CS 344 - Sections 5,6,7,8 - Spring 2021
Homework 5
Due Monday, April 12, 4:00pm
100 points total + 10 points EC

1 Problem 1 (25 points)

Given a directed graph $G = (V, E)$, we say that a vertex $v \in V$ is *fully inward* if for every vertex $x \neq v$, $(x, v) \in E$ but $(v, x) \notin E$. That is, v has no outgoing edges but has an incoming edge from every other vertex. (You can assume the graph has no self-loops, so there is never an edge from a vertex to itself.)

Consider the following problem

- INPUT: A directed grpah $G = (V, E)$ *given in adjacency matrix format.* That is, the input is a $|V|$ by $|V|$ matrix A such that $A[i, j] = 1$ if $(v_i, v_j) \in E$ and 0 otherwise.
- OUTPUT: a fully inward vertex v_i, or "no solution" if none exists. (Note: if v_i is fully inward, you can just output the number i. So for example if your algorithm detects tht v_{98} is fully inward, the algorithm can just output 98.)

write pseudocode that solves the above problem in $O(|V|)$ time. NOTE: $O(|V|)$ time, not $O(|E|)$ time! You will want to use the fact that the graph is given in adjacency matrix format.

HINT: Say your algorithm looks at some A[i][j]. If A[i][j] = 0, what can you deduce from that about which vertices can be fully inward? What about if A[i][j] = 1?

HINT 2: The algorithm I have in mind has a row pointer and a column pointer. Every time you look at some A[i][j], you will want to move either the row pointer or the column pointer. This will allow you to effectively ignore vertices which you have deduced cannot be fully inward.

GRADING NOTE: you only need to write pseudocode; no justification necessary. If your algorithm is slower than $O(|V|)$ you will get very little credit. In particular, a $O(|E|)$ algorithm will receive very little credit.

2 Problem 2 (25 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 3, 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?

3 Problem 3 (25 points total)

Part 1 (13 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 where each row corresponds to an iteration of the while loop, there is a column for "explored vertex"

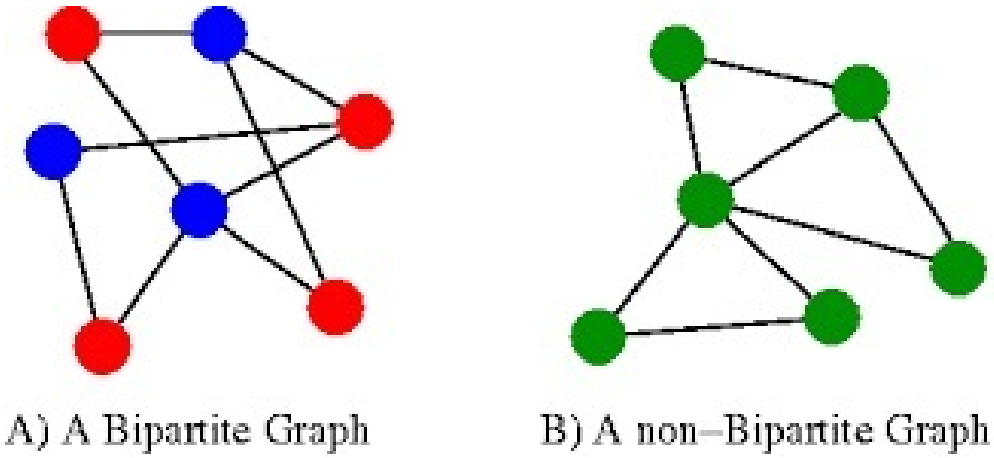
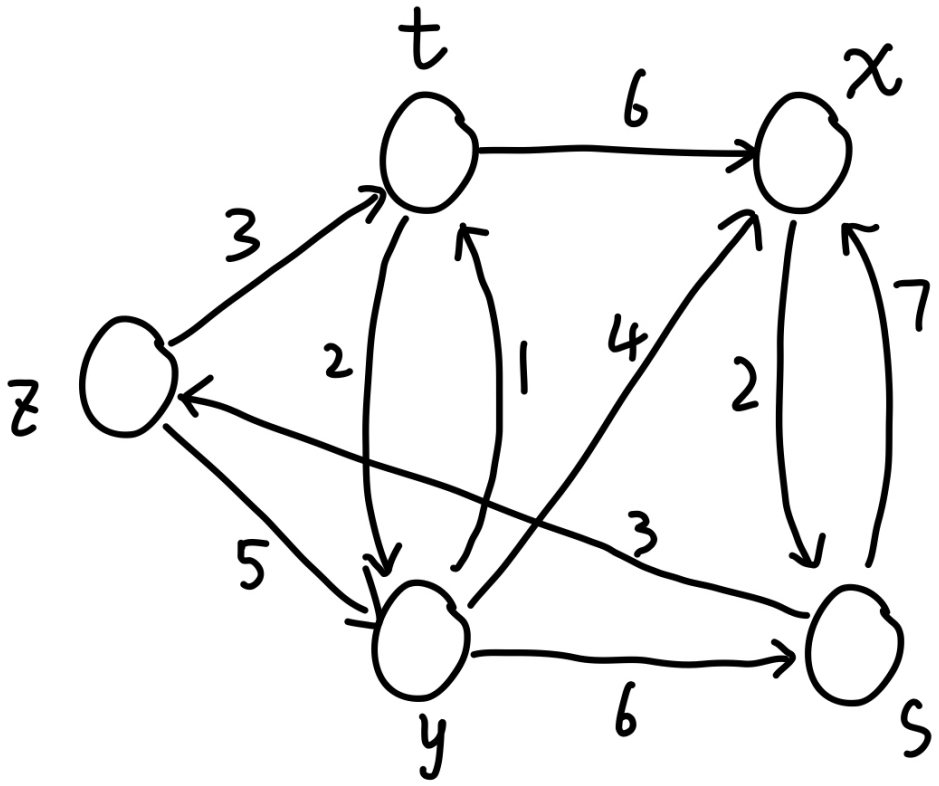


