Design and Analysis of Algorithms

Ekesh Kumar[*]

February 28, 2020

These are my course notes for CMSC 451: Design and Analysis of Algorithms, taught by Professor Clyde Kruskal. Gaps in lecture material are filled in with CLRS and Kleinberg & Tardos. Please send 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. $\square$

§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:

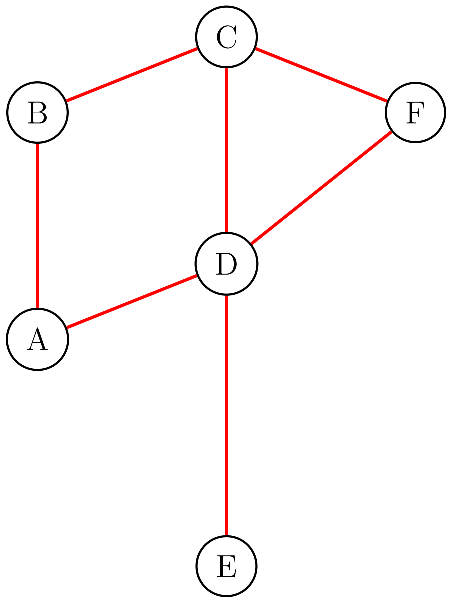

Figure 2: A Graph with an Articulation Point

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:

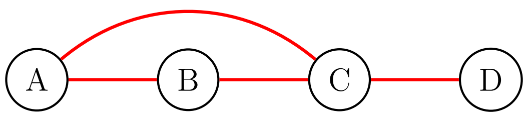

Figure 3: Articulation Point Example

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:

```cpp
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:

```cpp
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).

§7 Tuesday, February 18, 2020

Last time, we finished graph algorithms. Today, we'll begin **greedy algorithms**, which are a class of algorithms that repeatedly make "locally optimal" decisions in an attempt to find a globally optimal solution.

§7.1 The Union-Find Data Structure

§7.1.1 Motivating the Union-Find Data Structure

Before we introduce Kruskal's algorithm, we'll need to first introduce a data structure known as the **union-find** or **disjoint-set** data structure. Why? Because this data structure is used in the implementation of Kruskal's algorithm, which is one of the two minimum spanning tree algorithms we will be talking about.

The union-find data structure consists of a collection of disjoint sets (i.e. a set of sets). Each disjoint set is uniquely determined by a **set representative**, which is some member of the set. In most applications, it doesn't actually matter which member of the set is used as the representative; all we care is that, if we ask for the representative of a set twice without making any modifications, we should get the same answer both times.

The union-find data structure supports the following operations:

1. The `MAKE-SET(x)` operation creates a new set whose only member is x. Since x is the only member of this newly created set, x must also be the representative of this set. Moreover, since we require the sets to be disjoint, we require that x not already be in some other set.

2. The `UNION(x, y)` operation unites the two sets that contain the elements x and y. More precisely, if S_x and S_y are the sets containing x and y, then we remove both of these sets from our collection of sets, and we form a new set $S \overset{\text{def}}{=} S_x \cup S_y$, which is subsequently added to the collection of sets. What element becomes the representative of the new set? Typically, if S_x was originally larger than S_y, then we make the representative of S_x the representative of S. Otherwise, we make the representative of S_y the representative of S.

3. The `FIND-SET(x)` method takes in an element x and returns the representative of the set containing x. Note that this means that `FIND-SET(x)` might return `x` itself (if `x` is the representative of its set).

There are several applications of the union-find data structure. One of the many applications arises when we are trying to determine the connected components in an undirected graph. In particular, we can answer queries of the form "Are vertices u and v in the same connected component?" with a quick running time by using this data structure.

Consider the following pseudocode:

```
# Input:  A graph G.

# Output:  Nothing.  This function is called as a preprocessing step
# in order to use the function SAME-COMPONENT(u, v).

CONNECTED-COMPONENTS(G) {
   for each vertex v ∈ G {
     MAKE-SET(v)
   }
   for each edge (u, v) ∈ G {
     if FIND-SET(u) != FIND-SET(v) {
       UNION(u, v)
     }
   }
}

# Input:  Two vertices u and v.  CONNECTED-COMPONENTS(G) must be
# called prior to using this function.

# Output:  True if u and v are in the same connected component;
# otherwise false.

SAME-COMPONENT(u, v) {
   return (FIND-SET(u) == FIND-SET(v))
}
```

How does these functions work?

