Due April 11 (11:55 a.m.)

Instructions

- **Answer each question on a separate page.**
- **Honors questions are optional. They will not count towards your grade in the course. However you are encouraged to submit your solutions on these problems to receive feedback on your attempts. Our estimation of the difficulty level of these problems is expressed through an indicative number of stars ($'*' =$ easiest) to ($'*****' =$ hardest).**
- **You must enter the names of your collaborators or other sources as a response to Question 0. Do NOT leave this blank; if you worked on the homework entirely on your own, please write "None" here. Even though collaborations in groups of up to 3 people are encouraged, you are required to write your own solution.**

Question 0: List all your collaborators and sources: ($-\infty$ points if left blank)

Solution: None.

Question 1: (5 points)

The disjoint set data structure maintains a set of disjoint sets and supports the following operations.

1. createSet(x): creates a set containing the single element x.
2. findSet(x): returns a pointer to the unique set containing x.
3. merge(x, y): merges the unique sets containing x and y.

See CLRS Chapter 21 for a more detailed description. Consider the following example:

Operation performed	Collection of disjoint sets								
Initial sets	$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$	$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$
merge(a, b)	$\{a, b\}$		$\{c\}$	$\{d\}$	$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$
merge(c, d)	$\{a, b\}$		$\{c, d\}$		$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$
merge(d, f)	$\{a, b\}$		$\{c, d, f\}$		$\{e\}$		$\{g\}$	$\{h\}$	$\{i\}$
merge(a, h)	$\{a, b, h\}$		$\{c, d, f\}$		$\{e\}$		$\{g\}$		$\{i\}$
merge(e, g)	$\{a, b, h\}$		$\{c, d, f\}$		$\{e, g\}$				$\{i\}$
merge(b, e)	$\{a, b, e, g, h\}$		$\{c, d, f\}$						$\{i\}$

Now fill out the following table:

Operation performed	Collection of disjoint sets									
Initial sets	$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$	$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
merge(a, c)	$\{a, c\}$	$\{b\}$		$\{d\}$	$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
merge(b, d)	$\{a, c\}$	$\{b, d\}$			$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
merge(b, e)	$\{a, c\}$	$\{b, d, e\}$				$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
merge(f, i)	$\{a, c\}$	$\{b, d, e\}$	$\{f, i\}$				$\{g\}$	$\{h\}$		$\{j\}$
merge(f, j)	$\{a, c\}$	$\{b, d, e\}$	$\{f, i, j\}$				$\{g\}$	$\{h\}$		
merge(j, h)	$\{a, c\}$	$\{b, d, e\}$	$\{f, i, j, h\}$				$\{g\}$			
merge(g, e)	$\{a, c\}$	$\{b, d, e, g\}$	$\{f, i, j, h\}$							
merge(a, h)		$\{a, c, f, i, j, h\}$	$\{b, d, e, g\}$							
merge(h, d)		$\{a, c, f, i, j, h, b, d, e, g\}$								

How many distinct sets are present after this sequence of operations?

Solution: 1 set.

Question 2: (10 points)

Recall that the *adjacency matrix* representation of a directed graph $G = (V, E)$ is the $|V| \times |V|$ matrix M_G where $M_G[i][j]$ is 1 if there is an edge from vertex i to vertex j, and 0 otherwise. Describe an algorithm that, given the matrix M_G, checks if there is any vertex in G that has edges coming to it from *all other* vertices but no edges going out from it (this is known as a **universal sink**). You may assume no vertex in the graph has an edge going into itself (i.e., no self-loops). Your algorithm should run in $O(|V|)$ time. Justify why it is correct, as well as why it satisfies this run-time bound.

(Hint: Notice that for any pair of vertices i, j, the edge $i \to j$ is either present or absent. If it is present, then i cannot be a universal sink. If it is absent, then j cannot be a universal sink. Justify this fact and use it to create your algorithm.)

Solution:
If $i \to j$ is present, then we know there is at least one edge going out from it. In other words, we if $M_g[i][j]$ is 1, then vertex i is not a universal sink. If $i \to j$ is not present, we would know that vertex j does not have edges coming to it from all other vertices. Therefore, if $M_g[i][j]$ is 0, vertex j is not a universal sink.

