CS 344 - Fall 2022
Homework 5
100 points total + 15 points EC

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

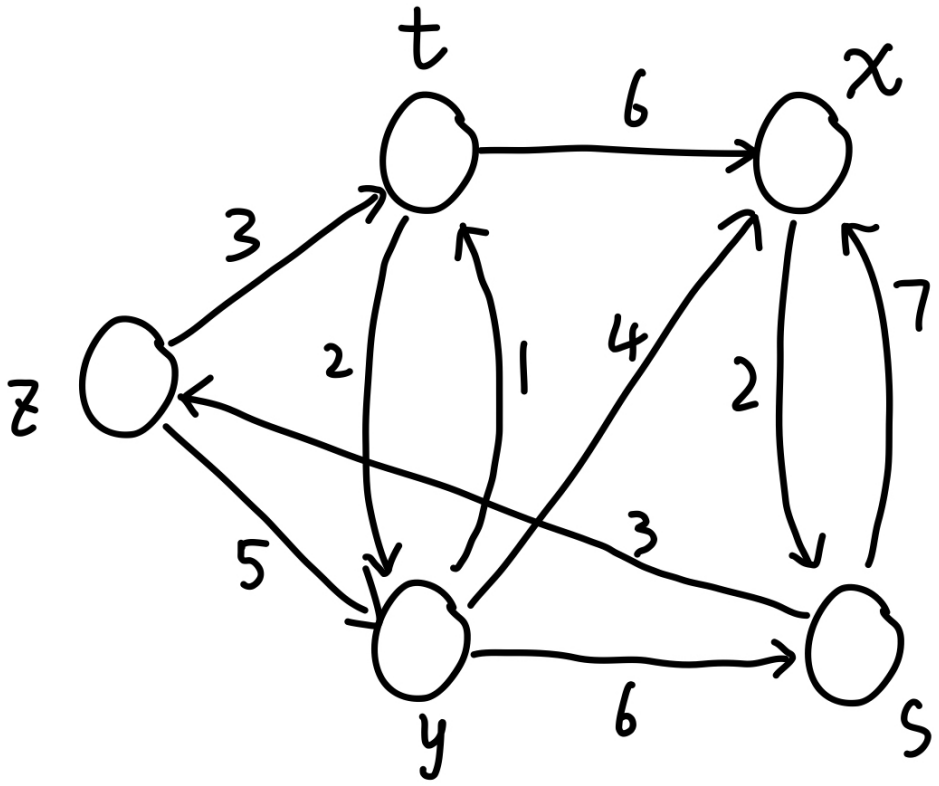
1 Problem 1 (30 points total)

1.1 Part 1 (7 points)

Consider the graph G in the figure below. Now consider running Dijkstra(G,s). (s is the bottom right vertex.)

Draw a table which indicates what the algorithm looks like after each execution of the while loop inside Dijkstra's. In particular, for each iteration of the loop you should indicate

- which vertex is explored in that iteration
- what is the label $d(v)$ for *every* vertex v at the end of that iteration.
- So all in all you should draw a table with 5 rows and 6 columns. Each row corresponds to an iteration of the while loop: so you can label the rows "iteration 1", "iteration 2", and so on up to "iteration 5". You then have six columns. The first column you should label "explored vertex", and this column indicates which vertex is explored in that iteration. You then have 5 other columns labeled d(z), d(t), d(x), d(y), and d(s), which show the label of each vertex at the end of every iteration.



NOTE: for an example of how Dijkstra is executed on a *different* graph, you may find it helpful to look at Figure 24.6 on page 659 of CLRS. In that example the actual graph is drawn at the end of each iteration. You can do it that way if you prefer, but you can also use a table instead, since that's easier for typing.

1.2 Part 2 (8 points)

In this problem, we want to show that Dijkstra only works if we assume all weights are non-negative. Your goal in this problem is to give an example of a graph $G = (V, E)$ with the following properties

- All edge weights in G are positive except there is EXACTLY ONE edge (x, y) with a negative weight
- G does not have a negative-weight cycle. (We will define this in class.)

