CS 577: Introduction to Algorithms Fall 2015

Graph Primitives

Instructor: Dieter van Melkebeek Scribe: Andrew Morgan

One of the most fundamental objects in finitary mathematics, including computer science, is the notion of a *graph*. Graphs provide a very general way by which to express relationships between objects in a set. For instance, the "is a friend of" relationship that appears in social networks has a very natural expression as a graph on the set of participants in the network. Flowcharts, dependency graphs, call graphs, and so on are all examples of graphs from the realm of software engineering. Accordingly, algorithms that perform certain operations to graphs, extract certain structure from graphs, or otherwise involve graphs are indispensable for computer science.

These notes present a few of the most basic of these algorithms. These algorithms are used as primitives in the remaining graph-theoretic algorithms that we will use in the rest of this course.

Often the applications of these algorithms are not purely black-box constructions: the primitives are modified slightly so as to help construct a relevant data structure, or use a pre-existing data structure to change the behavior of the primitive in some desirable way. As such, understanding these algorithms as mapping 'inputs' to 'outputs' is generally suboptimal; rather, we will present the graph primitives as algorithms, and then allow the more sophisticated algorithms we will see in the future serve as examples of the various ways these primitives can be used.

1 Graph Definitions

Most of this section should be a review, so these definitions are presented pretty quickly and with few examples.

Graphs are defined in many ways, some more general than others. Here, we will keep things simple and define a *graph* G to be 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*, and the elements of E are referred to as *edges*. Figure 1 has a few examples of graphs and non-graphs[1] drawn in the conventional way.

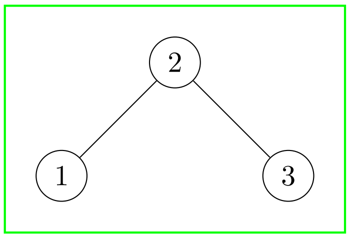
Our next definition is that of a subgraph. Like subsets, subgraphs are 'a part of' their containing graph, in the sense that structure of the subgraph is also present in the containing graph. Formally, we say that a graph $G = (V, E)$ is a subgraph of a graph $G' = (V', E')$ (symbolically: $G \subseteq G'$) when G is a graph, and $V \subseteq V'$ and $E \subseteq E'$ as sets.

Fix a graph $G' = (V', E')$. For a subset of vertices $V \subseteq V'$, let E be the set of all edges in E' for which both end points belong to V. We say that the graph $G = (V, E)$ is *induced by* the vertex set V. Equivalently we say, G is the *induced subgraph* of G' by V.

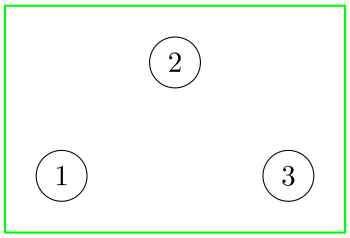
Graphs are usually regarded as having a loose notion of topology associated with them—people speak of 'neighbors', 'paths', 'connectivity', and so on. Let's formally define these notions as well.

[1] This is according to our definition of a graph. Various generalizations of our definition allow for each of the graphs in Figure 1, except no common definition allows the edge going into space in Figure 1d.

Figure 1: Graph Examples and Non-examples

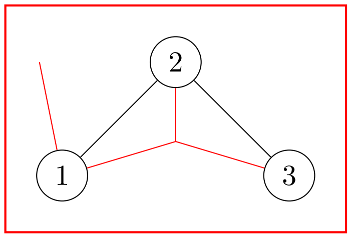


(a) 3 vertices with 2 edges

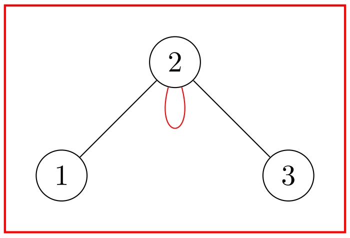


(b) 3 vertices with 0 edges

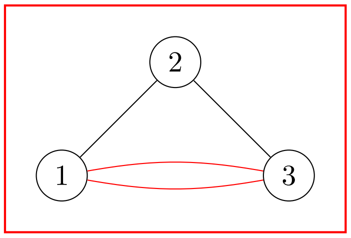
(c) 0 vertices with 0 edges



(d) Edges are between two vertices



(e) Graphs don't have self-loops



(f) At most one edge per pair

We say that a vertex u is a *neighbor* of a vertex v when there is an edge on u and v. A vertex is not considered to be a neighbor of itself, because we do not allow self-loops in our graphs. The *degree* of a vertex v is the number of neighbors of v.

A *path* is a sequence $v_1, v_2, \dots, v_{\ell+1}$ of vertices such that v_{i+1} is a neighbor of v_i for every $i = 1 \dots \ell$. The *length* of the path is the value ℓ, which counts the number of edges along the path. We say the path goes *from* v_1 *to* $v_{\ell+1}$. A *simple* path is a path $v_1, v_2, \dots, v_{\ell+1}$ for which no vertex appears more than once; *i.e.*, $v_i \neq v_j$ for all $i \neq j$. For every vertex v, there is a simple path from v to itself, namely the simple path 'v'.