Algorithm: With this strategy, we could design a algorithm that eliminate all the non universal sinks vertex. Starting from the first element in the adjacency matrix, $M_g[1][2]$, if it is zero then we check $M_g[1][3]$, and in this case we have eliminated vertex No.2. If $M_g[1][2]$ is one, then we proceeding on checking $M_g[2][2]$. Though this is not a very good example, since we don't need to check $M_g[2][2]$ because there are no vertex has an edge going into itself. Then we could proceeding on checking $M_g[2][3]$. To summarize, when we are on the entry $M_g[i][j]$, if it is 0, we would proceed on checking $M_g[i][j+1]$. If it is 1, we could proceed on checking $M_g[j][j+1]$. In the end, when the column entry hits then maximum length of the matrix, we stop. Let's say we at the element $M_g[i][n]$ when we stop. If it is 1, then we check whether vertex n is a universal sink by check if column n is all ones except $M_g[i][i]$ and row n is all zeros. If that condition holds, we have vertex n to be a universal sink. If not,

we have no universal sink in this graph. And if it is 0, we could check if column i is all ones except $M_g[i][i]$ and row i is all zeros to confirm if vertex i is a universal sink.

Correctness by Loop invariant:

Looping invariant: When we are at entry $M_g[i][j]$, all the vertex that their index is smaller than j besides vertex i cannot be a universal sink.

Initialization: We would start the algorithm at $M_g[1][1]$, by the looping invariant we claim that all the vertex that their index is smaller than 1 besides vertex 1 cannot be a universal sink. It is true because we have not yet eliminated any vertex yet and all the vertex that their index is smaller than 1 besides vertex 1 is a empty set.

Maintenance: When we are at entry $M_g[i][j]$, there are two possible scenarios, if $M_g[i][j] = 0$, then that implies vertex j can't be a vertex and we would proceed on $M_g[i][j+1]$. By the looping invariant before we have all the vertex that their index is smaller than j besides vertex i cannot be a universal sink. Combining the the previous step we would have all the vertex that their index is smaller than $j+1$ besides vertex i cannot be a universal sink which directly correspond to the looping invariant we are supposed to have at entry $M_g[i, j+1]$. The other scenario is $M_g[i][j] = 1$ and it implies that vertex i can't be a universal sink. By looping invariant, we could claim that all the vertex that their index is smaller than j besides vertex i cannot be a universal sink. And now we have showed that vertex i can't be a universal sink. When we have all the vertex that their index is smaller than j cannot be a universal sink. At the same time, we would proceed onto the entry $M_g[j][j+1]$. And the looping invariant at this point is all the vertex that their index is smaller than $j+1$ besides vertex j cannot be a universal sink, which is exactly that same as what we have showed.

Termination: Finally, we examine what we happens when the loop terminates. That will be when we have reached entry $M_g[i][n]$, for some $i \leq n$ and n is just the number of vertexes in the graph. At $M_g[i][n]$, we the looping invariant that all the vertex such that their index is smaller than $n-1$ besides vertex i cannot be a universal sink. Then if $M_g[i][n] = 0$ implying that vertex n can't be a universal sink. Then we will have that all vertex beside vertex v can't be a universal sink. Now, we would proceed on checking whether vertex v is a universal sink or not by checking its column and row to verify the definition. Other other case is when $M_g[i][n] = 1$, with looping invariant that we have all the vertex such that their index is smaller than n cannot be a universal sink. Then we could proceed on checking whether vertex n is a universal sink of not.

Another fact to notice is the algorithm is always checking the entries above the matrix diagonal because there are only two kinds of movement in this algorithm one is moving to the right by one entry and the other is moving to a entry that is one entry above the diagonal.

Runtime: The run time for the first part of the algorithm when we are iteration in the M_g matrix takes $O(n)$ time because each checking point takes constant time $O(1)$ and it only could take $|V|$ steps to reach termination because we are moving right by one step any ways. The second part of the algorithm is checking whether the vertex we output in part one is a universal sink or not. Let's say that vertex has index I. In this step, we would have need to check column i and row i in matrix M_g and in total we are checking $2|V| - 1$ entries in total. Then combining both parts together we have $T(n) = |V| + 2|V| - 1 = O(n)$.

