Design and Analysis of Algorithms
Section 0101

Ekesh Kumar*

February 13, 2020

These are my course notes for CMSC 451: Design and Analysis of Algorithms, taught by Professor Clyde Kruskal. Please e-mail corrections to ekumar1@terpmail.umd.edu

Contents

*Email: ekumar1@terpmail.umd.edu

§1 Tuesday, January 28, 2020

§1.1 Introduction

This is CMSC 451: Design and Analysis of Algorithms. We will cover graphs, greedy algorithms, divide and conquer algorithms, dynamic programming, network flows, NP-completeness, and approximation algorithms.

- Homeworks are due every other Friday or so; NP-homeworks are typically due every other Wednesday.
- There is a 25% penalty on late homeworks, and there's one get-out-of-jail free card for each type of homework.

§1.2 Stable Marriage Problem

As an introduction to this course, we'll discuss the **stable marriage problem**, which is stated as follows:

> Given a set of n men and n women, match each man with a woman in such a way that the matching is *stable*.

What do we mean when we call a matching is "stable"? We call a matching *unstable* if there exists some man M who prefers a woman W over the woman he is married to, and W also prefers M over the man she is currently married to.

In order to better understand the problem, let's look at the $n = 2$ case. Call the two men M_1 and M_2, and call the two women W_1 and W_2.

- First suppose M_1 prefers W_1 over W_2 and W_1 prefers M_1 over M_2. Also, suppose that M_2 prefers W_2 over W_1 and W_2 prefers M_2, then
- If both W_1 and W_2 prefer M_1 over M_2, and both M_1 and M_2 prefer W_1 over W_2, then it's still easy to see what will happen: M_i will always match with W_i.
- Now let's say M_1 prefers W_1 to W_2, M_2 prefers W_2 to W_1, W_1 prefers M_2 to M_1, and W_2 prefers M_1 to M_2. In this case, the two men rank different women first, and the two women rank different men first. However, the men's preferences "clash" with the women's preferences. One solution to this problem is to match M_1 with W_1 and M_2 with W_2. This is stable since both men get their top preference even though the two women are unhappy.

The solution to the problem starts to get a lot more complicated when the people's preferences do not exhibit any pattern. So how do we solve this problem in the general case? We can use the **Gale-Shapley algorithm**. Before discussing this algorithm, however, we can make the following observations about this problem:

- Each of the n men and M woman are initially unmarried. If an unmarried man M chooses the woman W who is ranked highest on their list, then we cannot immediately conclude whether we can match M and w in our final matching.This is clearly the case since if we later find out about some other man M_2 who prefers W over any other woman, W may choose M_2 if she likes him more than M. However, we cannot immediately rule out M being matched to W either since a man like M_2 may not ever come.
- Just because everyone isn't happy doesn't mean a matching isn't stable. Some people might be unhappy, but there might not be anything they can do about it (if nobody wants to switch).

Moreover, we introduce the notion of a man *proposing* to a woman, which a woman can either accept or reject. If she is already engaged and accepts a proposal, then her existing engagement breaks off (the previous man becomes unengaged).

Now that we've introduced these basic ideas, we can now present the algorithm:

```
# Input:  A list of n men and n women to be matched.

# Output:  A valid stable matching.

stable_matching {
   set each man and each woman to "free"
   while there exists a man m who still has a woman w to propose to {
     let w be the highest ranked woman m hasn't proposed to.

     if w is free {
       (m, w) become engaged
     } else {
       let m' be the man w is currently engaged to.
       if w prefers m' to m {
         (m', w) remain engaged.
       } else {
         (m, w) become engaged and m' loses his partner.
       }
     }
   }
}
```

Proposition 1.1
The Gale-Shapley algorithm terminates in $\mathcal{O}(n^2)$ time.

Proof. In the worst case, n men end up proposing to n women. The act of proposing to another person is a constant-time operation. Thus, the $\mathcal{O}(n^2)$ runtime is clear. □

§2 Thursday, January 30, 2020

§2.1 Optimality and Correctness of Gale-Shapley

Last time, we introduced the Gale-Shapley algorithm to find a stable matching. Today, we'll prove that the algorithm is correct (i.e. it never produces an unstable matching), and it is optimal for men (i.e. the men always end up for their preferred choice).

