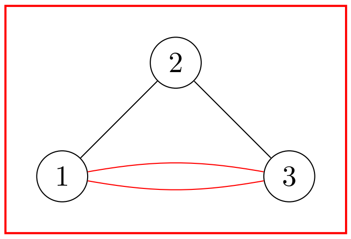


(b) 3 vertices with 0 edges

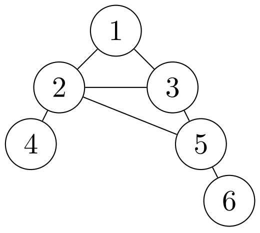
(c) 0 vertices with 0 edges



(d) Graphs don't have self-loops



(e) At most one edge per pair



(f) A graph containing cycles

We say that a vertex u is a *neighbor* of a vertex v when $\{u, v\}$ is an edge. A vertex is not considered to be a neighbor of itself as we do not allow self-loops. The *degree* of a vertex v is the number of neighbors of v, and is denoted by $\deg_G(v)$ or by $\deg(v)$ in case the graph G is clear from the context. The following relationship holds.

Proposition 1. *For every graph $G = (V, E)$*

$$\sum_{v \in V} \deg(v) = 2 \cdot |E|. \tag{1}$$

Equation (1) can be derived by counting the number of pairs $(v, e) \in V \times E$ such that $v \in e$ in two ways:

- Aggregating by vertex yields a contribution of $\deg(v)$ for each vertex v.
- Aggregating by edge yields a contribution of 2 per edge.

Equating the two sums yields (1).

The formula can also be proved using mathematical induction by considering subgraphs. A graph $G' = (V', E')$ is a *subgraph* of a graph $G = (V, E)$ (symbolically: $G' \subseteq G$) if $V' \subseteq V$ and $E' \subseteq E$ as sets. We say that G' is the subgraph *induced by* V' if E' contains all the edges in E for which both endpoints belong to V'. Two ways to obtain a subgraph G' of G are the following:

- Removing a vertex $v \in V$ and its incident edges. The resulting subgraph G' equals the subgraph induced by $V \setminus \{v\}$, and is denoted $G - v$.
- Removing an edge $e \in E$. The resulting subgraph $G' = (V', E')$ is given by $V' = V$ and $E' = E \setminus \{e\}$, and is denoted $G - e$.

Mathematical induction. Consider a statement of the form

$$(\forall t \in \mathbb{N})\, P(t) \tag{2}$$

$P(t)$ denotes a statement that depends on a parameter t ranging over the set $\mathbb{N} \doteq \{0, 1, 2, \dots\}$ of nonnegative integers. A proof of (2) by induction on t consists of two steps:

Base case Show that $P(0)$ holds.

Induction step Show that $(\forall t \in \mathbb{N})\, P(t) \Rightarrow P(t + 1)$ holds, where $\Rightarrow$ denotes logical implication. In words: Show that for every nonnegative integer t, if $P(t)$ holds, then $P(t + 1)$ holds. The premise $P(t)$ in this step is called the induction hypothesis.

Those two steps combined establish (2): We know that $P(0)$ holds by the base case. The base case combined with the induction step for $t = 0$ implies that $P(1)$ holds. The fact that $P(1)$ holds combined with the induction step for $t12$ implies that $P(2)$ holds, and so on.

To illustrate the principle of mathematical induction, we provide an alternate proof of Proposition 1 by induction on the number of edges, *i.e.*, we establish (2), where $P(t)$ denotes the statement that (1) holds for all graphs $G = (V, E)$ with $|E| = t$.

Proof. **Base case:** $P(0)$ states that the sum of the degrees in the empty graph is zero, which is a true statement.