- We use a single union-find data structure that is initially empty. At first, we create a new disjoint set for each vertex. Each disjoint set in our union-find data structure will represent a connected component in our graph.
- Next, we traverse every edge in our graph G. For each edge (u, v), we merge the two disjoint sets containing u and v (since they must be in the same component).
- Finally, we can call the `SAME-COMPONENT` function with two vertices u and v which simply compare the representatives of the sets u and v are in to determine whether the two vertices are in the same component.

§7.1.2 Implementation of the Union-Find Data Structure

In our connected components example, we use a union-find data structure, but we never explain how the functions `MAKE-SET`, `UNION`, or `FIND-SET` are implemented. In this section, we'll discuss how to implement these three methods.

Union-find data structures are typically implemented as a **disjoint-set forest** in which each member only points to its parent (the root of each tree is the representative of the disjoint set, and it is its own parent). The following figure from CLRS illustrates this idea:

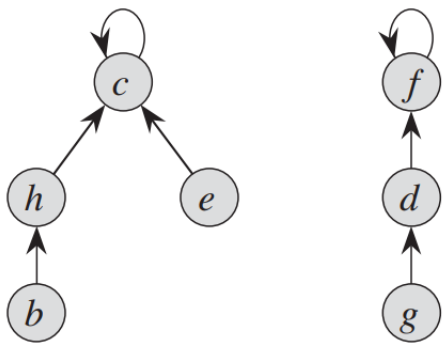

Figure 4: A Disjoint Forest

The disjoint-set forest above represents the two sets $\{c, h, b, e\}$ with representative c and $\{f, d, g\}$ with representative f. Note that the parent of any representative is itself.

How do we keep track of the parent of each vertex? This is easy — we can just include an array called `parent` as a part of our data structure implementation. For any vertex v, we can store the parent of v in `parent[v]`.

Now, we will discuss two heuristics to improve the running-time of various union-find operations. The first heuristic, known as the **union by rank heuristic**, is a heuristic that is applied when performing the `UNION` operation. In particular, this heuristic specifies to make the root of the tree with fewer nodes to point to the root of the tree with more nodes. Why? Because following the union by rank heuristic minimizes the overall depth of the resulting tree.

The following diagram illustrates the resulting tree that comes from performing the `UNION` operation on two elements in the disjoint sets from the previous figure:

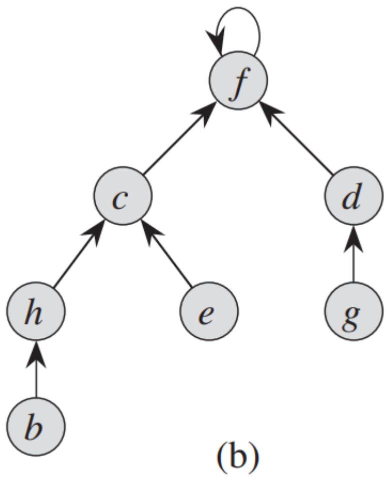

Figure 5: Our Disjoint Forest after performing `UNION`

Note that f is the representative of the resulting tree since we make the tree with fewer nodes point to the root of the tree with more nodes.

It would be computationally expensive to keep on recomputing the number of roots in each tree whenever we perform a `UNION` operation. Thus, we can instead just maintain an array `rank` which stores an upper bound on the height of each node. During a `UNION` operation, we simply make the root with a smaller rank point to the root with the larger rank.

The second heuristic, known as **path compression** is a heuristic that is used during `FIND-SET` operations to make each node on the find path point directly to the root. This technique is fairly easy to implement, and its purpose is to keep the depth of the tree small.