A *cycle* in a graph is a path $v_1, v_2, \dots, v_{\ell+1}$ for which $v_1 = v_{\ell+1}$, $\ell \geq 1$, and for which there is no backtracking. Backtracking intuitively means using the same edge twice in a row, and is formally as there being $v_i = v_{i+2}$ 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 $v_1, v_2, \dots, v_{\ell+1}$ for which $v_1, v_2, \dots v_\ell$ is a simple path; *i.e.*, the only repeated vertices are $v_1 = v_{\ell+1}$.

An graph is said to be *connected* if for every pair u, v of vertices, there is a path from u to v. Not every graph is connected, but every such graph can be regarded as a union of connected graphs:

Proposition 1. *Every graph $G = (V, E)$ has, for some number k, k connected subgraphs $G_i = (V_i, E_i)$ for $i = 1, \dots, k$, such that V_i and V_j are disjoint for $i \neq j$, and every edge $\{u, v\} \in E$ is present in E_i for some i (where i may depend on the edge). Furthermore, this decomposition is unique up to re-indexing.*

To prove Proposition 1, we will use the notion of an *equivalence relation*:

Equivalence relations Formally, for any set S, a (binary) *relation* on S is just a subset of $S \times S$; *i.e.*, it is just a set of ordered pairs of elements of S. An *equivalence relation* is a relation satisfying three additional properties. Usually, equivalence relations are written using the notation $a \sim b$ to indicate that the pair (a, b) is in the relation. The three properties satisfied by equivalence relations are then the following:

(reflexivity) For every element a in S, $a \sim a$.

(symmetry) For every pair of elements a, b in S, if $a \sim b$, then $b \sim a$.

(transitivity) For every triple of elements a, b, c in S, if $a \sim b$ and $b \sim c$, then also $a \sim c$.

An important property of equivalence relations is that they *paritition* the underlying set S. Formally, we have Proposition 2:

Proposition 2. *Let $\sim$ be any equivalence relation on S. Then there exists a family $\mathscr{C}$ of subsets of S so that*

1. *For every pair of sets A, B in $\mathscr{C}$, either $A = B$ or A and B are disjoint.*
2. *Every element of S is in some subset in $\mathscr{C}$.*
3. *For every pair of elements a, b in S, a and b are in the same subset in $\mathscr{C}$ if and only if $a \sim b$.*

Moreover, this family $\mathscr{C}$ is unique.

The subsets in $\mathscr{C}$ are referred to as the *equivalence classes* of $\sim$. Since each element of S belongs to exactly one equivalence class, and the equivalence classes are uniquely determined by $\sim$, we introduce the notation $[a]$ to represent the unique equivalence class of the element a of S. That is, $[a]$ denotes the set of all elements b so that $a \sim b$.

proof of Proposition 2. The idea of this proof is straightforward, but for concision is formulated somewhat abstractly.

For each element a in S, let C_a denote the set of elements b of S satisfying $a \sim b$. Let $\mathscr{C} \doteq \{C_a : a \in S\}$ be the set of all of these subsets of S. The claim is that $\mathscr{C}$ satisfies the above properties.

1. Let a and b be any two elements of S. If C_a and C_b are not disjoint, then they have a common element c. By definition, we have $a \sim c$ and $b \sim c$. Using the symmetry and transitivity properties of equivalence relations, we conclude that $a \sim c \sim b$, and hence $a \sim b$ and $b \sim a$. It then follows that for every element x in C_b, $a \sim b \sim x$, and thus $a \sim x$, and so x is in C_a; *i.e.*, $C_b \subseteq C_a$. Symmetrically, $C_a \subseteq C_b$, and so $C_a = C_b$.
2. Let a be any element of S. Then by reflexivity of $\sim$, we have $a \sim a$, and hence a is in C_a.
3. Let a, b be any pair of elements in S. Suppose that a and b are in the subset C_c of $\mathscr{C}$. Then we know that $c \sim a$ and $c \sim b$, so by symmetry and transitivity of $\sim$, we have $a \sim b$. Conversely, if $a \sim b$, then using parts 1 and 2, we conclude that $C_a = C_b$ is the unique subset of $\mathscr{C}$ containing a and b.

To prove uniqueness of $\mathscr{C}$, suppose that $\mathscr{C}'$ is some other family of subsets of S satisfying the above properties. Note that properties 1 and 2 imply that every element of S belongs to a unique set in $\mathscr{C}'$. Thus, for every a in S, let C'_a be the set in $\mathscr{C}'$ which contains a. Using property 3, it's easy to see that it must be the case that $C'_a = C_a$. This implies $\mathscr{C} \subseteq \mathscr{C}'$. The other direction is symmetric, and we conclude that $\mathscr{C} = \mathscr{C}'$. $\square$

With this essential notion of equivalence relations in hand, we can now prove Proposition 1:

proof of Proposition 1. The basic idea of the proof is to introduce an equivalence relation on the vertices in G so that the equivalence classes are exactly the (vertex sets of the) connected components of G.

To do this, define the relation $\sim$ on the vertices V by $u \sim v$ if and only if there exists a path $u \to v$ in G. This turns out to be an equivalence relation:

(reflexivity) For every vertex u in V, the trivial path 'u' is a path from u to itself.