Induction step: Let $G = (V, E)$ be a graph with $|E| = t + 1$ for some $t \in \mathbb{N}$. Let $e^* = \{u^*, v^*\}$ be an arbitrary edge of G (which exists as $|E| = t + 1 \geq 1$). Consider the subgraph $G' = G - e^*$. We have that $G' = (V', E')$ where $V' = V$ and $E' = E \setminus \{e\}$. Moreover,

$$\deg_{G'}(v) = \begin{cases} \deg_G(v) - 1 & \text{if } v = u^* \text{ or } v = v^* \\ \deg_G(v) & \text{otherwise} \end{cases}$$

It follows that

$$\sum_{v \in V} \deg_G(v) = 2 + \sum_{v \in V'} \deg_{G'}(v). \tag{3}$$

As $|E'| = t$, the induction hypothesis implies that $\sum_{v \in V'} \deg_{G'}(v) = 2|E'| = 2t$. Plugging this expression into (3) yields $\sum_{v \in V} \deg_G(v) = 2 + 2t = 2(t + 1) = 2|E|$.

□

A closely related proof strategy is *strong induction*, in which the induction step is modified as follows:

$$(\forall t \in \mathbb{N})\, P(0) \wedge P(0) \wedge \cdots \wedge P(t) \Rightarrow P(t + 1),$$

where "$\wedge$" denotes conjunction (logical and). Sometimes the smallest value in the range for t is an integer b other than 0; the base case changes accordingly to establishing $P(b)$. Sometimes there are multiple base cases, and the smallest value of the range for t in the induction step is larger than b.

Exercise 1. *Establish Proposition 1 by strong induction on the number of vertices.*

Connectivity. A *path* is a sequence $v_0, v_1, \ldots, v_\ell$ of vertices such that v_i is a neighbor of v_{i-1} for every $i \in [\ell] \doteq \{1, 2, \ldots, \ell\}$. We say the path goes *from* v_0 *to* v_ℓ. The *length* of the path is the value ℓ, which counts the number of edges along the path. We call the path *simple* if all vertices on the path are pairwise distinct except possibly $v_0 = v_\ell$. We call it *closed* if $v_0 = v_\ell$.

A *cycle* is a closed path for which there is no backtracking. Backtracking intuitively means using the same edge twice in a row, and is formally means that $v_i = v_{i+2}$ for some i. The formal definition corresponds to the intuitive one, since if $v_i = v_{i+2}$, then the path travels $v_i \to v_{i+1} \to v_i$, and follows the edge on v_i and v_{i+1} two times in a row. A *simple* cycle is a cycle that is a simple path, *i.e.*, *i.e.*, the only repeated vertices are $v_0 = v_\ell$. For every vertex v, there is a simple path from v to itself, namely the path 'v' of length 0; this path does not represent a cycle.[1]

A pair u, v of vertices is said to be *connected* if there is a path from u to v. A graph is connected if all pairs of vertices are connected. Not every graph $G = (V, E)$ is connected, but every graph can be decomposed into connected graphs as follows. If V' is a subset of V such that the induced subgraph G' is connected, and no strict superset of V' has that property, then G' is a *connected component* of G. The sets of vertices of the connected components of G form a partition of V, *i.e.*, they are pairwise disjoint and their union equals V.[2] Each edge of G is included in one (and only one) of the connected components of G. Thus, G is the *disjoint union* of its connected components.

[1] Our terminology is common but there are other standards. Some authors refer to our paths as "walks", our simple paths as "paths", and our simple cycles as "cycles".

[2] They are the equivalence classes of the connectivity relation on V.

Graphs in algorithms. In the context of computer science, there are a few extra issues to consider when it comes to graphs.

The first is purely notation. In graph problems, the "input size" is measured in terms of two variables: the number of vertices and the number of edges. Typically the letter n denotes the number of vertices in a graph, and the letter m denotes the number of edges. Since graphs on n vertices can have as many as $\binom{n}{2} = \Theta(n^2)$ edges and as few as zero edges, we need to differentiate between these two quantities in order to have a useful understanding of algorithmic efficiency. For instance, an algorithm that takes $\Theta(n^2)$ time performs much worse than an algorithm that takes $\Theta(n + m)$ time when the graph has relatively few edges (*e.g.*, trees, which have $m = \Theta(n)$).

The second issue is that of representation. There are two common ways to represent a graph inside a computer. The first is to use an *adjacency matrix*, and the second is to use an *adjacency list*. An adjacency matrix is simply the matrix whose rows and columns are indexed by the vertices. The entry corresponding to the u-th row and v-th column is 1 when u is a neighbor of v, and is 0 otherwise. An adjacency list is an array of lists: each vertex v corresponds to a position in this array, and the stored list contains the neighbors of v.

