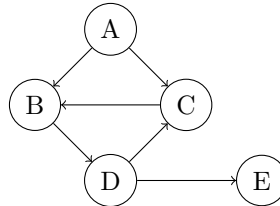


Exercises

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

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

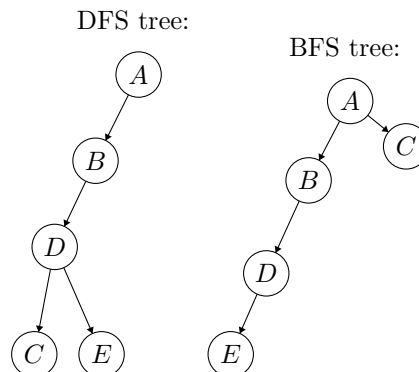


For the following parts you might want to use this website, which allows you to draw directed graphs in L^AT_EX: <http://madebyevan.com/fsm/>. (Note: On a Mac, fn+Delete will delete nodes or edges).

- (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.]

SOLUTION:



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 CS161Graphs 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 CS161Graphs 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 L^AT_EX.]

SOLUTION:

Graph A is bipartite and graph B is not. I had to make two changes to the BFS code. First, I had to change it to visit all vertices, even if they cannot all be reached from the starting vertex:

```
def fullBFS(w, G):
    for v in G.vertices:
        v.status = "unvisited"
    for v in G.vertices:
        if v.status == "unvisited":
            BFS(v, G)
```

Note: We were a little sneaky and made both graphs disconnected. B is a bipartite graph and a disjoint non-bipartite graph, while A is two disjoint bipartite graphs. So if you forgot to explore all connected components you may have gotten the wrong answer!

Next, I changed BFS to keep track of colors on alternate levels:

```
def BFS(w, G):
    n = len(G.vertices)
    Ls = [ [] for i in range(n) ]
    Ls[0] = [w]
    w.status = "visited"
    w.color = "blue"
    for i in range(n):
        for u in Ls[i]:
            for v in u.getOutNeighbors():
                if v.status == "unvisited":
                    v.status = "visited"
                    Ls[i+1].append(v)
                    if i%2 == 0:
                        v.color = "red"
                    else:
                        v.color = "blue"
    return Ls
```

And finally I checked to see if there were any pairs of neighbors with the same color after running fullBFS:

```
def isBipartite(G):
    fullBFS(G.vertices[0],G)
    for x in G.vertices:
        for y in x.getOutNeighbors():
            if x.color == y.color:
                return False
        for y in x.getInNeighbors():
            if x.color == y.color:
                return False
    return True
```

Then I got:

```
print(isBipartite(A),isBipartite(B))
True False
```

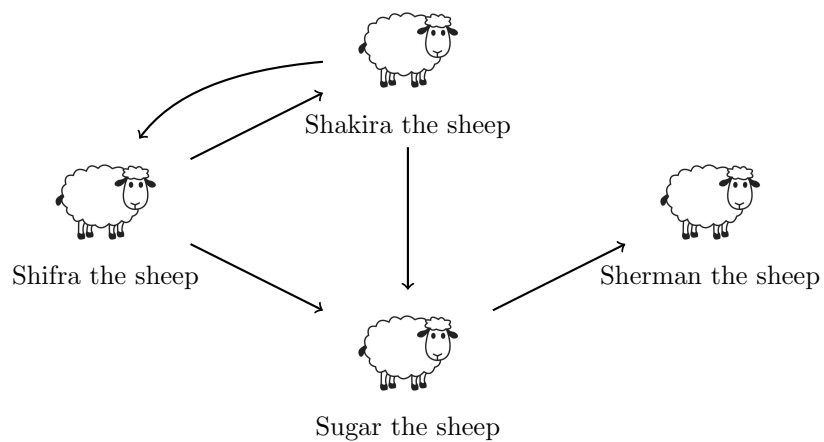
Problems

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 Baaahtter (it's like Twitter but for sheep¹). 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 $(a) \rightarrow (b)$ means that (b) will re-post anything that (a) posted. For example, if the social dynamics on the island were:

¹Also my new start-up idea



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 .

- (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 $O(n + m)$.]

SOLUTION:

- (a) First, we show that all source sheep are in the same SCC. Suppose that a and b are source sheep. Then there is a path from a to b , since b will eventually re-post anything a said. And, there is a path from b to a for the same reason. Then a and b are in the same strongly connected component. Since this holds for any pair of source sheep, all source sheep are in the same strongly connected component.

Next, we show that any sheep in that SCC is a source sheep. Suppose that a is a source sheep and there is another sheep c in the same SCC as a . Then c is also a source sheep: to see this, let d be any sheep on the island. There is a path from a to d (since a is a source sheep) and there is a path from c to a (since they are in the same SCC) so there is a path from c to d . Thus, any sheep in the same SCC as a source sheep is a source sheep.

