Lab 2 - Strongly Connected Components

Start Assignment

- Due Wednesday by 11:59pm
- Points 5
- Submitting a file upload

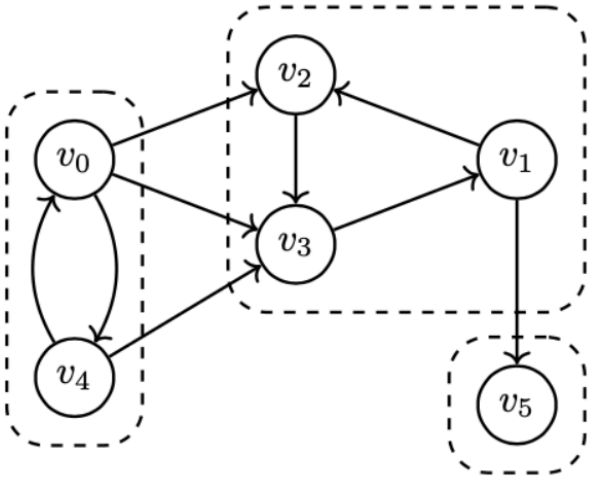
Credit: this lab is adapted from Christopher Siu. The sample files are created by Rodrigo Canaan.

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.

In this lab, you are given two sample scripts that read a file containing the

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: {v1,v2,v3}, {v0,v4}, and {v5}. Further recall that such components are maximal, in that no more vertices may be added to a component without breaking its strong connectedness. Thus, v0 alone does not form a component, because v4 can still be added.

Your job is to implement an algorithm that correctly assigns each vertex to a connected component.

Input

Your algorithm will receive as command line argument a text file that starts with a line containing a single integer, which represents *n*, the number of vertices, followed by *m* lines containing pairs of integers, separated by a comma, representing each of the *m* directed edges. For example, the above graph can be represented as shown below or in the file **lab2_example1 (https://canvas.calpoly.edu/courses/134261/files/14588042?wrap=1)** **(https://canvas.calpoly.edu/courses/134261/files/14588042/download?download_frd=1)** :

```
6
0, 2
0, 3
2, 3
0, 4
1, 5
4, 3
3, 1
1, 2
4, 0
```

You are also given a Python script **lab2_atomic_components.py (https://canvas.calpoly.edu/courses/134261/files/14588041?wrap=1)** **(https://canvas.calpoly.edu/courses/134261/files/14588041/download?download_frd=1)** that reads an input like the above and creates a list of *Node* instances. Each of these instances is already populated with a list of in- and out-going edges, and also initializes variables that can record useful information like pre- and post-visit numbers and the component assigned to each vertex. Feel free to create other class variables you deem necessary. This script also (incorrectly) assigns each Vertex to its own atomic component. An alternative (also incorrect) implementation **lab2_universal_components.py (https://canvas.calpoly.edu/courses/134261/files/14588016?wrap=1)** **(https://canvas.calpoly.edu/courses/134261/files/14588016/download?download_frd=1)** that assigns all vertices to a single universal component can also be found at .

Output

Your algorithm should implement a method called *strong_connectivity* that returns the connected components as a list of lists. Each of the inner lists represents a component and the elements in that contains the name (numerical id) of the vertices that belong to the component. The vertices within each component should be sorted in ascending order by their names, and the components themselves should be sorted in ascending order of their first element. I've added a method *sort component list* to the template that sorts the list of components in this fashion. Your *main* function should take the list of lists and print to the screen (using Python's default formatting of lists as strings). The *main* function in the template also does this for you.

Name your file studentID_lab2.py. Make sure your program prints nothing but the final answer to facilitate automatic grading

Usage example:

% python rcanaan_lab2.py lab2_example1

>>> [[0, 4], [1, 2, 3], [5]]

My automatic grader will attempt to read the list *returned* by the module, but if some unforeseen error happens, it will instead look at the console line output. The template handles all the formatting, so just focus on modifying the logic of *strong_connectivity*. You may also create helper methods. You shouldn't import anything other than sys, os, math and, optionally, **collections.deque** (https://docs.python.org/3/library/collections.html#collections.deque) for a generalized queue that supports O(1) insertion on either end

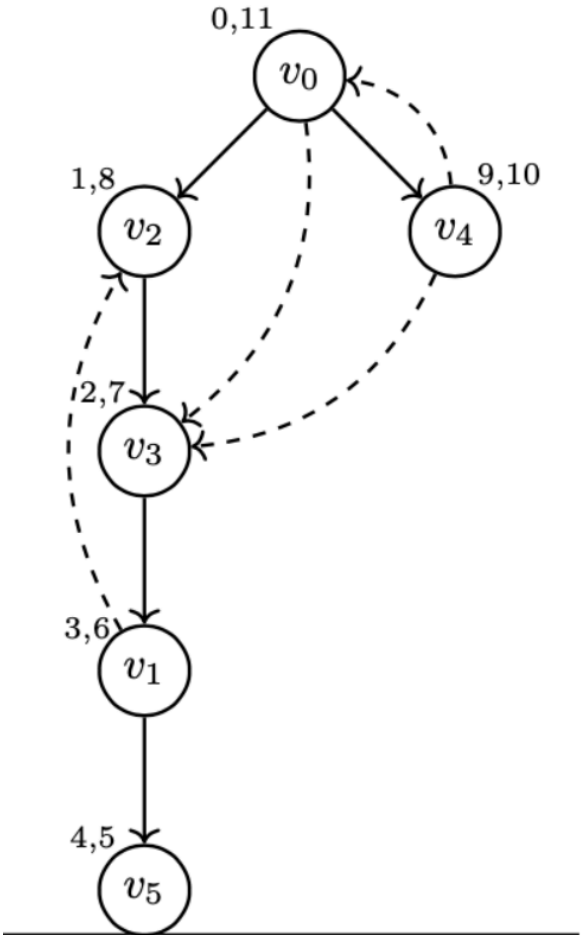
Tips

There are at least three linear time algorithms to calculate strongly connected components, as can be seen at **https://en.wikipedia.org/wiki/Strongly_connected_component** (https://en.wikipedia.org/wiki/Strongly_connected_component) . I based my reference implementation on **Kosaraju's Algorithm** (https://en.wikipedia.org/wiki/Kosaraju%27s_algorithm) , which is essentially the approach followed by the book. It consists on two passes of Depth First Search:

1. On the first pass, (called Visit on the Wiki), you iterate through all nodes and their **out-going** edges to calculate post-visit numbers.
2. On the second pass (called assign), you visit nodes in descending post-visit order and use their **in-going** edges to find the connected components. Make sure to assign a new number every time you start from a new root.

Note: you should not have to sort an array containing the post-visit numbers (since that would be inefficient). You can instead use a stack or equivalent structure (I used **collections.deque** (https://docs.python.org/3/library/collections.html#collections.deque) which supports O(1) insertion

The picture below show pre- and post-visit numbers resulting from the first pass in the graph above, starting from *v0*



Other tips:

- 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.