(symmetry) For every pair of vertices u, v in V, if $u \sim v$, then there is a path $u, v_2, v_3, \ldots, v_\ell, v$ from u to v, then the path $v, v_\ell, \ldots, v_3, v_2, u$ is a path from v to u, and so $v \sim u$.

(transitivity) Let u, v, w be any triple of vertices in V, and suppose that $u \sim v$ and $v \sim w$. This means there are paths $u, v_2, v_3, \ldots, v_\ell, v$ and $v, v_2', v_3', \ldots, v_k', w$. We can concatenate these paths to get the path $u, v_2, v_3, \ldots, v_\ell, v, v_2', v_3', \ldots, v_k', w$ from u to w; hence $u \sim w$.

Having defined $\sim$, we can use Proposition 2 and consider its family of equivalence classes $\mathscr{C}$. Let $V_1, V_2, \ldots, V_k$ be an arbitrary indexing of the elements of $\mathscr{C}$, and let $G_1 = (V_1, E_1), G_2 = (V_2, E_2), \ldots, G_k = (V_k, E_k)$ be the subgraphs of G induced by the respective vertex sets. The fact that $V_1, \ldots, V_k$ arose as a family of equivalence classes proves (via Proposition 2) that they are disjoint and cover every vertex in G. Thus we just have to show that (1) G_i is connected for every i, (2) every edge $\{u, v\}$ in E is present as an edge in E_i for some i, and (3) the decomposition $G_1, G_2, \ldots G_k$ is unique up to re-indexing.

That G_i is connected for every i follows from the definition of $\sim$: for every pair of vertices u, v in V_i, we have $u \sim v$, and hence there is a path $u, v_2, v_3, \ldots, v_\ell, v$ from u to v in G. Since the subpaths $u, v_2, v_3, \ldots, v_j$ for every $j = 2, 3, \ldots, \ell$ are also paths, we have $u \sim v_j$ for $j = 2, 3, \ldots, \ell$, which implies that each of $v_2, v_3, \ldots, v_\ell$ are all in V_i. Thus, since G_i is the subgraph of G induced by V_i, we have that the all the edges in the path $u, v_2, v_3, \ldots, v_\ell, v$ are present in G_i, and hence the path $u, v_2, v_3, \ldots, v_\ell, v$ is a path from u to v in G_i.

That every edge $\{u, v\}$ in E is present as an edge in E_i for some i follows likewise from the definition of $\sim$. Since the path u, v is a path in G, we have $u \sim v$, and hence u and v belong to the same vertex set, V_i for some i. Since G_i is the subgraph of G induced by V_i, the edge $\{u, v\}$ is present in G_i as well.

To show uniqueness, suppose that $G_1' = (V_1', E_1'), \ldots, G_{k'} = (V_{k'}', E_{k'}')$ is another decomposition of G into connected subgraphs with disjoint vertex sets so that every edge of G is present as an edge in G_i' for some i. Define the relation $\sim'$ on the vertices of G to be $u \sim v$ if and only if u and v belong to the same V_i'.

Because G_i' is connected for every i, we have that for every pair of vertices u, v in the same V_i', there exists a path $u \to v$ in G_i'. Since G_i' is a subgraph of G, we have that there exists a path $u \to v$ in G. Thus $u \sim v$. In other words, we have that $u \sim' v$ implies $u \sim v$ (or more formally, $\sim' \subseteq \sim$).

Similarly, because every edge of G is present in some G_i', we can infer that every path $v_1, v_2, \ldots, v_{\ell+1}$ in G is a path through G_i' for some i. It follows that, whenever $u \sim v$, we have $u \sim' v$, and hence $\sim' = \sim$.

Thus we can conclude that the vertex sets $V_1', V_2', \ldots, V_{k'}'$ are the equivalence classes of $\sim$. By the uniqueness statement of Proposition 2, we can conclude that the vertex sets V_i and the vertex sets V_i' are identical up to re-indexing. □

Since the decomposition in Proposition 1 is unique, we give it a name. The graph G is said to *decompose* into its *connected components*, which are the G_i's.

A *tree* is a graph that is *connected* and contains *no cycles*. A vertex in a tree which has degree exactly 1 is said to be a *leaf*. A graph, each of whose connected components is a tree, is called a *forest*. Trees satisfy the following properties:

Proposition 3. *Every tree with at least two vertices has at least one leaf.*

Proof. Let $G = (V, E)$ be a graph with $|V| \geq 2$. Let's suppose toward contraposition that G has no leaf vertex; *i.e.*, every vertex of G has degree zero or degree at least two. We will then show that G cannot be a tree.

If G has a vertex of degree zero, but at least two vertices, then G cannot be connected, and thus cannot be a tree. Thus we can assume that every vertex of G has degree at least two. We will show that, in this case, G must contain a cycle.

