**CS 577: Introduction to Algorithms** Fall 2015

# Graph Primitives

Instructor: Dieter van Melkebeek Scribe: Andrew Morgan

DRAFT

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

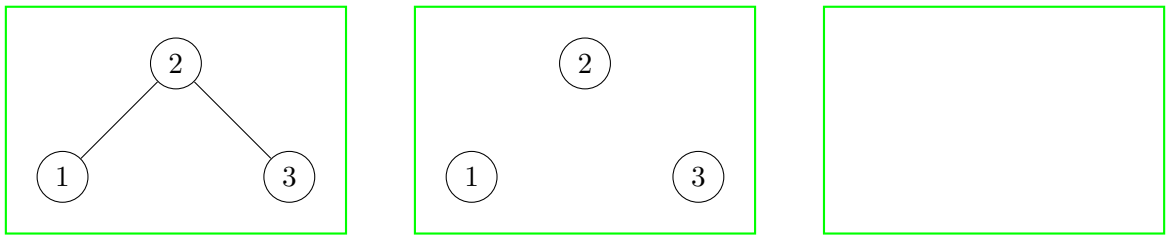
Most of this section should be a review, so these definitions are presented pretty quickly and with few examples.

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.

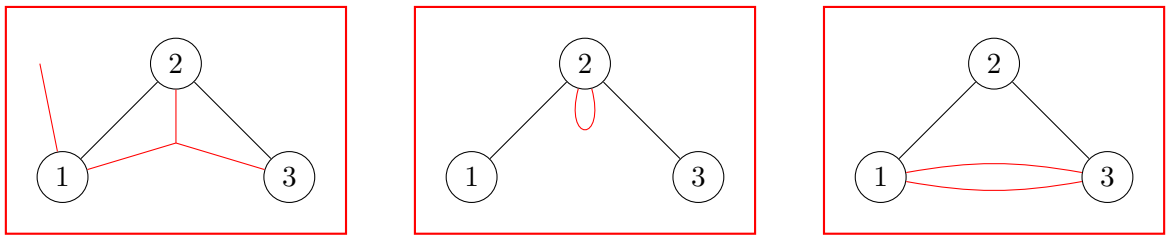
These kinds of graphs are sometimes referred to as *undirected graphs*, because the edges have no meaning of direction associated with them. *i.e.*, we do not differentiate between "an edge on $u$ and $v$" and "an edge on $v$ and $u$". However, differentiating these two cases is sometimes useful. For 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 2. 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.

[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

Our next definition is that of a subgraph. Like subsets, subgraphs $G$ of a graph $G'$ are 'a part of' $G'$, in the sense that structure of $G$ is also present in $G'$. 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.

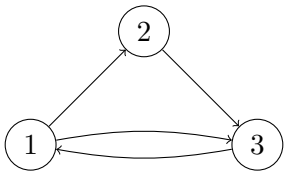
Fix a graph $G' = (V', E')$. For a subset of vertices $V \subseteq V'$, let $E$ be the set of all edges in $E'$ for which both end points belong to $V$. We say that the graph $G = (V, E)$ is *induced by* the vertex set $V$. Equivalently we say, $G$ is the *induced subgraph* of $G'$ by $V$.

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.

We say that a vertex $u$ is a *neighbor* of a vertex $v$ when there is an edge on $u$ and $v$. For directed graphs, we require there to be an edge from $v$ to $u$. That is, edges point *to* neighbors. Thus for undirected graphs the "is a neighbor of" relationship is symmetric (if $u$ is a neighbor of $v$, then $v$ is a neighbor of $u$), but for directed graphs, this need not be the case. 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 $v_1, v_2, \ldots, v_{\ell+1}$ of vertices such that $v_{i+1}$ is a neighbor of $v_i$ for every $i = 1 \ldots \ell$. The *length* of the path is the value $\ell$, which counts the number of edges along the path.

Figure 2: 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.

We say the path goes *from* $v_1$ *to* $v_{\ell+1}$. A *simple* path is a path $v_1, v_2, \ldots, v_{\ell+1}$ 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 '$v_1$'.

