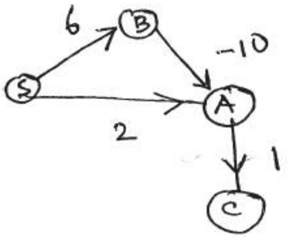
CS 721: Advanced Algorithms & Analysis

Homework 2 Solution, Fall 2018, Total 90 points

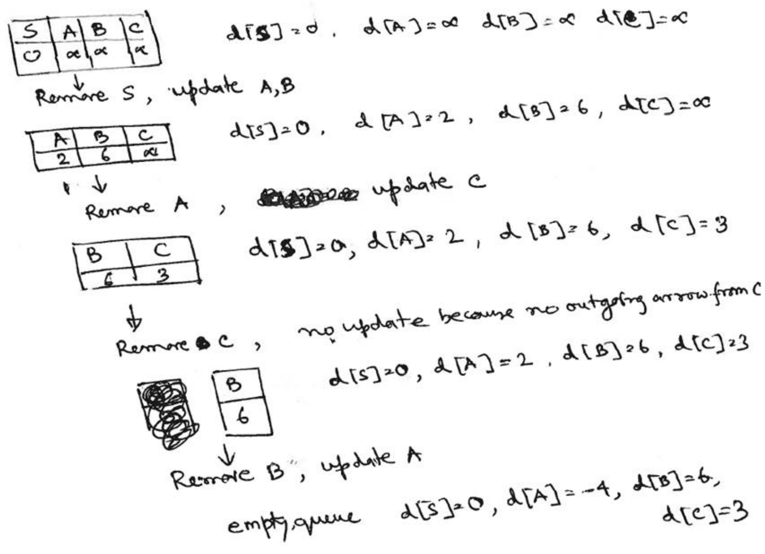
Assigned on: Tuesday, 08/18/2018
Due on: Thursday, 09/27/2018

1. In Dijkstra's algorithm a node is removed from the priority queue only when "the algorithm thinks" its shortest distance from the source node to this node is found. If all edge weights are positive, this shortest distance estimates will indeed be the correct shortest distance once a node is removed from the queue and this shortest distance estimate will never be updated again once it is removed from the queue, meaning that after a node's removal we can not find an alternate path to that node with a shorter distance. This does not hold if some edge weights



are negative. In the following figure once node C is removed, Dijkstra's algorithm estimates its shortest distance to be 3 ($S \rightarrow A \rightarrow C$), but can not find the correct alternate shortest distance which is -3 ($S \rightarrow B \rightarrow A \rightarrow C$). The reason is that as per Dijkstra's algorithm, node B is the last node to be removed from queue (after which the queue becomes empty), at that point only node A's shortest distance can be updated (since there is an outgoing edge from B to A) but not C's shortest distance.

The following figure shows the order in which nodes are removed as per Dijkstra's algorithm, queue contents with key values and information regarding which node is updated when. The notation $d[v]$ denotes shortest path estimate of node v from the start node S as per Dijkstra's algorithm.



S	A	B	C
0	∞	∞	∞

d[S] = 0 . d[A] = ∞ d[B] = ∞ d[C] = ∞

Remove S , update A,B

A	B	C
2	6	∞

d[S] = 0 , d[A] = 2 , d[B] = 6 , d[C] = ∞

Remove A , update C

B	C
6	3

d[S] = 0, d[A] = 2 , d[B] = 6, d[C] = 3

Remove C , no update because no outgoing arrow from C

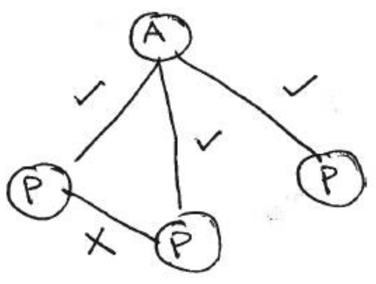
d[S] = 0 , d[A] = 2 . d[B] = 6 , d[C] = 3

B
6

Remove B , update A

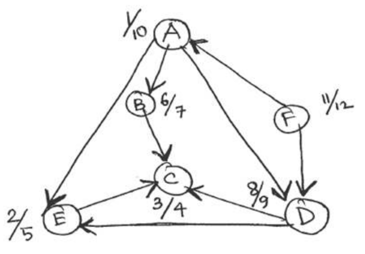
empty queue d[S] = 0 , d[A] = -4, d[B] = 6, d[C] = 3

