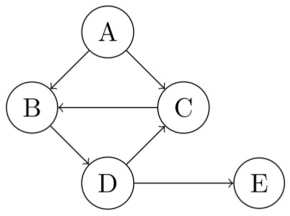
CS 161 **Problem Set 5**
Autumn 2017 **Due:** Friday November 10, 2017, at 3pm on Gradescope

Exercises

Exercises should be completed **on your own.**

1. **(2 pt.)** Consider the following directed graph:



For the following parts you might want to use this website, which allows you to draw directed graphs in LaTeX: `http://madebyevan.com/fsm/`. (Note: On a Mac, fn+Delete will delete nodes or edges).

(a) **(1 pt.)** Draw the DFS tree for this graph, starting from node A. Assume that DFS traverses nodes in alphabetic order. (That is, if it could go to either B or C, it will always choose B first).

(b) **(1 pt.)** Draw the BFS tree for this graph, starting from node A. Assume that BFS traverses nodes in alphabetic order. (That is, if it could go to either B or C, it will always choose B first).

[We are expecting: Pictures of your trees. No further explanation is required.]

2. **(3 pt.)** This exercise refers to `HW5.ipynb`, `graphStuff.py`, `HW5_mysteryA.pickle`, and `HW5_mysteryB.pickle`, available on the course website.

In class, we saw how to use BFS to determine whether or not a graph is bipartite; at least in theory. In this exercise, you will implement this idea (and hopefully also become familiar with the CS161Graph class that we'll be using for the whole graph unit).

In `HW5.ipynb`, we have provided the BFS code that we saw in class. This does Breadth-First Search beginning at a single vertex w of a graph, and it explores *all of the things reachable from w.* **(Note that it does not necessarily explore the whole graph).** `HW5.ipynb` imports `graphStuff.py`, which includes our implementation of `CS161Graph` and `CS161Vertex`.

Use the IPython notebook to load `HW5_mysteryA.pickle` and `HW5_mysteryB.pickle`, which load to `CS161Graph`s named A and B. The `CS161Graph` class supports directed graphs, but both graphs A and B are undirected in the sense that all edges go both directions: whenever there's an edge (v, w), there is also an edge (w, v).

One of A, B is bipartite and one of them is not. (Recall that a graph is *bipartite* if its vertices can be divided into two sets S and T so that there are no edges within S and no edges within T). Adapt the code in the IPython notebook to decide which is which. Your code should run on `CS161Graph`s in time $O(n + m)$, where n is the number of vertices in the graph and m is the number of edges.

Note: You should not have to change the code in `graphStuff.py`, although you may if you want to. Notice that `graphStuff.py` has been modified slightly from class so that a `CS161Vertex` has a `color` attribute, which may be helpful.

[We are expecting: Your answer (which graph is bipartite?) as well as the code that you used, and an English description of the changes that you had to make. You may wish to use the `verbatim` environment to include your code in LaTeX.]

Problems

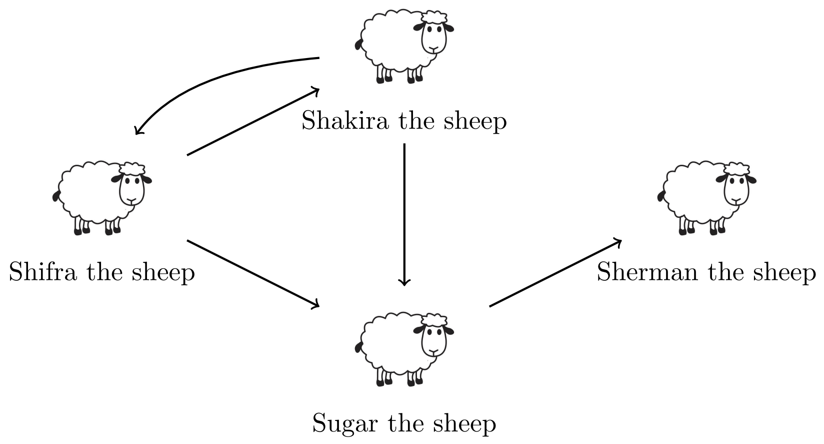
You may talk with your fellow CS161-ers about the problems. However:

- Try the problems on your own *before* collaborating.
- Write up your answers yourself, in your own words. You should never share your typed-up solutions with your collaborators.
- If you collaborated, list the names of the students you collaborated with at the beginning of each problem.

---

1. **(0 pt.)** Go over the problems on the midterm that you didn't get. You don't have to turn anything in, but this problem set is intentionally a little bit shorter to give you time to go over the midterm. Midterms will be handed back and solutions will be posted on Monday.

2. **(8 pt.)** You arrive on an island with n sheep. The sheep have developed a pretty sophisticated society, and have a social media platform called Baaaahtter (it's like Twitter but for sheep[1]). Some sheep follow other sheep on this platform. Being sheep, they believe and repeat anything that they hear. That is, they will re-post anything that any sheep they are following said. We can represent this by a graph, where $(a) \to (b)$ means that (b) will re-post anything that (a) posted. For example, if the social dynamics on the island were:



the Sherman the Sheep follows Sugar the Sheep, and Sugar follows both Shakira and Shifra, and so on. This means that Sherman will re-post anything that Sugar posts, Sugar will re-post anything by Shifra and Shikira, and so on. (If there is a cycle then each sheep will only re-post a post once).

