CS-UY 2413: Design&Analysis of Algorithms—Spring 2017 Prof. Boris Aronov
Department of Computer Science and Engineering Prof. Lisa Hellerstein

Homework 7 Solutions

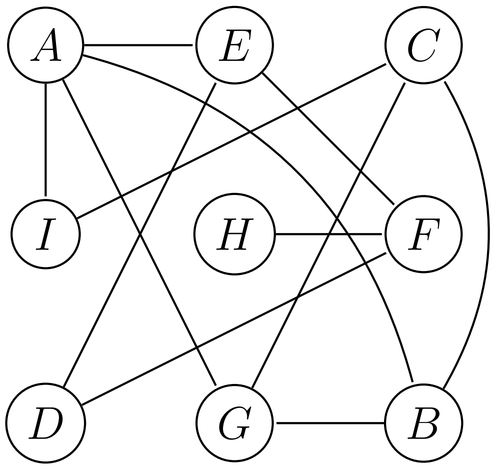
Due April 12 at the start of lecture (Aronov's sections).
Due April 13 at the start of lecture (Hellerstein's section).
This homework should be handed in on paper. No late homeworks accepted. Contact your professor for special circumstances.

Policy on collaboration on this homework: The policy for collaboration on this homework is the same as in previous homeworks. By handing in this homework, you accept that policy. Remember: A maximum of 3 people per group.

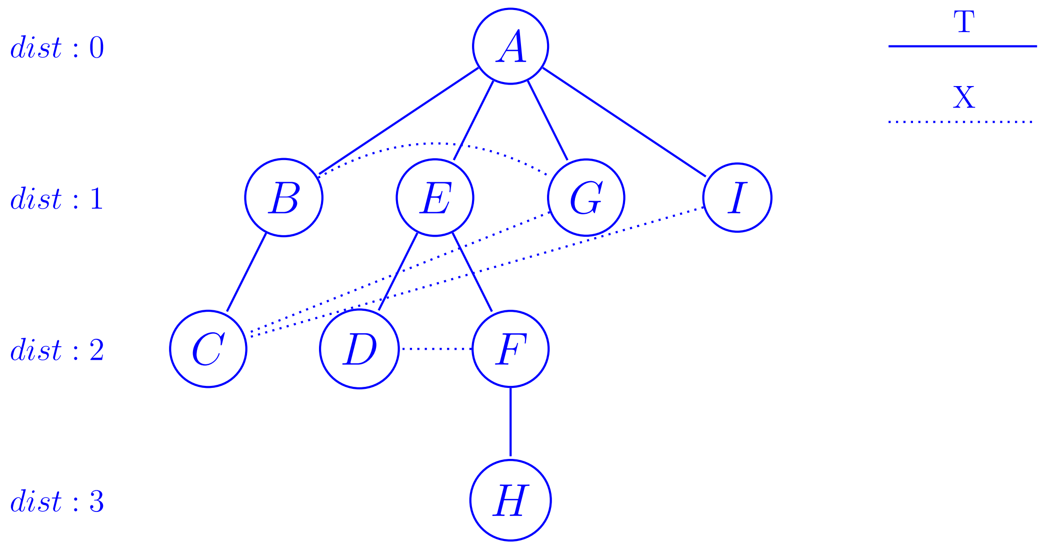
Notes: (i) Every answer has to be justified unless otherwise stated. Show your work! All performance estimates (running times, etc) should be in asymptotic notation unless otherwise noted.

(ii) You may use any theorem/property/fact proven in class or in the textbook. You do not need to re-prove any of them.

1. Run BFS on the given connected undirected graph, starting with vertex A. Show BFS tree and distance from A. Mark which edges are tree (T) edges and which are cross (X) edges.



