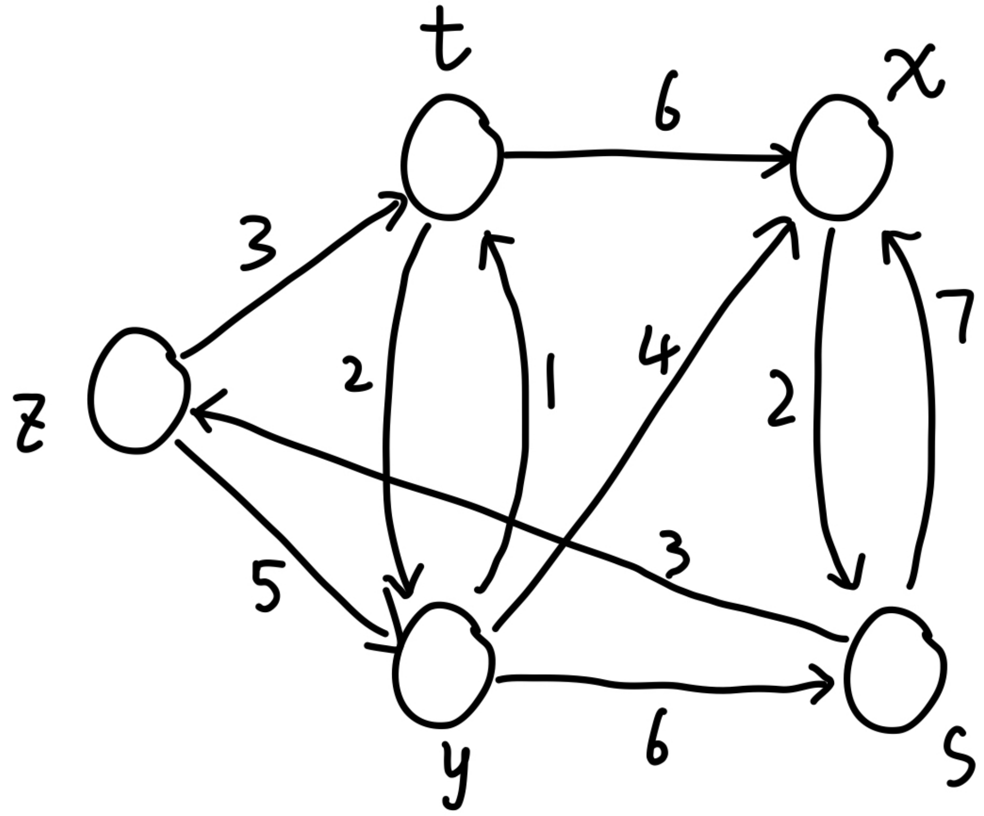
Homework #5 Solutions

CS 344: Design and Analysis of Computer Algorithms (Fall 2022)

Problem 1

(1) 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.



Each row shows updated distances after having explored the vertex at that row.

	explored vertex	$d(s)$	$d(x)$	$d(y)$	$d(t)$	$d(z)$
Init		0	∞	∞	∞	∞
iter1	**s**	**0**	7	∞	∞	3
iter2	**z**	0	7	8	6	**3**
iter3	**t**	0	7	8	**6**	3
iter4	**x**	0	**7**	8	6	3
iter5	**y**	0	7	**8**	6	3

(2) In this problem, we want to show that Dijkstra only works if we assume all weights are

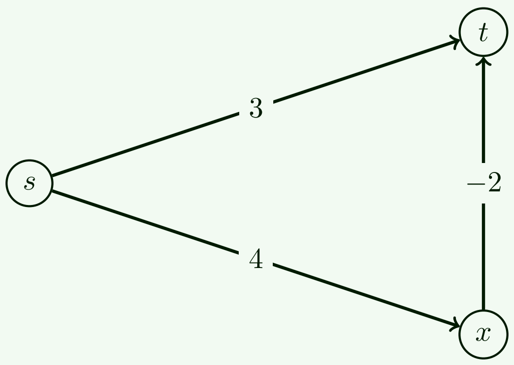
non-negative. Your goal in this problem is to give an example of a graph $G = (V, E)$ with the following properties

- 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).

