CS 161 **Problem Set 5**
Fall 2025 **Due:** Friday November 14, at 11:59pm on Gradescope.

Style guide and expectations: Please see the top of the "Homework" page on the course webpage for guidance on what we are looking for in homework solutions. We will grade according to these standards.

Make sure to look at the "**We are expecting**" blocks below each problem to see what we will be grading for in each problem!

Collaboration policy: You may do the HW in groups of size up to three. Please submit one HW for your whole group on Gradescope. (Note that there is an option to submit as a group). See the "Policies" section of the course website for more on the collaboration policy.

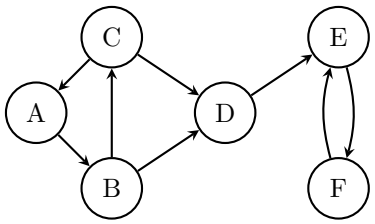
LLM policy: Check out the course webpage for best practices on how to productively use LLMs on homework, if you use them at all.

Exercises

We recommend you do the exercises on your own before collaborating with your group. The point is to check your understanding.

---

1. **(3 pt.)** Consider the following graph G:



   (a) **(1 pt.)** What is the SCC DAG (aka the *SCC meta-graph*) for G?
   [**We are expecting:** *A drawing or extremely clear description of the SCC meta-graph.*]

   (b) **(2 pt.)** Recall that Kosaraju's algorithm to identify SCCs does one run of DFS, reverses all the edges, and then does a second run of DFS. If you run Kosaraju's algorithm on G, which SCC in your SCC DAG does the *second* run of DFS start in? (That is, it starts at some vertex v of G. Which meta-vertex of the SCC DAG does v live in?)
   [**We are expecting:** *Clearly identify which SCC the second DFS run must start in. No explanation is required, but it would be good to think about why there is a unique answer to this question, regardless of where you started the first DFS run.*]

SOLUTION:

(a) The SCC DAG is $(ABC) \to (D) \to (EF)$, where (ABC) means "the SCC that contains A, B, and C", (D) is "the SCC with only D" and (EF) is "the SCC with E and F."

(b) The second run of DFS starts in the node (ABC). (Depending on where the first run of DFS starts, the second run may start at A or B or C, but it always must start in this (ABC) DAG.)

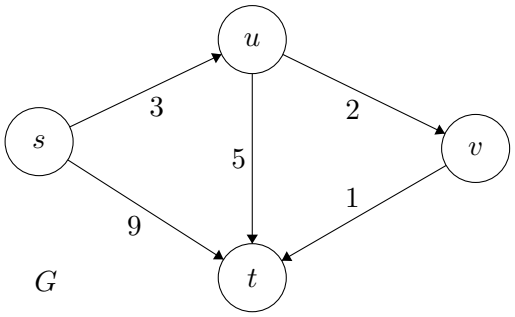
No explanation is required, but one way to see this is that this is the SCC that is first in the unique topological order on the SCC DAG. Another way to see it is that if you were to reverse all the edges, (ABC) is the node in the SCC DAG that would have no outgoing edges.

2. **(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
    while F isn't empty:
        x = a vertex v in F with minimum d[v]
        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)

    // 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 in the LaTeX template.



continued on next page...

[**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 removing the first element from F, the state is:	0	3	∞	9	None	s	None	s
Immediately after removing the second element from F, the state is:								
Immediately after removing the third element from F, the state is:								
Immediately after removing the fourth element from F, the state is:								

SOLUTION:

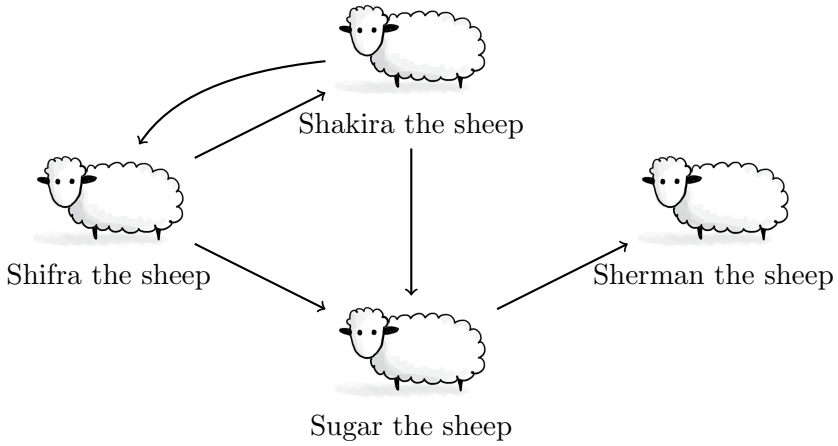
	$\mathtt{d}[s]$	$\mathtt{d}[u]$	$\mathtt{d}[v]$	$\mathtt{d}[t]$	$\mathtt{p}[s]$	$\mathtt{p}[u]$	$\mathtt{p}[v]$	$\mathtt{p}[t]$
When entering the first while loop for the first time, the state is:	0	∞	∞	∞	None	None	None	None
Immediately after removing the first element from F, the state is:	0	3	∞	9	None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

---

3. **(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 BaahGram (it's a platform for sheep to share photos of scenic pastures, new shearing styles, and so on[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:



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.*]

SOLUTION:

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

[1] Also my new start-up idea `#bleatthealgorithm`

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) **(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.]