A C++ implementation of the union-find data structure is presented below:

```
/* An implementation of the union-find data structure. */
class UnionFind {
   private:
       vector<int> parent;
       vector<int> rank;
   public:
       /* A constructor to initialize a union-find data structure with
          capacity N. */
       UnionFind(int N) {
           parent.assign(N, 0);
           rank.assign(N, 0);

           /* Each vertex is initially its own parent. */
           for (int i = 0; i < N; i++) {
               parent[i] = i;
           }
```

```
        }

    /* findSet(u) returns the representative of the set that u belongs
       to. */
    int findSet(int u) {
        if (parent[u] == u) {
            /* u is the representative of its set. */
            return u;
        }
        /* Path compression heuristic. */
        return parent[u] = findSet(parent[u]);
    }

    /* inSameSet(u, v) returns true if u and v are in the same set; false
       otherwise. */
    bool inSameSet(int u, int v) {
        /* We compare the set representatives. */
        return findSet(u) == findSet(v);
    }

    /* Union the sets that u and v belong in. */
    void unionSet(int u, int v) {
        if (!inSameSet(u, v)) {
            int rep1 = findSet(u);
            int rep2 = findSet(v);
            /* Union by rank heuristic. */
            if (rank[rep1] > rank[rep2]) {
                parent[rep2] = rep1;
            } else {
                parent[rep1] = rep2;
                if (rank[rep1] == rank[rep2]) {
                    rank[rep2]++;
                }
            }
        }
    }
};
```

§7.1.3 Analysis of Union-Find Operations

In order to discuss the running time of each of the union-find operations, we will need to use the **inverse Ackermann function**, denoted $\alpha(n)$. For our purposes, all we need to know is that this is an *extremely* slowly growing function (for all practical purposes, its value never exceeds 5).

While we won't derive the bound, we will take it for granted that the `UNION` and `FIND-SET` operations run in $\mathcal{O}(\alpha(n))$ time (approximately constant time). A full derivation is provided in CLRS 21.4.

§7.2 The Minimum Spanning Tree Problem

§7.2.1 Problem Statement

The **minimum spanning tree** problem is stated as follows:

> "Given a graph $G = (V_1, E_1)$, find a connected subgraph $H = (V_2, E_2)$ such that $V_1 = V_2$ and the quantity
>
> $$\sum_{(u,v)\in E_2} \texttt{weight}(u, v)$$
>
> is as minimal as possible."

The following proposition shows that H will always be a tree:

Proposition 7.1

If $H = (V_2, E_2)$ is a connected subgraph of $G = (V_1, E_1)$ with the properties described above, then H is a tree.

Proof. By definition, H must be connected. Thus, it suffices to show that H doesn't have any cycles. Suppose H contained a cycle C. Let e be an edge on C, and consider the graph $H \setminus \{e\}$. This graph is still connected since removing an edge in a cycle can't disconnect a graph, but this graph is also "cheaper" than H; this is a contradiction. $\square$

A simple brute force algorithm to find the minimum spanning tree would work by generating each possible spanning tree and storing the generated tree if its cost is less than our previously stored minimum. Unfortunately, this algorithm is not feasible since graph has exponentially many different spanning trees. Thus, we are compelled to look for more efficient solutions.

Tody, we will discuss **Kruskal's algorithm** and **Prim's algorithm**, both of which are used to find the minimum spanning tree of a graph.

Both of these algorithms are classified as **greedy algorithms** — they repeatedly make locally optimal choices in an attempt to find a globally optimal solution.

§7.2.2 Kruskal's Algorithm

Let S be an initially empty set, and let $G = (V, E)$ be our graph. Kruskal's algorithm works by iteratively adding edges the least weight to S as long as (u, v) does not form a cycle with any of the other edges in S. The algorithm terminates when adding any edge in $E \setminus S$ to S would result in a cycle.

How do we quickly check if adding an edge (u, v) to S will result in a cycle? This can be done quite easily using the union-find data structure.

The pseudocode for Kruskal's algorithm is below:

```
# Input:  A graph G.

# Output:  A set of edges that form a minimum spanning tree of G.

KRUSKAL(G) {
   let S be an empty set.

   for each vertex v ∈ G {
     MAKE-SET(v)
   }

   sort the edges in G.Edges into nondecreasing order by weight.

   for each edge (u, v) ∈ G.Edges taken in sorted order {
     if FIND-SET(u) != FIND-SET(v) {
       # (u, v) won't form a cycle.
       Add the edge (u, v) to S.
       UNION(u, v)
     }
   }
   return S
}
```

As mentioned earlier, there isn't too much to this algorithm:

1. First, we sort the edges in non-decreasing order by weight so that we can traverse the list of edges from lowest weight to highest weight.
2. For each weight we look at, we check whether we can add the weight without adding a cycle. This is done by maintaining a union-find data structure.
3. Finally, we return the set of edges that form our minimum spanning tree.