Adjacency matrices are generally simpler to work with when implementing algorithms. However, they can be rather inefficient, since they use $\Theta(n^2)$ space no matter how many edges are in the graph. They also suffer from the drawback that, in order to figure out the neighbors of a vertex v, one needs to look at all n entries in the v-th row of the matrix.

On the other hand, adjacency lists are only marginally more complicated to work with, and only use $\Theta(n + m)$ space. They also have the advantage that, in order to look at all the neighbors of a vertex v, one only needs to look at d_v entries, where d_v is the degree of v in the graph. In particular, in an adjacency list representation, we can enumerate all the edges of a graph in time $\Theta(m)$: we just iterate over all the vertices, and, for each vertex v, enumerate the edges indicated by the adjacency list for v. Note, however, that this formulation actually lists every edge twice—the edge $\{u, v\}$ appears both in the adjacency list for u and for v.

2 Trees

In graph theory a *tree* is defined as a connected graph that contains no cycles. A *rooted tree* is a tree together with a designated vertex, called the root. A rooted tree is often drawn like a real tree, but often also upside down (with the root on top). A *leaf* of a tree is a vertex of degree 1. Unlike real trees, the root of a rooted tree can also be a leaf. A graph, each of whose connected components is a tree, is called a *forest*.

Nonempty rooted trees can also be defined constructively in the following recursive fashion:

1. A graph consisting of a single vertex is a rooted tree with root that vertex.

2. Let $G_i = (V_i, E_i)$ for $i \in [k]$ be a rooted tree with root r_i such that the sets V_i are pairwise disjoint, and let r be a new vertex. The disjoint union of the G_i together with the vertex r and edges between r and each r_i is a rooted tree with root r.

The equivalence of this recursive definition and the above graph-theoretic definition follows from the discussion on graph traversal in Section 3.

Recursive constructive definitions of objects lend themselves to proving properties of those objects by induction on the number of times a constructor rule is applied. We refer to such a proof strategy as *structural induction*. In the case of nonempty rooted trees, the base case corresponds to the first constructor rule, and the induction step to the second one. We illustrate structural induction for the following proposition.

Proposition 2. *For any nonempty tree $G = (V, E)$, $|E| = |V| - 1$.*

Proof. We prove the proposition by strong structural induction for rooted trees, i.e., we show $(\forall t \in \mathbb{Z}^+)\, P(t)$, where $P(t)$ denotes that $|E| = |V| - 1$ holds for all rooted trees $G = (V, E)$ obtained by t applications of the constructor rules.

Base case: $P(1)$. The only non-empty rooted trees that can be obtained by one application of the constructor rules consist of a single vertex, as obtained via the first constructor rule. For those $|V| = 1$ and $|E| = 0$, so $|E| = |V| - 1$ holds.

Inductive step: For $t \in \mathbb{Z}^+$, consider a rooted tree $G = (V, E)$ obtained by $t + 1$ applications of the constructor rule. If the last rule applied is rule 1, we are done by the base case. If not, let $G_i = (V_i, E_i)$ with root r_i be as in rule 2. By construction, we have that $|E| = k + \sum_{i \in [k]} |E_i|$ and $|V| = 1 + \sum_{i \in [k]} |V_i|$. Note that each G_i is a rooted tree obtained by at most t applications of the constructor rules. By the induction hypothesis that $P(s)$ holds for $s \in [t]$, we have that $|E_i| = |V_i| - 1$ for each $i \in [k]$. Thus,

$$|E| = k + \sum_{i \in [k]} |E_i| = k + \sum_{i \in [k]} (|V_i| - 1) = \sum_{i \in [k]} |V_i| = |V| - 1.$$

□

As a corollary to Proposition 1 and Proposition 2 we have that every tree with at least two vertices has at least two leaves.

The property in Proposition 2 can replace one of the defining conditions of a tree as follows.

Proposition 3. *A nonempty graph $G = (V, E)$ is a tree if and only if it satisfies at least two of the following:*

1. *G is connected*
2. *G has no cycles*
3. *$|E| = |V| - 1$*

in which case it satisfies all three.