First, we'll show that the algorithm is correct:

Proposition 2.1
The matching generated by the Gale-Shapley algorithm is never an unstable matching.

Proof. Suppose, for the sake of contradiction, that m and w prefer each other over their current partner in the matching generated by the Gale-Shapley algorithm. This can happen either if m never proposed to w, or if m proposed to w and w rejected m. In the former case, m must prefer his partner to w, which implies that m and w do not form an unstable pair. In the latter case, w prefers her partner to m, which also implies m and w don't form an unstable pair. Thus, we arrive at a contradiction. □

Next, we'll prove that the algorithm is optimal for men. However, before presenting the proof, observe that it is not too hard to see intuitively that the algorithm "favors" the men. Since the men are doing all of the proposing and the women can only do the deciding, it turns out that the men always ends up with their most preferred choice (as long as the matching remains stable).

Proposition 2.2
The matching generated by the Gale-Shapley algorithm gives men their most preferred woman possible without contradicting stability.

Proof. To see why this is true, let A be the matching generated by the men-proposing algorithm, and suppose there exists some other matching B that is better for at least one man, say m_0. If m_0 is matched in B to w_1 which he prefers to his match in A, then in A, m_0 must have proposed to w_1 and w_1 must have rejected him. This can only happen if w_1 rejected him in favor of some other man — call him m_2. This means that in B, w_1 is matched to m_0 but she prefers m_2 to m_0. Since B is stable, m_2 must be matched to some woman that he prefers to w_1; say w_3. This means that in A, m_2 proposed to w_3 before proposing to w_1, and this means that w_3 rejected him. Since we can perform similar considerations, we end up tracing a "cycle of rejections" due to the finiteness of the sets A and B. □

§3 Tuesday, February 4, 2020

Today, we'll recap graph terminology and elementary graph algorithms.

§3.1 Graph Terminology

Definition 3.1. A **graph** $G = (V, E)$ is defined by a set of vertices V and a set of edges E.

The number of vertices in the graph, $|V|$, is the **order** of the graph, and the number of edges in the graph, $|E|$, is the **size** of the graph. Typically, we reserve the letter n for the order of a graph, and we reserve m for the size of a graph.

Definition 3.2. We say a graph is **directed** if its edges can only be traversed in one direction. Otherwise, we say the graph is **undirected**.

Definition 3.3. A graph is called **simple** if it's an undirected graph without any loops (edges that start and end at the same vertex).

Definition 3.4. A graph is **connected** if for every pair of vertices u, v, there exists a path between u and v.

§3.2 Graph Representations

There are two primary ways in which we can represent graphs: **adjacency matrices** and **adjacency lists**.

An adjacency matrix is an $n \times n$ matrix `A` in which `A[u][v]` is equal to 1 if the edge (u, v) exists in the graph; otherwise, `A[u][v]` is equal to 0. Note that the adjacency matrix is symmetric if and only if the graph is undirected.

On the other hand, an adjacency list is a list of $|V|$ lists, one for each vertex. For each vertex $u \in V$, the adjacency list `Adj[u]` contains all vertices v for which there exists an edge (u, v) in E. In other words, `Adj[u]` contains all of the vertices adjacent to u in G.
Each graph representation has its advantages and disadvantages in terms of runtime. This is summarized by the table below.

	Adjacency List	Adjacency Matrix
Storage	$\mathcal{O}(n+m)$	$\mathcal{O}(n^2)$
Add vertex	$\mathcal{O}(1)$	$\mathcal{O}(n^2)$
Add edge	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Remove vertex	$\mathcal{O}(n+m)$	$\mathcal{O}(n^2)$
Remove edge	$\mathcal{O}(m)$	$\mathcal{O}(1)$

Figure 1: Adjacency Matrix vs Adjacency List

An explanation of these runtimes are provided below:

- An adjacency list requires $\mathcal{O}(n+m)$ since there are n lists inside of the adjacency list. Now for each vertex v_i, there are $\deg(v_i)$ vertices in the i^{th} adjacency list. Since $\sum_i \deg(v_i) = \mathcal{O}(m)$, we conclude that the adjacency list representation of a graph requires $\mathcal{O}(n+m)$ space. On the other hand, the adjacency matrix representation of a graph requires $\mathcal{O}(n^2)$ space since we are storing an $n \times n$ matrix.

