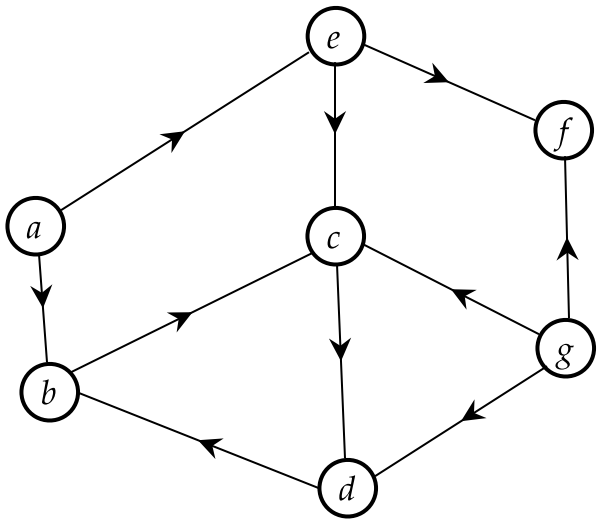
CMPSC 465 Data Structures & Algorithms
Spring 2022 Chunhao Wang and Mingfu Shao

Solution Part2

1. (8 + 6 + 6 = 20 pts.) Consider the directed graph G given below.

1. Run DFS-with-timing on this graph G: give the pre and post number of each vertex. Whenever there is a choice of vertices to explore, always pick the one that is alphabetically first.
2. Draw the meta-graph of G.
3. What is the minimum number of edges you must add to G to make it strongly connected (i.e., it consists of a single connected component after adding these edges)? Give such a set of edges.

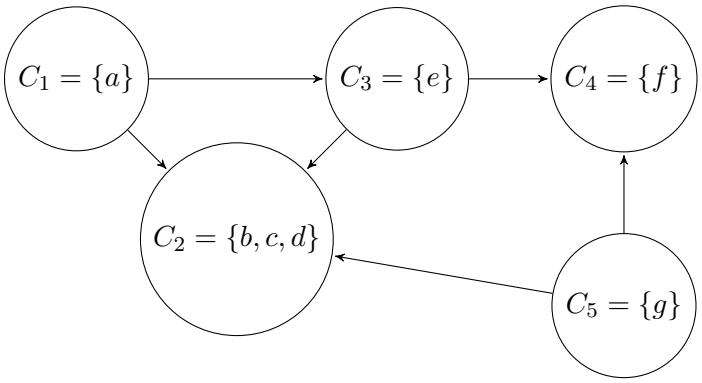


Solution:

1. Pre and post numbers of each vertex are as the following.

Vertex	Pre	Post
a	1	12
b	2	7
c	3	6
d	4	5
e	8	11
f	9	10
g	13	14

2. The meta-graph of G is as the following.



3. Since there are two sources C_1 and C_5, and two sinks C_4 and C_5 in the meta-graph, we need to add at least 2 edges to make G strongly connected itself. Two edges should be from a vertex in a sink meta-node to a vertex in a source meta-node and not share a common meta-node. There are two possibilities in terms of meta-nodes such as $\{(C_2, C_1), (C_4, C_5)\}$ or $\{(C_2, C_5), (C_4, C_1)\}$. Therefore, as an example, we can add two edges (b, a) and (f, g).

2. (20 pts.) You are given a directed graph $G = (V, E)$. Design an algorithm to determine if there exist two distinct vertices $u, v \in V$ such that for every vertex $w \in V \setminus \{u, v\}$ either w can reach u, or w can reach v, or both. Describe your algorithm, analyze its running time, and explain why it is correct. Your algorithm should run in $O(|V| + |E|)$ time. *(Hint: You may find Problem 3 in Assignment 06 relevant.)*

Solution: Firstly, we can transfer the problem to: for the reverse graph G^R of G, design an algorithm to determine if there exist two distinct vertices $u, v \in V$ such that for every vertex $w \in V \setminus \{u, v\}$ either w is reachable from u, or w is reachable from v, or both. We can make the following observations before designing an algorithm:

- **If such u, v exist, either u or v, or both must be in the source connected component(s) of G^R.** If both u, v are in non-source connected component(s), they wouldn't be able to reach to any vertex in a source connected component which contradicts with the requirement that all other vertices are reachable from either u or v, or both.
- **There can be at most two source connected components of G^R.** If there is only one source connected component, we can have either u or v, or both in the source connected component. If there are two source connected components scc_1, scc_2, u, v must in different source connected component (i.e. u in scc_1 and v in scc_2, or v in scc_1 and u in scc_2). If there are more than two source connected components, a vertex in one source connected component wouldn't be able to reach a vertex in another source connected component which contradicts with the requirement as well.

