Homework 2: BFS, DFS, and Graphs

Due Date: The written parts of this assignment will be due at 11:59 PM on Friday January 26. As always, that means that the TAs will grade **every** written problem you submit to gradescope before that deadline. You will also be able to resubmit two written problems with every assignment, starting with this homework.
Collaboration: You are allowed (and encouraged!) to discuss these problems at a high level with others. But, please read the full collaboration policy to ensure you are keeping your discussions at the right level, and remember to cite any collaborators.

Problems to Submit: We will count your 1 best mechanical question and your 3 best long-form questions.
We strongly recommend you skim all the problems in a section before attempting them. While we try to make problems in a given section of approximately equal difficulty, you may find some to be easier than others.

Directions Unless otherwise noted, you are allowed to use algorithms described in class (or prerequisite courses) as though you had library implementations of them. For example, you can say "run the BFS-based 2-coloring algorithm from class on the graph G" or "run the bipartite checking algorithm from lecture 5 on G." We also have a list of data structures and algorithms you can use from 373 here.

Since this is a course about efficient algorithms, algorithms that are slower than necessary are unlikely to get "E" scores, but (unless otherwise noted) we do not care about constant factors. In general, an algorithm that is correct but (mildly) slower than optimal will get an equal or higher score to an algorithm that is fundamentally incorrect, but fast.

Mechanical Problems

Both of the mechanical questions on this homework are about cut edges.

Definition: Cut Edge

In an undirected graph, a *cut edge* (also called a "bridge") is an edge such that if the edge is removed, the number of connected components will increase.

The mechanical questions in this assignment are designed to hint at an algorithm that discovers all cut edges in a graph.

First some background: when we run DFS in an undirected graph, we classify an edge based on how it was first discovered. So if we discover u first, then follow an edge to v, we think of (u, v) as going from u to v (even though the edge itself is undirected). When we process v, we don't go back along u, so we don't try to change its classification.

1. A Short Proof

Prove that in an undirected graph, if an edge is a cut edge, then it must be a tree edge in DFS. (For the sake of simplicity, you may assume that the original graph is connected.)

More formally, suppose u and v are the endpoints of a cut edge in an undirected graph, and let w be an arbitrary vertex (i.e. it doesn't necessarily need to be u or v). Argue that said cut edge must be a tree edge with respect to *any* DFS tree rooted at w.

Our proof is about 4 sentences. Here are some tips to guide you:

Hint 1: convince yourself that there are no cross or forward edges when DFS is run on an undirected graph (you may use this result without supplying a formal proof).

Hint 2: consider doing a proof by contrapositive. While we tend to think of the DFS tree as having directed edges, it might be helpful here to surpress the directedness and instead think of it as a connected subgraph of the original undirected graph that only has edges that have been classified as tree edges, *i.e. any two vertices have a (undirected) path between them consisting entirely of tree edges*.

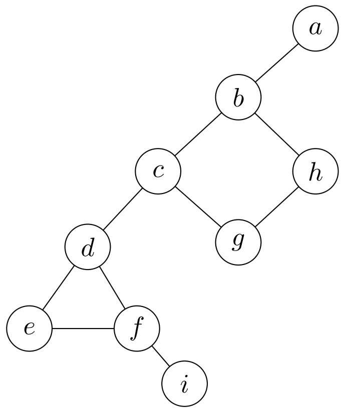
2. Run the Algorithm Yourself

To help us find out whether an edge is a cut edge, it will help to know how "high up" in the tree you could get from u by going down tree edges and taking a back edge up. Define the quantity "low" on each vertex:

$$\texttt{u.low} = \min\left\{\texttt{u.start}, \min_{v:(u,v) \text{ is a back edge}} \{\texttt{v.start}\}, \min_{d:d \text{ is a descendant of } u} \{\texttt{d.low}\}\right\}$$

In words, `u.low` is the minimum of: `u.start`, the `start` values of vertices that are (directly) connected to u by a back edge and the `low` values of the descendants of u in the DFS tree (i.e. any vertex which will be seen for the first time while u is on the stack)

(a) Run DFS starting from a in this graph. Break ties by putting the alphabetically first one on the stack first. Record `start`, `end`, `low` for each vertex.



(b) An edge is a cut edge if:

(i) It is a tree edge, discovered going from u to (new discovery) v, and

(ii) `v.low` is greater than `u.start`

List the cut edges in the graph above. We recommend you use this problem as a chance to check your numbers in the previous part!

Long-form Problems

3. Modify DFS

(a) Write pseudocode for a modification of DFS that takes as input a directed graph G and returns `true` if G is strongly connected (i.e., if you can get from every vertex to every other, and back the other way), and `false` otherwise.

For this problem, we want to see you actually make the modifications (e.g., you may not use another graph algorithm as a black box for this problem), but you may add any fields you like to the vertices and edges.

You can use code like `DFSWrapper` (from Lecture 7, slide 11) to control how DFS is called, but you must use only one run of DFS (or `DFSWrapper`)

Hint: You might use something like the `low` numbers from problem 2.