- G contains two vertices s and t such that running Dijkstra(G,s) returns the wrong answer for dist(s,t).

WHAT TO WRITE: In your answer, make sure to clearly indicate

- Your graph G itself along with all the edge weights
- The two vertices s and t in G
- What is the actual shortest s-t distance
- what is the shortest s-t distance returned by Dijkstra

Your graph G should have at most 7 vertices. You can get away with even fewer than 7 vertices.

1.3 Part 3 (8 points)

Consider the following problem:

- INPUT
  - A directed graph G where all edges have weight -1, 0, or 1
  - Two fixed vertices s,t.
- OUTPUT: dist(s,t)

Consider the following algorithm for this problem:

Bad Algorithm

- Create a new graph G' which is the same as G except that all weights are increased by 1. So G' uses weight function w', where $w'(x, y) = w(x, y) + 1$.
- Run Dijkstra(G',s)
- Let P be the shortest s-t path in G' returned by Dijkstra(G',s)
- Output $w(P)$ as your final answer, where $w(P)$ is the weight of path P in the original graph G.

The Problem: We want to show that the above algorithm does NOT work. Give an example graph G where the above algorithm produces an incorrect solution. Concretely, you need to show a graph G such that the shortest $s-t$ path in G' is NOT the shortest $s-t$ path in G.

NOTE: your graph should contain at most 7 vertices. you can get with many fewer vertices.

WHAT TO WRITE: In your answer, make sure to clearly indicate

- Your graph G itself along with all the edge weights
- The two vertices s and t in G
- What is the actual shortest s-t distance
- what is the shortest s-t distance returned by Bad Algorithm.

1.4 Part 4: 7 points

Recall that Dijkstra requires a data structure D that can handle three operations. There exists a fancy data structure D with the following running times.

- D.insert(key k, value v) takes $O(\log(n))$ time
- D.decrease-key(new key k, value v) takes $O(1)$ time.
- D.delete-min() takes $O(\log(n))$ time.

The Problem: if one used the above fancy data structure inside of Dijkstra's algorithm, what would the resulting running time of Dijkstra's be in big-O notation? How does this runtime compare to the $O(|E|\log(|V|))$ running time we got in class by using a min-heap? For what edge-density of graphs (if any) does the fancy data structure better give a better running time (in terms of big-O) than $O(|E|\log(|V|))$? For what edge-density (if any) does it have the same running time? For what edge-density (if any) does it have a worse running time?

2 Problem 2 (20 points)

We say that an *undirected* graph $G = (V, E)$ is bipartite if it is possible to color every vertex either red or blue in such a way that for every edge $(u, v) \in E$, u and v have different colors. For example, in Figure 1, the graph on the left is bipartite because such a red-blue coloring is possible, but the graph on the right is not bipartite: you can check for yourself that no matter how you color the vertices of the right graph red/blue, there will always be an edge between two vertices of the same color.

The Question: Assume you are given an *undirected* graph G that is connected: that is, there is a path from every vertex to every other vertex. Give pseudocode for an algorithm CheckBipartite(G) that outputs TRUE if the graph is bipartite (i.e. if there exists a valid red/blue coloring) or FALSE if the algorithm is non-bipartite. Your algorithm should run in $O(|E|)$ time.

NOTE: you only need to write pseudocode. No justification or running time analysis necessary.

HINT 1: Start by picking an arbitrary vertex s and coloring it red. Now, proceeding from s, try to color the rest of the vertices in a way that doesn't create any illegal edge (by illegal I mean blue-blue or red-red.)

HINT 2: BFS will prove very useful here. One approach is to modify the BFS algorithm. An even better approach is to just use BFS(G,s) as a black-box. After you run BFS(G,s) you will know dist(s,v) for every vertex v (you don't have to explain how it works, since BFS is in our library). Is there a way to use all the dist(s,v) to determine if the graph is bipartite?

