CS 344 - Fall2022
Homework 6
100 points total

General Assumption: You can assume that all graphs given in this problem set have zero isolated vertices, so $|E| \geq |V|/2$.

1 Problem 1 (30 points)

Consider the following problem:

- INPUT:
  - directed, *unweighted* graph $G = (V, E)$. Note that the graph is unweighted. Moreover, every vertex in the graph is either red or black. You can access the color of a vertex v in $O(1)$ time by looking at v.color. So for each vertex v, v.color is either "red" or "black"
  - Two vertices s, t.
- OUTPUT: output TRUE if there exists a path s to t that contains at most $\sqrt{|V|}$ red vertices. Output FALSE otherwise. Note that you don't need to output a path; you just need to determine if one exists.

The Problem: Write pseudocode for an algorithm which solves the above problem in time $O(|E| \log(|V|))$. **For this problem you *must* use graph transformation.** That is, your algorithm should create a new graph G', and then run an existing graph algorithm from our library on G'; that is, the algorithm you run on G' should be one of the graph algorithms in our library, without any modifications to that algorithm.

OPTIONAL ASSUMPTION: If it makes things easier for you, you can assume that both s and t are colored black. If your algorithm requires this optional assumption, please write at the beginning of your solution "I am using the optional assumption that s and t are colored black". You will not lose any points for making this assumption.

HINT: note that none of the algorithms in our library work on graphs where the vertices have colors. This means that in G' the vertices will not be colored. G' will also be different from G in several other ways as well.

HINT: first think about how you would solve this problem if *all* vertices were red; in this case, you just want to figure out if there is an st path with at most $\sqrt{n}$ vertices. This is significantly easier, and you can do it without any graph transformation just by calling one of the algorithms from our library directly on G. Now think about the harder case where only some vertices are red and think about how graph transformation can help here.

HINT: In the solution I have in mind, the graph G' you construct will have edge-weights. You will also have to add some additional vertices and edges.

HINT: this problem might look similar to problem 4 because in both you have colored vertices, but it's really not. In that problem we have a DAG; here we do not. In that problem the input graph G is weighted; here the input graph G is unweighted. Most importantly, in this problem you need to use graph transformation, whereas in that one you should not. So don't expect the solutions of these two problems to have much in common.

WHAT TO WRITE:

- Make sure to very clearly describe the graph G'
- State what algorithm you are using to compute distances in G'
- State how to use the distances you computed in G' (previous step) to figure out the solution to the original problem in G.

2 Problem 2 (10 points)

Consider the DAG G in Figure 1. This graph has exactly two valid topological orderings. You should write down BOTH of them. No need to justify work. Your solution only needs to include the two orderings; no need for explanation or anything else.

3 Problem 3 (30 points)

Recall the topological ordering problem from class.

INPUT: a DAG $G = (V, E)$

OUTPUT: a topological ordering $v_1, ..., v_n$ of the vertices.