Question 3: (10+10=20 points)

Let P be a program which, given as input a directed graph $G = (V, E)$ as well as two vertices $s, t \in V$, outputs the shortest path between them. Let $T(P)$ be the run-time of this program. Imagine that P is already highly optimized (say for running on a GPU) such that $T(P)$ runs significantly faster than your own shortest path algorithm. As such, we will use P to solve the following problem.

We are given a directed graph G where each of its edges is colored either red or blue. We want to find the shortest path from some vertex s to some other vertex t, with the stipulation that our path must be *separable*. In order to be separable, the path we output must consist of first some number (possibly 0) of red edges followed by some number (possibly 0) of blue edges. In other words, once you use a blue edge, all following edges must be blue.

1. Design an algorithm to find the shortest separable path from s to t. Your algorithm should invoke P *once* on a carefully constructed graph, and it should run in time $O(|V| + |E|) + T(P)$. You should *not* explore the graph yourself (i.e., do not implement BFS); use the optimized program P.

   (Hint: Consider the subgraph formed by restricting the edges to only the red ones. Then, consider the same process with the blue edges. Notice that you are to spend some number of steps in the first part and then move to the second part and stay there. Can you combine the two graphs somehow?)

Solution:
Step 0:
First, we will do as hinted that restricting the edges to be only the red ones and the only blue ones. That we constructed the red graph $G_{red} = (V, E_{red})$ and the blue graph $G^*_{blue} = (V^*, E^*_{blue})$. One thing to notices here is, V^* is represent the same vertex as V but using different notation to differentiate them from V and E^*_{blue} represents the edges of E that are marked blue but they are connecting vertex V^* instead of V. In other words, if we have vertex v in V, we would have v^* in V^*. And if we have directed blue edge $(u, v) \in E$, we would have $(u^*, v^*) \in E^*_{blue}$.
Step 1:
Then we need to identify the vertexes that could potential be a transitive vertex such that it has at least a red edge going into it and at least a blue edge coming out of it. To identify them, we could create a two dimensional matrix M that has $|V|$ columns and 3 rows. As the first row is filled will all the vertexes. The second row represent whether there is a red edge going to the corresponding vertex at the first row with binary expression 0 and 1. The third row represent whether there is a blue edge coming out of the corresponding vertex with binary expression 0 and 1. The following table is a example of such matrix M,

vertex	v_1	v_2	v_3	$\cdots$	$v_{\|V\|}$
red	1	1	1	$\cdots$	1
blue	0	1	0	$\cdots$	0

We could also think of this as a dictionary in python to make sure each access would have constant time.
Step 2:
At initialization of matrix M, the second and third row of M are initialized to be all zeros. Then we would traverse the data set of edges. For each directed edge going from vertex u to vertex v as represented by (u, v), if this edge is marked red, then we would change the red entry

responding to vertex v to be 1. And if edge (u, v) is marked blue, then we would mark the blue entry corresponding to vertex v to be 1.

Step 3:

After we have traversed all the edges, we would traverse the M by the vertexes on the first row and initialize an empty collection of edges $\tilde{E}$. While traversing M, if both the red and blue entries of the vertex v have been marked as 1, then we would add directed edge going from v to $v*$ expressed as (v, v^*) to $\tilde{E}$ and this edge is marked grey. At last, we would also add two extra direct grey edges $(t, t*)$ and $(s, s*)$ to $\tilde{E}$.

Step 4:

In second last step, we would create a new graph that combine all the vertex together and all the edges together such that,

$$\tilde{G} = \{V + V^*, E_{red} + E^*_{blue} + \tilde{E}\}$$

Now, we can input graph $\tilde{G}$ and vertex s and vertex t^* in the optimized algorithm P, finding the shortest path between vertex s and vertex t^*. When we have the output as a sequence of edges or a sequence of vertexes, let's say the output is a sequence of vertex, $s, v_1, v_2, \cdots, v^*_{k-1}v^*_k, t^*$.