Figure 1: Graphs for Problem 2. The left graph is bipartite. It is not hard to check that the right graph is non-bipartite, because no matter how you color the vertices, there will be a red-red edge or a blue-blue edge.

which indicates which vertex is explored in that iteration, and there is a column for every vertex v which indicates the value of $d(v)$ at the end of the 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.

Part 2 (12 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 contains two vertices s and t such that running Dijkstra(G,s) returns the wrong answer for dist(s,t).

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.

4 Problem 4 – 5 points + 10 points EC

Part 1 (5 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(value v, key k') 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?

Part 2 (10 points – EXTRA CREDIT) Say that you are given a directed graph $G = (V, E)$ where all edge weights are integers in the set $\{1, 2, ..., C-1, C\}$ for some positive integer C. Show a data structure D such that if we run Dijkstra using data structure D on a graph G with edge weights in $\{1, 2, ..., C-1, C\}$ then the running time is $O(|E| + C|V|)$. The space used by the data structure should be $O(C|V|)$.

Make sure to write pseudocode for how to implement each of the three operations of the data structure (the three operations we need for Dijkstra, covered in class.)

NOTE: the data structure is simple and doesn't require you to use anything beyond basic things such as arrays and lists.

5 Problem 5 (20 points)

Let $G = (V, E)$ be a weighted directed graph with non-negative edge weights and let $s \in V$ be some source vertex. Say that you are given a label $d(v)$ for every vertex v such that $d(s) = 0$ and such that for *every* edge $(x, y) \in E$ we have the following KEY PROPERTY:

$$d(x) + w(x, y) \geq d(y).$$

The Problem: Prove that if $d(s) = 0$ and the key property holds for *all* edges in the graph then $d(v) \leq$ dist(s,v) for all vertices $v \in V$.

HINT: it's a proof by contradiction, similar to how we proved BFS and Dijkstra. Here's how I would start the proof

1. Say the claim were false
2. Then there would be at least one vertex v for which $d(v) >$ dist(s,v)
3. let $B \subseteq V$ be the set of all bad vertices. That is, $B = \{v \in V \mid d(v) > dist(s, v)\}$.
4. Let x be the vertex in B with a certain property. For example, if the correctness proof of BFS and Dijkstra we let x be the "first" bad vertex. For this problem there is no notion of first, but there is some easy-to-define property that we want x to have. (Just try to do the bullet point below and think about what property you will need from x for the proof to go through.)
5. Show by using the KEY PROPERTY, combined with other basic observations from class, we can deduce that $d(x) \leq$ dist(s,x), so $x \notin B$, which is a contradiction because in the previous bullet point we chose $x \in B$.

NOTE: you don't have to use the proof strategy I outlined above; that's just meant as a hint. In fact, it is possible to prove the claim by induction instead of proof by contradiction. You are welcome to do that instead if you prefer.