- We can add a vertex in constant time in an adjacency list by simply inserting a new list into the adjacency list. On the other hand, to insert a new vertex in an adjacency matrix, we need to increase the dimensions of the adjacency matrix from $n \times n$ to $(n+1) \times (n+1)$. This requires $\mathcal{O}(n^2)$ time since we need to copy over the old matrix to a new matrix.

- We can insert an edge (u, v) into an adjacency list in constant time by simply appending v to the end of u's adjacency list (and u to the end of v's adjacency list if the graph is undirected). Similarly, we can insert an edge in an adjacency matrix in constant time by setting `A[u][v]` to 1 (and also seting `A[v][u]` to 1 if the graph is undirected).

- Removing a vertex requires $\mathcal{O}(n+m)$ time in an adjacency list since we need to traverse the entire adjacency list and remove any incoming our outgoing edges to the vertex being removed. Similarly, this operation takes $\mathcal{O}(n^2)$ time in an adjacency matrix since we need to traverse the entire matrix to remove incoming and outgoing edges.

- Removing an edge (u, v) requires $\mathcal{O}(m)$ time in an adjacency matrix since we only need to search the adjacency lists of u and v (in the worst case, these vertices have all m edges in their adjacency list). On the other hand, this operation takes constant time in an adjacency matrix since we're just setting `A[u][v]` to 0.

§3.3 Graph Traversal

Before discussing recapping the two primary types of graph traversal, we will introduce some more terminology.

Definition 3.5. A **connected component** of a graph is a maximially connected subgraph of G. Each vertex belongs to one connected component as does each edge.

There are two primary ways in which we can traverse graphs: using **breadth-first search** or **depth-first search**. These two methods of graph traversal are very similar, and they allow us to explore every vertex in a connected components of a graph.

1. Breadth-first search starts at some source vertex v and all vertices with distance k away from v before visiting vertices with distance $k+1$ from v. This algorithm is typically implemented using a queue, and it can be used to find the shortest path (measured by the number of edges) from the source vertex.

2. Depth-first search starts from a source vertex and keeps on going outward until we cannot proceed any further. We must subsequently backtrack and begin performing the depth-first search algorithm again. This algorithm is typically implemented using a stack, whether it be the data structure or the function call stack.

Both of these algorithms run in $\mathcal{O}(n^2)$ time on an adjacency matrix and $\mathcal{O}(n+m)$ time on an adjacency list.

Since breadth-first search and depth-first search are guaranteed to visit all of the vertices in the same connected component as the starting vertex, we can easily write an algorithm that counts the number of connected components in a graph.

Some C++ code is provided below.

```
/* visited[] is a global Boolean array. */
/* AdjList is a global vector of vectors. */
void dfs(int v) {
    visited[v] = true;
    for (int i = 0; i < AdjList[v].size(); i++) {
        int u = AdjList[v][i];
        if (!visited[u]) {
            dfs(v);
        }
    }
}

int main(void) {
    /* Assume AdjList and other variables have been declared. */
    int numCC = 0;
    for (int i = 0; i < num_vertices; i++) {
        if (!visited[i]) {
            numCC = numCC + 1;
            dfs(i);
        }
    }
}
```

§4 Thursday, February 6, 2020

Today, we'll discuss algorithms to find articulation points and biconnected components.

§4.1 Articulation Points

Definition 4.1. An **articulation point** or **cut vertex** is a vertex in a graph $G = (V, E)$ whose removal (along with any incident edges) would disconnect G.

Definition 4.2. A graph is said to be **biconnected** if the graph not have any articulation points.