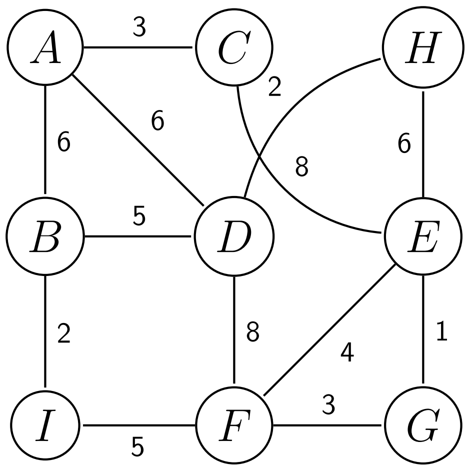
Solution:



2. Use Dijkstra's Algorithm to find the shortest distance from A to all other vertices of the graph given below. Draw a table showing the values in the `dist` and `prev` arrays after each iteration of the while loop in the pseudocode for Dijkstra's algorithm in Figure 4.8 of the textbook; use the format below.

Set	A	B	C	D	E	F	G	H
{}	0/nil	∞/nil	∞/nil	∞/nil	∞/nil	∞/nil	∞/nil	∞/nil
A	0/nil	6/A	3/A	6/A	∞/nil	∞/nil	∞/nil	∞/nil

Use alphabetical ordering to break any ties (favoring the letter that is closer to A). (Note: the weight of CE is 8 and that of DH is 2.)



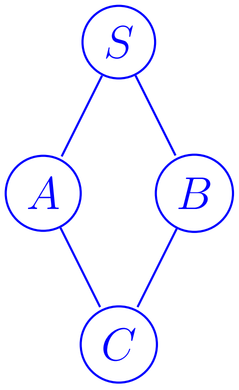
Solution:

Set	A	B	C	D	E	F	G	H	I
{}	0/nil	∞/nil	∞/nil	∞/nil	∞/nil	∞/nil	∞/nil	∞/nil	∞/nil
A	0/nil	6/A	3/A	6/A	∞/nil	∞/nil	∞/nil	∞/nil	∞/nil
AC	0/nil	6/A	3/A	6/A	11/C	∞/nil	∞/nil	∞/nil	∞/nil
ACB	0/nil	6/A	3/A	6/A	11/C	∞/nil	∞/nil	∞/nil	8/B
ACBD	0/nil	6/A	3/A	6/A	11/C	14/D	∞/nil	8/D	8/B
ACBDH	0/nil	6/A	3/A	6/A	11/C	14/D	∞/nil	8/D	8/B
ACBDHI	0/nil	6/A	3/A	6/A	11/C	13/I	∞/nil	8/D	8/B
ACBDHIE	0/nil	6/A	3/A	6/A	11/C	13/I	12/E	8/D	8/B
ACBDHIEG	0/nil	6/A	3/A	6/A	11/C	13/I	12/E	8/D	8/B
ACBDHIEGF	0/nil	6/A	3/A	6/A	11/C	13/I	12/E	8/D	8/B

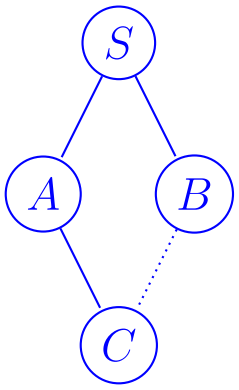
3. For each of the following situations, state whether it is possible. If not, explain why not. If yes, give a (small) example when it happens:

   (a) We run BFS on a connected undirected graph starting with vertex s. We remove an edge of the BFS tree and re-run BFS from s. The graph remains connected and all the distances to s remain the same.

   Solution: It is possbile.



Running BFS from S would produce the following tree:



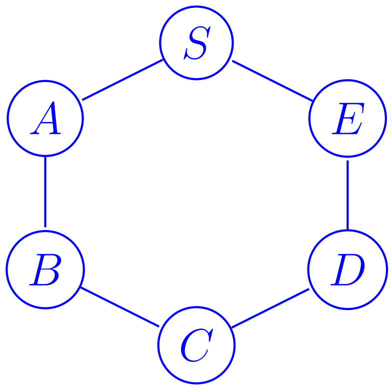
If tree edge AC is removed, then the cross edge BC would become a tree edge when BFS is run from S again. This would maintain C's distance of 2 while the distances to A and B will remain the same.