With these observations, the algorithm simply needs to check if there are at most two source connected components of G^R.

Algorithm:
Step 1: construct the reverse graph G^R of G, use G^R as the input graph for the following steps.
Step 2: based on Claim 1 in lecture A11 typed note and the fact that any meta-graph is a DAG, we know that the vertex with the largest post number is in a source connected component after running DFS-with-timing. So, we do this and get that vertex i.e. the first vertex in the postlist, store as v_1.
Step 3: run *explore* on v_1 and get the visited array $visited_1$. If all other vertices are visited in the procedure, the algorithm returns True. Otherwise, we know that there is certain connected component that the source connected component containing v_1 can't reach, go to step 4.
Step 4: for all unvisited vertexes in step 3, find the vertex with the largest post number, store as v_2.

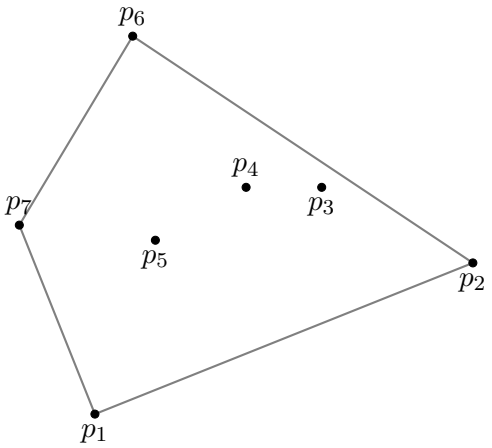
Step 5: run *explore* on v_2 and get the visited array $visited_2$. If all unvisited vertexes in step 3 are visited in the procedure, the algorithm returns True. Otherwise, we know that there is certain connected component that the source connected components containing v_1 and v_2 can't reach. Thus, it must be the case where there are more than two source components, and the algorithm can safely return False.

Running time: step 1 costs $O(|V|+|E|)$; step 2 costs $O(|V|+|E|)$ because DFS-with-timing simply records pre, post numbers for each vertex and creates a postlist as it does DFS; step 3 costs $O(|V| + |E|)$; step 4 costs $O(|V|)$; step 5 costs $O(|V| + |E|)$ (similar with step 2). Overall, the time complexity is $O(|V| + |E|)$.

Correctness explained in observations and algorithm.

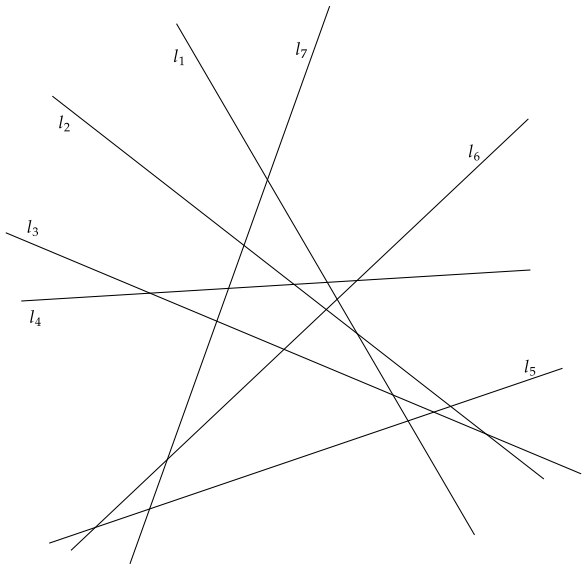
3. (7 + 7 = 14 pts.)

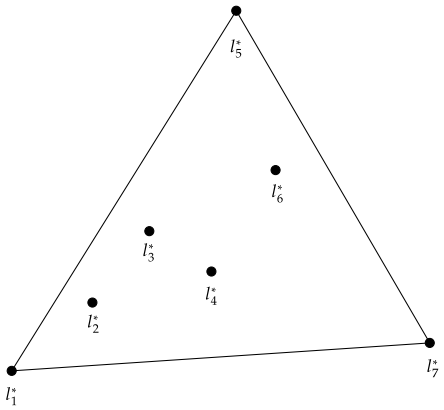
1. Design an instance of convex hull problem with 7 points, such that if Graham-Scan algorithm runs on your instance, the sequence of stack operations is (push, push, push, push, push, pop, pop, pop, push, push). Include a visual illustration of these 7 points and the convex hull. You will need to guarantee no three points in your instance locate on the same line.
   Solution: Based on the Gramham-Scan-Core pseudo-code covered in class and the stack operation sequence, we first push p_1 and p_2 on to the stack.



