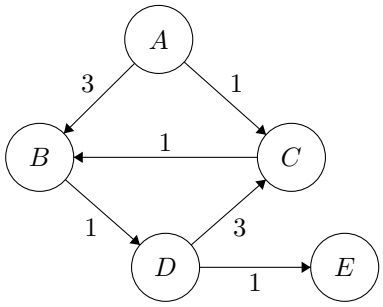
Exercises

Please do the exercises on your own.

1. **(6 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). It is also fine to include an image created in your favorite drawing program, or a photo/scan of a hand-drawn graph.

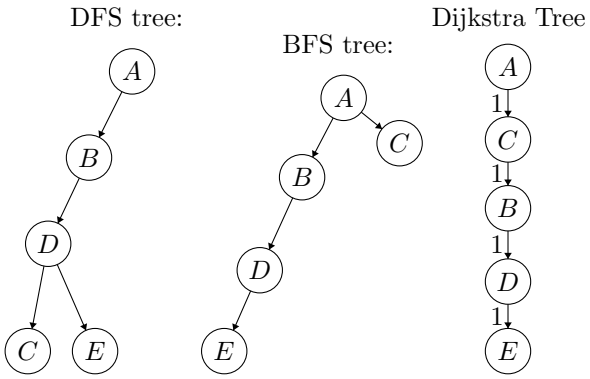
(a) **(2 pt.)** Draw the DFS tree for this graph, starting from node A. Assume that DFS traverses nodes in alphabetical order. (That is, if it could go to either B or C, it will always choose B first).

(b) **(2 pt.)** Draw the BFS tree for this graph, starting from node A. Assume that BFS traverses nodes in alphabetical order.

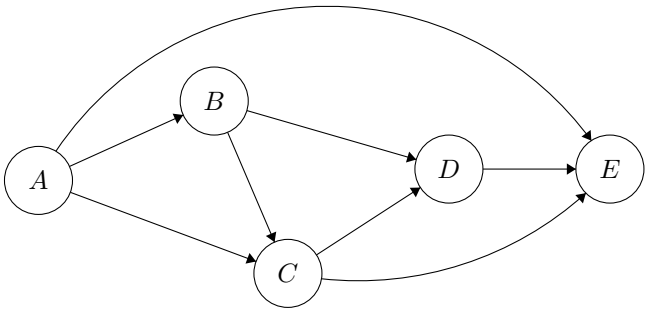
(c) **(2 pt.)** Draw the "Dijkstra's algorithm tree" for this graph, starting from node A. Assume that Dijkstra's algorithm breaks ties in alphabetical order.

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

SOLUTION:



2. **(4 pt.)** Consider the following directed acyclic graph (DAG):



In class, we saw how to use DFS to find a topological ordering of the the vertices; in the graph above, the unique topological ordering is A, B, C, D, E. We saw an example where we happened to start DFS from the first vertex in the topological order. In this exercise we'll see what happens when we start at a different vertex. Recall that when you run DFS, if it has reached everything it can but hasn't yet explored the graph, it will start again at an unexplored vertex.

(a) Run DFS starting at vertex C, breaking any ties by alphabetical order.[1]
   i. What do you get when you order the vertices by **ascending** start time?
   ii. What do you get when you order the vertices by **descending** finish time?

(b) Run DFS starting at vertex C, breaking any ties by **reverse** alphabetical order.[2]
   i. What do you get when you order the vertices by **ascending** start time?
   ii. What do you get when you order the vertices by **descending** finish time?

[**We are expecting:** *For all four questions, an ordering of vertices. No justification is required.*]

SOLUTION:

(a) Alphabetical order:
   i. By start time: C, D, E, A, B
   ii. By finish time: A, B, C, D, E

(b) Reverse alphabetical order:
   i. By start time: C, E, D, B, A
   ii. By finish time: A, B, C, D, E

[1] For example, if DFS has a choice between B or C, it will always choose B. This includes when DFS is starting a new tree in the DFS forest.

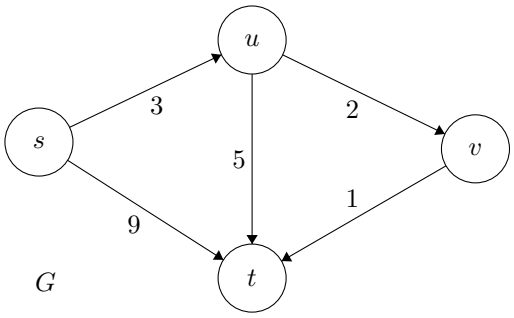
[2] For example, when DFS has a choice between B or C, it will always choose C. This includes when DFS is starting a new tree in the DFS forest.