(b) We run BFS on a connected undirected graph starting with vertex s. We remove an edge that does not belong to the BFS tree. Some distances to s change.

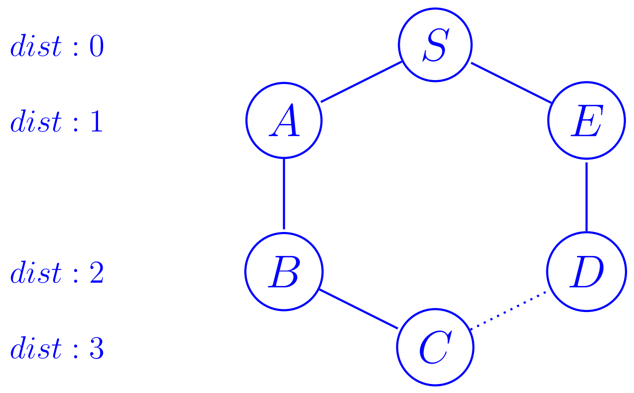
Solution: It is not possible. A BFS tree guarantees that any path from the source to a vertex within the tree is the shortest distance. If a non-tree edge is removed, it will not effect the resulting distances because the BFS algorithm would still be able to traverse the tree edges which provide the shortest distances.

(c) We run BFS on a connected undirected graph with 6 vertices, starting with vertex s. We remove an edge of the BFS tree and re-run BFS from s. The graph remains connected, but there is a vertex v whose distance from s increases by 4.

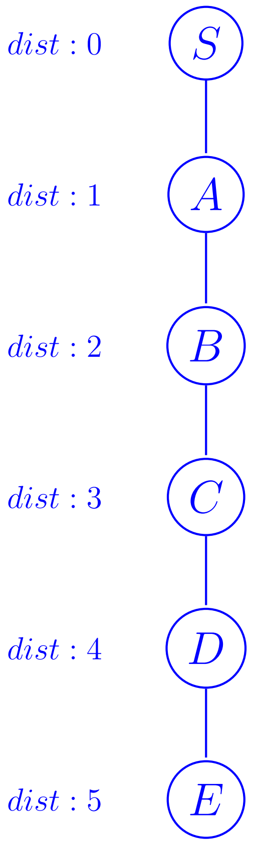
Solution: It is possible.



Running BFS from S would produce:



If edge SE is removed and BFS is run from S again, then the following would be produced:



The distance from S to E changed from 1 to 5.

4. Suppose we are given a directed graph G in adjacency list form: We have an array of list headers. Each list header $A[v]$ contains a pointer to the first element of a singly-linked list containing the vertices adjacent to vertex v and nothing else. G has n vertices and m edges. Assume vertices of G are numbers 1 through n.

   (a) For each of the following operations, explain how to implement it and give its worst-case running time, as a function of n and m.

- Given a pair (v, u), determine if it is an edge of G.
  Solution: Loop through the linked list of $A[v]$ and check if u exists in the linked list. If u is in $A[v]$ then (v, u) is an edge of G. This would take $O(m)$ in the worst case since all the edges could be in $A[v]$.
  Note: It can also be said that the worst case running-time is $O(n)$ since a vertex can have at most $n-1$ out-going edges in a directed graph. For dense graphs, this would be a tighter bound, but for sparse graphs, when $m < n$, $O(m)$ would be a tighter bound, both would still work as upper bounds.
- Given a vertex v, determine if it is a *source*: its *in-degree* is 0.
  Solution: Loop through all $A[k] : k \in V$ and check if v is in any of the linked lists. If it is in any of the linked lists, then it is a sink because there is an incoming edge to v. This would take $O(n+m)$ in the worst case because every vertex and every edge would have to be checked.
- Given a vertex v, determine if it is a *sink*: its *out-degree* is 0.
  Solution: Check if $A[v]$ is empty. If it is empty then there are no outgoing edges and it is a sink. This would be an $O(1)$ operation.

