Assignment 3 — Undirected Graphs

Due: October 18th

Graphs allow us to mathematically model and computationally explore the relationships that occur in real-world situations. Once we have modeled such situations as graphs, many interesting questions can be answered simply by finding paths from one vertex to another.

Deliverables:

GitHub Classroom: `https://classroom.github.com/a/khBul5Oo`

Required Files: `report.pdf`, `compile.sh`, `run.sh`

Optional Files: `*.c`, `*.h`, `*.py`, `*.java`, `*.js`, `*.json`, `*.ts`, `*.clj`, `*.kt`, `*.jl`, `*.rs`

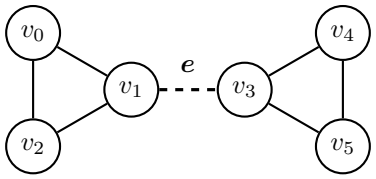
Part 1: Bridges

Consider the following graph definition:

- Vertices represent cities.
- Two vertices are connected by an edge if and only if there is a highway directly between those cities.

In anticipation of some hypothetical natural disaster, we might ask, "Which roads, if destroyed, would prevent travel from any city to any other city?"

Given a connected graph $G = (V, E)$, an edge $e = (u, v)$ is a *bridge* if and only if $G' = (V, E - \{e\})$ is not connected. That is to say, it is an edge whose removal disconnects the graph. For example:



In the above graph, the edge $e = (v_1, v_3)$ is a bridge. Its removal disconnects the graph, resulting in the components $\{v_0, v_1, v_2\}$ and $\{v_3, v_4, v_5\}$.

In your programming language of choice (see Assignment 1), design and implement an algorithm to find all of the bridges in a given graph. For the purposes of this assignment, it will suffice to find all bridges with complexity $O(|E|^2)$. To help you get started, note the following observations:

- If an edge $e = (u, v)$ is a bridge, then its endpoints u and v must be in different components after e is removed.
- If e is a bridge, then any and all paths that "crossed over" from one of those components to the other must have passed through u and v while traversing e.
- Thus, if there exists an alternative path from u to v that does not traverse e, then e is not a bridge: even if e is removed, it will still be possible to "cross over" via that alternative path.

Each input graph will be provided as an edge list: each edge in the graph will be represented by a comma-separated pair of vertex identifiers, indicating an edge between the first vertex and the second. You may assume that vertex identifiers are contiguous natural numbers — they begin at 0, and there will be no "gaps" in the identifiers used. You may further assume that the graph will be simple and connected.

For example, the above graph could be represented as:

```
0, 1
1, 2
2, 0
1, 3
3, 4
4, 5
5, 3
```

Your program must accept as a command line argument the name of a file containing an edge list as described above, then print to `stdout` all of the bridges according to the following format:

- Each bridge must appear on its own line. Within each bridge, the vertices must be comma-separated and sorted in ascending order: the vertex with the lower identifier must appear first.
- Bridges must be sorted in ascending order: a bridge with a lower first vertex must appear earlier. If two bridges have the same first vertex, the bridge with the lower second vertex must appear earlier.

For example:

```
>$ ./compile.sh
>$ ./run.sh in1.txt
Contains 1 bridge(s):
1, 3
```

Your program will be tested using `diff`, so its printed output must match *exactly*.

Part 2: Mathematically Characterizing Bridges

As we move away from surface-level inductive proofs of recursive pseudocode, such as those you wrote for Assignment 2, your proofs now will focus on the key underlying concepts that your algorithms rely upon. Prove the following theorem:

> *Theorem:* Let $G = (V, E)$ be a connected graph. An edge $e \in E$ is a bridge if and only if there exist no cycles traversing e in G.

While this fact is certainly instrumental in designing a correct algorithm, it is important to realize that, given only proof of the above, we cannot conclude that your algorithm is correct. Such a proof may formalize your intuition for the problem, but it does not demonstrate that your algorithm correctly acts on that intuition.

Part 3: Submission

The following files are required and must be pushed to your GitHub Classroom repository by the deadline:

- `report.pdf` — Pseudocode for an efficient algorithm to identify bridges in simple, connected graphs, along with proof of the theorem given in Part 2 and analysis of complexity for the pseudocode.
- `compile.sh` — A Bash script to compile your submission (even if it does nothing), as specified.
- `run.sh` — A Bash script to run your submission, as specified.

The following files are optional:

- `*.c`, `*.h`, `*.py`, `*.java`, `*.js`, `*.json`, `*.ts`, `*.clj`, `*.kt`, `*.rs` — The source code of a working program to identify bridges in graphs, as specified.

Any files other than these will be ignored.