CMPUT 275 - Tangible Computing **Winter 2020**
Weekly Exercise #5

Graph Concepts

In class we built an instance of the `Digraph` class based on the description of a graph read from the input and discussed graph search algorithms to find the set of vertices that can be reached from a given vertex (i.e., there exists a path between them). You will do the following tasks in this weekly exercise:

1. Implement a function that counts the number of *connected components* in a given graph in linear time.
2. Implement a function that reads data (in a comma-separated format) from a file describing a city's road network and builds an instance of the `Digraph` class corresponding to the undirected version of that road network. That is, for every edge uv in the graph file, you add both (u, v) and (v, u) to the directed graph.
3. Implement a function that builds the undirected version of the Edmonton road network, calls the other function to count the number of connected components, and prints this number to the standard output.

You will write all these functions in `graph_concepts.cpp`. This file must be in the same directory as the `edmonton-roads-2.0.1.txt` file which contains the description of the Edmonton's road network. Your solution must build off of the files `digraph.cpp`, `digraph.h`, and `breadthfirstsearch.cpp` which are provided as starter code for this weekly exercise. We elaborate these three tasks below.

Task #1: Determining the Number of Connected Components

Implement a function `count_components(`g`)` that takes a single parameter g which is an instance of the `Digraph` class and returns an integer which is the number of connected components in graph g. Recall that a connected component of an undirected graph is a subset of vertices $C \subseteq V$ such that there is a path between any two vertices in C and there is no path between a vertex in C and a vertex lying outside of C.

For full marks this function should run in linear time, i.e. $O(|V| + |E|)$, where $|V|$ is the number of vertices and $|E|$ is the number of edges in the graph. We will assume that an insertion or lookup takes $O(1)$ time in unordered_map and unordered_set of STL.

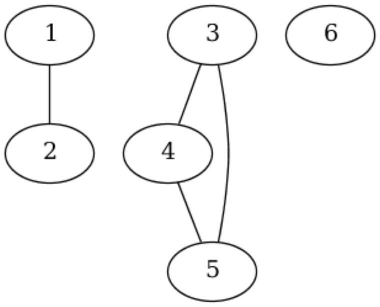
The code snippet below shows an example of calling the `count_components` function on an undirected graph modelled as a directed graph:

```
Digraph graph;
int nodes[] = {1, 2, 3, 4, 5, 6};
for (auto v : nodes)
  graph.addVertex(v);
int edges[][2] = {{1, 2}, {3, 4}, {3, 5}, {4, 5}};
for (auto e : edges) {
  graph.addEdge(e[0], e[1]);
  graph.addEdge(e[1], e[0]);
}
cout << count_components(&graph) << endl;
graph.addEdge(1, 4);
graph.addEdge(4, 1);
cout << count_components(&graph) << endl;
```

Output

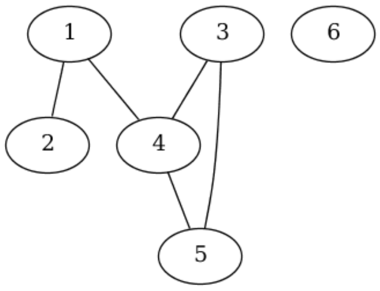
```
3
2
```

The first call to `count_components` finds the number of connected components in the following graph:



This graph has three connected components, i.e., $\{1, 2\}$, $\{3, 4, 5\}$ and $\{6\}$.

After adding the edge $(1, 4)$, the resulting graph (shown below) will have two connected components, i.e., $\{1, 2, 3, 4, 5\}$ and $\{6\}$.



This number is returned in the second call to this function.

Task #2: Building Graph of Road Network

Implement a function `read_city_graph_undirected(filename)` that takes the name of a plain text file which contains the description of a road network. This function builds an instance of our `Digraph` class corresponding to the undirected version of the graph described in this file. Recall that an undirected graph is modelled as a directed graph by adding both (u, v) and (v, u) directed edges for each undirected edge uv.

The file format will be in exactly the same format as the Edmonton graph file. That is, there are two types of lines:

- A line of the form `V,ID,Lat,Lon` describing a vertex.

  Here `V` is the single character '`V`', `ID` is an integer that is the vertex identifier (label), and `Lat` and `Lon` are floating point numbers describing the geographic coordinates (latitude and longitude) of this vertex.

  Important: The `ID` of a vertex is unique. Hence, no two lines starting with '`V`' will have the same `ID`. The vertices of the graph you construct should be the `ID` values provided in these lines.

- A line of the form `E,start,end,name` describing an edge/street.

  Here `E` is the single character '`E`', `start` and `end` are the IDs of two vertices connected by the edge, and `name` is a nonempty string giving the name of the street.

  Important: There may be spaces in name, but no commas. Every vertex ID used to define an edge has appeared earlier in the file, in a vertex description line.

An example of a road network with only 4 vertices and 3 edges is below:

```
V,1,53.430996,-113.491331
V,2,53.434340,-113.490152
V,3,53.414340,-113.470152
V,4,53.435320,-113.480152
E,1,3,St Albert Road
E,2,3,80 Avenue  North-west
E,2,1,None
```

You must treat each street as an **undirected** edge in this exercise. The x and y coordinates, and street names are not used in this exercise so you can ignore them after parsing each line. This information will be used in Assignment 2.

Hints:

- To open and read from a file, you can use an object of class `ifstream`.
- The functions `getline`, `find`, `substr`, and `compare` may come handy.

Task #3: Counting Connected Components in the Edmonton Graph

Your final task is to count the number of connected components in the undirected version of the Edmonton graph described in `edmonton-roads-2.0.1.txt`. Your program will take the name of this file as a command line argument and print the number of connected components:

```
make graph_concepts
./graph_concepts edmonton-roads-2.0.1.txt
```

Use `read_city_graph_undirected` to build the graph and pass this graph to `count_components` to find the number of connected components in this graph.
Note: do not just hard code the integer and print it!

Why will there be more than one connected component?
We obtained the road network description from `OpenStreetMap` by asking for all vertices and edges contained in a bounding box around Edmonton. Thus, the roads that crossed the boundary of this box are not included in the file, causing some vertices near the edges of the bounding box to be disconnected from other vertices.

Submission Details:

You should submit the following files as `components.tar.gz` or `components.zip` file:

- `graph_concepts.cpp` containing your solution to all three tasks along with `digraph.cpp`, `digraph.h`, and all other files needed to compile your code.
- a custom Makefile with at least these targets: (a) the main target `graph_concepts` which simply links `graph_concepts.o` and `digraph.o`, (b) the target `graph_concepts.o` which compiles the object, (c) the target `digraph.o` which compiles the object, and (d) the target `clean` which removes the object and the executable.
- your `README`, following the Code Submission Guidelines