(b) Now design a new way to represent the graph, that still takes $O(n+m)$ space (so no adjacency matrices!) that speeds up as many of the above operations as you can. Describe the new representation, and the new implementation of each operation, and its worst-case running time.

No hashing, please.

Solution: Instead of $A[v]$ being a linked list, the neighbors reachable from v could be stored as a sorted array of vertices. To determine if an edge (v, u) is an edge of G, a binary search could be used on $A[v]$ to check if u exists in the array, this would take $O(\log m)$ time. A separate array could also be used to speed up determining if a vertex is a source. We can have $In[v]$ specify a vertex v's in-degree. With this extra array, determining whether a vertex v is a source would take $O(1)$ simply by checking whether $In[v]$ is zero. To check for whether the vertex v is a sink, we can just check if the array stored in $A[v]$ is empty. If the array is empty, it means v has no reachable neighbors and it's out-degree is zero. This would be a $O(1)$ operation. The amount of space used is still $O(m+n)$.

Alternatively, the reachable neighbors of v could be stored in $A[v]$ as a self-balancing binary search tree. The worst-case runtime for determining if an edge exists in G would be the same as a sorted array, $O(\log m)$. However, it offers improvement if edges are going to be added or removed. To add or remove an edge, it would take $O(\log m)$ for the self balancing binary search tree, but would take $O(m)$ for the sorted array representation. The extra array storing the in-degree would still be needed here to speed up determining if a vertex is a source. To determine if the vertex v is a sink, we can check whether there is any data stored in the search tree of $A[v]$. If there is no data, it means that there are no reachable vertices and the out-degree is 0. This would be a $O(1)$ operation. The amount of space used is still $O(m+n)$.

5. Modify the pseudocode for Dijkstra's algorithm in Figure 4.8 so that for every vertex v, it computes a Boolean value mulitple(v) which is True if there is more than one shortest path from s to v, and False otherwise. Write out the entire pseudocode, including your modification. (You do not need to include the descriptions of the Input and Output.)

   Give a brief explanation, in English, of how and why your modification works.

   Solution:

```
procedure dijkstra($G, l, s$)
Input:    Graph $G$ = ($V$, $E$), directed or undirected;
          positive edge lengths $\{l_e : e \in E\}$; vertex $s \in V$
Output:   For all vertices $u$ reachable from $s$, dist($u$) is set
          to the distance from $s$ to $u$, multiple($u$) is set to
          $True$ if there is more than one shortest path from
          $s$ to $u$.

for all $u \in V$:
   dist($u$) = $\infty$
   prev($u$) = nil
   multiple($u$) = $False$
dist($s$) = 0

$H$ = makequeue ($V$) (using dist-values as keys)
while $H$ is not empty:
   $u$ = deletemin($H$)
   for all edges ($u$, $v$) $\in E$:
      if dist($v$) > dist($u$) + $l$($u$, $v$):
         dist($v$) = dist($u$) + $l$($u$, $v$)
         prev($v$) = $u$
         multiple($v$) = multiple($u$)
         decreasekey($H$, $v$)
      else if dist($v$) == dist($u$) + $l$($u$, $v$):
         multiple($v$) = $True$
```