NOTE: you need to write full pseudocode. So if you modify BFS, you need to write the full pseudocode of your modified BFS. That's why I recommend using the approach in Hint 2, since then you can use BFS as a black-box, without needing to write any pseudocode for it.

3 Problem 3 (20 points)

Definition: We say that a directed graph $G = (V, E)$ contains a cycle if there exists a set of edges from a vertex back to itself. Formally, G contains a cycle if there exists a set of vertices $x_1, ..., x_k$ such that

- All the x_i are different vertices AND
- edge $(x_i, x_{i+1}) \in E$ for all $1 \leq i \leq k-1$ AND

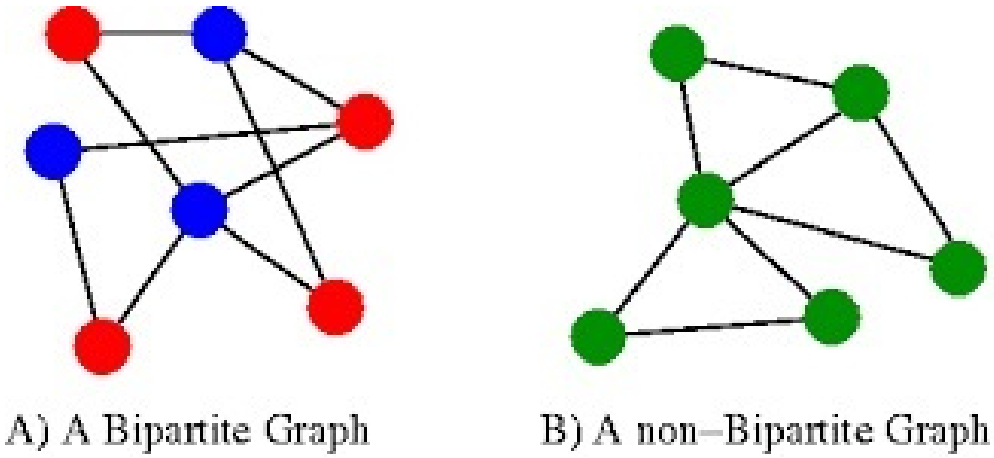


Figure 1: Graphs for Problem 2. The left graph is bipartite. It is not hard to check that the right graph is non-bipartite, because no matter how you color the vertices, there will be a red-red edge or a blue-blue edge.

- edge $(x_k, x_1) \in E$.

Assumtpion: Like we did in the rest of this class, we assume for this problem that a vertex cannot have an edge to itself. That is, there are no edges of the form (v, v).

The Problem: Consider the following problem:

- INPUT: a directed, unweighted graph G, where G has the following additional property: every vertex v in G has in-degree(v) ≥ 1. That is, for every vertex v in G, IN(v) is non-empty
- OUTPUT: a cycle in G.

Write pseudocode for an algorithm that solves the above problem in time $O(|V|)$. Note that the runtime should be $O(|V|)$ and not $O(|E|)$; an algorithm with runtime $O(|E|)$ will receive very few points.

NOTE: your pseudocode must return all the vertices on the cycle, in the order of the cycle. So e.g. if the cylce has edges (v_1, v_4), (v_4, v_8), (v_8, v_1), then v_4, v_8, v_1 is a valid output but v_4, v_1, v_8 is NOT a valid output. I am not

picky if your pseudocode outputs the cycle as a list, or an array, or simply prints the vertices in the correct order.

NOTE: you will need to take advantage of the fact that every vertex in G has at least one in-neighbor.

HINT: to return the actual cycle in the end, I recommend using parent pointers. That is, each vertex v will store a v.parent, which is initially NIL, but which your algorithm can change.

4 Problem 4 (30 points)

4.1 Problem 4 Part 1 (20 points)

Let $G = (V, E)$ be an *undirected, unweighted* graph with the following property: for *every* vertex $v \in G$, we have that degree(v) $\geq 100|V|^{1/5}$. That is, every vertex v has at least $100|V|^{1/5}$ neighbors.