   - Since $p_1 \to p_2 \to p_3$ is "turning left",the 3rd operation is push.
   - Since $p_2 \to p_3 \to p_4$ is "turning left",the 4th operation is push.
   - Since $p_3 \to p_4 \to p_5$ is "turning left", the 5th operation is push.
   - Since $p_4 \to p_5 \to p_6$ is "turning right", the 6th operation is pop.
   - Since $p_3 \to p_4 \to p_6$ is "turning right", the 7th operation is pop.
   - Since $p_2 \to p_3 \to p_6$ is "turning right", the 8th operation is pop,.
   - Since $p_1 \to p_2 \to p_6$ is "turning left", the 9th operation is push.
   - Since $p_2 \to p_6 \to p_7$ is "turning left", the 10th operation is push.

2. Give 7 lines such that the convex hull of their dual points has 3 vertices. Include a visual illustration of your 7 lines and their dual points, and label which line corresponds to which dual point. You will need to guarantee no two lines are parallel and no three lines intersect at the same point.
   Solution: We know that upper-envelope and lower-envelope of lines is essentially the same with lower- and upper-hull in the dual plane. As we are asked to get only 3 points on the convex hull in dual plane, we need to have only 3 lines involved in upper and lower envelopes combined in the primal plane.



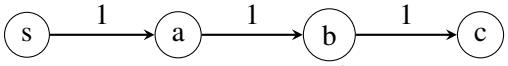


4. (8 + 15.5 = 23.5 pts.) You are given a directed graph $G = (V, E)$, a vertex $s \in V$, edge length $c(e)$ for each $e \in E$, and an integer $d > 0$; $c(e)$ could be negative but the graph does not contain negative cycles. We aim to find the length of the shortest path from s to each vertex such that the number of edges on it is at most d. In other words, we do not consider paths with more than d edges; among the paths from s to each vertex with at most d edges, we aim to find the shortest one.

1. A simple intuition is to use Bellman-Ford algorithm but run d instead of $|V| - 1$ rounds. Give an example where this algorithm does not work.

2. Design an algorithm that does work, analyze its running time, and explain why it is correct. Your algorithm should run in $O(|V| \cdot |E|)$ time. *(Hint: modify Bellman-Ford algorithm by adding a data structure companied with $dist$ array to keep track of the number of edges.)*

Solution:

1. 

Consider the graph above, $d = 2$. If Bellman-Ford goes through the edges in the order of $\{s, a\}$, $\{a, b\}$, and $\{b, c\}$, then just one iteration of the outer loop is enough to find the shortest path from s to c and update $dist[c]$ to be 3. However, the path contains 3 edges, as opposed to our limit of 2. So, this is incorrect considering the constraint in the question i.e. $dist[c]$ should not be updated and stay at the initial value ∞.
2. We modify Bellman-Ford algorithm for the constraint. Besides keeping track of the shortest distance from s to all the vertices, we also store the number of edges each shortest path has (let's say in an array $edge$, initialized as $edge[v] = 0$ for every $v \in V$). This can be updated in the update procedure. When the condition is met, $edge[v] = edge[u] + 1$ in addition to what's already in the procedure. Another change we need to make to the update procedure is that we have to include another condition $edge[u] < d$ to make sure $edge[v] \leq d$ after the potential $edge[v]$ update, besides the existing condition $dist[v] > dist[u] + l(u, v)$. If either condition is not met, we don't consider updating $dist[v]$ and $edge[v]$. These changes reflect the constraint in the question, so Bellman-Ford algorithm with these

changes is able to solve this shortest path question with the additional constraint on edges.
For your information, the pseudo-code for the modified Bellman-Ford algorithm looks like this:

Algorithm modified-Bellman-Ford ($G = (V, E)$, $l(e)$ for any $e \in E$, $s \in V$)
initialize an array $dist$ of size $|V|$;
$dist[s] = 0$; $dist[v] = \infty$ for any $v \neq s$;
initialize an array $edge$ of size $|V|$;
$edge[v] = 0$ for every $v \in V$;
for $k = 1 \rightarrow |V| - 1$ **do**
  for *each edge* $(u, v) \in E$ **do**
    $update(u, v)$;
  end
end
end algorithm;

procedure update (edge $(u, v) \in E$)
if $dist[v] > dist[u] + l(u, v)$ *and* $edge[u] < d$ **then**
  $dist[v] = dist[u] + l(u, v)$;
  $edge[v] = edge[u] + 1$;
end
end procedure;

We have three added step:

(a) initializing the array $edge$. This takes $O(|V|)$ time.

