Design and Analysis of Algorithms
Section 0101

Ekesh Kumar[*]

February 11, 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:

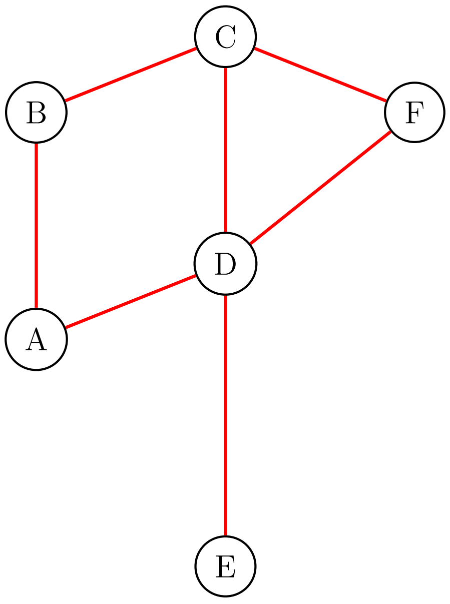

Figure 2: A Graph with an Articulation Point

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.

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:

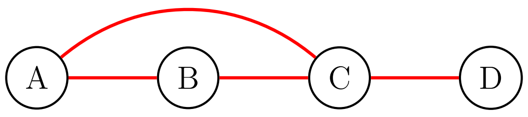

Figure 3: Articulation Point Example

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 Thursday, 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.

When implementing this algorithm, it's important to note to use recursion to our advantage.
In terms of
Recall that last time