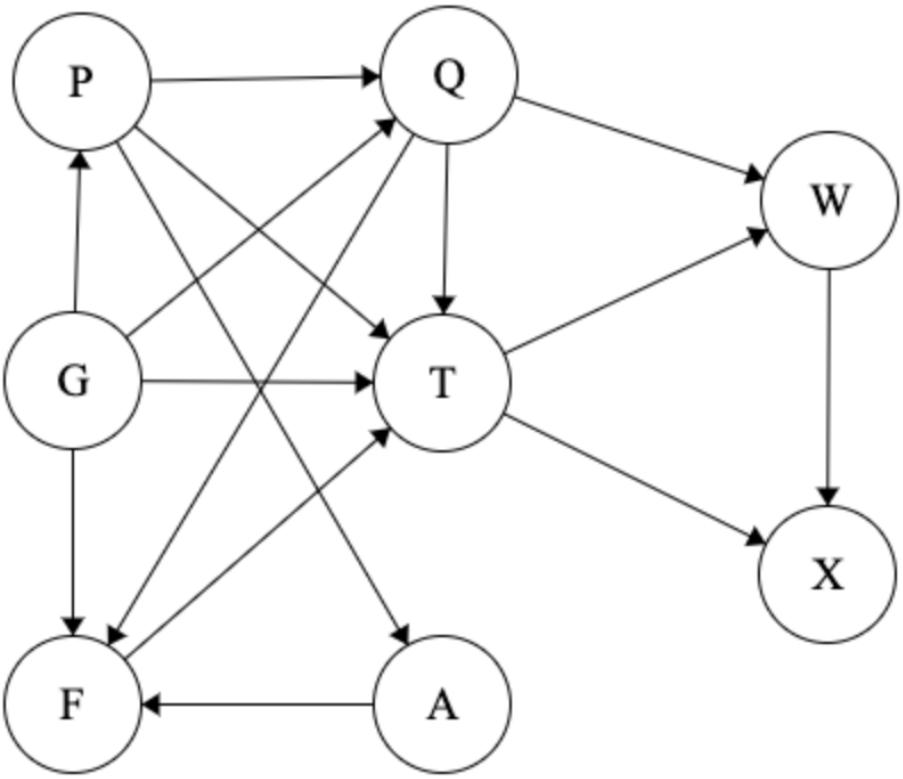

Figure 1: Graph for problem 2

Write pseudocode for an algorithm that solves this problem in $O(|E|)$ time.

IMPORTANT NOTE: to keep notation consistent, please use the following notation for this problem. Assume that the original vertex set is $V = v'_1, ..., v'_n$, which are NOT in topological order. (I am using v'_i to avoid confusion with the v_i of a topological ordering.) Your goal is then to sort the vertices so they are in topological order. So for example, if the correct topological order is v'_2, v'_3, v'_1, then you should output the list v'_2, v'_3, v'_1. Alternatively, you could output

- $v_1 \leftarrow v'_2$
- $v_2 \leftarrow v'_3$
- $v_3 \leftarrow v'_1$,

You can use either of the two output methods described above.

HINT: you will want to follow the approach from class of repeatedly finding zero-degree vertices. Think about how to implement that efficiently. Note that it's not necessary to literally remove edges/vertices as long as you have an efficient way of finding the next vertex in the topological order in each step. But you are welcome to remove vertices if that's easier for you, as long as the overall runtime is $O(|E|)$.

HINT: recall that in our representations of graphs, each vertex is represented with a vertex-object that can store additional information about the vertex. For this problem, I recommend have a v.count at each vertex. Your algorithm will sometimes update parameter v.count for different vertices v. This hint is intentionally vague: all I want to communicate is that it will help you to store an additional parameter v.count for each vertex. I leave it to you to figure out what information you should store in each v.count and how your algorithm should update this parameter.

4 Problem 4 (30 points)

Consider the following problem:

- INPUT:

  - a *directed, weighted* graph $G = (V, E)$ where G is guaranteed to be a DAG. Moreover, every $v \in V$ has a color, which is either red or black. For any vertex v, you can access its color in $O(1)$ time by doing v.color. So for every v, v.color will be 'red' or 'black'.
  - two vertices $s, t \in V$.

- OUTPUT: find the length of shortest path from s to t such that the path contains at most 100 red vertices. That is, among all $s - t$ paths that contain at most 100 red vertices, you have to find the one with smallest total weight. Note that there is no restriction on the number of black vertices on the path.

Write pseudocode for an algorithm that solves the above problem in $O(|E|)$ time.

CLARIFICATION: if there exists no st path with ≤ 100 red vertices, then output "no solution".

IMPORTANT NOTE: You don't need to return the actual path; just the weight of the path.

HINT: I do NOT know of a good way to solve this problem using graph transformation. The solution I have in mind just requires writing pseudocode for a new algorithm, which is pretty similar to the algorithm for finding shortest paths in a DAG that we covered in class.

HINT: Recall that finding a shortest path in a DAG is a dynamic programming problem. For the modified version in this problem, in addition to keeping track of the weight of the path, we also need to keep track of the number of red vertices. What is our general strategy for modifying dynamic programming problems when we need to keep track of extra information?