(b) the added condition $edge[u] < d$. Checking this takes constant time.

(c) the added update $edge[v] = edge[u] + 1$. This also takes constant time.

Overall, the modified Bellman-Ford algorithm takes $O(|V|) + O(|V| \cdot |E|) = O(|V| \cdot |E|)$ time as required.

Rubrics:

Problem 1, 20pts

- – 8pts : Provided the correct pre and post numbers.
  - – 7pts : Provided incorrect pre/post numbers but the order of them is consistent with the correct one.
  - – 5pts : Provided incorrect pre/post numbers but the correct DFS order from pre numbers.
  - – 6pts : Provided the correct one but missed the vertex g.
  - – 5pts : Provided incorrect pre/post numbers but the order of them is consistent with the correct one but missed the vertex g.
  - – 3pts : Provided incorrect pre/post numbers but the correct DFS order from pre numbers but missed the vertex g.
  - – 4pts : Did DFS but not in the alphabetical order and provided the correct one.
  - – 4pts : Did DFS on G^R and provided the correct one.
- – 6pts : Drew the correct meta-graph of G.
  - – 5pts : Drew an incorrect one with tiny mistakes such as adding/missing a few edges, putting opposite directions, or not specifying directions.
  - – 3pts : Drew an incorrect one but provided correct connected components.
- – 3pts : Found the correct minimum number of required edges.
  - – 3pts : Provided an edge set making G strongly connected.
  - – 1.5pts : Provided wrong answers but established an idea of sources and sink.
- 10% for each subproblem : I don't know how to answer this question.

Problem 2, 20pts

- 20 pts: Correct algorithm, running time analysis and appropriate explanation
- 18 pts: Partial credits for algorithm of checking two sources cc (G^R) or two sinks cc (G)
- 15 pts: Partial credits for algorithm of checking two source vertices or two sink vertices
- 13 pts: Partial credits for algorithm of checking one sources cc (G^R) or one sinks cc (G)
- 10 pts: Partial credits for DFS/BFS, largest postnumber, etc.
- 2 pts: I don't know how to answer this question.

Problem 3, 14pts

1. • 7 points: Provided correct example (i.e., all stack operations are correct).
   - 5 points: partially correct with atleast 7 stack operations correct
   - 3 points: partially correct with atleast 5 stack operations correct

- 2 points: less than 5 stack operatons are correct.
- 0.7 points: I don't know how to answer this question.

2.
- 4 points: Provided correct represntations in primal plane (total number of lines involved in upper and lower hyper planes is 3) .
- 3 points: partially correct representation with parallel lines.
- 3 points: partially correct representation with more than 2 lines interseting at same point.
- 2 points: partially correct representation with parallel lines and more than 2 lines interseting at same point.
- 3 points: Provided correct represntations in dual plane..
- 2 points: Partially correct representation with incorrect position of points in dual plane.
- 0.7 points: I don't know how to answer this question.

Problem 4, 23.5pts

1.
- 4 points: The provided graph is a correct counter-example.
- 4 points: Correctly explained why the naive solution doesn't work for the example.
- 3 points: Overall correct explanation but missed key information (e.g. the presumed order of edges we go through when running Bellman-Ford algorithm).
- 2 points: Unclear explanation.
- 0.8 points: I don't know how to answer this question.

2.
- 7.5 points: Provided an algorithm that does work.
- 6 points: Overall correct algorithm but missed certain detail (e.g. how to initialize $edge$ array).
- 4 points: The provided algorithm has correct aspects but doesn't work overall.
- 4 points: Correctly explained why the proposed algorithm works.
- 2 points: Missed explanations on a few key parts of the algorithm or the explanations on them are incorrect.
- 4 points: Correctly explained why the proposed algorithm has running time $O(|V| \cdot |E|)$.
- 2 points: Missed run time analysis on a few key parts of the algorithm or the analyses on them are incorrect.
- 1.55 points: I don't know how to answer this question.