3. **(6 pt.)** In class, we saw pseudocode for Dijkstra's algorithm which returned shortest distances but not shortest paths. In this exercise we'll see how to adapt it to return shortest paths. One way to do that is shown in the pseudocode below:

```
Dijkstra_st_path(G, s, t):
    for all v in V, set d[v] = Infinity
    for all v in V, set p[v] = None
    // we will use the information p[v] to reconstruct the path at the end.
    d[s] = 0
    F = V
    D = []  // D is the list of "done" vertices
    while F isn't empty:
        x = a vertex v in F so that d[v] is minimal
        for y in x.outgoing_neighbors:
            d[y] = min( d[y], d[x] + weight(x,y) )
            if d[y] was changed in the previous line, set p[y] = x
        F.remove(x)
        D.add(x)

    // use the information in p to reconstruct the shortest path:
    path = [t]
    current = t
    while current != s:
        current = p[current]
        add current to the front of the path
    return path, d[t]
```

Step through `Dijkstra_st_path`(G, s, t) on the graph G shown below. Complete the table below (on the next page) to show what the arrays `d` and `p` are at each step of the algorithm, and indicate what path is returned and what its cost is. If it is helpful, the LaTeXcode for the table is reproduced at the end of the PSET.



[**We are expecting:** *The following things:*

- *The table below filled out*
- *The shortest path and its cost that the algorithm returns.*

No justification is required.]

	$\mathtt{d}[s]$	$\mathtt{d}[u]$	$\mathtt{d}[v]$	$\mathtt{d}[t]$	$\mathtt{p}[s]$	$\mathtt{p}[u]$	$\mathtt{p}[v]$	$\mathtt{p}[t]$
When entering the first while loop for the first time, the state is:	0	∞	∞	∞	None	None	None	None
Immediately after the first element of D is added, the state is:	0	3	∞	9	None	s	None	s
Immediately after the second element of D is added, the state is:								
Immediately after the third element of D is added, the state is:								
Immediately after the fourth element of D is added, the state is:								

SOLUTION:

	$\mathtt{d}[s]$	$\mathtt{d}[u]$	$\mathtt{d}[v]$	$\mathtt{d}[t]$	$\mathtt{p}[s]$	$\mathtt{p}[u]$	$\mathtt{p}[v]$	$\mathtt{p}[t]$
When entering the first while loop for the first time, the state is:	0	∞	∞	∞	None	None	None	None
Immediately after the first element of D is added, the state is:	0	3	∞	9	None	s	None	s
Immediately after the second element of D is added, the state is:	0	3	5	8	None	s	u	u
Immediately after the third element of D is added, the state is:	0	3	5	6	None	s	u	v
Immediately after the fourth element of D is added, the state is:	0	3	5	6	None	s	u	v

The final path is $s \to u \to v \to t$ and the final cost is 6.

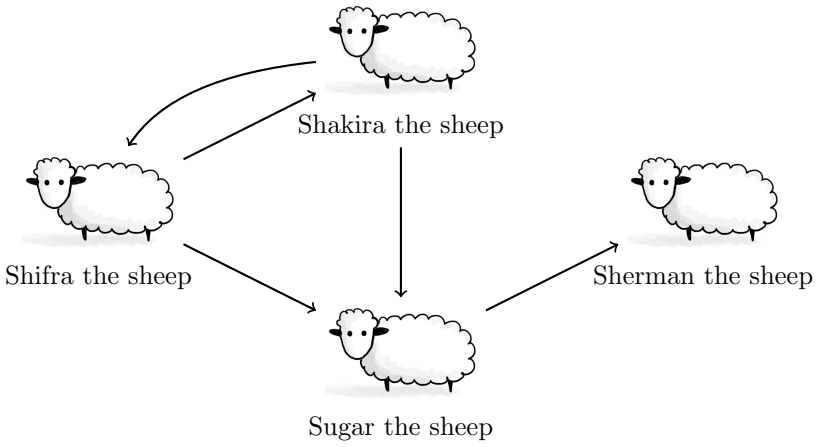
Problems

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.

---

4. **(8 pt.)** (**Wake up, Sheeple!**) 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[3]). 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:



then 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 directed, unweighted graph on the n sheep. Let m denote the number of edges in G.

(a) **(2 pt.)** Call a sheep an **influencer** 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 influencers.

Prove that all influencers are in the same strongly connected component of G, and every sheep in that component is an influencer.

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

[3] Also my new start-up idea

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

[**We are expecting:** *The following things:*

- *Pseudocode or a very clear English description of your algorithm*
- *an informal justification that your algorithm is correct*
- *an informal justification that the running time is* $O(n+m)$

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 an influencer. Give an algorithm that runs in time $O(n+m)$ and either returns an influencer or returns `no influencer.` You may use any algorithm we have seen from class as a subroutine, and you may also use your algorithm from part (b) as a subroutine.

[**We are expecting:** *The following things:*

- *Pseudocode or a very clear English description of your algorithm*
- *an informal justification that your algorithm is correct*
- *an informal justification that the running time is* $O(n+m)$

You may use any statement we have proved in class without re-proving it.]