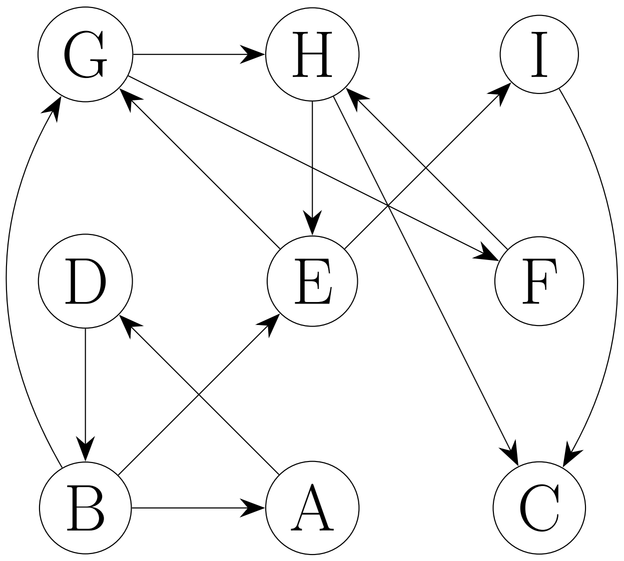
In the initialization phase, we set `multiple(u)` to be *False* for all $u \in V$ because we do not know if a vertex has more than one shortest path yet. When we update the shortest path information for vertex v (as in Dijkstra's Algorithm), we set `multiple(v)` to `multiple(u)` because the number of paths found for a larger distance are no longer relevant, but if there are multiple paths to u then it means there are multiple paths to v. To check for when Dijkstra's Algorithm has found a separate shortest path to v, we add a separate check to see if the distances of the path are the same. If they are the same, then we have found a separate shortest path to v and set `multiple(v)` to *True*.

6. **Hellerstein's Section Only:**

   You did not do the Strongly Connected Components problems in Homework 6. To give you practice on this subject, you should do Problems 5c and 5d from Homework 6, but you don't need to hand them in. Check the solution set to Homework 6 to see if your answers are correct.

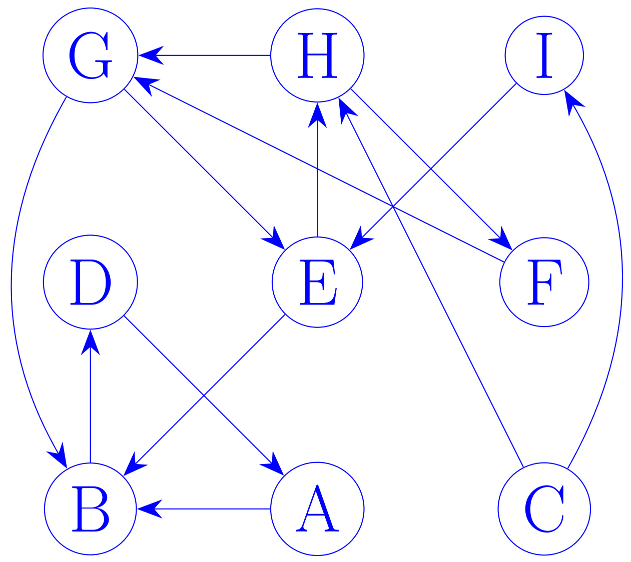
   Then, do the following problem, and HAND IT IN with your solutions to the rest of the problems on this homework.

   On the given directed graph, run the strongly connected components (SCC) algorithm described in the textbook. Show the first DFS forest, give the list of vertices in decreasing order of their post times, and finally show the strongly connected components identified by the algorithm. (You do not need to show the second DFS forest that is computed.)