How fast is Kruskal's algorithm? Firstly, note that the first for-loop performs V `MAKE-SET` operations. Subsequently, we sort the list of edges; doing so requires $\mathcal{O}(E\log(E))$ time. Finally, the second for-loop performs $O(E)$ `FIND-SET` and `UNION` operations. Putting everything together, we have a runtime of $\mathcal{O}((V+E)\alpha(V))$ time. But since $\alpha(|V|) = \mathcal{O}(\log(V)) = \mathcal{O}(\log(E))$ (where the second equality follows due to the fact that $|E| \geq |V| - 1$ in a connected graph), the total running time of Kruskal's algorithm as $\mathcal{O}(E\log(E))$.

§7.2.3 Prim's Algorithm

Prim's algorithm works by starting with an empty set S and iteratively adding edges to S until our minimum spanning tree is complete. We start by adding an arbitrary vertex to S, and at each step we add a vertex that is connected to some other vertex in S.

Some pseudocode illustrating how Prim's algorithm works is shown below:

```
# Input:  A graph G and a source vertex v.

# Output:  A set S containing the edges that represent a minimum
# spanning tree.

PRIM(G, v) {
   let key[1...V] be an array.
   let Q be an empty minimum priority queue

   for each vertex u ∈ G {
     key[u] = ∞
     parent[u] = NIL
     enqueue u into Q.
   }
   key[v] = 0

   # This is the main loop.
   while Q isn't empty {
     let u = EXTRACT-MIN(Q)
     for each vertex v in Adj[u] {
       if v ∈ Q and weight(u, v) < key[v] {
         key[v] = weight(u, v)
         parent[v] = u
       }
     }
   }
}
```

How does this algorithm work?

- We start building our minimum spanning tree from an arbitrary vertex v. This vertex is passed in as a parameter to our function.

- Next, we process enqueue all of our vertices into a minimum priority queue Q which allows us to extract elements with the minimum `key` value in logarithmic time.
- While Q isn't empty, we take the vertex with the smallest `key` value from Q; denote this vertex by u. Note that, on the first iteration, the vertex we grab is always v.
- For each neighbor v of u, we check whether the edge (u, v) is cheaper than the stored `key` value of v. If so, we update the key value of v to the weight of edge `(u, v)`. We additionally store the vertex u from which we took v.

The purpose of the minimum priority queue is to iteratively identify the cheapest edge that we can add to our minimum spanning tree. The `key` value of a vertex v represents the "cheapest" amount that we can pay in order to add that vertex to our spanning tree.

Prim's algorithm greedily selects the pair (u, v) in front of the priority queue—which has the minimum weight w—if the end point of this edge, namely v, has not been taken before. When the `while` loop terminates, the minimum spanning tree consists of the set of edges

$$A = \{(v, \texttt{parent}[v]) \mid v \in V - \{r\} - Q\}.$$

§8 Thursday, February 20, 2020

Today, we'll discuss two more algorithmic problems, both of which have greedy optimal solutions.

§8.1 Interval Scheduling

The first problem we'll discuss is known as the **interval scheduling problem**, which is stated as follows:

> Given a pair of parallel arrays `start[1...N]` and `finish[1...N]`, call a set of indices S **compatible** if, for any pair of indices $i, j \in S$, the intervals `(start[i], finish[i])` and `(start[j], finish[j])` are disjoint. Moreover, call a compatible set S **optimal** if its cardinality is maximal. The goal is to find an optimal set.

Why do we care about this problem? For each index $1 \le k \le N$, we can interpret the quantity `start[k]` and `finish[k]` as the starting time and ending time of an event. Under this interpretation, our task is to fit as many events as possible into our calendar.