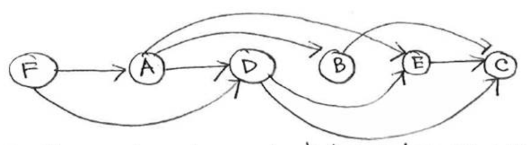
2. Create an undirected graph where each vertex represents a golf player and each edge represents a rivalry. The graph will have n vertices and r edges. Perform as many BFS's as needed to visit all vertices. Assign all golf players whose distance is even to be professionals and all golf players whose distance is odd to be amateurs. Then check each edge to verify that it goes between a professional and an amateur. If in the given graph rivalry always exists between professionals and amateurs and not between amateurs and amateurs or professions and professionals then node designation after BFS will be enough. However, if rivalry also exists between professionals and professionals or between amateurs and amateurs then no designation of nodes such that rivalry exists between professionals and amateurs is possible. This can be done by going through each edge to check that opposite end nodes of this edge are of different type. For example, in the following graph, while checking each edge after BFS, it is clear that there is also a rivalry between professional and professional. That means no designation of nodes, such that rivalry is always between professionals and amateurs, is possible in this case.



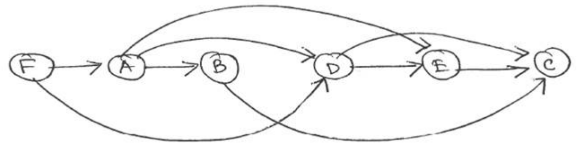
The solution will take $O(n+r)$ time for BFS, $O(n)$ time to designate each golf player as a professional or amateur (looking at odd or even distance value from root node), and $O(r)$ time to check each edge. Thus total time is $O(n+r)$.

3. (a) Topoplogically sorted graph is as follows





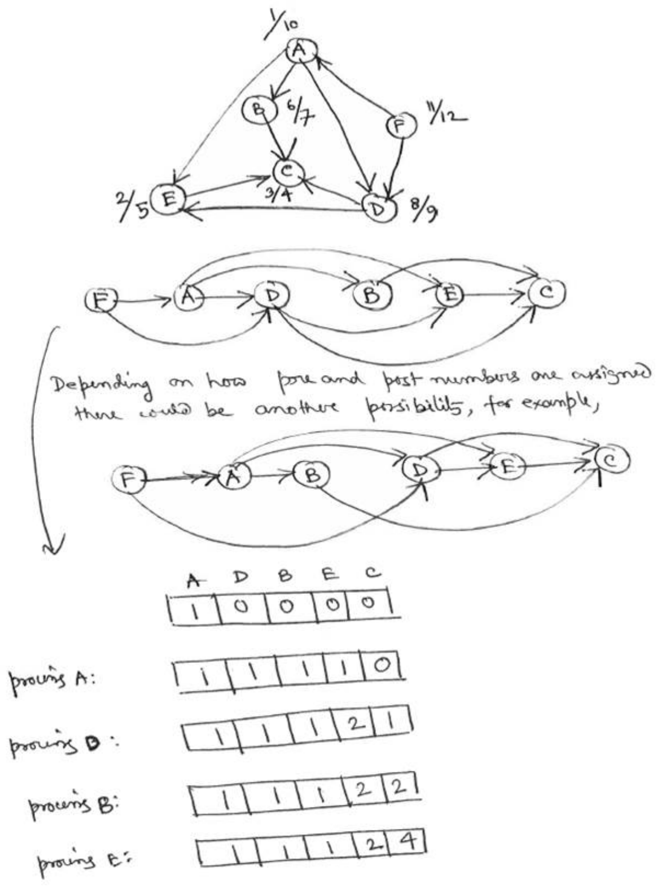
Depending on how pre and post numbers are assigned there could be another possibility, e.g.;



(b) Find a linearized ordering of the nodes of this graph (topological sorting). Any node that appears in simple path between s and t must appear between nodes s and t in this ordering. Count the number of nodes appearing between s and t (including s and t). Say this number be t. Assign an array of size t, one for each of the t nodes in linearized order(array index) which will contain the number of paths from s to that node. First index of this array corresponds to node s, the second index corresponds to the node next to s in the linearized ordering and so on. The last index corresponds to node t. For any node u let $f(u)$ be the number of paths from s to u. Since $f(s) = 1$ and initialize the array as follows. Set the array element corresponding to index s to 1 and all other array elements to zero.