The Problem: prove that the graph G above contains a cycle with at most 10 edges.

HINT: pick an arbitrary vertex s and consider doing a BFS from s. How do you know when you have found a cycle using BFS? How can you use the fact that every vertex has high degree to prove that you will find a cycle within a small number of BFS layers, and so the cycle must contain few edges? I'm being a bit vague on purpose; these are meant to be hints that point you in the right direction, but I leave it to you to figure out the details.

GRADING NOTE: You don't need a proof by induction or anything like that. By "prove" I just mean a detailed justification that would fully convince someone who is seeing this problem for the first time.

4.2 Problem 4 Part 2 (10 points)

Now we want to show that the above claim does not hold for a directed graph. In fact, it's so far from being true that we can construct a graph with no cycle at all.

The Problem: Give an example of an unweighted, directed graph G with $|V|$ vertices such that every for *every* vertex, in-degree(v) + out-degree(v) $\geq \sqrt{|V|}$, and yet G does not contain ANY cycles.

HINT: in the example graph I have in mind, half the vertices have in-degree(v) = 0, while half the vertices have out-degree(v) = 0. But there are

many other graphs that also work just as well.

NOTE: your example must work for arbitrary $|V|$. It's not enough to just draw a graph with 10 vertices. You need to show that for any n you can construct a graph such that $|V| = n$ and the above properties hold. If it makes it easier for you, I'm ok with you fixing $|V|$ to be a large number: in particular, $|V|$ = one million. That is, to get full credit it is enough for you to describe a graph with the following properties

- G is unweighted, directed, has exactly a million vertices $v_1 ..., v_{1000000}$ AND
- Every vertex v has in-degree(v) + out-degree(v) ≥ 1000 AND
- the graph G contains no cycles.

5 Problem 5 – extra credit (15 points)

Given a directed weighted graph $G = (V, E)$ with weight function w, such that G contains no negative-weight cycle, let G' be the graph obtained when running Johnson's algorithm, covered in class. To recap what we did in class, we define the transformed graph G' as follows.

- Add a new vertex s to G and add an edge (s, v) of weight 0 to every $v \in V$ (note: G does not contain any edges into s, only edges out of s)
- Set $\phi(v) = \text{dist}_G(s, v)$ for every $v \in V$.
- Now define G' as follows
  - G' has the same edges and vertices and G, but with different edges weights
  - For any edge $(u, v) \in E$ I will use $w(u, v)$ to denote the original edge-weight in G, and $w'(u, v)$ to refer to the new edge weight in G'.
  - For every edge $(u, v) \in E$, set $w'(u, v) \leftarrow w(u, v) + \phi(u) - \phi(v)$.

The Problem: Say you have graph G with the following properties

- G has *exactly* 7 vertices. You should label them a,b,c,d,e,f,g

- G is directed and weighted and all edge weights are either -3,0,or 2.
- G contains no negative-weight cycles.

Question: given a graph G with the above properties, what is the maximum possible edge weight in G'? That is, what is the highest possible value of $\max_{(u,v)\in E} w'(u,v)$. Note that different graphs G will yield different G', so your goal is to find the graph G such that G' has the maximum possible edge weight.

WHAT TO WRITE:

- You have to clearly draw all the vertices and edge weights in G. Again label your vertices a,b,c,d,e,f,g
- The price functions $\phi(a), \phi(b), ..., \phi(g)$ you get from running Johnson's algorithm on G.
- Draw ALL vertices and edge weights in G' as well
- Indicate what maximum edge-weight you get in G'
- Please make your drawing extremely legible. In particular, draw everything large, as that will make it easier for the graders to interpret.

GRADING NOTE: you really must write all of the above. If you skip parts you will get very little credit. If you draw a graph G, and then your prices ϕ and your graph G' correctly correspond to what would happen if you ran Johnson's algorithm on G, but the graph G you picked does not actually achieve the maximum possible edge weight in G', the amount of partial credit will depend on how close you get to the maximum.