Proof. The definition of a tree and Proposition 2 show that if G is a tree, then it satisfies all three of the listed conditions.

For the other direction, we work in cases.

Suppose that conditions (1) and (2) hold. Since G is connected and has no cycles, it is by definition a tree.

Now suppose that conditions (1) and (3) hold. Since G is connected, it has a spanning tree, $G' = (V', E')$. Since G' is a spanning tree, $V' = V$, and by Proposition 2, $|E'| = |V| - 1$. Since $|E| = |V| - 1$, we have $|E| = |E'|$. Since spanning trees are subgraphs, $E' \subseteq E$. Because these sets are finite, $E' \subseteq E$ and $|E'| = |E|$ implies that $E = E'$, and so $G = G'$, and thus G is a tree.

Finally, suppose that conditions (2) and (3) hold. Suppose that G has k connected components, $G_i = (V_i, E_i)$ for $i \in [k]$. Since G has no cycles, no G_i has a cycle. Since each G_i is connected, it follows that G_i is a tree for every i. By Proposition 2, we know $|E_i| = |V_i| - 1$ for every i. Since every vertex and every edge of G appears in exactly one G_i, it follows that $|V| = |V_1| + |V_2| + \cdots + |V_k|$ and $|E| = |E_1| + |E_2| + \cdots + |E_k|$. Applying condition (3), we have the following:

$$\begin{aligned}\left(\sum_i |V_i|\right) - 1 &= \sum_i |E_i| \\ &= \sum_i (|V_i| - 1) \\ &= -k + \sum_i |V_i|\end{aligned}$$

By canceling the summations, we have $k = 1$. Thus $G = G_1$ is connected. □

Spanning trees. For a graph $G = (V, E)$, a *spanning tree* of G is a subgraph $G' = (V', E')$ of G which is a tree and for which $V' = V$ (*i.e.*, every vertex of G is in G'). The following proposition characterizes the existence of spanning trees:

Proposition 4. *For any graph G, G has a spanning tree if and only if G is connected.*

Proof. If G has a spanning tree, say G', then since G' is connected, it follows that G is connected. For the other direction, we can use induction on the number of edges in G.

Base case: Suppose that G has no edges. Then because G is connected, it follows that G must have either zero vertices or one vertex. In either case, G is a spanning tree of itself.

Inductive step: G is assumed to be connected, so we just need to show that G has no cycles. Indeed, if $v_0, \ldots, v_\ell$ is a cycle in G, then consider $G' = G - e$ where e denotes the edge $\{v_0, v_\ell\}$ G' is a subgraph of G that is still connected. This is because any path in G that uses the edge e can be transformed into a path in G' between the same vertices by replacing the edge e by the path $v_0, \ldots, v_\ell$. Since G' has one edge less than G, we can apply to our inductive hypothesis to G', and obtain a spanning tree T of G'. Since G' and G have the same vertices, it follows that T is also a spanning tree of G.

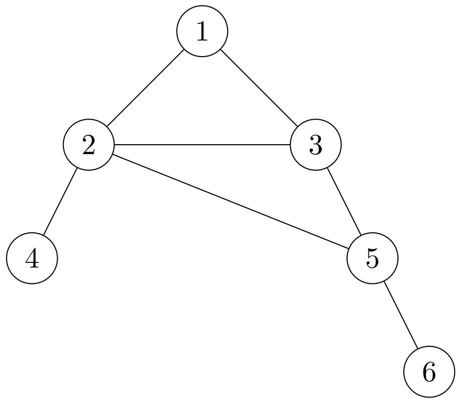
□

For graphs G that are not connected, we can consider its decomposition into connected components, and obtain spanning trees of each of the connected components. The subgraph that is the union of all these spanning trees is a *spanning forest* of G. Thus, every graph has a spanning forest.

