Assignment 2 — Directed Graphs

Due: Monday, February 1st at 8pm

Credit: This is a lab by Christopher Siu.

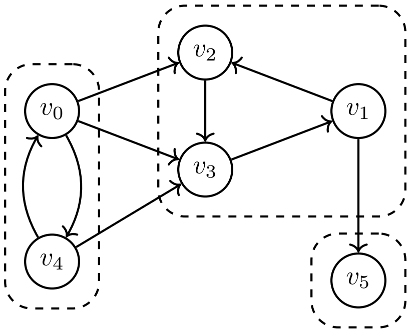
In a directed graph, the existence of a path from a vertex u to a vertex v does not imply the existence of a path from v back to u. As a result, finding a path from u to v does not necessarily mean that they are in the same component. Finding the components of a directed graph requires a more sophisticated approach than a pure depth-first search.

Deliverables:

GitHub Classroom:	`https://classroom.github.com/a/_bqy--Pj`
Required Files:	`compile.sh`, `run.sh`
Optional Files:	`*.py`, `*.java`, `*.clj`, `*.kt`, `*.js`, `*.sh`

Part 1: Strongly Connected Components

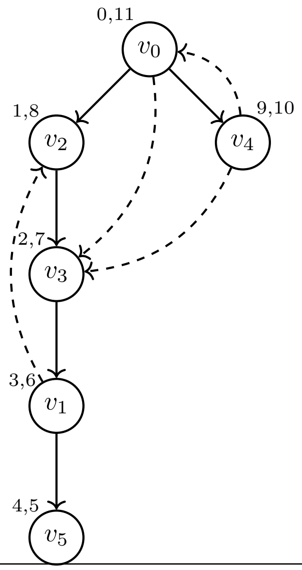
Recall that a *strongly connected component* of a directed graph is a set of vertices within which there exists a directed path between every pair of vertices. For example, consider the following directed graph:



This graph contains three strongly connected components: $\{v_1, v_2, v_3\}$, $\{v_0, v_4\}$, and $\{v_5\}$. Further recall that such components are *maximal*, in that no more vertices may be added to a component without breaking its strong connectedness. Thus, v_0 alone does not form a component, because v_4 can still be added.

In your programming language of choice per Assignment 1, implement an algorithm to find the strongly connected components of a directed graph. To help you get started, note the following observations:

- No strongly connected component can span multiple trees in a forest generated by a depth-first search.
- Assigning pre- and postvisit numbers to vertices during a depth-first search allows identification of back edges.
- Finding a back edge during a depth-first search indicates the existence of a cycle.
- All vertices along a cycle must be in the same strongly connected component.
- If two cycles share any vertices, then they are both in the same strongly connected component.



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 from the first vertex to 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 also assume that the graph will be simple and will not contain any isolated vertices.

For example, the above graph could be represented as:

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

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` the strongly connected components according to the following format:

- Vertices within the same component appear on a single comma-separated line. Vertices in different components appear on different lines.
- Each line's vertices must be sorted in ascending order. Note that vertex identifiers are integers, not strings. The lines themselves must be sorted in ascending order using their smallest vertex identifiers.

For example:

```
>$ ./compile.sh
>$ ./run.sh in1.txt
3 Strongly Connected Component(s):
0, 4
1, 2, 3
5
```

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

Part 2: Submission

The following files are required and must be pushed to your GitHub Classroom repository by 8pm of the due date:

- `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:

- `*.py`, `*.java`, `*.clj`, `*.kt`, `*.js`, `*.sh` — The Python, Java, Clojure, Kotlin, Node.js, or Bash source code files of a working program to find strongly connected components, as specified.

Any files other than these will be ignored.

Test run: On the date before the due date, the grader will be run after 8pm and provides feedback containing only what your program scores. Only submissions made before 8pm are guaranteed to receive feedback. What your program scores on the official run (the due date) is the final score.

Late submission:

- Late submissions made within 48 hours of the deadline receive no penalties.
- Late submissions made after 48 hours of the deadline receive a .8 multiplier. For example, if your late submission scores 80%, the final score will be 80% * .8 = 64%.
- If you decide to make a late submission, please send an email to vnguy143@calpoly.edu.
- Please do not include `compile.sh` and `run.sh` in your repository until you want to submit.

Resubmission: No resubmissions will be accepted. What you get on a submission (other than the test run) will be your final score.