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.



Actual shortest path: $s \to x \to t$ where $d(s, t) = 2$.
Shortest path from Dijkstra: $s \to t$ where $d(s, t) = 3$.

(3) 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.

Your graph should contain at most 7 vertices. you can get with many fewer vertices. 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.

$$\begin{tikzcd}
s \arrow[r, "0"] \arrow[d, "1"'] & t \\
x \arrow[r, "-1"] & y \arrow[u, "-1"']
\end{tikzcd}
\qquad G$$

$$\begin{tikzcd}
s' \arrow[r, "1"] \arrow[d, "2"'] & t' \\
x' \arrow[r, "0"] & y' \arrow[u, "0"']
\end{tikzcd}
\qquad G'$$

Actual shortest path: $s \to x \to y \to t$ where $d(s,t) = -1$.
Shortest path from Bad Algorithm: $s \to t$ where $d(s,t) = 0$.

(4) 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.

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?

Insert - $O(|V|)$ times
Decrease-key - $O(|E|)$ times
Delete-min - $O(|V|)$ times

$$T(|V|,|E|) = O(|V| * log(|V|) + |E| * 1 + |V| * log(|V|)) = O(|E| + |V|log(|V|))$$

When $|E| \approx |V|$, $O(|E| + |V|log(|V|)) = O(|E|(log(|V|) + 1)) = O(|E|log(|V|))$.

comparing $|E| + |V|\log(|V|)$ **to** $E\log(|V|)$

- When $|E| \leq |V|\log(|V|)$ they are the same
- When $|E| > |V|\log(|V|)$, the fancy data structure (i.e. the one that gets $|E| + |V|\log(|V|)$) is better
- The fancy data structure is never worse.

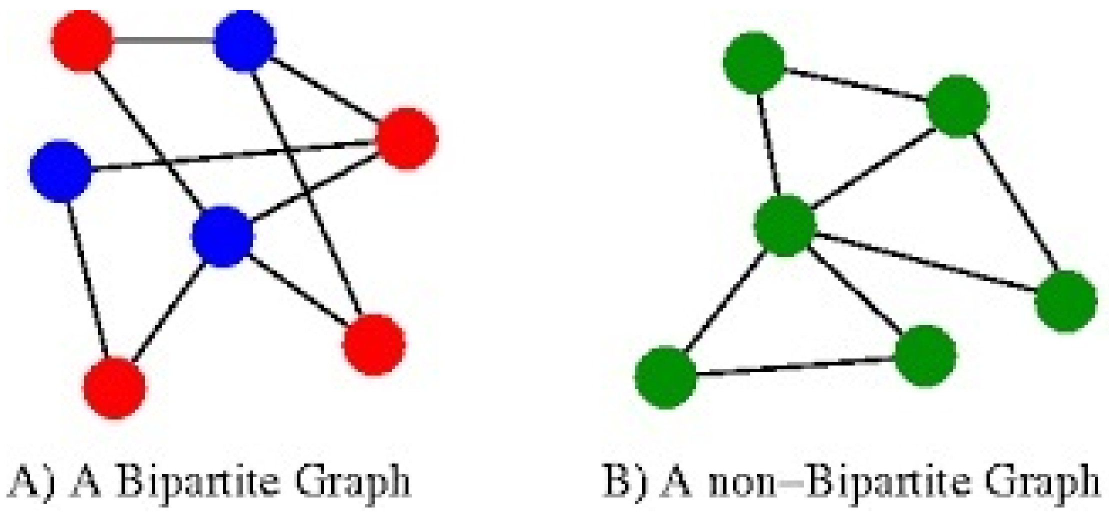


Figure 1: Graphs for Problem 2. The left graph is bipartite. It is not hard to check that the right graph is non-bipartite, because no matter how you color the vertices, there will be a red-red edge or a blue-blue edge.