There are several algorithms that we can implement that following the greedy heuristic:

1. One approach is to always select the next available event that always starts the earliest (i.e. keep on picking $\text{argmin}_{k \in \{1,2,\ldots N\}}$ `start[k]`), and remove k from our set afterwards. This method, however, is not optimal. A counterexample can be generated by considering the case in which the event with the earliest start time is very very long. By accepting this request, we'll miss out on many other events.

2. A second approach is to keep on picking $\text{argmin}_{k \in \{1,2,\ldots,N\}}$ `finish[k] - start[k]` and remove the index we picked from our set. While this is better than the other approach, this isn't optimal either.

3. A third approach is to pick the next request that finishes first (that is, pick $k = \text{argmin}_{k \in \{1,2,\ldots,N\}}$ `finish[k]`) over and over again. This algorithm seems similar to our first idea. Surprisingly, however, this is the optimal solution.

Some pseudocode illustrating how this procedure works is presented below:

```
# Input:  A set $S$ representing the
# Output:  An optimal solution $A$.
SCHEDULING(S) {
   let A be the empty set.
```

```
    while S isn't empty {
        let e be the event in S with the smallest finishing time.
        add request e to set A.
        remove any events that aren't compatible with e from S.
    }
    return A
}
```

Next, we'll prove that the set A returned by this algorithm is an optimal solution.

Proposition 8.1
The set A returned by our algorithm is a compatible set of events.

Proof. On each iteration, we add an event, and we remove any events that *aren't* compatible with the event we just added. Since the compatibility relationship between events is symmetric, we know that we'll never have a pair of incompatible events in A. □

Now we need to show that the set A produced by this algorithm has maximal cardinality. In order to do so, let $\mathcal{O}$ be an optimal set of intervals. We want to show $|\mathcal{O}| = |A|$.

In other words, if $A = \{i_1, i_2, \ldots, i_k\}$ and $\mathcal{O} = \{j_1, j_2, \ldots, j_m\}$, then our goal is to show $k = m$.

In order to show that this is true, we need to make use of the following lemma:

Lemma 8.2
For any indices $r \leq k$, we have `finish[`i_r`]` $\leq$ `finish[`j_r`]`.

Proof. For brevity, this proof writes $f(k)$ and $s(k)$ represent `finish[k]` and `start[k]`, respectively.

When $r = 1$, the statement holds since our algorithm always picks the index i_1 corresponding to the event with the minimum finish time. Now suppose $f(i_{r-1}) \leq f(j_{r-1})$. We want to show $f(i_r) \leq f(j_r)$. But this is clearly true since $f(j_{r-1}) \leq s(j_r)$ implies $f(i_{r-1}) \leq s(j_r)$. This means that j_r is in the set S of compatible events at the time when the greedy algorithm chooses i_r. Since the greedy algorithm always picks the event with the minimum finish time, we must have $f(i_r) \leq f(j_r)$. □

Lemma 8.2 means precisely that our greedy algorithm's intervals are finished at least as soon as the corresponding intervals in $\mathcal{O}$.

We can now prove our original claim:

Proposition 8.3
The set A returned by our greedy algorithm has maximal cardinality.

Proof. If A doesn't have maximal cardinality, then an optimal set $\mathcal{O}$ must have more requests. In other words, we require $m > k$. Applying our previous lemma with $r = k$, we find $f(i_k) \leq f(j-k)$. But since $m > k$, there exists some request j_{k+1} in $\mathcal{O}$. Since this request starts after the event corresponding to j_k ends, deleting all of the requests that aren't compatible with $i_1, \ldots, i_k$ will still contain j_{k+1}. However, this means that the greedy algorithm stops with a request present in set, when it's actually only supposed to stop when S is empty. □

§8.1.1 Extensions: Minimizing Lateness

Once again, consider the situation in which we have a set of n events that we want to schedule in an interval of time. But now, instead of a start time and a finish time, each event has a *deadline*. We say an event k is **late** if our finish time is greater than its deadline. Moreover, we define the *lateness* of a late event as the difference between the time at which it was finished and the time of the deadline. The objective of this problem is to minimize the number of late events.

The greedy algorithm in this problem is to sort the jobs in increasing order of their deadlines, and schedule them in this order (i.e. we process the events with the earliest deadline first). We will not prove the correctness of this algorithm.