Let v_1 be an arbitrary vertex of G. Since v_1 has degree at least two, it must have some neighbor v_2. Since v_2 has degree at least two, it must have some neighbor, v_3, *which is not* v_1. Continuing in this way, we can construct v_4 as a neighbor of v_3 not equal to v_2, v_5, v_6, *etc.*

Now suppose that for some i and j with $i < j$, we have $v_i = v_j$. Then if we look at the subsequence $v_i, v_{i+1}, \ldots, v_j$, we will have a cycle: it is a path by construction, satisfies the condition that $v_i = v_j$, and moreover satisfies the condition that v_k and v_{k+2} are distinct for every k. Thus, if we can just show that, for some distinct indices i and j, $v_i = v_j$, we will have shown that G contains a cycle.

This follows easily from the pigeon-hole principle. We can building the path $v_1, v_2, \ldots$ arbitrarily long; in particular, we can build it to have length $\ell = |V|$. When we do this, there are $|V| + 1$ occurrences of vertices on the path. Since there are only $|V|$ many vertices, it follows that some vertex occurs at least twice. Let $i < j$ be the two distinct positions where this vertex appears. Then $v_i = v_j$, so we are done. □

Proposition 4. *For any nonempty tree* $G = (V, E)$, $|E| = |V| - 1$.

Proof. We proceed by induction on $|V|$.

Base case: If $|V| = 1$, then there can be no edges in G, *i.e.*, $|E| = 0$.

Inductive step: Suppose that $|V| \geq 2$. Applying Proposition 3, G must have a leaf, say v.

Let $G' = (V', E')$ be the subgraph of G whose vertices are the vertices of G that are not v, and whose edges are the edges of G except the single edge on v. Every cycle in G' is also a cycle in G, so since G had no cycles, G' also has no cycles. Furthermore, it's easy to see that G' must be connected, since G was connected and we only removed a leaf. We can show this formally by observing that every path can be 'simplified' to a simple path, and the only simple paths involving a leaf must begin or end at the leaf. (We leave the complete formalism as an exercise.)

Applying our inductive hypothesis to G', we see that $|E'| = |V'| - 1$. Thus since $|V| = |V'| + 1$ and $|E| = |E'| + 1$, basic algebra then tells us that $|E| = |V| - 1$.

□

An important kind of tree is the notion of a spanning tree. 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 5. *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.[2]

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 $v_1, \ldots, v_{\ell+1}$ is a cycle in G, then we can remove the edge $\{v_1, v_\ell\}$ from G to get a subgraph G'. It is easy to see that G' must be connected; informally, it is because paths that used the $\{v_1, v_\ell\}$ edge could use the path $\{v_1, \ldots, v_\ell\}$ instead. Since G' has fewer edges 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'.

$\square$

Of course, for graphs which are not connected, we can consider partitioning it into its connected components, and then take spanning trees of each of these subgraphs. The subgraph which is the union of all these spanning trees is called a *spanning forest*. Following Proposition 5 and Proposition 1, it follows that every graph has a spanning forest.

Trees have many equivalent characterizations. Some simple ones are stated in Proposition 6:

Proposition 6. *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 4 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 4, $|E'| = |V| - 1$. 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.

[2] Fun fact: While this direction may seem to be obviously true for infinite graphs (to which our inductive argument does not apply), this proposition stated for infinite graphs is actually equivalent to the axiom of choice.

Finally, suppose that conditions (2) and (3) hold. Suppose that G has k connected components, $C_1 = (V_1, E_1), C_2 = (V_2, E_2), \ldots, C_k = (V_k, E_k)$. Since G has no cycles, none of $C_1, \ldots, C_k$ has a cycle. Since each C_i is connected, it follows that C_i is a tree for every i. By Proposition 4, we know $|E_i| = |V_i| - 1$ for every i. Since every vertex and every edge of G appears in exactly one of $C_1, \ldots C_k$, 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 $G = C_1$ is connected. $\square$

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 Traversal Algorithms

We now move on to the basic traversal algorithms for graphs. The main purpose of a graph traversal algorithm is to provide a means of enumerating all the vertices of a graph in a *purely graph-theoretic* way, in the sense that the order of enumeration should be decided only based on the "is a neighbor of" relationship, rather than on any underlying interpretation of the vertices. For instance, if the vertices of a graph are $\{1, \mathrm{cat}, ☺\}$, then the order of enumeration should depend on the edges in the graph, and not on any particular understanding of what "1", "cat", or "☺" is.

The two traversal algorithms we will cover are the classic breadth-first search and depth-first search algorithms. The two approaches can have very similar implementations, differing only on the data structure in use. However, this change of data structure turns out to drastically influence the behavior of these algorithms, and each approach will end up revealing different structural properties of the graph. We will quickly summarize the algorithms, provide pseudocode for each, and discuss some of their most generally useful properties.

2.1 BFS

Recall that breadth-first search uses a queue at its core. Its *modus operandi* is to pop the front element off the queue, add all of its unvisited neighbors to the back of the queue, and then repeat until the queue is empty. By appropriately seeding this procedure (*i.e.*, telling it where to start each time it finishes a connected component), BFS will explore every vertex in a graph. The pseudocode for this is given in Algorithm 1.