SOLUTION:

(a) First, we show that all influencers are in the same SCC. Suppose that that a and b are influencers. 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 influencers, all influencers are in the same strongly connected component.
Next, we show that any sheep in that SCC is an influencer. Suppose that a is an influencer and there is another sheep c in the same SCC as a. Then c is also an influencer: to see this, let d be any sheep on the island. There is a path from a to d (since a is an influencer) 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 an influencer is an influencer.

(b) There are at least two correct answers the 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 influencer, so let B be the SCC that contains all the influencers, 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 an influencer. Thus the algorithm returns an influencer.
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 an influencer, we can do:

```
def getSourceSheep_ifItExists(G):
     a = getSourceSheep(G)
     run DFS starting from a
     if DFS reaches all sheep:
           return a
     return "No influencer."
```

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

5. **(5 pt.)** (**Dijkstra with negative edges**) For both of the questions below, suppose that G is a connected, directed, weighted graph, which may have negative edge weights, containing vertices s and t, and refer to the pseudocode for `Dijkstra_st_path` from Exercise 3. 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. (Note that in a directed graph, a *path* must follow the direction of the edges; recall that a *shortest path* is one which minimizes the sum of the edge weights along that path).

   [**We are expecting:**

   - *A small example (at most* 5 *vertices)*
   - *An explanation of why there is no 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 `Dijkstra_st_path`(G, s, t) does not return one.

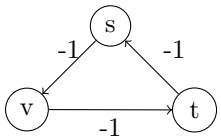
   [**We are expecting:**

   - *A small example (at most* 5 *vertices)*
   - *An explanation of what* `Dijkstra_st_path` *does on this graph and why it does not return a shortest path.*

   ]

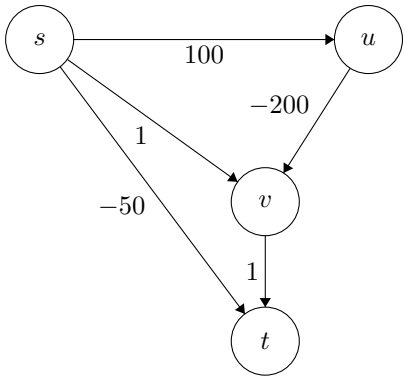
SOLUTION:

(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, `Dijkstra_st_path`(G, s, t) returns the path $s \to t$ which has cost -50, while the shortest path (which does exist) is $s \to u \to v \to t$ 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.

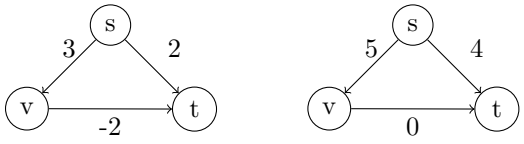
6. **(4 pt.)** (**A fix for Dijkstra?**) During class (Lecture 11), someone suggested the following fix to make Dijkstra's algorithm to deal with negative edge weights. Let $G = (V, E)$ be a weighted graph with negative edge weights, and let w^* be the smallest (most negative) weight that appears in G. Consider a graph $G' = (V, E')$ with the same vertices as G. Then to construct the edges E', we do the following: for every edge $e \in E$ with weight w, we add 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 Dijkstra_st_path(G',s,t)
```

In class, Prof. Wootters said it wouldn't work but now she's not so sure...does this suggestion work? (That is, does it always return a shortest path from s to t in G if it exists?) Either prove that it is correct (that is, prove that this algorithm correctly finds shortest paths in weighted directed graphs), or give a counter-example.

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

SOLUTION: Despite being a very reasonable suggestion, this does not work! The reason is that the shortest path G is not always the same as the shortest path in G'. For example:



G $\qquad\qquad$ G'

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, this algorithm would fail and return the wrong path.

Helpful LaTeXcode

Here is the code for the table from Exercise 3:

```
\begin{center}
\def\arraystretch{1.5}
\newcommand{\td}{\texttt{d}}
\newcommand{\tp}{\texttt{p}}
\begin{tabular}{|p{6cm}||c|c|c|c||c|c|c|c|}
\hline
& \td[$s$] & \td[$u$] & \td[$v$] & \td[$t$] & \tp[$s$] & \tp[$u$] & \tp[$v$] & \tp[$t$] \\
\hline
When entering the first while loop for the first time, the state is:&
0 & $\infty$ & $\infty$ & $\infty$ & None & None & None & None \\
\hline
Immediately after the first element of $D$ is added, the state is: &
0 & $3$ & $\infty$ & $9$ & None & s & None & s \\
 \hline
Immediately after the second element of $D$ is added, the state is: &
 & & & & & & & \\
 \hline
Immediately after the third element of $D$ is added, the state is: &
 & & & & & & & \\
 \hline
Immediately after the fourth element of $D$ is added, the state is: &
 & & & & & & & \\
 \hline
\end{tabular}
\end{center}
```