§8.2 Caching

A **cache** is a piece of hardware or software that stores data in a special location so that future requests for that data can be served in a high-speed manner. The idea of caching is to store frequently-used values in a special area so that we can access the values in a quick manner. If a value is *not* cached, then we say that the value is stored in **main memory**.

In order to have an effective cache, it should usually be the case that when we're trying to access a piece of data, it's already present in the cache. Today, we'll talk about a cache maintenance algorithm that determines what to keep in the cache and what to toss out of the cache when new data is brought in.

Our problem is stated as follows:

> Let U be a set containing n pieces of data stored in main memory, and let C denote our cache that can hold $k < n$ pieces of memory. Given a sequence of data items $d_1, d_2, \ldots, d_m$ drawn from U, we must process them in order and determine which of the k items to keep in the cache. When an item d_i is presented that isn't in C, we say a **cache miss** occurs (we want to minimize these), and we have the option to evict some other

data element in C in exchange for d_i. Thus, our problem consists of computing the minimum number of cache misses necessary to process our data sequence.

Example 8.4 (Caching Example)

Suppose $U = \{a, b, c\}$, and our cache size is $k = 2$. Moreover, suppose we are presented with the sequence

$$a, b, c, b, c, a, b.$$

If the cache initially contains items a and b, then on the third item in the sequence, we can evict a to bring in c, and on the sixth item, we could evict c to bring in a. This results in two total cache misses. It can be shown that no solution can have fewer than two cache misses.

§8.3 Farthest in Future Algorithm

Surprisingly, the solution to the caching problem is fairly short. When data element d_i needs to be brought into the cache, we should always evict the item that is needed the farthest into the future. This is known as the **Farthest-in-Future algorithm**, and it was discovered by Belady.

We won't prove optimality; however, it's important to take note that that greedy algorithms might not always be obvious (why do we evict the element farthest in the future as opposed to the least frequent element?)

§9 Tuesday, February 25, 2020

§9.1 Huffman Codes

When storing data in a computer, we often need to use data compression techniques to minimize the amount of space we need. Today, we'll talk about **Huffman codes**, which are very effective — they typically result in savings between 20% and 90%. Huffman's greedy algorithm uses a table containing the frequency of each character to build an optimal solution of representing each character as a binary string.

§9.2 Prefix Codes

One particular class of encoding schemes are **prefix codes**. A prefix code for a set S of letters is a function γ that maps each letter $x \in S$ to some sequence of zeros and ones in such a way that for any $x, y \in S$ with $x \neq y$, the sequence $\gamma(x)$ is not a prefix of the sequence $\gamma(y)$. Why do many encoding schemes fall into this class? Because it removes ambiguity: — if there exists a pair of letters where the bit string that encodes one letter is a prefix of the bit string that encodes the other, then there might be multiple interpretations of the same string.

The ambiguity of encoding schemes that aren't prefix codes is demonstrated through the following example:

Example 9.1 (Ambiguity Morse Code)

In Morse code, we typically encode letters with dashes and dots. For our purpose, we can think of dots and dashes as zeros and ones. Suppose e maps to 0 (a single dot), t maps to 1, and a maps to 01. Then the string 0101 can have several interpretations: it can mean $eta, aa, etet$, or aet. If the morse code were a prefix code, then this problem wouldn't be present.

Now, here's an example illustrating the ease of using a prefix code:

Example 9.2 (Prefix Code Example)

Suppose we have a set $S = \{a, b, c, d, e\}$ with the encoding $\gamma(a) = 11, \gamma(b) = 01, \gamma(c) = 001, \gamma(d) = 10, \gamma(e) = 000$. This defines a prefix code since no encoding is a prefix of any other. The string *cecab* is encoded as 0010000011101, and a recipient of this message can decipher this message to our single unique message.

In order to efficiently decipher a prefix code, we need an effective way to represent the prefix code so that we can easily pick off the codeword. This is typically done with a binary tree in which the leaves of the tree store the characters of our alphabet. How does this work? We interpret the binary codeword for a character as a simple path from the root to that character; the bit 0 tells us to go to the left child, whereas the bit 1 tells us to go to the right child.