For example, consider the following graph:

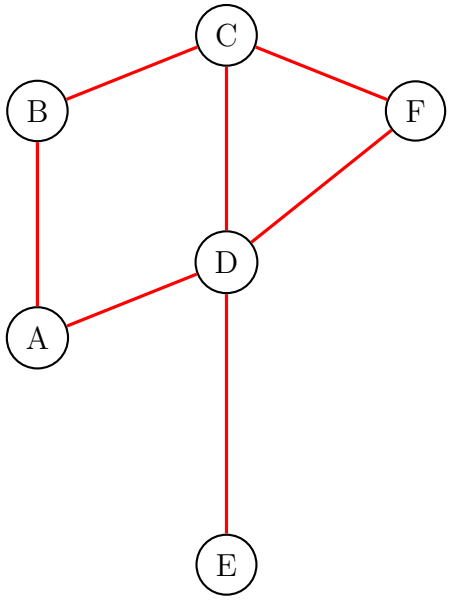

Figure 2: A Graph with an Articulation Point

In the diagram above, Vertex D is an articulation point. To see why, note that if we were to remove Vertex D (and any incident edges to D) from the graph, then we would end up with two connected components: the first component would contain the vertices A, B, C, and F, whereas the second component would only contain the vertex E.

Why are articulation points important? One example in which searching for articulation points is important is in the study of networks. In a network modeled by a graph, an articulation point represents a vulnerability: it is a single point whose failure would split the network into two or more components (preventing communication between the nodes in different networks).

How do we find an articulation points? The brute force algorithm is as follows:

1. Run an $\mathcal{O}(V+E)$ depth-first search or breadth-first search to count the number of connected components in the original graph $G = (V, E)$.

2. For each vertex $v \in V$, remove v from G, and remove any of v's incident edges. Run an $\mathcal{O}(V + E)$ depth-first search or breadth-first search again, and check if the number of connected components increases. If so, then v is an articulation point. Restore v and any of its incident edges.

This naive algorithm calls the depth-first search or breadth-first search algorithm $\mathcal{O}(V)$ times. Hence, it runs in $\mathcal{O}(V \times (V + E)) = \mathcal{O}(V^2 + VE)$ time.

While this algorithm *works*, it is not as efficient as we can get. We will now describe a linear-time algorithm that runs the depth-first search algorithm just *once* to identify all articulation points and bridges. This algorithm is often accredited to Hopcraft and Tarjan.

In this modified depth-first search, we will now maintain two numbers for each vertex v: `dfs_num(v)` and `dfs_low(v)`. The quantity `dfs_num(v)` represents a label that we will assign to nodes in an increasing fashion. For instance, the vertex from which we call depth-first search would have a `dfs_num` of 0. The subsequent vertex we visit would be assigned a `dfs_num` of 1, and so on.

On the other hand, the quantity `dfs_low(v)`, also known as the **low-link value** of the vertex v, represents the smallest `dfs_num` reachable from that node while performing a depth-first (including itself).

Here's an example. Consider the following directed graph:

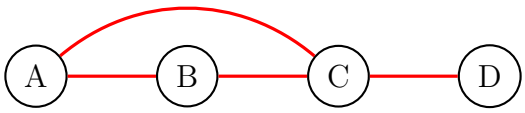

Figure 3: Articulation Point Example

Suppose we perform a depth-first search starting at Vertex A.

- Vertex A will be assigned a `dfs_num` of 0 since this is the first vertex that we're visiting. Moreover, 0 is the smallest `dfs_num` that is reachable from A (all other vertices have their `dfs_num` set to `nil` or `INFINITY`). Hence, we set `dfs_num(A) = 0` and `dfs_low(A) = 0`.

- Next, we visit vertex B. Vertex B is assigned a `dfs_num` of 1 since it's the second vertex we're visiting. Moreover, Vertex B has a `dfs_low` value of 0 since we can reach a vertex with a `dfs_num` value of 0 through the path $B \to C \to A$. Note that it would be invalid to say that the path $B \to A$ causes `dfs_low(B)` to equal 0 since we cannot go backwards in the depth-first search traversal.