A *cycle* in a *directed* graph is a path $v_1, v_2, \ldots, v_{\ell+1}$ for which $v_1 = v_{\ell+1}$ and for which $\ell \geq 1$. The condition $v_1 = v_{\ell+1}$ means that the path starts and ends at the same vertex, and $\ell \geq 1$ means the path is nontrivial. In undirected graphs, we additionally require cycles to not backtrack, *i.e.*, we require the path to satisfy $v_i \neq v_{i+2}$ for every $i = 1 \ldots \ell - 1$. For instance $1, 2, 1$ is *not* a cycle in the graph in Figure 1a, even though it is a path. However, $3, 1, 3$ *is* a cycle in the graph in Figure 2, because the graph is directed. A *simple* cycle is a cycle $v_1, v_2, \ldots, v_{\ell+1}$ for which $v_1, v_2, \ldots v_\ell$ is a simple path; *i.e.*, the only repeated vertices are $v_1 = v_{\ell+1}$.

An undirected graph is said to be *connected* if for every pair $u, v$ of vertices, there is a path from $u$ to $v$. For directed graphs, we say the graph is *strongly connected* if it satisfies this property. This is mainly to help distinguish the fact that, in undirected graphs, the existence of a path from $u$ to $v$ implies the existence of a path from $v$ to $u$, while this is not the case for directed graphs.

Not every undirected graph is connected, but every such graph can be regarded as a union of connected graphs:

**Proposition 1.** *Every undirected graph $G = (V, E)$ has, for some number $k$, $k$ connected subgraphs $G_i = (V_i, E_i)$ for $i = 1, \ldots, k$, 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.*

*Proof.* A formal proof would use the notion of an equivalence relation, but we haven't defined those, so we will keep our proof informal.

For each vertex $v$, let $V_v$ denote the set of vertices which are reachable from $v$, and let $G_v$ be the subgraph of $G$ induced by $V_v$. Some of these graphs will be identical—for instance if there is a path from $u$ to $v$, then $G_u = G_v$. But sometimes they will not—if there is no path from $u$ to $v$, then $V_u$ and $V_v$ are actually disjoint. In any case, if we choose our $G_i$'s to be the distinct graphs $G_v$ that show up, then they will satisfy the properties above.

To show uniqueness, we need to show that, given two decompositions satisfying the above conditions, they are the same decomposition. Toward this end, let $G_1, G_2, \ldots, G_k$ and $H_1, \ldots, H_\ell$ be two decompositions of $G$ into connected graphs with no crossing edges.

Let $u$ and $v$ be two vertices in $G_i$ for some $i$; then $u$ and $v$ are connected by some path through $G$, so they must also belong to the same $H_j$. In other words, for each component $G_i$, it must be that $G_i \subseteq H_j$ for some component $H_j$. Reversing this argument shows that, for each choice of $H_j$, there is some component $G_i$ so that $H_j \subseteq G_i$. Thus we have a correspondence between the components in the first decomposition and the components in the second decomposition, which shows that they are the same decomposition up to re-indexing. □

Since the above decomposition 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 an undirected 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 2.** *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 3.** *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 2, $G$ 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 an undirected 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 $G$.[2]

[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.

**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 $G'$.

□

Of course, for graphs which are not connected, we can consider partitioning it into its connected components, and taking spanning trees of each of these subgraphs. The subgraph which is the union of all these spanning trees would be called a *spanning forest*. Following Proposition 4 and

**Proposition 5.** *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 3 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 3, $|E'| = |V| - 1$. 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, $C_1 = (V_1, E_1), C_2 = (V_2, E_2), \ldots, C_k = (V_k, E_k)$. 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 3, 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. □

**Graphs in Algorithms** In the context of computer science, there are a few 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. Often the letter $n$ denotes the number of vertices in a graph, and the letter $m$ denotes the number of edges. Since undirected 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, in which $m = O(n)$).

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, and there is a 1 in the $u$-th row and $v$-th column when $u$ is a neighbor of $v$, and there is a 0 otherwise. An adjacency list is an array of lists: each vertex $v$ corresponds to a position in this array, and the associated list contains the neighbors of $v$.

Adjacency matrices are generally much 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.