The following binary search tree illustrates a prefix code representation:

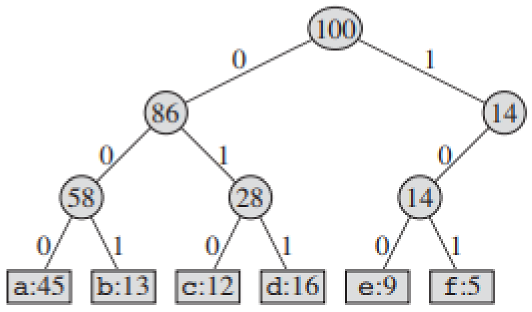

Figure 6: Prefix Code Representation

If we had the sequence 001011101, then we could start at the root, and we'd scan our sequence from left to right. First, we counter two zeros, so we go to the left twice. At this point, we'd be at the vertex labelled 58. Next, we encounter a 1, so we go to the right. Thus, we obtain the character b. Next, we start at the root again, and we follow our procedure again. The next character that we decipher is d. This process continues until there are no more bits to process.

Given a tree T corresponding to a prefix code, we can now easily compute the number of bits required to encode a file. In particular, for each character c in our alphabet S, we can let `freq[c]` denote the frequency of c in our file. Moreover, we can let $d_T(c)$ denote the depth of c's leaf in the tree. With this notation, the number of bits required to encode a file is given by

$$\sum_{c \in S} \texttt{freq[c]} \cdot d_T(c).$$

We call this the **cost** of the tree T.

§9.2.1 Constructing a Huffman code

Now that we've introduced prefix codes, we'll talk about an optimal prefix code known as a **Huffman code**, whose tree representation has minimum cost. The algorithm constructing the tree is presented below:

```
# Input:  A set C representing the set of all possible characters that
# might appear in our text, and an array freq[] in which freq[k] represents
# the frequency of the character k in our text.
# Output:  The root of a binary representing our encoding minimum cost.

HUFFMAN(C, freq) {
   let Q be a minimum priority queue
   for each element c in C { enqueue c into Q }

   for i = 1 to n - 1 {
     let z be a new node
     x = EXTRACT-MIN(Q)
     y = EXTRACT-MIN(Q)
     z.left = x
     z.right = y
     freq[z] = freq[x] + freq[y]
     insert z into Q.
   }
   return EXTRACT-MIN(Q) /* Return the root of the tree.  */
}
```

How does this algorithm work?

1. Firstly, we enqueue all of the characters in C into our minimum priorty queue Q.
2. The for-loop repeatedly extracts the two vertices with the lowest frequency and replaces them in the queue with a new node representing their "merger" (parent). The frequency of z is the sum of the frequencies of x and y.
3. After $n - 1$ merges, there's only one node left in the queue, which is the root of the code tree.

If we're using a binary heap, then the algorithm runs in $\mathcal{O}(n \log(n))$ time since we perform $\mathcal{O}(n)$ calls to `EXTRACT-MIN`, which is an $\mathcal{O}(\log(n))$ operation.
While we won't show it, it can be shown that this construction of a tree is optimal. This procedure counts as a greedy algorithm since, at each step, we greedily extract the characters with the lowest frequency.

§9.3 Matrix Multiplication

The next problem we'll discuss is stated as follows:

> Given two $n \times n$ matrices A and B, compute the $n \times n$ matrix C whose $(i, j)^{\text{th}}$ entry is defined by $c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$. In other words, we want to compute the product $C = AB$.

The brute force solution is $\mathcal{O}(n^3)$ (which is n^3 when $n = m = p$). In this algorithm, we just use three loops, and we compute each value c_{ij} in C as the summation provided in the problem statement. Of course, we want to do better.

Another idea is to perform a divide-and-conquer technique on the matrix. In particular, we can divide the matrix into four submatrices (top left corner, top right corner, bottom left corner, bottom right corner), and we can calculate the products recursively. The time complexity of this algorithm is given by the recurrence $T(n) = 8T(n/2) + \mathcal{O}(n^2)$. Unfortunately, by Master's Theorem, we know that the solution to this recurrence will be $\mathcal{O}(n^3)$, which isn't any better.