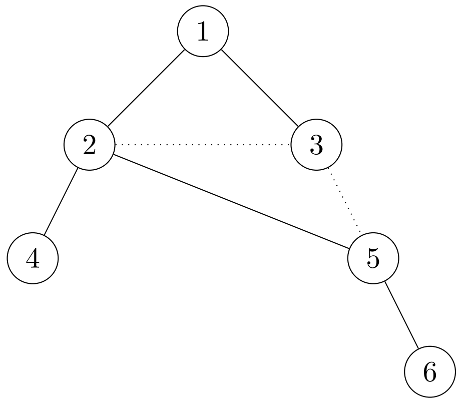
3 Graph Traversal

A graph traversal visits every vertex and edge of a given connected graph G. The two best-known traversal algorihtms are breadth-first search (BFS) and depth-first search (DFS). BFS is iterative

Figure 2: BFS Tree Example



(a) Input graph



(b) BFS tree from Figure 2a

in nature; DFS is recursive in nature. Both run in linear time. See the Powerpoint presentations from class.

BFS and DFS can be viewed as two instantiations of an algorithm scheme for graph traversal that is parameterized by the choice of a data structure. Instantiating the scheme with a queue yields BFS, and with a stack yields DFS. See Section 5.5 and 5.6 of the text by Erickson for more details.

Each instantation also implicitly computes a spanning tree of G, namely the union of the edges along which new vertices are discovered. The spanning trees computed by BFS and DFS have different interesting properties. We briefly describe them and illustrate how they can be exploited algorithmically.

3.1 BFS

An example of a BFS tree for a connected graph is given in Figure 2.

If we group vertices of the spanning tree by their distance from the root, then we can see that every non-tree edge of the graph either stays within a single level, or crosses from one level to an *adjacent* level. This follows from the fact that, when BFS processes a vertex, *all* of its unprocessed neighbors become its children in the BFS tree.

Shortest paths. One use of BFS is to find shortest-paths in (unweighted) graphs. Suppose we are given a graph G, and a pair of vertices s and t of G, and we want to know a shortest path from s to t in G. We can do this with a BFS seeded by s to build the BFS tree for the connected component of G that contains s. If this connected component doesn't contain t (*i.e.*, BFS never visits t), then there is no path from s to t. Otherwise, t appears in the BFS tree at some level ℓ (starting with s at level zero). The claim is that t must be at distance exactly ℓ from s in G. That it is at distance at most ℓ is because the edges in the BFS tree are also edges in G, and t being at level ℓ means the unique a path of from s to t in the BFS tree has length ℓ.

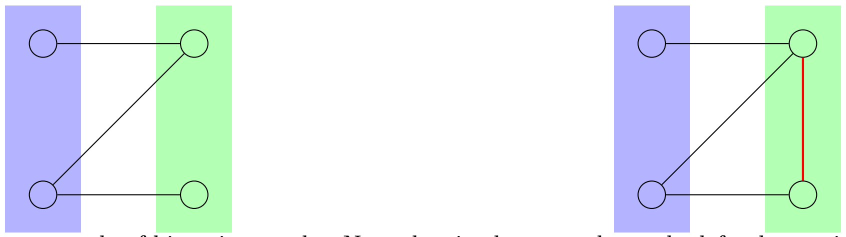
We can argue that the distance is also at least ℓ by using the structural property of BFS trees. In particular, any edge on a path in G can move the level in the BFS tree by at most one. It follows that at least ℓ edges are needed to go from s to t.

Bipartiteness. There are many special kinds of graphs. One particular kind that will play a significant role later in the course is that of a *bipartite graph*. Informally, a bipartite graph is a graph where the vertices can be partitioned into two parts such that the edges are never between two vertices in the same part. Formally, $G = (V, E)$ is a bipartite graph if there are nonempty sets of vertices $L, R \subseteq V$ so that the following properties hold:

1. $L \cup R = V$. Every vertex in V is in L or in R.
2. $L \cap R = \varnothing$. No vertex in V is in both L and R.
3. For every edge $\{u, v\}$ in E, either u is not in L or v is not in L, and, either u is not in R or v is not in R.

A visual depiction of bipartite graphs is given in Figure 3.

Figure 3: Bipartite graphs



An example and nonexample of bipartite graphs. Note that in the example on the left, the vertices are partitioned into two parts, denoted by the blue and green shaded regions. In the nonexample on the right, the presence of the red edge forces the graph not to be bipartite.