- Applying similar reasoning, we find that vertex C ends up with a `dfs_num` value of 2, and it also has a `dfs_low` value of 0 (we can reach vertex A).

- Finally, vertex B ends up with a `dfs_num` value of 3; however, no vertices with a lower `dfs_num` value are reachable from D. Hence, the `dfs_low` value of D is also equal to 3. Note that it is incorrect to say that D has a `dfs_low` value of 0 through the path $D \to C \to A$ since we cannot revisit vertices while performing the depth-first search algorithm.

Why do we care about these `dfs_num` and `dfs_low` values? It becomes more clear when we consider the depth-first search tree produced by calling the depth-first search algorithm. The quantity `dfs_low(v)` represents the smallest `dfs_num` value reachable from the current depth-first search spanning subtree rooted at the vertex v. The value `dfs_low(v)` can only be made smaller if there's a back edge (an edge from a vertex v to an ancestor of v) in the depth-first search tree.

This leads us to make the following observation: If there's a vertex u with neighbor v satisfying `dfs_low(v) >= dfs_num(u)`, then we can conclude that vertex u is an articulation point. Note that this makes sense intuitively since it means that the *smallest* numbered vertex that we can ever reach starting from vertex v is greater than or equal to the number we assigned to u. Hence, removing u would disconnect `v` from any vertex with smaller `dfs_num` than `dfs_num(u)`.

Going back to the previous graph figure, we can note that the following:

```
3 = dfs_num(D) >= dfs_low(C) = 0
```

As stated previously, this implies that Vertex C is an articulation point. Note that removing Vertex C would disconnect the vertices A and B from Vertex D.

Now, there's one special case to this algorithm. The root of the depth-first search spanning tree (the vertex that we choose as the source in the first depth-first search call) is an articulation point only if it has more than one children. This one case is not detected by the algorithm; however, it is easy to check in implementation.

§5 Tuesday, February 11, 2020

§5.1 Articulation Point Algorithm Implementation

Last time, we introduced the algorithm to find articulation points. Recall that if there's a vertex u with neighbor v satisfying `dfs_low(v) >= dfs_num(u)`, then vertex u is an articulation point.

In terms of the depth-first search tree, the quantity `dfs_low(v)` is the lowest value that you can reach by going down the depth-first search tree rooted at v and possibly taking a back edge up (we can't visit the immediate parent of v). The inequality `dfs_low(v) >= dfs_num(u)` implies that we cannot visit any vertex with `dfs_num` less than `dfs_num(u)` when we start a depth-first search from v (there aren't any back edges that go to a vertex visited before vertex u).

Furthermore, recall that the root of the depth-first search tree is an exception — this vertex is an articulation point only if it has more than one child.

When actually implementing this algorithm, we need to be clever in order to maintain a linear time complexity. A pseudocode implementation is provided at http://www.cs.umd.edu/class/spring2020/cmsc451/biconnected.pdf.

§5.2 Strongly Connected Components

Recall that an undirected graph $G = (V, E)$ is called **connected** provided that for any pair of vertices $u, v \in V$, there exits a path between u and v.

The corresponding analogue for connectivity in a directed graph is presented below:

Definition 5.1. We call a *directed* graph **strongly connected** if, for every pair of vertices $u, v \in V$, there exists a directed path $u \rightsquigarrow p$.

We're often interested in checking whether or not a graph is strongly connected (e.g. starting from *anywhere* in a directed graph, is it possible to reach *everywhere* else?).

Like connected components in an undirected graph, strongly connected components in a directed graph form a partition of the set of vertices. This is formalized through the following result:

Lemma 5.2 (Klekleinberg and Tardos, 3.17)
For any two nodes s and t in a directed graph, their strong components are either identical or disjoint.

Proof. Consider any two nodes s and t that are mutually reachable. We claim that the strong components containing s and t are identical. This is clearly true due to the definition of a strongly connected component — for any node v, if s and v are

mutually reachable, then t and v are mutually reachable as well (we can always go $s \rightsquigarrow t \rightsquigarrow v$). Similarly, if t and v are mutually reachable, then s and v must be mutually reachable as well.

