In this lab, we will implement the Girvan-Newman algorithm for community detection on undirected, weighted graphs. This algorithm comes from their 2002 PNAS paper:

Girvan and Newman. Community structure in social and biological networks. PNAS 2002.
Available on arXiv: `https://arxiv.org/pdf/cond-mat/0112110v1.pdf`.

The steps are:

1. Calculate the *edge betweenness* of all edges in the network. Recall that the edge betweenness is defined as

$$C_B(u,v) = \sum_{s,t \in V} \frac{\sigma(s,t|(u,v))}{\sigma(s,t)}, \tag{1}$$

   where $C_B(u,v)$ is the edge betweenness centrality of edge $(u,v) \in E$, $\sigma(s,t)$ is the number of shortest paths between s and t, and $\sigma(s,t|(u,v))$ is the number of shortest paths between s and t that pass through e.

   (a) For two nodes $s,t \in V$, you need to return *all* shortest paths.
   (b) You need to return all shortest paths for *all* pairs of nodes $u,v \in V$.
   (c) You can assume that the graph is connected.

   You may want to keep data structures that store this information.

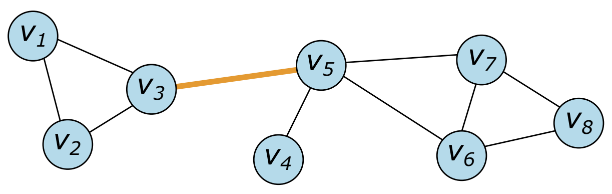
2. Remove the edge with the highest betweenness.

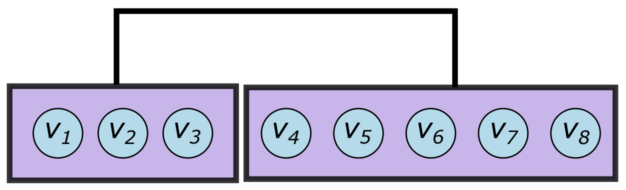
   (a) You need to compute the $C_B(u,v)$ for *all* edges $(u,v) \in E$.
   (b) Keep a *partition* of nodes at each iteration. When you remove an edge, you will modify this partition to reflect the split. (Sometimes the partition will remain unchanged).

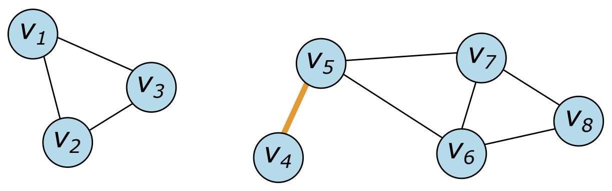
3. Recalculate betweenness for all edges affected by the removal.

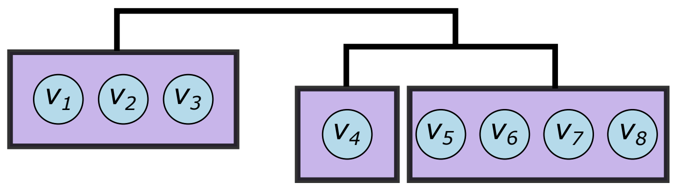
   (a) You can either re-compute all shortest paths for all pairs of nodes, or only compute the shortest paths for the node pairs that contained at least one shortest path with (u,v). It is relatively quick to run either way.

4. Repeat from Step 2 until no edges remain.









1 Calculate Edge Betweenness

The first major hurdle is to write a function that calculates the edge betweenness $C_B(u,v)$ of an edge $(u,v) \in E$. To do this, we first need to enumerate all shortest paths between two nodes, then determine a way to calculate σ in Equation (1). This will give us the components to calculate edge betweenness. **Use the `toy_dataset.txt` file to start.**

1.1 Modify Dijkstra's to return all shortest paths

Dijkstra's algorithm from HW5 works on undirected, weighted graphs. If there are multiple shortest paths, only one is stored in the predecessor (π) variable. Modify `dijkstra()` from HW5 to track all shortest paths (you will modify the predecessor dictionary, π). The pseudocode for the original Dijkstra's algorithm is in Lab7.

1.2 Copy or write a function to return all shortest paths

We must also modify the `get_path()` function from HW6 to return all paths between a source s and a target t. It will take as input whatever your Dijkstra algorithm returns (we'll call it π') from a source node $s \in V$.

`get_paths()`: *Return all shortest paths between two nodes*
Inputs: Predecessor dictionary π' and target node $t \in V$
Returns: List of lists denoting the paths that connect s and t.