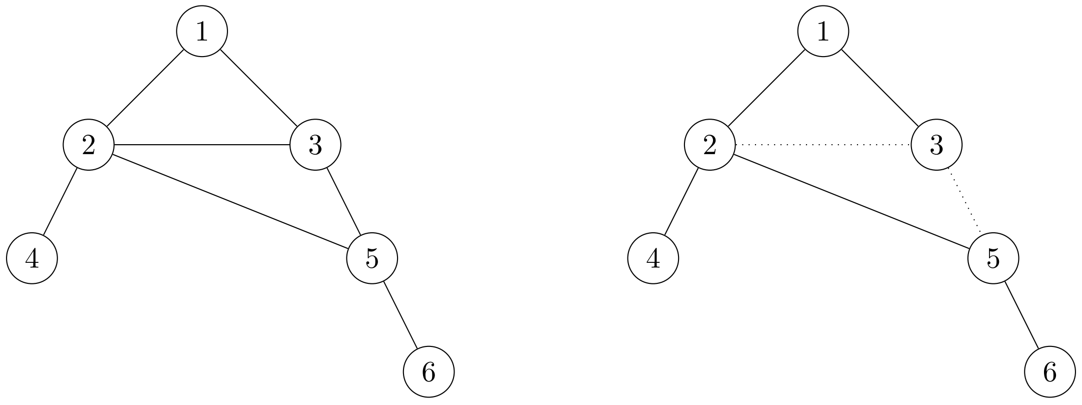
Algorithm 1

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

Output: (Nothing)

```
 1: procedure BREADTH-FIRST-SEARCH($A$)
 2:     Color ← array of size $n$ initialized to Unvisited
 3:     for $v \in V$ do
 4:         if Color[$v$] $\neq$ Unvisited then
 5:             continue
 6:         Queue ← an empty queue
 7:         Queue.PUSH($v$)
 8:         Color[$v$] ← InProgress
 9:         while ¬ Queue.EMPTY do
10:             $u$ ← Queue.POP
11:             if Color[$u$] = Complete then
12:                 continue
13:             for $u' \in A[u]$ do
14:                 if Color[$u'$] = Complete then
15:                     continue
16:                 Queue.PUSH($u'$)
17:                 Color[$u'$] ← InProgress
18:             Color[$u$] ← Complete
```

Figure 2: BFS Tree Example



(a) Input graph (b) BFS tree from Figure 2a

When given an adjacency list representation of the input graph, BFS uses time $\Theta(n+m)$ to traverse the graph.

BFS Trees One of the most generally useful aspects of breadth-first search is the spanning forest it implicitly generates on its input graph. As BFS runs, it pops an "active vertex", u, off the queue. While processing u, we may add some of its neighbors u' to the queue. The edges $\{u, u'\}$ added in this manner ultimately form the edges of a spanning forest of the input graph. A pictorial example is given in Figure 2.

The forest generated by BFS is not just any spanning forest. In fact, if we break the trees down into levels (grouping vertices of the tree by their distance from the root), then we can see that every 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.

BFS for shortest paths One use of BFS is that it can find shortest-paths in graphs. Suppose we are given a graph G, and a pair of vertices s and t of G, and we want to know the shortest path from s to t through 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 distance exactly ℓ from s in G. That it is distance at most ℓ is clear—the edges in the BFS tree are also edges in G, and t being at level ℓ means there is a path of length ℓ from s to t through the BFS tree.

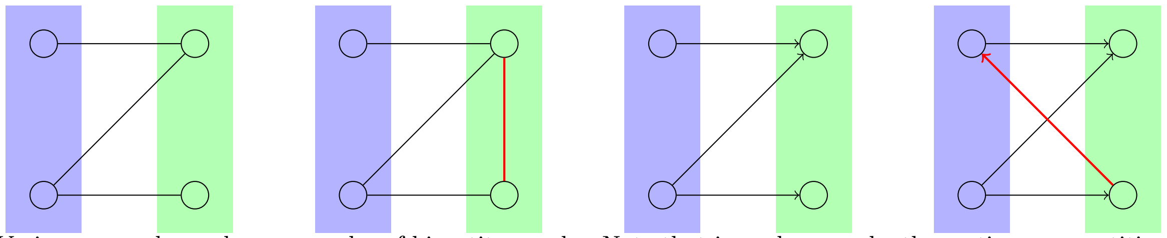
We can argue that the distance is also at least ℓ by using the structural property of BFS trees. In particular, let $s = v_1, v_2, \cdots, v_{k+1} = t$ be an arbitrary path of length k from s to t. Since each edge of G can move by at most one level through the BFS tree, it follows that the level of v_i in this path is at most $i-1$. Thus v_{k+1} is at level at most k. Since $v_{k+1} = t$, this means that $k \geq \ell$ as desired.

BFS and bipartite graphs There are many special kinds of graphs. One particular kind that will show up later 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



Various examples and nonexamples of bipartite graphs. Note that in each example, the vertices are partitioned into two parts, denoted by the blue and green shaded regions. Red edges denote edges whose presence causes the graph to be come *not* bipartite.

For most of our applications of bipartite graphs, the decomposition of the graph's vertices into two parts will be obvious. However, there are settings in which such a decomposition is less obvious, and it may not even be clear whether a graph is even bipartite in the first place. That said, it turns out that one can compute such a decomposition (or discover that none exist) in linear time.