(b) Explain how your algorithm works – for any fields or variables you added, briefly describe what they do, and add a sentence or two to describe overall how the field(s)/variable(s) work together to get you the correct output. You do **not** need a full proof.

4. Flowers

You have designed a machine learning system to classify different flowers – given a picture of a flower, the system prints out what type of flower it thinks it is.

You'd like to build a new dataset to test your system on; you've scraped the internet for TONS of flower pictures. A few of the pictures came "pre-labeled" with the species of flower, but many of them weren't labeled. And they won't make for a good test set without knowing what the right answer is supposed to be. In this problem we'll figure out how to quickly extend a few accurate labels to the rest of the pictures we've scraped.

You can use an online survey system (like Mechanical Turk) to get humans to identify the pictures for you, but the average person isn't an expert on flowers, and might not identify the flower in a picture accurately. What humans can do consistently is tell whether two pictures of flowers are the same type or not. So you set up your survey to show people a few flower pictures at the same time. The user will then identify (exactly) two of the pictures they were shown to say they believe those flowers are the same. You'll have each survey taker repeat this process for many sets of pictures (each time identifying exactly two flowers of the same species).

You get the responses from your first survey participant. But you know that surveys like this aren't reliable – some people just click randomly (or are really bad at identifying flowers). You'd like to make sure a single person's responses are **consistent** with your starting data. For example, that from one person's data you can't find them saying picture A is the same species as picture B, that B is the same species as C, and that both A and C are pre-labeled flower pictures that you know are different species.

More formally, a person's responses are consistent if there is no sequence of pictures $p_1, \ldots, p_k$ such that p_1 and p_k are pre-labeled with different species, and for every i, the user marked p_i and p_{i+1} as the same species.

Sample Input I: Pre-labeled images: A is a rose, B is a lily, C is a orchid, D,E,F,G are unlabeled
User input: A and D are the same species, C and F are the same species, E and G are the same species, D and F are the same species.

Correct Output: Inconsistent

Sample Input II: Pre-labeled images: A is a rose, B is a rose, C is a lily, D,E,F,G are unlabeled
User input: A and B are the same species, A and E are the same species, F and G are the same species, D and F are the same species.

Correct Output: Consistent

Describe an algorithm to check if someone's responses are consistent with the starting data. When analyzing the running time, assume you have p unlabeled pictures, ℓ labeled-pictures, and that the participant has made k observations (i.e. identified k pairs as each being the same species).

(a) Describe a graph you can use in this problem:

(i) What are your vertices?

(ii) What are your edges?

(iii) Briefly justify that given your starting data you could build your graph, and state how much time it would take (in terms of p, ℓ, k). (Our justification is 1 sentence; depending on what you choose for your graph, you may require more justification)

(b) Describe an algorithm to solve this problem. For this part, you should assume that the graph that you described in part (a) is already built; **do not rebuild it again here.**

(c) Give an informal argument *separate from your algorithm description* for why your algorithm produces the correct output (reports "consistent" when the observations really are consistent and reports "inconsistent" when the observations really are inconsistent, or something logically equivalent).

5. Some Snakes are Sus

You are an algorithm developer who has recently been contracted by a newbie herpetologist (a zoologist who specializes in the study of amphibians and reptiles). Unfortunately, because of their limited experience, they can't tell the difference between eastern coral snakes and scarlet kingsnakes. Eastern coral snakes are venomous while scarlet kingsnakes aren't, so it's really important you can help them classify these two snake species correctly.

Thankfully, the herpetologist *is* competent enough to tell if two snakes are the same species as each other or not. This is the only information you have for designing the classification tool.

For example, given two photos they can say that "those are the same snake" or "those are different snake." But are not able to look at a photo and say "that's a scarlet kingsnake." As a result, you do a bunch of comparisons and write them all down; finally, you pick one image and say, "this is an A snake." You goal for this tool is to perfectly identify each snake image as part of either snake group (snake A or snake B).

Given your data, you need to respond with one of three options:

- Inconsistent: you have analyzed the pictures in such a way that you can't possibly classify them all into snake A and snake B correctly (for example the herpetologist said photo 1 and 2 were the same snake species, photo 2 and 3 were different snakes but 1 and 3 were the same snakes).
- Underspecified: your answers aren't inconsistent, but there is at least one picture that (from the data you've collected so far) could validly be either snake A or snake B. Intuitively, this means there is an image that is not connected (even indirectly) to the image you started with.
- Exact answer: your answers aren't inconsistent, and every photo can be only one of snake A or B.

Describe an algorithm to return which of those three cases the input matches, and if you're in the exact answer case to store the species labels for each photo.

Sample Input I: There are 4 images.
Image 1 and 2 are different snakes
Image 3 and 4 are the same snakes
Image 1 is "snake A"

Correct Output: Underspecified

Sample Input II: There are 4 images.
Image 1 and 2 are different snakes
Image 1 and 4 are the same snakes
Image 3 and 4 are different snakes
Image 1 and 3 are different snakes
Image 1 is "snake A"

Correct Output: Exact Answer

Sample Input III: There are 4 images.
Image 1 and 2 are different snakes
Image 1 and 4 are the same snakes
Image 3 and 4 are different snakes
Image 1 and 3 are the same snakes
Image 1 is "snake A"

Correct Output: Inconsistent

(a) Give pseudocode to run for this problem. You need to construct at least one graph (but you might want to make more than one, or alter the first one you make), you can assume creating a vertex or an edge takes $O(1)$ time each.