For the parts below, let G denote this graph on the n sheep. Let m denote the number of edges in G.

[1]Also my new start-up idea

(a) **(2 pt.)** Call a sheep a **source sheep** if anything that they post eventually gets re-posted by every other sheep on the island. In the example above, both Shifra and Shakira are source sheep. Prove that all source sheep are in the same strongly connected component of G, and every sheep in that component is a source sheep.

[We are expecting: A short but rigorous proof.]

(b) **(4 pt.)** Suppose that there is at least one source sheep. Give an algorithm that runs in time $O(n + m)$ and finds a source sheep. You may use any algorithm we have seen in class as a subroutine.

[We are expecting: Pseudocode or an English description of your algorithm; a short justification that it is correct; and short justification of the running time. You may use any statement we have proved in class without re-proving it.]

(c) **(2 pt.)** Suppose that you don't know whether or not there is a single source sheep. Give an algorithm that runs in time $O(n + m)$ and either returns a source sheep or returns `no source sheep`. You may use any algorithm we have seen from class as a subroutine.

[We are expecting: Pseudocode or an English description of your algorithm, and an informal justification of why it is correct and runs in time $O(n + m)$.]

3. Let $G = (V, E)$ be a directed graph. A *path* from s to t in G is formally defined as a sequence of edges

$$(u_0, u_1), (u_1, u_2), (u_2, u_3), \dots, (u_{M-1}, u_M)$$

so that $u_0 = s$, $u_M = t$, and $(u_i, u_{i+1}) \in E$ for all $i = 0, \dots, M-1$. A *shortest path* is a path $((u_0, u_1), \dots, (u_{M-1}, u_M))$ so that

$$\sum_{i=0}^{M-1} \text{weight}(u_i, u_{i+1}) \le \sum_{i=0}^{M'-1} \text{weight}(u'_i, u'_{i+1})$$

for all paths $((u'_0, u'_1), \dots, (u'_{M'-1}, u'_{M'}))$.

As we saw in class, Dijkstra's algorithm can be used to find shortest paths in weighted graphs, when there are non-negative edge weights. In this exercise, we'll see what happens when there *are* negative edge weights.

Consider the following version of Dijkstra's algorithm, which has been modified to return a shortest path. (Note, this is pseudocode, not working Python code, although it is adapted from the IPython notebook for Lecture 11).

```
# Finds a shortest path from s to t in G
def dijkstraShortestPath(G,s,t):
    Initialize a data structure d which uses vertices as keys and stores real numbers.
    d[v] = Infinity for all v in G.vertices
    d[s] = 0
    unsureVertices = G.vertices
    while len(unsureVertices) > 0:
        u = a vertex in unsureVertices so that d[u] is minimized
        if d[u] == Infinity:
            # then there is nothing more that I can reach
            break
        # update u's neighbors
        for v in u.getOutNeighbors():
            if d[u] + weight(u,v) < d[v]:
                d[v] = d[u] + weight(u,v)
                v.parent = u
        unsureVertices.remove(u)

    # now I have all the information I need to find the shortest path from s to t.
    if d[t] == Infinity:
        return "Can't reach t from s!"
    path = []
    current = t
    while current!= s:
        path.append(current)
        current = current.parent
    path.append(current)
    path.reverse()
    return path
```

For all the questions below, suppose that G is a directed, weighted graph, which may have negative edge weights, containing vertices s and t. Suppose that there *is* some path from s to t in G.

(a) **(2 pt.)** Give an example of a graph where there is a path from s to t, but no shortest path from s to t.

[We are expecting: A small example (fewer than 5 vertices) and an explanation of why there is not shortest path from s to t.]

(b) **(3 pt.)** Give an example of a graph where there *is* a shortest path from s to t, but `dijkstraShortestPath`(G, s, t) does not return one.

[We are expecting: A small example (fewer than 5 vertices) and an explanation of what `dijkstraShortestPath` does on this graph and why it does not return a shortest path.]

(c) **(3 pt.)** Your friend offers the following way to patch up Dijkstra's algorithm to deal with negative edge weights. Let G be a weighted graph, and let w^* be the smallest weight that appears in G. (Notice that w^* may be negative). Consider a graph $G' = (V, E')$ with the same vertices, and so that E' is as follows: for every edge $e \in E$ with weight w, there is an edge $e' \in E'$ with weight $w - w^*$. Now all of the weights in G' are non-negative, so we can apply Dijkstra's algorithm to that:

```
modifiedDijkstra(G,s,t):
     Construct G' from G as above.
     return dijkstraShortestPath(G',s,t)
```

Does your friend's approach work? (That is, does it always return a shortest path from s to t if it exists?) Either prove that it is correct or give a counter-example.

[We are expecting: Your answer, along with either a short proof or a counter-example.]