The basic idea is to take advantage of the structure of BFS trees. In particular, let $G = (V, E)$ be a graph, and let T be a BFS tree. 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 whose height in T is odd, and let R be the vertices whose height in T is even. Since following an edge in G changes the parity of the level in T by one, 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.

2.2 DFS

Recall that depth-first search uses a stack at its core. Its *modus operandi* is to pop the front element of the stack, add all of its unvisited neighbors to the top of the stack, and then repeat until the stack is empty. By appropriately seeding this procedure (*i.e.*, telling it where to start each time

Algorithm 2

```
Input: $A[v \in V]$, an adjacency list representation of the graph $G = (V, E)$, where $n = |V|$ and
    $m = |E|$
Output: (Nothing)
 1: procedure DEPTH-FIRST-SEARCH($A$)
 2:     Color ← array of size $n$ initialized to Unvisited
 3:     for $v \in V$ do
 4:         if Color[$v$] $\neq$ Unvisited then
 5:             continue
 6:         Stack ← an empty stack
 7:         Stack.PUSH($v$)
 8:         Color[$v$] ← InProgress
 9:         while ¬ Stack.EMPTY do
10:             $u$ ← Stack.POP
11:             if Color[$u$] = Complete then
12:                 continue
13:             for $u' \in A[u]$ do
14:                 if Color[$u'$] = Complete then
15:                     continue
16:                 Stack.PUSH($u'$)
17:                 Color[$u'$] ← InProgress
18:             Color[$u$] ← Complete
```

it finishes a connected component), DFS will expore every vertex in a graph. The pseudocode for this is given in Algorithm 2. Note the substantial similarity to BFS.

The efficiency of DFS is essentially the same as that of BFS: its running time is $\Theta(n + m)$.

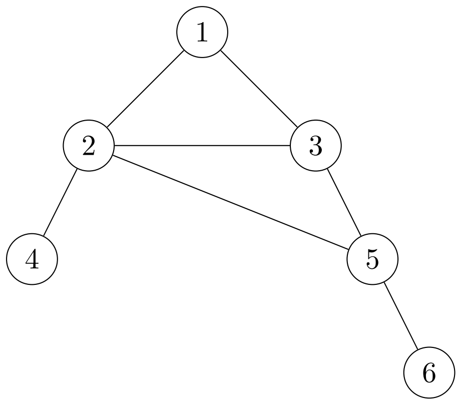
DFS Trees Similar to BFS, DFS also implicitly constructs a spanning forest of its input graph. This spanning forest is in general a bit different from the one constructed by BFS. However, it still has an extremely useful property: For every edge $\{u, v\}$ in the input graph, either u is an ancestor of v in the DFS tree, or else v is an ancestor of u in the DFS tree.

Bridge-finding in linear time via DFS trees One application of DFS trees is that of finding every bridge 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. An easy way to find all the edges in a graph is just to try removing each edge, and check to see if this increases the number of connected components. This approach has a running time of $\Theta(m \cdot (n + m))$.

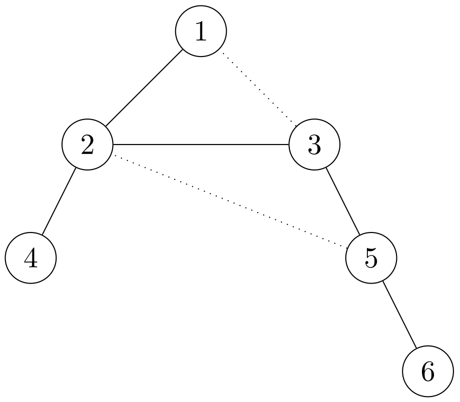
We can actually do better by using the structural properties of DFS trees. In fact, we can construct an algorithm that runs in time $\Theta(n + m)$, and is essentially a modified DFS.

The first step toward this is to give a different characterization of bridges. The characterization is as follows: Let $\{u, v\}$ be any edge in a graph G, and let $G - \{u, v\}$ denote the graph G with the edge $\{u, v\}$ removed. Then $\{u, v\}$ is a bridge if and only if there is *not* a path from u to v in $G - \{u, v\}$. Put another way, $\{u, v\}$ is a bridge if and only if *every* path from u to v uses the edge $\{u, v\}$. These two propositions are straightforward to prove; the details are left as a exercises.

Figure 4: DFS Tree Example



(a) Input graph



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

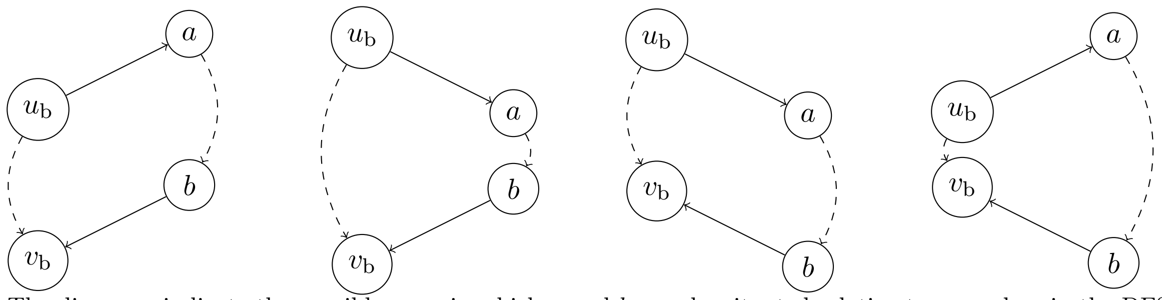
With this characterization under our belts, we can move on to connecting the notion of a bridge with DFS trees. Fix a graph G. For simplicity, assume that G is connected; the following generalizes to general graphs, but focusing on connected G helps keep the argument simple. Now fix a DFS tree T of G. We know that for any edge $\{u, v\}$ of G, either u is an ancestor of v, or vice versa.