Problem 2

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.

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.

We can use the distances returned by BFS to color the vertices based on the parity (divisibility by 2) of their distance to the BFS source. If there is no edge whose endpoints are colored the same, the graph is bipartite. If, on the other hand, any edge has endpoints colored the same, the graph is not bipartite.

For notational convenience, we allow vertices to index into arrays.

```
CheckBipartite($G = (V, E)$):
    $dist[v_0..v_{n-1}] \leftarrow \mathrm{BFS}(G, v_0)$
    $color[v_0..v_{n-1}] \leftarrow$ Nil in each entry
    for $v \in V$
        $color[v] = \begin{cases} \textsc{Red} & \text{if } dist[v] \text{ is even} \\ \textsc{Blue} & \text{if } dist[v] \text{ is odd} \end{cases}$
    for $(u, v) \in E$
        if $color[u] = color[v]$
            return False
    return True
```

Correctness. Notice that, if G were bipartite, coloring v_0 Red forces the coloring of all other vertices since neighboring vertices must be oppositely colored; the neighbors of v_0 must be colored Blue, their neighbors colored Red, their neighbors' neighbors colored Blue, and so on. The coloring given by CheckBipartite matches this forced coloring as vertices at layer i (distance i to v_0) of the BFS are neighbors to vertices at layer $i-1$ (distance $i-1$ to v_0), and v_0 is colored Red (since $dist(v_0, v_0) = 0$). So if G were bipartite, CheckBipartite would return True.

Suppose, in the other direction, that CheckBipartite returns True. All the vertices of G are colored Red or Blue in the first loop, and no edge of G has endpoints that are colored the same or else it would have been discovered in the second loop. G is thus bipartite by the definition of bipartite.

Time Complexity. Straightforward: $O(|E|)$ (we are promised that G is connected and so $|E| \geq |V|$).

Problem 3

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
- 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$(v) \geq 1$. 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.

We begin a process starting from an arbitrary vertex where we follow in-edges (every vertex has at least one to follow). Once a vertex has been visited twice (call it v^*), which must happen in no more than $|V|$ steps, a cycle has been found. We recover the cycle by taking all vertices visited in the process after the first time v^* was visited.

```
FindCycle($G = (V, E)$):
    $v^* \leftarrow V[0]$
    // If we have visited a vertex already, its parent has been set.
    while $v^*.parent = $ Nil
        $u \leftarrow IN(v^*)[0]$
        $v^*.parent \leftarrow u$
        $v^* \leftarrow u$
    // $v^* = u$ is the vertex that has been visited twice.
    // Recover a cycle, stored in $V_C$ which is initialized to contain just $v^*$.
    $V_C \leftarrow$ BuildList($v^*$)
    while $u.parent \neq v^*$
        $V_C.add(u.parent)$
        $u \leftarrow u.parent$
    // $V_C$ was in reverse-order, because $u.parent$ points to $u$.
    return Reverse($V_C$)
```

Problem 4

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

Prove that the graph G above contains a cycle with at most 10 edges.

Consider running BFS(G, s) on an arbitrary vertex s of G.

Layer 1 of the BFS-tree must have at least $100n^{1/5}$ distinct vertices since the degree of s is at least $100n^{1/5}$. If there is any edge $(u, v) \in E$ with both u, v in layer 1, we have found s, u, v – a cycle of length 3. If any pair u, v of vertices in layer 1 share a common neighbor $w \neq s$, we have found s, u, w, v – a cycle of length 4. If, on the other hand, none of the previous two events occur, then layer 2 of the BFS-tree must have at least $100n^{1/5} \cdot (100n^{1/5} - 1) \geq 100n^{2/5}$ distinct vertices.