In some settings the decomposition of the graph's vertices into two parts is obvious. In some other settings such a decomposition is less obvious, and it may not even be clear whether a graph is bipartite in the first place. Using BFS we can can compute such a decomposition (or discover that none exists) in linear time.

We once again take advantage of the the structure of BFS trees. Let $G = (V, E)$ be a graph, and let T be a BFS tree of G. Suppose that every edge of G is between *different* levels of T; *i.e.*, every edge of G either goes up one level in T, or else goes down one level in T. Then we can let L be the vertices at odd levels in T, and let R be the vertices at even levels in T. Since following an edge in G changes the parity of the level in T, it follows that this decomposition satisfies the required properties for a bipartite decomposition of G.

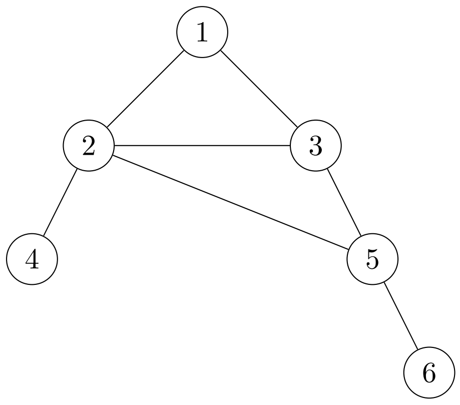
On the other hand, it may be the case that T contains an edge between two vertices at the same level. Let u and v be these vertices, and consider the path from u, to the root of T, back down to v, and finally across the edge to u. This path is a cycle, and it moreover has an odd length. No bipartite graph can have an odd cycle, so this means that G cannot be bipartite.

3.2 DFS

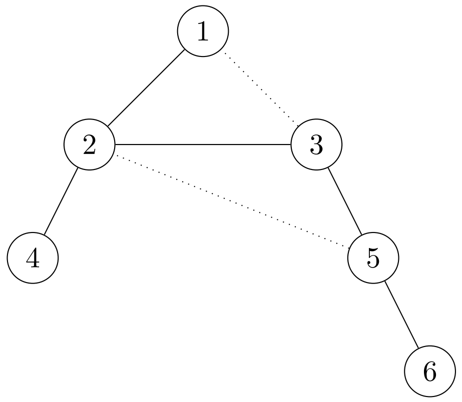
Figure 4 gives an example of a DFS tree for the same connected graph for which Figure 2 depicts a BFS tree.

DFS trees have different properties than BFS trees. A particularly useful property of DFS trees is the following: For every edge $\{u, v\}$ in the input graph, either u is an ancestor of v in the DFS tree, or else v is an ancestor of u in the DFS tree.

Figure 4: DFS Tree Example



(a) Input graph



(b) DFS tree from the graph in Figure 4a when starting from vertex 1

Bridges. One application of this property of DFS trees is to efficiently find all bridges in a graph. A *bridge* in a graph G is an edge e so that removing e from G increases the number of connected components. A straightforward way to find all bridges is to explicitly check for each edge whether removing it increases the number of connected components. This approach has a running time of $\Theta(m \cdot (n + m))$.

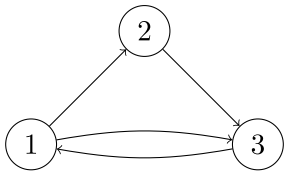
Using the above property of DFS trees, we can do it in linear time by running DFS and keeping track of some additional information (like the level in the BFS application above). We omit the details.

4 Directed Graphs

In our discussion of graphs so far, we have focused entirely on *undirected* graphs. The qualifier undirected refers to the fact that the edges are unordered pairs; there is no sense of direction associated with them. Assigning a direction to edges allows us to capture relationships that are not symmetric. For this, we define the notion of a *directed graph* to be a pair (V, E), where V is again some finite set, and E is a set of *ordered* pairs of distinct elements of V. Directed graphs are also known as *digraphs* for short. When the pair (u, v) is in the edge set E, we say that there is an edge *from u to v*. The conventional way to draw a directed graph is to add arrows to the edges, with the arrowhead pointing to the 'to' vertex, as demonstrated in Figure 5.