Suppose that $\{u_\mathrm{b}, v_\mathrm{b}\}$ is a bridge in G, with u_b an ancestor of v_b. Then something interesting happens within the DFS tree T: Let $\{a, b\}$ be any edge in G, and choose a so that a is the ancestor of b in T. Then either it is the case that a and b *both* below v_b in T, or else a and b are *both* above u_b in T.

This is because the only other possible option is to have a above v_b and b below u_b; but, if this is the case, then there is a path from u_b to v_b that doesn't use the edge $\{u_\mathrm{b}, v_\mathrm{b}\}$: simply follow T from u_b to a, take the edge $\{a, b\}$, and then follow T from b to v_b. (Figure 5 gives a visualization of the different cases.) Note that this implies that $\{u_\mathrm{b}, v_\mathrm{b}\}$ will be an edge in the DFS tree T.

So now suppose we can compute the vertex $a(v)$ for each vertex v in T, where $a(v)$ is the highest vertex in T which is reachable from v or some descendant of v in one step, ignoring the

Figure 5: Bridges in DFS trees



The diagrams indicate the possible ways in which a and b may be situated relative to u_b and v_b in the DFS tree; being positioned closer to the top of the page denotes being "higher" or "more ancestral" in the DFS tree. Solid arrows indicate tree edges, and dashed lines indicate edges that may or may not be tree edges. Note that, in any configuration, the path $u_\mathrm{b} \to a \to b \to v_\mathrm{b}$ does *not* need to use the edge $\{u_\mathrm{b}, v_\mathrm{b}\}$.

edge between v and its parent in T. For example, in Figure 4, $a(1) = 1$ (1 is the root), $a(2) = 1$ (via $2 \to 3 \to 1$), $a(3) = 1$, $a(4) = 4$, $a(5) = 2$, and $a(6) = 6$.

Considering the bridge $\{u_b, v_b\}$, our earlier observation says that $a(v_b)$ can never be higher in T than v_b, and thus $a(v_b) = v_b$. Conversely, if $\{u, v\}$ is an edge with u above v in T, then $a(v)$ is v or some ancestor of v, and if $a(v)$ is v itself, then it follows that $\{u, v\}$ is a bridge. Thus, finding bridges can be done in linear time once we compute $a(v)$ for every vertex v.

We can compute $a(v)$ for each vertex v in linear time from the DFS tree. First, note that we can easily compute the height of each vertex in T, which we will denote $h(v)$, in linear time. Then, for leaves v of T, $a(v)$ is simply the most ancestral vertex reachable from v in one step; v has no other descendants. This can be computed by just enumerating the edges $\{u, v\}$ of G incident on v, and computing which has the smallest value of $h(u)$. For non-leaf vertices v of T, $a(v)$ is either a vertex u such that $\{u, v\}$ is an edge of G, or else there is some descendant v' of v in T so that $a(v') = u$. In this second case, we actually only need to check v' so that $\{v, v'\}$ is an edge in T. Thus we can just enumerate all the tree-edges $\{v, v'\}$ of v (excluding the one to v's parent in T), and check $h(a(v))$, and enumerate all the non-tree edges $\{v, u\}$ from G, and check $h(u)$. Whichever vertex, v' or u, has either $h(a(v'))$ or $h(u)$ smallest will be the $a(v)$. This check also takes linear time in the number of edges incident to v.

Altogether, the above process takes time linear in the size of G. It is possible to compute the values $a(\cdot)$ and $h(\cdot)$ on the fly during DFS, which obviates the need to explicitly represent the DFS tree.

3 Directed Graphs

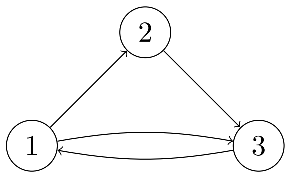
In our discussion of graphs so far, we have focused entirely on *undirected* graphs. The qualifier undirected refers to the fact that the edges are unordered pairs—there is no sense of direction associated with them.

However, assigning a direction to an edge is sometimes extremely useful. For this, we define the notion of a *directed graph*[3] 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 6.

The various topological notions of graphs change slightly when we want to discuss digraphs. The most fundamental change is that the "is a neighbor of" relation fails to be symmetric, as we

[3] We will follow the convention that a "graph" (without any qualifiers) will be undirected, and use specify "directed graph" or "digraph" when we mean to use directed graphs.

Figure 6: 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.

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 (leaving) u.

Another consequence of this loss of symmetry is that paths only follow edges in the forward direction. Furthermore, paths cannot backtrack in the intuitive sense that we wished to rule out when we defined a cycle. Thus a cycle in a digraph is just a path $v_1, v_2, \ldots, v_{\ell+1}$ which has $v_1 = v_{\ell+1}$ and no other conditions.