This will be a *recursive function*, where your function will call itself. The function is tricky to debug! You have two options: (a) You can copy the code directly from `get_all_paths.py` and use it, or (b) for a challenge, you can write your own version.

In either case, **make a toy predecessor dictionary** π' that will contain multiple shortest paths and confirm that it returns the data structure you intended (in the code provided, it will contain a list of lists). You can hard-code this test in your code (be sure to comment it!).

1.3 Compute edge betweenness for all edges in the graph

Finally, you can write a function to compute edge betweenness for an edge in the graph. Consider pre-computing all shortest paths and passing this in as a data structure.

`edge_betweenness()`: *Write a function to calculate edge betweenness for an edge in the graph.*
Inputs: Undirected, weighted graph $G = (V, E)$.
Returns: The edge betweenness of every edge $(u,v) \in E$ as shown in Eq. (1).

2 Implement the GN algorithm on the toy network

Once we have an `edge_betweenness()` function, we can use it as a subroutine to implement the Girvan-Newman algorithm.

> `girvan_newman()`: *Identify communities using the Girvan-Newman algorithm.*
> **Inputs:** Weighted, undirected graph $G = (V, E)$
> **Returns:** The communities (groups of nodes) from G for $k = 1, 2, \ldots, n$.

Some considerations:

1. Note that you return *all* partitions. For the toy network, your partitions may look like these:

```
[['C', 'A', 'H', 'D', 'B', 'E', 'G', 'F']]
[['C', 'A', 'B'], ['H', 'D', 'E', 'G', 'F']]
[['D', 'E', 'G', 'F'], ['H'], ['C', 'A', 'B']]
[['D'], ['E', 'G', 'F'], ['H'], ['C', 'A', 'B']]
[['A', 'B'], ['C'], ['D'], ['E', 'G', 'F'], ['H']]
[['E', 'G'], ['F'], ['A', 'B'], ['C'], ['D'], ['H']]
[['A'], ['B'], ['E', 'G'], ['F'], ['C'], ['D'], ['H']]
[['E'], ['G'], ['A'], ['B'], ['F'], ['C'], ['D'], ['H']]
```

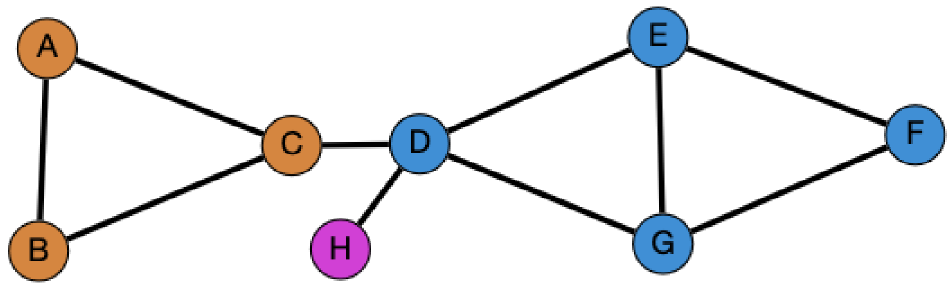
   Think about how you want to store these partitions.

2. At each step, you remove an edge, This is one way to determine whether you are done (though there are other ways).

3. When you remove an edge, you may need to modify the partition.

2.1 Post the toy network for some k, colored by k groups.

Post a graph to GraphSpace that contains the *original* edges of the network (you can read them in again or store a copy before you start removing edges). For $k = 3$, your graph will look like this:



3 Post the Badger Social Network

Now, run your code on the `badger_edge_costs.txt` file. Here, the weights (contact time in minutes) have been converted to costs, where lower is better.

1. Post a badger network corresponding to $k = 8$, the number of setts in the original dataset.
2. Read in the original sett information from `badger-info.txt` and denote these by node shapes. The available shapes are:

   ```
   'rectangle','roundrectangle','rhomboid',
   'ellipse','triangle','star','diamond','vee',
   'pentagon','hexagon','heptagon','octagon'
   ```

   Look for places of agreement & disagreement. Nothing to hand in here, but think about whether GN reflects sett membership.

Challenges

1. Run your code with $k < 8$ communities. Which groups are merged first?
2. Run your code with $k > 8$ communities. Which groups are split first?
3. We assumed that the graph is connected. Modify your code to handle the case where the graph is *not* connected.

Hand In: *Submit the code via Moodle (just the .py files). Share the badger social network with* $k = 8$ *to the Biol331F19 group (you may also share other numbers besides 8).*