The various topological notions of graphs change slightly for digraphs. The most fundamental

Figure 5: A directed graph with 3 vertices with 4 edges



The edge $(1, 2)$ is drawn as a line from the vertex 1 to the vertex 2, with an arrowhead on the 2-side of the edge.

change is that the “is a neighbor of” relation fails to be symmetric, as we define v to be a neighbor of u precisely when there is an edge from u to v. One consequence of this is that the degree of a vertex is the number of edges *leaving* the vertex. To help emphasize this, and also to give a name to the number of edges entering a vertex, we say the *in-degree* and *out-degree* of a vertex u are, respectively, the number of edges entering and leaving u. Another consequence of this loss of symmetry is that paths and cycles only follow edges in the forward direction.

Every undirected graph can be regarded as a directed graph, where the edge $\{u, v\}$ in the undirected graph is regarded as the edges (u, v) and (v, u) in the directed graph. The topological notions of “path”, “is a neighbor of”, and so on are preserved in this transformation, except that now paths of the form u, v, u are cycles in the directed interpretation, whereas they were not in the undirected sense.

Another topological notion that gets more complicated is that of connectivity. Since the “is a neighbor of” relation is no longer symmetric, we can no longer make the inference “if there is path from u to v, then there is a path from v to u”. To address this, we will think of two vertices u and v as being “connected” if there is a path from u to v *and* a path from v to u. We say a directed graph is *strongly connected* if every pair of vertices is connected in this sense. The qualifier “strong” is introduced to distinguish the difference in definition between connectivity as intended for undirected graphs, and this new notion adapted for digraphs.

As with undirected graphs, every directed graph decomposes into strongly connected components. However, in the case of directed graphs, there can be edges that do not belong to any connected component.

Representation of digraphs. We note that directed graphs can also be represented in the adjacency matrix and in the adjacency matrix formats. The idea is still to encode the “is a neighbor of” relation, so that the vertices indicated as adjacent to a vertex u are exactly the vertices v for which there is an edge from u to v. The drawbacks of the adjacency matrix approach as compared to the adjacency list approach apply to the digraph setting as well.

There is one improvement that happens when working with adjacency lists for digraphs: when we enumerate the edges of the graph by first enumerating all vertices, and then enumerating all the edges leaving this vertex, we only enumerate each edge exactly *once*.

5 Digraph Traversal

BFS and DFS can be defined for directed graphs similarly to how they are defined for undirected graphs. We can also define the notions of BFS and DFS trees as before. However, the loss of symmetry in the “is a neighbor of” relation induces some loss of structure in the BFS and DFS trees obtained in this way.

For BFS trees, the loss of structure is that, while edges can still only go down by at most one level, they can go up *any* number of levels. This still allows us to find shortest paths in directed graphs using BFS.

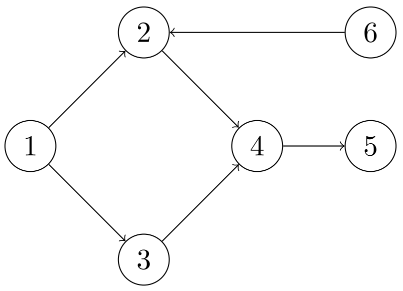
For DFS trees, in the case of undirected graphs, non-tree edges can only connect to ancestral vertices. In the case of directed graphs, non-tree edges can point to any vertex appearing earlier in the order of vertices considered by DFS. This is also still useful. For example, based on DFS one can find all strongly connected components of a digraph in linear time. We omit the details.

We will also see DFS appear in the context of finding topological sorts of directed acyclic graphs.

6 Direct Acyclic Graphs

A *directed acyclic graph*, or DAG for short, is just as the name implies: it is a directed graph $G = (V, E)$ which has no cycles. DAGs have important applications throughout computer science. Figure 6 gives an example of a DAG:

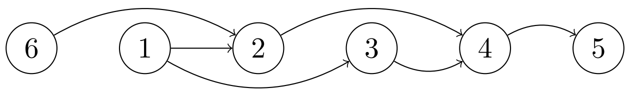
Figure 6: A directed acyclic graph



DAGs can be *topologically sorted* (also referred to as *linearly ordered*, or some combination thereof). What this means it that there is an enumeration of the vertices $v_1, v_2, \ldots, v_n$ so that every edge (v_i, v_j) is oriented such that $i < j$; *i.e.*, it points from earlier in the enumeration to later in the enumeration.