We can continue a similar line of reasoning to assert that layer 3 of the BFS-tree must have at least $100n^{3/5}$ distinct vertices, layer 4 of the BFS-tree must have at least $100n^{4/5}$ distinct vertices and, finally, that layer 5 of the BFS-tree must have at least $100n$ distinct vertices.

The last assertion is not possible, since $100n > n$, and so one of the events

- There is an edge $(u, v) \in E$ with u, v in layer $i < 5$ of the BFS-tree
- A pair u, v in layer $i < 5$ of the BFS-tree share a common neighbor in layer $i + 1$ of the BFS-tree

must have occurred; this means a cycle of length ≤ 10 must exist within the 0th through 5th layers of the BFS-tree.

(2) 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.

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

Let $G = (V, E)$ where $E = \{(v_i, v_j) \mid 1 \leq i < j \leq n\}$.

Every vertex $v \in V$ satisfies *in-degree*(v) + *out-degree*$(v) = n - 1 \geq \sqrt{n}$. Moreover, G does not contain any cycle since edges point from vertices with a smaller index to vertices with a larger index (for any cycle, there must be a cycle-edge which points from a vertex with a larger index to a vertex with a smaller index).

Problem 5

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.

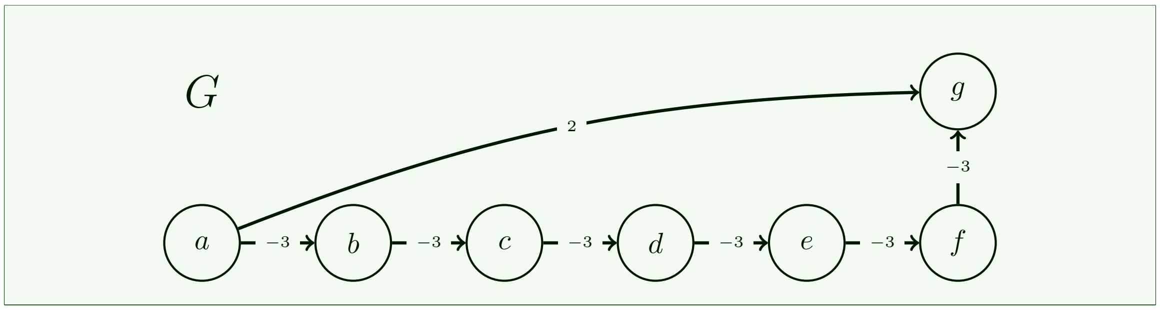
- Add a new vertex s to G and add an edge (s, v) of weight 0 to every $v \in V$ (note: G does not contain any edges into s, only edges out of s)
- Set $\phi(v) = \text{dist}_G(s, v)$ for every $v \in V$.
- Now define G' as follows
  - G' has the same edges and vertices and G, but with different edges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

Say you have graph G with the following properties

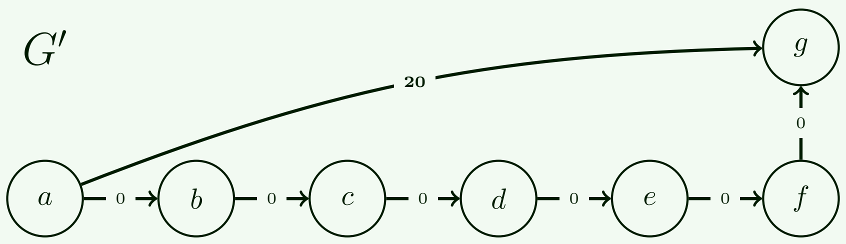
- G has *exactly* 7 vertices. You should label them a,b,c,d,e,f,g
- G is directed and weighted and all edge weights are either -3,0,or 2.
- G contains no negative-weight cycles.

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.

- 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'



$\varphi(a) = 0$
$\varphi(b) = -3$
$\varphi(c) = -6$
$\varphi(d) = -9$
$\varphi(e) = -12$
$\varphi(f) = -15$
$\varphi(g) = -18$



The largest weight is 20. In the diagram, this is the weight of edge (a, g) in G'.