Conversely, suppose s and t are *not* mutually reachable. Then there cannot be a node v in the strong component of both s and t. Suppose such a node v existed. Then s and v would be mutually reachable, and v and t would be mutually reachable. But this would imply that s and t are mutually reachable, which is a contradiction. □

A brute force algorithm to check whether a grpah is strongly connected is presented below:

1. For each vertex $v \in V$ in our input graph $G = (V, E)$, perform a depth-first search starting with vertex v.
2. If there exists some vertex u that we cannot from a vertex v, then we can conclude that G is not strongly connected.
3. If we finish iterating over all vertices with no issues, we can conclude that our graph is strongly connected.

Since we perform $\mathcal{O}(V)$ depth-first search calls in the algorithm above, the runtime of this algorithm runs in $\mathcal{O}(V \times (V + E)) = \mathcal{O}(V^2 + VE)$ time on an Adjacency List. However, this is not as efficient as we can get.

It turns out that we can solve the problem of determining whether a graph is strongly connected in linear time using two depth-first search calls. Before presenting this algorithm, we'll need the following terminology:

Definition 5.3. Given a directed graph $G = (V, E)$, the **transpose graph** of G is the directed graph G^T obtained by reversing the orientation of each edge from (u, v) to (v, u).

A summary of Kosaraju's algorithm is presented below:

1. Pick an arbitrary vertex $v \in V$ in our initial graph $G = (V, E)$.
2. Perform a depth-first search from v and verify that every other vertex in the graph can be reached from v. If there exists some vertex u that cannot be reached from v, then we can immediately conclude that G is not strongly connected.
3. Compute G^T, the transpose graph of G^T. Perform a depth-first search on G^T with the same source vertex v. If we can reach every vertex from v in G^T as well, then we can conclude that G is strongly connected.

Why does this work? Because a graph and its transpose always have the same connected components (for each directed $u \rightsquigarrow v$ path, we can just go in the reverse direction).

Now, this algorithm tells us *if* a graph is strongly connected; however, it doesn't tell us *what* the strongly connected components are (i.e. if a graph has many strongly connected components, which component does an arbitrary vertex v belong in?). To answer this question, we'll first present a way to classify the edges in a depth-first search tree.

We will see that this edge-classification system is very closely related to finding strongly connected components in a graph.

§5.3 Classifying Edges in a DFS Tree

While performing a depth-first search traversal, we generate a depth-first search spanning tree. In particular, this DFS tree's root is the source vertex from which we started the DFS traversal, and we add the edge (u, v) if we traverse the edge (u, v) during the DFS procedure.

Within the depth-first search tree, we can classify each edge into exactly one of four disjoint categories:

1. **Tree edges** are edges traversed by the depth-first search traversal (i.e. they are neighbors in the original graph, and we go from one of the vertices to the other). These are the only type of edges that are actually explored.
2. **Back edges** are edges that are part of a cycle in the original graph. In particular, a back edge is an edge (u, v) that we discover when we have started (but not finished) a DFS traversal from v and we're exploring the neighbors of vertex u.
3. **Forward edges** and **cross edges** are edges of the form (u, v) where we have started (but not finished) the depth-first search traversal from u, and we find a vertex v that has already been fully explored.

§6 Thursday, February 13, 2019

Last time, we started discussing strongly connected components, and we presented an edge-classification system. Today, we'll show how we can use our edge-classification system to identify what vertices lie in strongly connected components.

§6.1 Kosaraju's Algorithm

Now, we'll show how we can identify strongly connected components in linear time. The algorithm that we will describe is **Kosaraju's algorithm**.

The pseudocode corresponding to the algorithm is presented below:

```
procedure kosarajuSCC(graph G) {

    for each node v in G:
        color v gray.

    let L be an empty list.
    for each node v in G:
        if v is gray:
            run DFS starting at v, appending each node to list L when it
                is we've finished processing that node.

    let G' be the transpose graph of G

    for each node v in G':
        color v gray.

    let SCC be a new array of length n.
    let index = 0

    for each node in v in L, in reverse order:
        if v is gray:
            run DFS on v in G' and set scc[u] = index
            for each node u visited during the traversal.
        index = index + 1


    return scc
}
```

How is this working?