SOLUTION:

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) **(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:

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.`

4. **(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 2. 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).

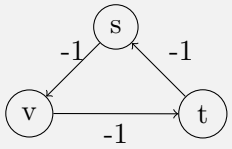
   [**We are expecting:**

   - *A small example (at most 5 vertices)*
   - *An explanation of why there is no shortest path from s to t.*

   ]

   SOLUTION:

   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) **(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.

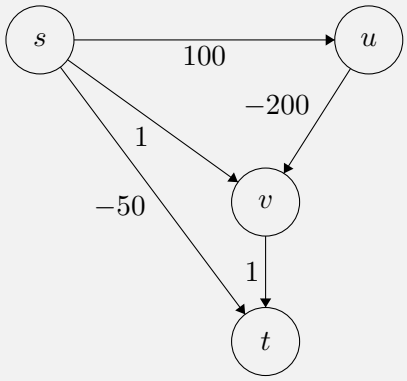
[**We are expecting:**

- *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:

(c) 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.

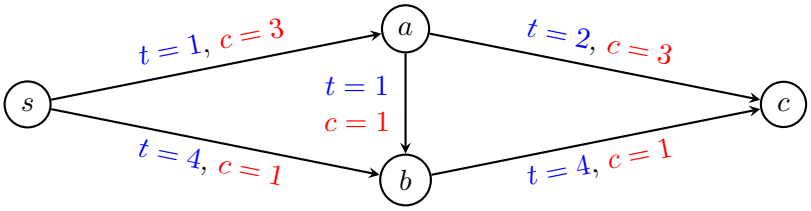
First, we will update from s, and set $d[t] = -50$ and $p[t] = s, d[v] = 1, p[v] = s$, and $d[u] = 100$, $p[u] = s$. Next we will choose t and then v to update; neither of these will change anything. Finally, we will choose u; this updates $d[v] = -100$ and $p[v] = u$. However, at the end we have $d[t] = -50$ 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.

5. **(10 pt.) [The Shortest Green-Enough Path]** Let $G = (V, E)$ be a directed graph, where each edge (u, v) has *two* weights. It has a weight $t(u, v)$ that represents the time (in minutes) it takes to traverse that edge; and a weight $c(u, v)$ that represents the carbon footprint cost (in kilograms of CO_2) it takes to traverse that edge. Suppose that for all $(u, v) \in E$, $t(u, v)$ and $c(u, v)$ are non-negative integers.

   Your goal is to solve the following version of the single-source shortest-path problem: Given a starting vertex s, and a carbon budget B (a positive integer), find, for all vertices $v \in V$, the shortest path (in terms of time) from s to v *subject to the constraint* that the total cost of the path (in terms of carbon) is *strictly less* than B kg of CO_2.

   For example, let G be the graph below:



   On this graph with a budget of $B = 6$ and starting vertex s, your algorithm should return the following information (in a format of your choosing, it doesn't need to be a table):

Destination	Time (min)
a	1
b	2
c	6

   For example, to get from s to c, we should travel along the path $s \to a \to b \to c$, which takes time 6 minutes and costs 5 kilograms of CO_2. The path $s \to a \to c$ is faster (time only 3 minutes), but the cost is $6 \geq B$ kilograms, which is too large.

   Your algorithm should run in time $O((Bn + Bm)\log(Bn))$, where n is the number of vertices in G and m is the number of edges. You may (and, hint, may want to) use any algorithm we have seen in class as a black box.

   [**Hint:** *Consider modifying $G = (V, E)$ to create a new graph $G' = (V', E')$ with Bn vertices, by replacing each vertex $v \in V$ with B copies of v, $(v, 0), (v, 1), \ldots, (v, B-1) \in V'$. Think of the vertex $(v, j) \in V'$ as "the vertex $v \in V$ in the case that it took j kilograms of CO_2 to get to v from s". Then replace each edge in E with up to B edges in E'... How would you place these edges in a way that is consistent with the hinted interpretation of (v, j) above?*]

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

   - *A clear description of your algorithm;*
   - *An explanation of why it is correct;*
   - *A short justification of the running time.*

   You do not need to give pseudocode, but you may if you think it makes your solution clearer. Your explanation doesn't need to be a formal proof, but it should be clear to the grader.]

SOLUTION: Algorithm:

Following the hint, let $G' = (V', E')$ so that for all $v \in V$, there are B copies $(v, 0), \ldots, (v, B-1)$ of v in V'. For each edge $(u, v) \in G'$ with time cost t and carbon cost c, add an edge from (u, i) to $(v, i + c)$ with time cost t whenever $c + i < B$. (And if $c + i \geq B$, don't add any edge for that i).