Now start processing each node in the linearized order from s to t. While processing a node say u, consider each outgoing edge from this node (u, v), and update the array element corresponding to the node v to be $f(v) + f(u)$, that is increment the content of array element at index v by the content of the array element indexed by u. The meaning of this increment is as follows, suppose there 3 ways to reach to v from s and there are 2 ways to reach to u from s. Since we are processing the nodes in the linearized order, if there is an edge from u to v, v now can be reached from s in 3+2=5 different ways.

The steps to compute number of simple paths from A to C is shown below.


1/10
A
B
6/7
F
11/12
C
3/4
2/5
E
D
8/9
Depending on how pre and post numbers are assigned there could be another possibility, for example,

A	D	B	E	C
1	0	0	0	0

procing A:	1	1	1	1	0
procing D:	1	1	1	2	1
procing B:	1	1	1	2	2
procing E:	1	1	1	2	4

Time for topological sorting is $O(|V| + |E|)$. The additional processing for simple path takes at most another $O(|V| + |E|)$ time. Thus total running time is $O(|V| + |E|)$.

4. (a) The problem will occur if the node with lowest post number computed on G^R happens to lie in the source strongly connected component of G^R. Then if DFS is started from this node, it may be possible that all nodes G^R are reachable from this node, and thus the algorithm will say there is only one strongly connected componnet which is wrong.

   (b) The following example illustrate this idea. If DFS is started from node B in G^R then A might have lowest post number.

5. (a) For the same graph, set the edge weight of an edge (u, v) to be $l(u, v) = \log\left(\frac{1}{r(u,v)}\right)$. Given two vertices s and t, find shortest path on this graph from s to t using Dijkstra's algorithm. Running time is $O(|V| + E|) \log(|V|)$.

Consider

$$\begin{tikzcd} G: & A \arrow[r, bend left] & B \arrow[l, bend left] & C \arrow[l] \end{tikzcd}$$

$$\begin{tikzcd} G^R: & \overset{2/3}{A} \arrow[r, bend right] & \overset{1/6}{B} \arrow[l, bend right] \arrow[r] & \overset{4/5}{C} \end{tikzcd}$$

Order of increasing post numbers in G^R: A, C, B
Starting DFS with A, will identify {A,B,C} as strongly connected component where as true strongly connected components are {A, B} and {C}

Why this works? Suppose the shortest path from s to t is given by $s \to u \to v \to w \to t$. Then the total weight along this shortest path is $\log\left(\frac{1}{r(s,u)}\right) + \log\left(\frac{1}{r(u,v)}\right) + \log\left(\frac{1}{r(v,w)}\right) + \log\left(\frac{1}{r(w,t)}\right)$. Using properties of logarithm, this quantity is as same as $\log\left(\frac{1}{r(s,u)\times r(u,v)\times r(v,w)\times r(w,t)}\right)$. Because logarithm is a monotonically increasing function, this will be minimum when $r(s,u) \times r(u,v) \times r(v,w) \times r(w,t)$ is maximum.

(b) If the directed graph does not contain a cycle, we can topologically sort and use similar ideas as in Q.3 to get the solution in $O(|V| + |E|)$ time. Basically, we process the nodes in linearized order from left to right and for each outgoing arrow, we update the shortest distance value of the node at the other end. The pseudo-code will look something like this:

```
for each $u \in V$ :
  $dist[u] = \infty$
  $prev[u] = nil$
$dist[s] = 0$

for each $u \in V$ in linearized order from left to right:
    for all edges $(u, v) \in E$ :
        if $dist[v] > dist[u] + w(u, v)$ :
            $dist[v] = dist[u] + w(u, v)$
            $prev[v] = u$
```

Essentially we get rid of the deletemin and decreasekey operations of Dijkstra's algorithm which takes $O(\log |V|)$ time each. the above algorithm processes each vertex and each edge exactly once and thus its running time is $O(|V| + |E|)$.