Step 5:

At last, we would traverse through the sequence of vertex as we go along, we would transform any vertex in the form v^* to its original form v in the sequence, and keep track of the previous vertex in the sequence. And if after transformation the vertex we have is the identify to the previous vertex, then we know we are onto one of the grey edges we created. Then we would eliminate one of the identify vertexes. After this is done, we output the new sequence output as the shortest path from vertex s to vertex t.

2. Justify the correctness of your algorithm, and show that it runs in time $O(|V| + |E|) + T(P)$.

Solution:

Correctness: The intuition of the algorithm is since we are trying to find a separable path. In other words, we only take red edges in the first part of the path call it p_red and only take blue edges in the other part, call it p_blue. When we add them together, $p_red + p_blue = p$ which represent the optimal solution. Then p_red is only takes edges in the graph $G_red = \{V, E_red\}$ and p_blue is only takes edges in the graph $G_blue = \{V, E_blue\}$. Then the problem of shortest separable path between vertex s and vertex v becomes finding p_red and p_red such that p_red begins on vertex s and ends on some vertex u, and p_blue begins on vertex u and ends on some vertex t. But not all vertex could be put in the position of u since it requires that u to have at least a red edge directing to it and a blue edge directing out of it. This is what I am doing in step 1 in the above algorithm. Now it's very intuitive if we connect G_red and G_blue together at the vertexes that satisfied the properties described above. The procedure is we make a copy of G_blue, $G^*_{blue} = (V^*, E^*_{blue})$. And connect any vertex u that have at least a red edge directing to it and a blue edge directing out of it to their counter part u^* in the copy of the graph by creating a collection of edges $\tilde{E}$ that contains directed grey edges (u, u^*). Since these grey edges are directed, then if we take them once, we would go from the red graph to the blue graph and they guarantee that the output path is separable. Because the greys are the only connecting edges

between the red graph and the blue graph and they are all directed as they goes from a vertex in the red graph to their counterpart in the blue graph, so once we are in the blue graph there is no way going back to the red graph. But we can't perform program P right now because the if the optimal separable solution is completely red or blue, it could not be produced by our construction because if s or t^* is not connected to a grey edge, then we have to take steps in the red graph first to find a vertex that could help us transit into the blue graph. The solution to this issue is to connect s and t to their counterpart in V^*, as we add grey edges (s, s^*) and (t, t^*) to collection $\tilde{E}$. Now we have a new graph, $\tilde{G} = \{V + V^*, E_{red} + E^*_{blue} + \tilde{E}\}$, if we perform program P with input graph $\tilde{G} =$, vertex s, and vertex t^*. And if we modify the output path a little bit as described in the algorithm we will get the optimal separable path between vertex s and vertex t. For any path $\tilde{p}$ that would be taken in graph $\tilde{G}$ would take one extra step than their counter part in counterpart in the regular graph G because we need transit from the red graph to the blue graph which would take one extra step.

The above paragraph is the intuition, we could also prove it rigorously by contradiction. Assume, the output of algorithm is $p*$, a path that takes place in $\tilde{G}$ and its counter part in graph G after modification as described in step 5 is p. I would ignore the part that proof that p is a legitimate path between vertex s and t. Let's assume that there exist some other path $\tilde{p}$ that is a shorter path in comparison to path p. We could map the part of $\tilde{p}$ that is red into graph G_red and map the blue part of $\tilde{p}$ into graph G^*_{blue}, also make the connection where it transit from red to blue, then it becomes a path in $\tilde{G}$ that start from vertex s to vertex t^* call it $\tilde{p^*}$ By assumption $\tilde{p^*}$ would be a shorter path than p^*, however p^* is the shortest path in $\tilde{G}$ that start from vertex s to vertex t^*, which we have a contradiction. Then we prove the correctness of the algorithm by contradiction.

Runtime:
We can find we the run time by calculating the runtime by adding all of them together.
$T(\text{step } 0) = |E| + |V|$. We traverse over all the edges and split them into the the red graph and the blue graph which takes $|E|$. And it would take $|V|$ to copy V into V^*.

$T(\text{step } 1) = 1$. We are only initializing the matrix here, so it is constant time and I would write it as 1.

$T(\text{step } 2) = |E|$. Since we are traversing over all the edges and marking them into the matrix, for each edge it takes constant time to mark.