- (b) There are at least two correct answers to this. The first is:

```
def getSourceSheep(G):
    Run DFS on G (starting from any sheep).
    Return the sheep with the largest finish time.
```

To see that this works, consider running DFS on the SCC-DAG that we defined in class. There is at least one source sheep, so let B be the SCC that contains all the source sheep, as guaranteed by part (a). Then, there is a path from B to every other SCC in the SCC-DAG. As we showed in class, the SCC with the biggest finish time is the first SCC in a topological sort of the SCC-DAG, so this means that B is the SCC with the largest finish time. Now, by definition of the finish time of an SCC, there is a sheep $b \in B$ with the same finish time as B , and this sheep is the one returned by the algorithm

above. Thus, the algorithm returns $b \in B$, and by definition of B every sheep in B is a source sheep. Thus the algorithm returns a source sheep.

Here's another way to do this, which is perhaps easier to reason about but is slower:

```
def getSourceSheep(G):
```

```
    Run the SCC algorithm from class to identify the SCCs.
```

```
    Run the topological sort algorithm from class on the SCC DAG.
```

```
    Return any sheep in the first SCC returned by the topological sort algorithm.
```

This algorithm does exactly the same thing the first algorithm at the end of the day, but it runs three runs of DFS (two for the SCC algorithm, one for topological sort) instead of one.

In either case, the running time is $O(n + m)$, the time to run DFS.

- (c) If we do not know whether or not there is a source sheep, we can do:

```
def getSourceSheep_ifItExists(G):
```

```
    a = getSourceSheep(G)
```

```
    run DFS starting from a
```

```
    if DFS reaches all sheep:
```

```
        return a
```

```
    return "No source sheep."
```

That is, if there is a source sheep, then our algorithm from part (b) will find it, and we will return it. If there is no source sheep, our algorithm from part (b) will return a sheep a which is not a source sheep. Then DFS will not reach all of the island from a , and we will return `No source sheep`.

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

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

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(s)
    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:

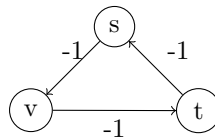
```
modifiedDijkstra(G,s,t):
    Construct G' from G as above.
    return dijkstraShortestPath(G',s,t)
```

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.]

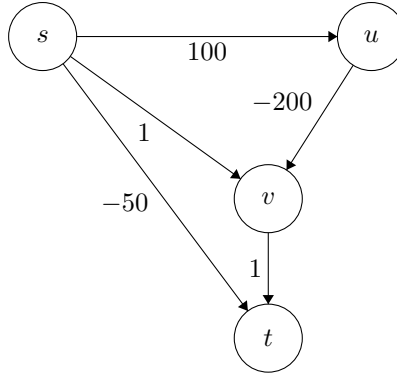
SOLUTION:

- (a) Here's an example:



Then there are paths of arbitrarily small negative cost (looping many times around the cycle), and a minimum-cost path does not exist.

- (b) Here is our example:

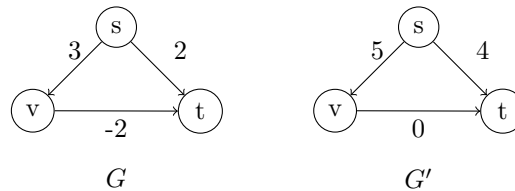


On this graph G , $\text{Dijkstra_st_path}(G, s, t)$ returns the path $s \rightarrow t$ which has cost -50 , while the shortest path (which does exist) is $s \rightarrow u \rightarrow v \rightarrow t$ which has cost -99 . To check this, we can trace through the functionality of the pseudo-code.

First, we will update from s , and set $d[t] = -50$ and $p[t] = s$, $d[v] = 1$, $p[v] = s$, and $d[u] = 100$, $p[u] = s$. Next we will choose t and then v to update; neither of these will change anything. Finally, we will choose u ; this updates $d[v] = -100$ and $p[v] = u$. However, at the end we have $d[t] = -50$ and $p[t] = s$, and so Dijkstra's algorithm will return that the shortest path goes from s to t (with a cost of -50), which is not correct.

Note: This problem is a bit tricky! If you don't include this edge from s to t , for example, then Dijkstra's algorithm (as above) will return the wrong value, but it will return the correct path.

- (c) Your friend's logic is pretty shaky. Underlying their approach is the assumption that the shortest path G is the same as the shortest path in G' . This is not true, as the following counter-example shows.



In G , the shortest path from s to t is through v , with cost 1. In G' , the shortest path is the edge directly from s to t , with cost 4. Thus, even though there is a shortest path from s to t in G , your friend's algorithm would fail and return the wrong path.