## 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 $\{1, \text{cat}, ☺\}$, 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. As these algorithms should be review, 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 ∈ V do
4:          if Color[v] ≠ 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′ ∈ A[u] do
14:                 if Color[u′] = Complete then
15:                     continue
16:                 Queue.PUSH(u′)
17:                 Color[u′] ← InProgress
18:             Color[u] ← Complete
```

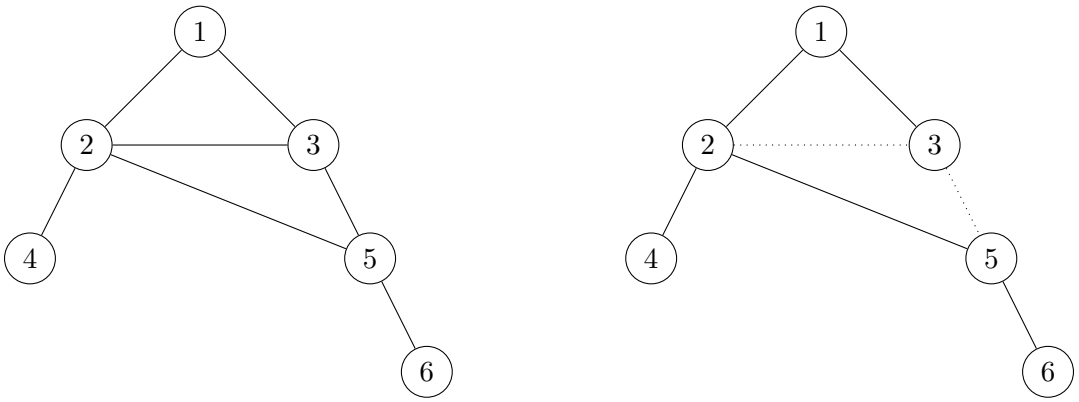
When given an adjacency list representation of the input graph, BFS uses time $\Theta(n + m)$ to traverse the graph.

**BFS Trees** When considering undirected graphs, 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 3.

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 $\ell$ (starting with $s$ at level zero). The claim is that $t$ must be distance exactly $\ell$ from $s$ in $G$. That it is distance at most $\ell$ is clear—the edges in the BFS tree are also edges in $G$, and $t$

Figure 3: BFS Tree Example




(a) Input graph

(b) BFS tree from Figure 3a

being at level $\ell$ means there is a path of length $\ell$ from $s$ to $t$ through the BFS tree.

We can argue that the distance is also at least $\ell$ 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.

## 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 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: Suppose the input graph is an undirected graph. Then, 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 an undirected graph. A *bridge* in an undirected 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.

**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 ∈ V do
4:          if Color[v] ≠ 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′ ∈ A[u] do
14:                 if Color[u′] = Complete then
15:                     continue
16:                 Stack.PUSH(u′)
17:                 Color[u′] ← InProgress
18:             Color[u] ← Complete
```

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 an undirected 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.

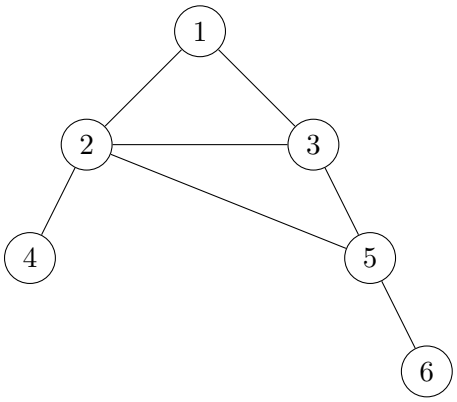
With this characterization under our belts, we can move on to connecting the notion of a bridge with DFS trees. Fix an undirected graph $G$. For simplicity, assume that $G$ is connected; the following generalizes to general graphs, but focusing on connected $G$ helps keep the argument simple. Now fix 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 $\{u_b, v_b\}$ 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 $T$.

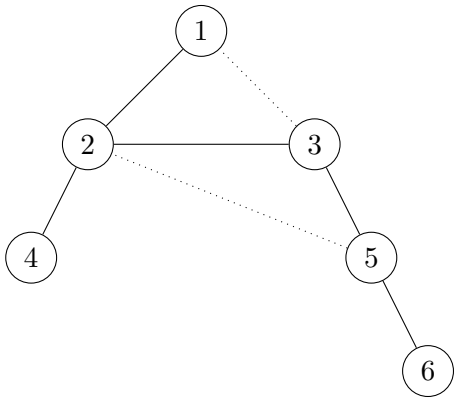
This is because the only other possible option is to have $a$ above $v_b$ and $b$ below $u_b$; but, if this is the case, then there is a path from $u_b$ to $v_b$ that doesn't use the edge $\{u_b, v_b\}$: simply follow $T$ from $u_b$ to $a$, take the edge $\{a, b\}$, and then follow $T$ from $b$ to $v_b$. (Figure 5 gives a visualization of the different cases.) Note that this implies that $\{u_b, v_b\}$ 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 edge between $v$ and its parent in $T$. For example, in Figure 4, $a(1) = 1$ (1 is the root), $a(2) = 1$