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 weird 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 mainly to help distinguish the slight 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.[4] However, the main difference is that, in the case of directed graphs, there could still be edges from one component to another. These edges have to satisfy some additional structural properties, however, which we will mention when we cover the topic of directed acyclic graphs in Section 5.

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*.

4 Traversal algorithms in digraphs

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. We can still use BFS trees to find shortest paths in

[4] The proof of this is even essentially same: define $u \sim v$ if and only if u and v are 'connected', *i.e.*, there is a path from u to v *and* a path from v to u.

directed graphs, but this loss of structure does break our bipartiteness-detection algorithm. In this latter case, however, we can regard the directed graph as an undirected graph by *forgetting* the orientations of the edges. Once we've done this, the resulting undirected graph is bipartite if and only if the initial directed graph is bipartite, so we can use our procedure on the undirected graph.

For DFS trees, the loss of structure is more profound. Before, non-tree edges could only connect to ancestral vertices, but now non-tree edges can point to any vertex appearing earlier in the order of vertices considered by DFS.

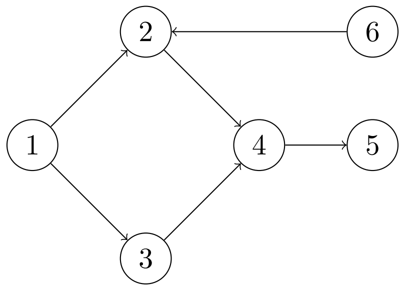
However, DFS is still useful in the world of directed graphs. One can use DFS to put together a linear-time algorithm for computing the strongly connected components of any digraph. The algorithm is nontrivial to devise; however, we do not cover it here.

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

5 DAGs and Topological Sort

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 many important applications throughout computer science (and mathematics in general), and we can only touch on a few throughout this course. Figure 8 gives an example of a DAG:

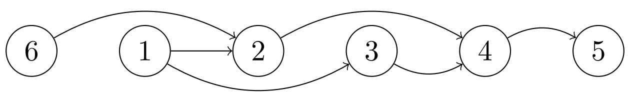
Figure 7: A directed acyclic graph



One of the most important properties of DAGs is that they can be *topologically sorted*, *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 **??**, which is a linear order of the vertices from Figure 8.

Figure 8: A linear ordering of the DAG in Figure 8



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 8. Another linear order would be "1, 6, 2, 3, 4, 5", or "1, 3, 6, 2, 4, 5".

In fact, DAGs are precisely the types of graphs that can be linearly sorted:

Proposition 7. *Let $G = (V, E)$ be a directed graph. Then G can be linearly sorted if and only if G is a DAG.*

Proof. $\Rightarrow$ This direction is easy; we can prove it 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.

$\Leftarrow$ This direction is a little trickier, but still fairly easy. The proof is by induction:

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

$|V| = k + 1$ Since G is a DAG, we can find a vertex v of G which has zero out-going edges: If we pick v from G arbitrarily and it has an out-going edge, we just follow the edge to pick a new v, and repeat this indefinitely. If this procedure never stops, then G would have a cycle, so it must have stopped at some point, and thus there is a vertex of G with out-degree 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. Then suppose we add v to the end of this; *i.e.*, we set $v_{k+1} = v_k$. Then the sequence $v_1, \ldots, v_k, v_{k+1}$ 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 into v, and hence cannot cause trouble when v is at the end of the sequence.

□

In fact, the proof of the $\Leftarrow$ direction of Proposition 7 tells us how to compute the topological ordering of a given DAG in linear time! With a simple tweak to DFS, we can compute the topological order starting with the last element and working forward.

However, there is a different linear-time algorithm for computing the topological sort. The idea here is to start with vertices with *in*-degree zero, and place them at the front of the order. As we process these, we think of removing them from the graph as decreasing the in-degree of all of their neighbors. Then, as we proceed, vertices that originally had positive in-degree change to having in-degree zero, in which case we add them to the queue of in-degree-zero vertices to be added to the order. A pseudocode implementation of this is given in Algorithm 3. We leave its correctness as an exercise.

Algorithm 3

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

```
1: procedure TOPOLOGICAL-SORT($A$)
2:     $D[v \in V] \leftarrow$ array of integers, initialized to zero
3:     for $v \in V$ do
4:         for $v' \in A[v]$ do
5:             $D[v'] \leftarrow D[v'] + 1$
6:     $Q \leftarrow$ empty queue of vertices
7:     for $v \in V$ do
8:         if $D[v] = 0$ then
9:             $Q$.PUSH($v$)
10:    $k \leftarrow 0$
11:    while $\neg$ $Q$.EMPTY do
12:        $k \leftarrow k + 1$
13:        $v_k \leftarrow Q$.POP
14:        for $v' \in A[v_k]$ do
15:            $D[v'] \leftarrow D[v'] - 1$
16:            if $D[v'] = 0$ then
17:                $Q$.PUSH($v'$)
18:    return $v_1, v_2, \ldots, v_k$        ▷ If $G$ were not acyclic, then we would have $k < n$ here
```