1. Firstly, we look at the original graph $G = (V, E)$, and we perform a depth-first search on the components of G. Once we've finished visiting each node v, we append v to the end of a list L (we are placing the vertices into L in reverse-topological order). The list L ends up being sorted in reverse-order of

finishing time. The entire purpose of this first depth-first search traversal is to be able to number the vertices according to their finish time.

2. Next, we'll construct the transpose graph G^T, and we'll iterate over L in reverse-order. Recall that the strongly connected components in G^T are exactly the same as those in G. Also, we mark each

3. For each vertex v we visit in L, if we haven't already call DFS on while iterating over L, any set of vertices that we visit forms a strongly connected component.

Some more intuition is provided below.

Note that, when performing a depth-first search in G^T in post-order from a node v, the depth-first search first visits nodes that can reach v followed by v itself, and finally followed by nodes that cannot reach v. On the other hand, when we perform a depth-first search in pre-order on the original graph G from a node v, the depth-first search first visits v, followed by any nodes reachable from v, and finally the nodes that are not reachable from v.

§6.2 Topological Sorting

Next, we'll begin discussing our next problem. First, we'll present a couple of definitions.

Definition 6.1. A **directed acyclic graph**, also known as a "DAG," is (as its name suggests), a directed graph that doesn't have any cycles.

Definition 6.2. A **topological sort** of a directed acyclic graph $G = (V, E)$ is a linear ordering of all its vertices such that if G contains an edge (u, v), then u precedes v in the ordering.

Clearly, a graph with a cycle cannot be topologically sorted — we wouldn't be able to order the vertices that form the cycle.
It's important to remember that, unlike number sorting algorithms, topological sorts are not unique. Each graph G can have multiple valid topological sorts.

Topological sorts are really helpful when we're considering a graph that represents precedences among events or objects. Here are a few examples:

Example 6.3 (Figure 22.7, CLRS)
Professor Bumstead gets dressed in the morning. The professor must wear certain garments before others (e.g. socks before shoes), whereas other pairs of items can be put on in any order (e.g. socks and pants). We can represent this situation with a directed acyclic graph $G = (V, E)$ in which a directed edge (u, v) indicates that garment u must be donned before garment v. The professor can topologically sort this graph in order to get a valid order for getting dressed.

Here's another example.

Example 6.4 (Pick-up Sticks)

The game of *pick-up sticks* involves two players. The game consists of dropping a bundle of sticks. Subsequently, players take turns trying to remove sticks without disturbing any of the others. In order to model this game, we can use a directed graph $G = (V, E)$ in which each vertex represents a stick. We place a directed edge (u, v) between sticks u and v if stick u is on top of stick v. By topologically sorting the graph, we can find a valid way to pick up the sticks on top first.

Now, we've seen a couple of examples in which topological sorts can be useful, but how do we perform a topological sort?

It turns out we can topologically sort a graph in linear time. We will present two algorithms.

Firstly, we present **Kahn's algorithm**, which relies on the following fact:

Proposition 6.5

Every directed acyclic graph has at least one vertex with in-degree 0.

Proof. Suppose not. For each vertex v, we can move backwards through an incoming edge. But due to the finiteness of the graph G and absence of a cycle, this process must eventually terminate. The vertex we terminate must have in-degree 0. □

Now that we've established this fact, a summary of Kahn's algorithm is presented below:

1. Enqueue all vertices with in-degree 0 into a priority queue Q. At least one such vertex must exist due to Proposition 6.5.
2. Let L be an empty list. This will store our vertices in topologically sorted order.
3. While the Q isn't empty, extract the next vertex u from Q. Remove the vertex u from the original graph G along with any incident edges, and add u to L. If this removal causes another vertex v to have in-degree 0, then enqueue v into Q.
4. Once the while-loop terminates, L will contain every vertex in topologically sorted order.