$T(\text{step } 3) = |V| + 2$. Since we are iteration over the columns of the matrix and there are $|V|$ columns and each iteration takes constant time. Beside, we are also adding two extra grey edges, (s, s^*) and (t, t^*), to the collection, so there is add two in the time.

$T(\text{step } 4) = 1 + T(P)$. Generate the graph $\tilde{G}$ would take constant time as I have put 1 as the first term. And run the fast program P would take $T(P)$ time.

$T(\text{step } 5) = O(|E|)$. In this step, we are traversing the output of P, and the optimal path could not have a length that is longer than the number of edges. Because by pigeon hole principle, if that is the case, we are going over some edge twice which contradicts the optimal nature of the

path. Then the runtime of step 5 is bounded by $|E|$.

Then the runtime T of my algorithm is given by

$$\begin{aligned} T &= T(\text{step 0}) + T(\text{step 1}) + T(\text{step 2}) + T(\text{step 3}) + T(\text{step 4}) + T(\text{step 5}) \\ &= |E| + |V| + |E| + |V| + 2 + 1 + T(P) + O(|E|) \\ &= O(|V| + |E|) + T(P) \end{aligned}$$

Question 4: (5+5=10 points)

1. Consider the graph in Figure 1. Say we begin a DFS traversal starting at node A. List the discovery (i.e., when we mark a node "gray") and finishing times (i.e., when we mark a node "black") of all the vertices (you may assume each successive step in the traversal takes time 1). Assume that the adjacency lists in the representation of the input graph are in alphabetical order.

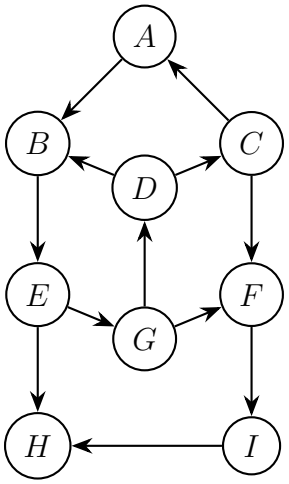


Figure 1: Graph for Question 4.1

Solution:

vertex	discovery time	finishing time
A	1	18
B	2	17
E	3	16
G	4	15
D	5	14
C	6	13
F	7	12
I	8	11
H	9	10

2. Consider the directed acyclic graph in Figure 2. Consider a DFS traversal of this graph starting from vertex A. List the discovery and finishing times of all the vertices. Draw the vertices on a line in decreasing order of finish time. Now draw the graph edges between these vertices. What's special about the edges drawn in this way? (Assume that the adjacency lists in the representation of the input graph are in alphabetical order.)

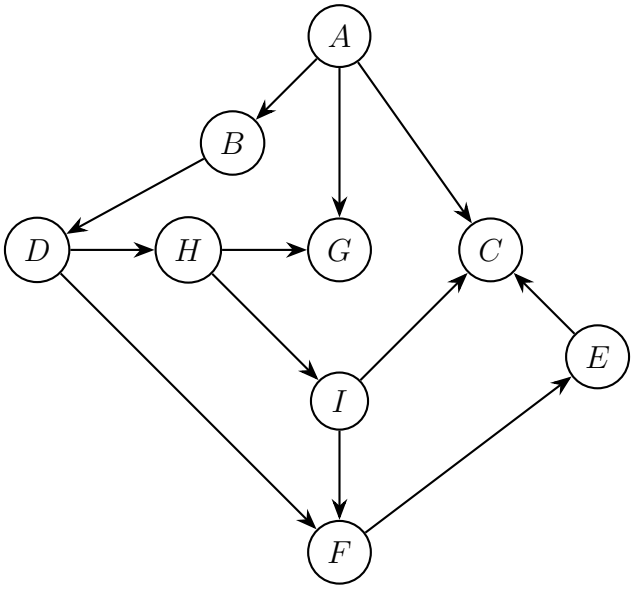
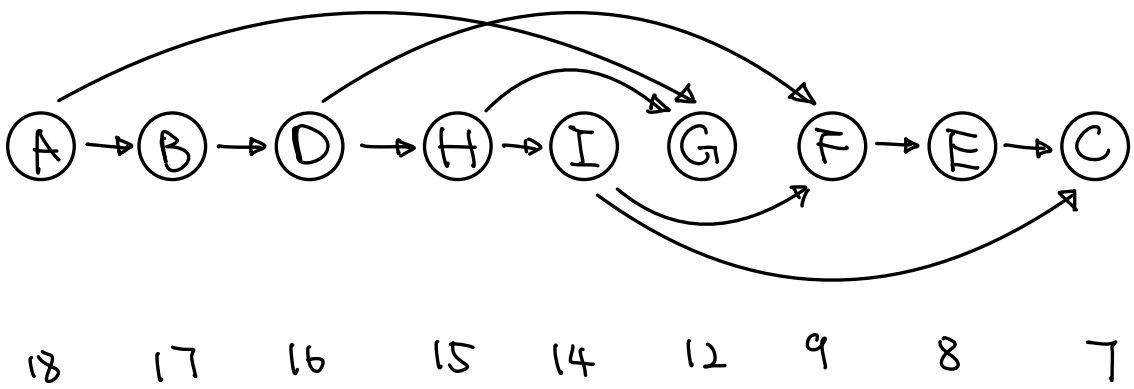


