Taking Off (Project 4)

Okay, Limgrave doesn't have enough locations so we're taking things into the real world!

Before you begin:

Familiarize yourself with the concept of (minimum) vertex covers..

Also, if you don't know what `heuristic` means, read the first couple sentences here.

To help clarify this, think spanning trees. These guys create a tree graph that spans all vertices. Vertex covers are a little different; instead of connecting all vertices to each other, covers create a set of vertices such that *at least one* (vertex) endpoint of *every* edge in the graph is included in the cover set. Another way of saying it is that, a vertex cover is a subset of a graph's vertices such that all edges connect to at least one vertex in the subset. A minimum vertex cover is a vertex cover using the minimum number of vertices.

The Premise

Now Professor's been doing a little traveling (not really, but I'm using the age-old of tactic of lies and deception to create a realistic premise).

The International Civil Aviation Organization (ICAO) wants to keep track of his (and any other planar pioneer's) whereabouts. As such, they have decided to **place one TSA agent** in airports that have a flight between them, using the given flight tables as references. However, they're cheap and want to minimize the number of TSA agents they hire, so they want to *approximate this quantity.*

If you don't care for the premise, we'll just be heuristically approximating minimum vertex cover for a Graph that you read in.

Ultimately, since it's finals season, we keep things brief.

Task 1: Taking Stock

Let's begin (for the last time this semester)! First, we'll need to read in the data of our flight tables.

You'll notice three text files already provided:

1. `sm_flights.txt`
2. `md_flights.txt`
3. `lg_flights.txt`

Each of these are flight tables of varying sizes (say on a given day using a UTC timezone), where each flight is represented by a row, from a given aiport (eg. `JFK`) to another airport (eg. `KIX`).

Notice that in this situation, we do not care about the duration nor direction of the flight. We only need to concern ourselves with the fact that there exists a flight between the two airports.

Thus, we'll make an undirected graph, where each Vertex is the three-letter code (as a string) representing an airport, and whose edges are stored as an adjancency list of other airport codes.

Now, we *could* create a new struct called `Vertex` or `Node` to represent vertices in the graph and figure out an entire Graph class. We can instead, just use a map from our the vertex to a collection of its adjacent vertices.

- Notice, however, we want to use *airport codes* for our vertices which are strings.
- We'll also be deleting from our edges within our Graph rather frequently (you'll see in the later task), so we'll benefit from storing neighbors within a container that supports fast accesses/deletions.

Hence, we opt for expressing our adjancency list representation of a Graph using an `unordered_map<string, unordered_set<string>>`. That is, each airport code maps to an (unordered) set of airport codes (which it has flights to).

```
using Vertex = std::string;
using Neighbors = std::unordered_set<Vertex>;
using Graph = std::unordered_map<Vertex, Neighbors>;
```

Here's an example:

If we have a flight from `JFK` to `LGA` & `JFK` to `KIX`, our graph would look like:

```
{
    "JFK" : { "LGA", "KIX" },
    "LGA" : { "JFK" },
    "KIX" : { "JFK" }
}
```

We'll make a function that reads from a given file and returns a graph, or throws a runtime error if no such file exists.

```
/**
* @brief Reads the contents of a flight table
* as specified by the filename, into an undirected
* Graph.
*
* @param filename (a const. string reference) The filename of the file to be read
* @return (Graph) The resultant Graph object described by the file's contents.
*
* @throws (std::runtime_error) If the file cannot be opened for some reason (eg.
*/
VertexCover::readFromFile
```

Task 2: Taking Flight

Now that we've read in the graph, let's move on to our algorithm portion: finding the minimum vertex cover for the flight table we just read in.

Mind you, finding the actual minimum vertex cover is an NP-hard optimization problem, so most of the time we *cannot* find a solution in a feasible amount of time.

Hence, we'll opt for a simple heuristic: choose the vertex with the greatest number of edges (or any vertex tied for the most edges), add it to the cover set, remove it (along with all adjacent edges, from the original graph), and continue until we've generated a vertex cover for the Graph.

That's it!

```
/**
 * @brief Generates a sub-optimal minimumum vertex cover
 * by repeatedly choosing the largest degree vertex, removing
 * it & its edges from the Grpah & adding it to the cover set.
 *
 * @param g (Graph) The graph object
 * for which to generate a vertex cover.
 * NOTE: This is NOT a const. reference
 *
 * @return (std::unordered_set<Vertex>) The set of vertices
 * that forms a vertex cover of the graph.
 */
VertexCover::cover_graph
```

Submission

All in all, you'll only be submitting two files:

1. `Graph.hpp`
2. `Graph.cpp` with both functions in the `VertexCover` namespace.

That's it! (I know *that's it?*). Happy finals and happy travels!