Now run Dijkstra's algorithm on G' with the starting vertex $(s, 0)$. This returns a data structure that stores the vertices (v, i) and with keys $d[(v, i)]$, so that $d[(v, i)]$ is the cost of the shortest path from $(s, 0)$ to (v, i) in G'. Then initialize an array D, indexed by the vertices of V, and set

$$D[v] \leftarrow \min_{i \in \{0,1,\ldots,B-1\}} d[(v, i)].$$

Then return D, and assert that $D[v]$ is the shortest path from s to v with budget at most B.

Justification of correctness:

To see why this works, consider any path in G'. It looks something like $(s, 0) \to (v_1, c_1) \to (v_2, c_2) \to \cdots \to (v_r, c_r)$. Moreover, c_r is the carbon cost of this path. Indeed, if we took an edge from $(s, 0)$ to (v_1, c_1), then $c_1 = 0 + c(s, v_1) = c_1$ by construction; and then $c_2 = c_1 + c(v_1, v_2) = c(s, v_1) + c(v_1, v_2)$, and so on.

The above logic shows that paths from $(s, 0)$ to (v, c) in G' correspond to paths from s to v in G with cost $c < B$. Thus, the shortest B-constrained path from s to v in G corresponds to the shortest path from $(s, 0)$ to (v, c) for any c. And this is precisely what our algorithm returns.

Justification of running time:

The running time is the time it takes to (a) create G'; (b) run Dijkstra on G'; and (c) loop over the output d to construct D. (a) takes time $O(B(n+m))$ with an adjacency-list format, since we need to loop over all the vertices and all the edges, and add up to B copies of each. (b) takes time $O((Bn + Bm)\log(Bn))$ (using the implementation of Dijktra with a Red-Black Tree). For (c), initialize D to be an array of all ∞. Then read out all of the (key, item) pairs $(d[(v, i)], (v, i))$ from the RBTree (eg with in-order traversal, which takes $O(nB)$ time). As we loop over all these items, whenever we see $(d[(v, i)], (v, i))$, we update $D[v] \leftarrow \min(D[v], d[(v, i)])$. So step (c) takes time $O(Bn)$ total. (Note: This amount of detail is not required for full credit—it's okay to say, e.g., "for (c), we loop through all of the items once so that's $O(nB)$").

Altogether, (b) dominates, and the running time is $O(B(n + m)log(Bn))$, as desired.

6. **(6 pt.) The Shortest Greenest Path (Ethics):** Given your expertise after having completed Problem 5, a ride-sharing company taps you to help reduce its carbon footprint in a particular city. The company represents the city as a weighted graph, where each edge weight represents the expected travel time between intersections. The company also wants to factor in carbon emissions on each route segment, and as in the previous problem they have an estimate of this as well. But when the engineers try to combine the two into a single "cost" to minimize, they struggle to decide how to weigh one against the other. They could do what you did in Problem 5 and impose a total carbon budget B. But is that the "right" thing to

do? Is there any "right" way to combine the two costs?

(a) **(3 pt.)** Using the concept of **incommensurability** from one of the Embedded Ethics lectures, explain why the team has difficulty deciding how to combine travel time and carbon footprint into a single weight.
[**We are expecting:** *We are expecting: 1–2 sentences explaining why these values are incommensurable.*]

(b) **(3 pt.)** The Embedded Ethics lecture discusses other real-world examples of algorithms that treat incommensurable values as if they were commensurable, like credit scoring. Give another real-world example where an algorithmic model treats incommensurable values as if they were commensurable. What ethical risks arise when distinct kinds of value are collapsed into one scale?
[**We are expecting:** *We are expecting: 2–4 sentences describing an example algorithmic model and one risk of value simplification.*]

SOLUTION:

(a) *Responses should include a variation of the following:*
Incommensurability refers to the lack of a common value measurement. In this case, travel time and carbon footprint are measured in fundamentally different units, making them incommensurable. Mapping one value to another requires a conversion factor: for example, how many minutes of travel time are equivalent to one kilogram of carbon emission. Each stakeholder (e.g. ride-share company, environmental regulators, urban planners) involved in the development process will normatively prescribe a weight that reflects their own value judgement. Therefore, there is no "right" way to define a weight (*"cost"*) to minimize. Implementing trade-off analyses in the value measurement process will allow stakeholders to select a weight that aligns with their priorities.

(b) *Responses should include a variation of the following:*
An example of an algorithm that treats incommensurable values as if they were commensurable is **college admissions systems**. This algorithmic model combines diverse factors - such as academic performance, personal essays, extracurriculars, and demographic characteristics - into a singular score that determines their admission decisions. These factors reflect educational opportunity and intellectual ability as incommensurate values, yet are condensed into a single metric for admission. Collapsing these qualities into a single metric risks unfair outcomes, as it obscures the value judgement that is embedded wtihin the model.

Another example is **social media engagement algorithms**. These models recommend content based on engagement metrics (likes, comments, time spent watching a video). In doing so, these models treat incommensurate values, like information accuracy and emotional reaction to content, as equal values of engagement. This value judgement risks the proliferation of sensationalized media, which undermines the validity of online information.