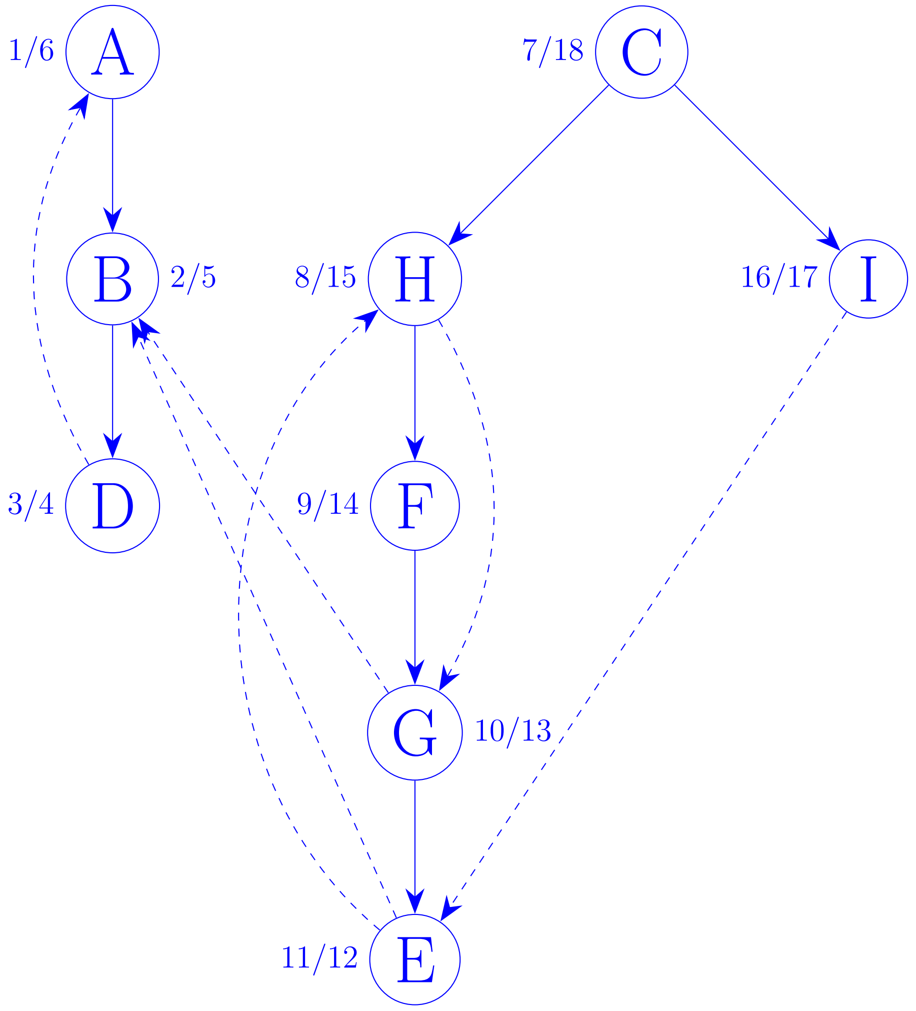


   Solution: The SCC algorithm consists of running DFS on G in the order of decreasing post numbers of the DFS forest of G^R

   The reversed graph, G^R:

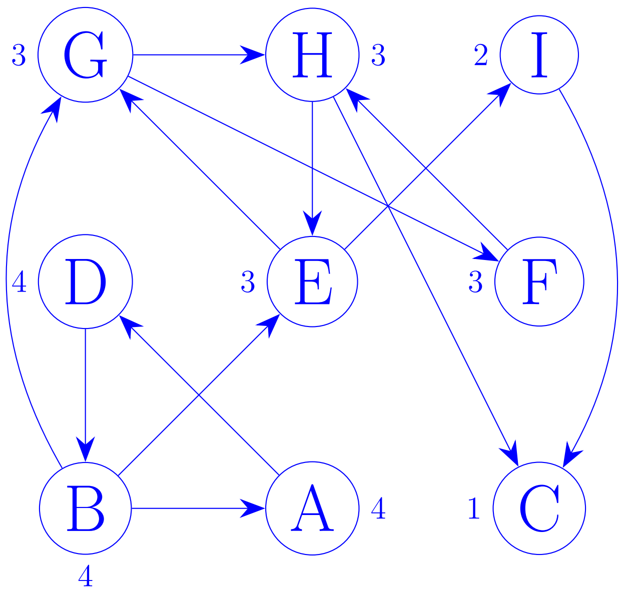


The DFS forest of G^R:



The ordering used by the modified DFS procedure to find SCCs will be: {C, I, H, F, G, E, A, B, D}.

Running the modified DFS procedure on the original graph using the decreasing post number ordering of G^R:



Grouping the connected component numbers will give back the strongly connected components: {{C}, {I}, {E, F, G, H}, {A, B, D}}