Figure 4: DFS Tree Example

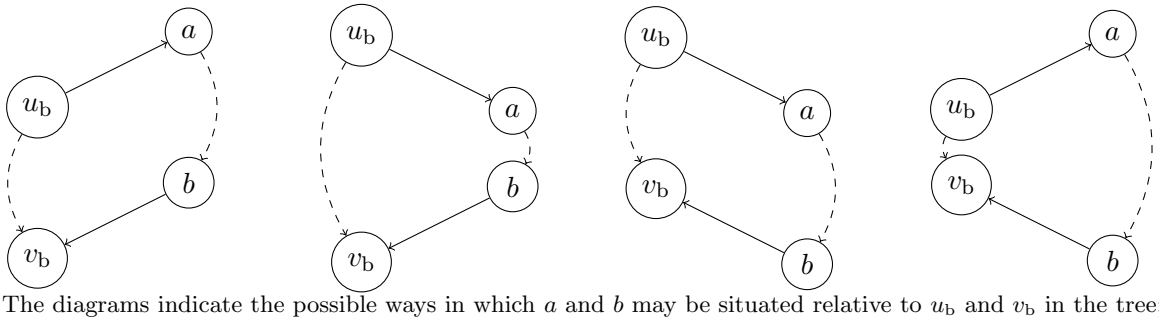



(a) Input graph




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

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 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 $u_b \to a \to b \to v_b$ does *not* need to use the edge $\{u_b, v_b\}$.

(via $2 \to 3 \to 1$), $a(3 = 1$, $a(4) = 4$, $a(5) = 2$, and $a(6) = 6$.

Considering the bridge $\{u_\text{b}, v_\text{b}\}$, our earlier observation says that $a(v_\text{b})$ can never be higher in $T$ than $v_\text{b}$, and thus $a(v_\text{b}) = v_\text{b}$. 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 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 (possibly directed) 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:

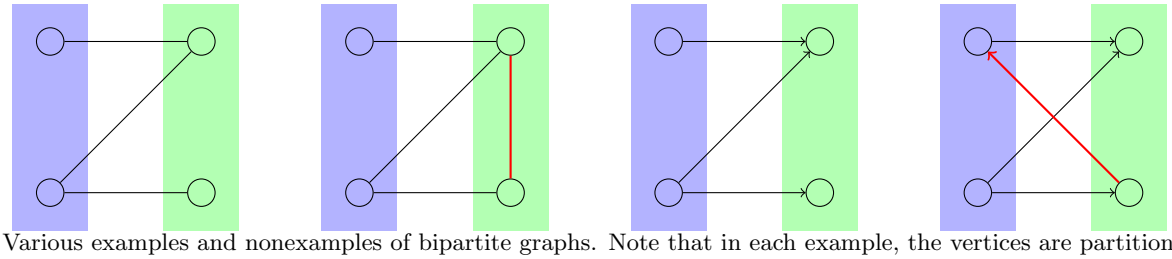
1. $L \cup R = V$. Every vertex in $V$ is in $L$ or in $R$.
2. $L \cap R = \varnothing$. No vertex in $V$ is in both $L$ and $R$.
3. - In undirected graphs, for every edge $\{u, v\}$ in $E$, either $u$ is not in $L$ or $v$ is not in $L$, and, either $u$ is not in $R$ or $v$ is not in $R$.
   - In directed graphs, we require the edges to go from $L$ to $R$. *i.e.*, for every edge $(u, v)$ in $E$, we insist that $u \in L$ and $v \in R$.

A visual depiction of bipartite graphs is given in Figure 6.

**Detecting bipartiteness** For most of our applications of bipartite graphs, the decomposition of the graph's vertices into two parts will be obvious. However, 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.

For undirected graphs, the basic idea is to take advantage of the structure of BFS trees. In particular, let $G = (V, E)$ be an undirected graph, and let $T$ be its BFS tree. Suppose that every edge of $G$ is between two *different* levels of $T$. Then we can let $L$ be the vertices whose height in $T$

Figure 6: 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.

is odd, and let $R$ be the vertices whose height in $T$ is even. It's easy to see 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.

In the case of directed graphs, a bipartite partitioning can also be computed in linear time if it exists, and the construction can be made to detect non-bipartite graphs. We leave the details as an exercise.

# 4 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.

One of the most important properties of DAGs is that they can be *topologically sorted*, or *linearly sorted*. 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. In fact, DAGs are precisely the types of graphs that can be linearly sorted:

**Proposition 6.** *Let $G = (V, E)$ be a directed graph. Then $G$ has a 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 $G$ cannot be linearly ordered.

$\Leftarrow$ This direction is a little tricker, 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 the procedure never stops, then $G$ would have a cycle; so it must have stopped at some point.

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 6 tells us how to compute the topological ordering given a DAG! A simple tweak can be made to DFS which computes this ordering in linear time.