While we won't prove correctness for this algorithm, it should be a little clear as to why it works. Since we're always choosing vertices with in-degree 0, we know that there is no other vertex that should come before the vertex we're choosing. Hence, the vertices we pick are always "safe." This is pretty similar to the selection sort algorithm used to sort numbers in which we repeatedly pick the minimum element in an array to place at the front of the array. This algorithm runs in $\mathcal{O}(V + E)$ time

on an adjacency list.

Here's a second algorithm that correctly performs a topological sort. This is just a slight modification to the DFS algorithm.

1. Let $G = (V, E)$ be our original graph. Mark each vertex $v \in V$ as "unvisited."
2. For each unvisited vertex, call `DFS(v)`, and prepend `v` into an array `A` once we've finished visiting all of its neighbors.
3. Once we've finished visiting every vertex in G, the array `A` will be in reverse-topological order. We can reverse the array in linear time, and we're done.

This algorithm runs in $\mathcal{O}(V + E)$ time as the runtime is dominated by our depth-first search calls.

Once again, we won't prove correctness of this algorithm, but it should be clear why this algorithm works. Our call to depth-first search will end pushing vertices with out-degree 0 onto the stack first (because they won't have any more neighbors to visit), which are always safe to place at the end of the topological ordering since no vertex is "greater" than them. This is followed by other vertices in ascending order of out-degree.

A C++ implementation of this algorithm is presented below:

```
vector<vector<int>> AdjList; /* Our graph. */
vector<int> toposort; /* Global array to store topological sort. */
bool visited[10000];

void dfs(int u) {
    visited[u] = true;
    for (int i = 0; i < AdjList[u].size(); i++) {
        int v = AdjList[u][i];
        if (!visited[v]) {
            dfs(v);
        }
    }
    toposort.push_back(u);
}

int main(void) {
    memset(visited, false, sizeof(visited));
    for (int i = 0; i < V; i++) {
        if (!visited[i]) {
            dfs(i);
        }
    }
    reverse(toposort.begin(), toposort.end());
    /* Topological sort is complete. */
}
```

§6.3 Bipartite Graphs

Finally, we'll discuss bipartite graphs.

Definition 6.6. A graph $G = (V, E)$ is called **bipartite** if we can partition its vertex set V into two disjoint sets U and V such that each edge $(u, v) \in E$ has one endpoint in U and the other endpoint in V.

Here's an equivalent definition that we sometimes like to use:

Definition 6.7. A graph $G = (V, E)$ is said to be **bipartite** if we can color each vertex either black or white such that no two adjacent vertices have the same color.

In order to test whether a graph is bipartite, we can perform a graph search in which we color vertices as we go along. Although we can use either breadth-first search or depth-first search for this check, breadth-first search is often the more natural approach. Pretty much, we start by coloring the source vertex with value 0, color the direct neighbors of the source vertex with 1, the neighbors of the neighbors of the source vertex with color 0, and so on. If we encounter any violations (i.e. two adjacent vertices with the same color) as we go along, then we can conclude that the given graph is not bipartite.

A C++ implementation is provided below:

```
vector<vector<int>> AdjList; /* Our graph. */

bool isBipartite(int src) {
    queue<int> q;
    q.push(src);
    vector<int> color(V, INFINITY);
    color[src] = 0;
    bool isBipartite = true;

    while (!q.empty() && isBipartite) {
        int u = q.front(); q.pop();

        for (int i = 0; i < AdjList[u].size(); i++) {
            int v = AdjList[u][i];
            if (color[v] == INFINITY) {
                /* We haven't colored v yet. */
                color[v] = 1 - color[u];
                q.push(v);
            } else if (color[v] == color[u]) {
                /* We've found a violation. */
                isBipartite = false;
                break;
            }
        }
    }
    return isBipartite;
}
```

The runtime of this algorithm is dominated is $\mathcal{O}(V + E)$ on an adjacency list since we're just performing a breadth-first search.

Another useful fact regarding bipartite graphs is the following:

Fact 6.8. A graph is bipartite if and only if it has no odd cycles (i.e. cycles of length $3, 5, 7$, etc).