A useful analogy is to think of the vertices as being placed on a line, with v_1 placed at the origin, then v_2 one unit to the right, then v_3, and so on. Our condition that every edge (v_i, v_j) is oriented with $i < j$ translates to drawing the edge from left to right. An example is given by Figure 7, which is a linear order of the vertices from Figure 6.

Figure 7: A linear ordering of the DAG in Figure 6



Note that all the edges are oriented from left to right. Thus the order "6, 1, 2, 3, 4, 5" is a linear ordering of the vertices in Figure 6. Another linear order would be "1, 6, 2, 3, 4, 5", or "1, 3, 6, 2, 4, 5".

In fact, DAGs are precisely the digraphs that can be linearly sorted:

Proposition 5. *Let $G = (V, E)$ be a directed graph. Then G can be linearly sorted if and only if G is a DAG.*

Proof. $\Rightarrow$ We prove this direction by contraposition. Suppose that G is not a DAG, *i.e.*, that G has a cycle. Let $v_1, v_2, \ldots, v_k$ be this cycle. Then we have the edges $v_1 \to v_2 \to \cdots \to v_k \to v_1$, which implies that any linear ordering of G satisfies

$$1 < 2 < \cdots < k < 1$$

which is impossible for any order.

$\Leftarrow$ We prove this direction by induction on the number of vertices.

$|V| = 0$ The base case is trivial; there are no vertices to order!

$|V| = t + 1$ Since G is a DAG, we can find a vertex v of G that has zero incoming edges: If we pick v from G arbitrarily and it has an incoming edge, we just follow the edge in reverse to pick a new v, and repeat this indefinitely. If this procedure never stopped, then G would have a cycle. So, it must stop at some point, and thus there is a vertex of G with indegree zero. Let v be this vertex.

When we remove v from G, G is still an acyclic graph, so we can apply our inductive hypothesis to get a sequence of vertices $v_1, \ldots, v_k$ which is a linear order of $G - v$. Then suppose we add v to the the beginning of this order; *i.e.*, we set $v_0 = v$. Then the sequence $v_0, \ldots, v_k$ is a topological sort of the vertices of G: Every edge either exists in G after removing v, or else involves v, but the latter kind of edges only point out of v, and hence cannot cause trouble when v is at the beginning of the sequence.

□

In fact, the proof of the $\Leftarrow$ direction of Proposition 5 leads to a linear-time algorithm for finding a topological order (or detecting that none exists). The algoritm can be based on DFS and the properties of DFS trees.

We describe an alternate linear-time implementation from first principles. It maintains some additional datastructures to facilitate an efficient execution, namely a table D with the indegrees of all vertices in the remaining digraph, and a queue Q of the ones of indegree zero. Pseudocode is given in Algorithm 1. We leave the analysis as an exercise.

Algorithm 1

Input: $A[v \in V]$, an adjacency list representation of the directed acyclic graph $G = (V, E)$, where $n = |V|$ and $m = |E|$

Output: A topological order $v_1, v_2, \ldots, v_n$ of V

```
procedure TOPOLOGICAL-SORT($A$)
    $D[v \in V] \leftarrow$ array of integers, initialized to zero
    for $v \in V$ do
        for $v' \in A[v]$ do
            $D[v'] \leftarrow D[v'] + 1$
    $Q \leftarrow$ empty queue of vertices
    for $v \in V$ do
        if $D[v] = 0$ then
            $Q$.PUSH($v$)
    $k \leftarrow 0$
    while $\neg$ $Q$.EMPTY do
        $k \leftarrow k + 1$
        $v_k \leftarrow Q$.POP
        for $v' \in A[v_k]$ do
            $D[v'] \leftarrow D[v'] - 1$
            if $D[v'] = 0$ then
                $Q$.PUSH($v'$)
    return $v_1, v_2, \ldots, v_k$          ▷ If $G$ were not acyclic, then we would have $k < n$ here
```