(b) Describe the intuition behind your algorithm (in a few sentences of English). You don't need a full "proof" here, but you should spend some time describing what your vertices/edges are and a brief description of why the algorithm(s) you run are doing the correct thing.

(c) Give the running time of your algorithm. For analyzing the running time, assume that you have p photos, s pairs identified as "the same" and d pairs identified as "different." Give a big-$\mathcal{O}$ bound in terms of p, s, d.

Note that your input in this problem is different from the last problem. In the last problem, you only got information of "these two are the same" (there was no way to identify two pictures as different). Here, you will have some pairs identified as "the same as each other" and others identified as "different from each other."

This problem has multiple valid solutions – just because you and another student are coming at the problem differently, don't assume that only one of you can have a right idea!

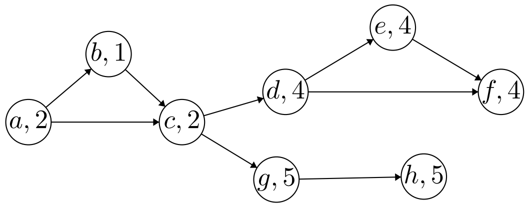
6. Points!

You are given a **directed** graph G, and a start vertex u. Every vertex in the graph contains a non-negative number of `points` you can collect on the first time you visit the vertex (if you visit the vertex again you get no points the second time, but there's no penalty). You would like to figure out how many points you could collect in the graph.

For this question, you do not have to prove your algorithms correct for any part. Instead, you should explain why you think it works in 2-3 sentences per part.

(a) Suppose G is strongly connected. Describe a walk that collects the maximum possible points (you don't need the shortest walk, just a walk). Describe an algorithm to find the number of points in this walk. **Hint:** Don't find the walk itself! It takes longer to find the walk than to find its value.

(b) Suppose G is a DAG. Describe an algorithm to find the walk that collects the most points. We recommend you consider the example below, where the best walk is: a, b, c, d, e, f. (hint: This step should also take $\mathcal{O}(V + E)$ time)



(c) Now, stitch together the last two parts and describe an algorithm to handle any directed graph.

7. Running out of rooms

In Madison, Wisconsin[1] essentially every apartment lease ends on August 14 with new leases beginning on August 15th. Dissatisfied with the current system (which leaves some people sleeping with their belongings in U-hauls overnight while waiting to move into their new apartment[2]), the City of Madison has given you the power to fix the problem. You will receive a list of all n people moving in the city, their current apartment (or `null` if they are newly moving to Madison from far away), and their new apartment (or `null` if they are moving far away from Madison). Your goal is to find an ordering of people that will allow for *easy movement*. By that we mean, for every person, by the time you reach them in the list, the apartment they are moving to will have already been vacated by its current occupant (if any), ensuring no one ends up taking a nap in a U-haul. It may be that some apartments are currently vacant (or will become vacant after the move is completed); if an apartment appears only as someone's "new apartment" you may assume that it is currently vacant.

For simplicity, you may assume moving in or out of an apartment is instantaneous for each individual and that each apartment has only one tenant.

[1]home of *the other* UW.

[2]Yes, really: https://onwisconsin.uwalumni.com/traditions/moving-day/.

Given a list. you will return either

- `Easy Movement` and an ordering of all n people, allowing for easy movement.
- `No Easy Movement` and list of people who prevent easy movement.

For example,

Sample Input I
Swati [123 Fake St., 200 State St.]
Ewin [200 State St., 1000 Main St.]
Xin [`null`, 123 Fake St.]
Victor [567 Broadway, `null`]

You might return `Easy Movement` and [Victor, Ewin, Swati, Xin] (there are other valid lists to return here, you only need to give one).

Sample Input II
Swati [123 Fake St., 200 State St.]
Ewin [200 State St., 1000 Main St.]
Xin [1000 Main St, 123 Fake St.]
Victor [567 Broadway, `null`]

You would return `No Easy Movement` and [Swati, Ewin, Xin].
There is no easy movement in this example, because none of Swati, Ewin, and Xin can move first – they each need one of the others to go first.

(a) Describe an algorithm to solve this problem. **Hint:** By using algorithms from class and 373 (see the list in the directions), we only need a few sentences of English for this problem.

(b) Briefly explain why your algorithm works. This doesn't need to be a full proof, but it should be clear what graph object(s) you're looking for and why they correspond to `Easy Movement` or `No Easy Movement`.

(c) If your list has n people, what is the worst-case running time. Briefly (1-2 sentences) explain.

Resubmissions

You may resubmit up to two problems this week. The due date for resubmissions is Monday, January 29th. You will both: (1) submit a pdf to gradescope **in the resubmission box** (not the box where you submitted the problem originally), and then fill out the google form to tell us which one you resubmitted (we will post the form on the assignments page on the course website).