Figure 2: Graph for Question 4.2

Solution:

vertex	discovery time	finishing time
A	1	18
B	2	17
D	3	16
F	4	9
E	5	8
C	6	7
H	10	15
G	11	12
I	13	14



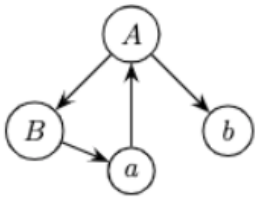
All edges are directed in the direction of a earlier finish time. In other words, that are no edges that direct a vertex to another vertex with a earlier finishing time.

Question 5: (5+5=10 points)

As in the previous question, for any vertex a in directed graph G, we denote by $a.\mathsf{d}$ the time at which DFS on G discovers (i.e., marks as "gray") the vertex a. Similarly, denote by $a.\mathsf{f}$ the time at which DFS fully explores (i.e., marks as "black") the vertex a. Give counterexamples to the following statements:

1. If a directed graph G contains a path from a to b, and if $a.\mathsf{d} < b.\mathsf{d}$ in a depth-first search of G, then b is a descendant of a in the depth-first forest produced.

Solution: For the graph below, if we begin a DFS traversal starting at node A. And assume that the adjacency lists in the representation of the input graph are in alphabetical order and capitalized letters will always be arranged in front of lower case letters. The traversal of the following graph would be.

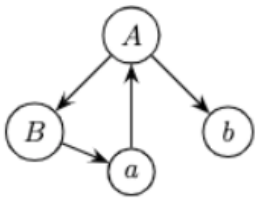


vertex	discovery time	finishing time
A	1	8
B	2	5
a	3	4
b	6	7

In this example, $a.\mathsf{d} = 3$ and $b.\mathsf{d} = 6$. And there is a path from a to b that there is a edge from a to A and another edge from A to b. But as the white path theorem we have proved in class states that "in a depth-first forest of a (directed or undirected) graph $G = (V, E)$, vertex v is a descendant of vertex u if and only if at the time $u.\mathsf{d}$ that the search discovers u, there is a path from u to v consisting entirely of white vertices"(CLRS). However, in our case, when vertex a is discovered, vertex A is already colored because the it is the vertex we started from. Thus we don't have a path from a to b consisting entirely of white vertices. Then b is not a descendant of a which make it a counter example.

2. If a directed graph G contains a path from a to b, then any depth-first search must result in $b.\mathsf{d} < a.\mathsf{f}$.

Solution: I will use the same counter example. For the graph below, if we begin a DFS traversal starting at node A. And assume that the adjacency lists in the representation of the input graph are in alphabetical order and capitalized letters will always be arranged in front of lower case letters. The traversal of the following graph would be.



vertex	discovery time	finishing time
A	1	8
B	2	5
a	3	4
b	6	7

It is very obvious that we have $b.\mathsf{d} = 6$ and $a.\mathsf{f} = 4$ and it contradict with $b.\mathsf{d} < a.\mathsf{f}$ which make this graph a counter example to the question.