Computer Science CSC263H — March 15, 2018
St. George Campus — University of Toronto

Solutions for Homework Assignment #5

Answer to Question 1. CLAIM: $\frac{5}{2}$ is the best (i.e., smallest) upper bound for $T(n)/n$ in the list L.

- **First show for all n: $T(n)/n \leq \frac{5}{2}$.**

  To show this, we use the accounting method, and charge operations as follows:

  (a) $\frac{5}{2}$ for each INSERT(x) operation, and (b) 0 for each OUTPUTANDREDUCE().

  We claim that the following *credit invariant* holds:

  The total credit of any array A of size n is at least $n \times \frac{3}{2}$ (*)

  Suppose (*) holds just before an operation o on an array A of size n, we need to show that (*) also holds immediately after operation o. There are two cases:

  (a) o is an insert operation. Since (*) holds, before o is executed the array A has at least $n \times \frac{3}{2}$ credits. After executing o, the array's amount of credits is at least $n \times \frac{3}{2} + (\frac{5}{2} - 1) = (n+1)\frac{3}{2}$, and the array's size is $n+1$. So (*) still holds.

  (b) o is a OUTPUTANDREDUCE() operation. Before o, the array A has at least $n \times \frac{3}{2}$ credits. Executing o costs exactly n credits (because o prints the n elements of A), so array A has at least $n \times \frac{3}{2} - n = n \times \frac{1}{2} = \frac{n}{3} \times \frac{3}{2}$ credits left.

  Since the size of A is now at most $n' = n/3$, A has at least $n' \times \frac{3}{2}$ credits, and therefore the invariant (*) still hold.

  So (*) always holds. From (*), the amount of accumulated credits is always positive. So the total amount charged to any sequence of $n \geq 1$ operation (more than) covers the total cost of executing these operations. Since we charge at most $\frac{5}{2}$ per operation, we have $T(n)/n \leq \frac{5}{2}$.

  ALTERNATIVE PROOF:

  To show $T(n)/n \leq \frac{5}{2}$, we prove that if we charge:

  (a) $\frac{5}{2}$ for each INSERT(x) operation, and (b) 0 for each OUTPUTANDREDUCE()

  then we always cover the total cost of any sequence of operations.

  Since the actual cost of doing an insert is exactly 1, after an element is inserted, it has a remaining credit of $\frac{5}{2} - 1 = \frac{3}{2}$. In fact, we will maintain the invariant that every element in the array always has $\frac{3}{2}$ credit on it.

  To pay for the cost of executing a OUTPUTANDREDUCE() operation, we first take one credit from every element that is printed (so if this prints k elements, this takes k credits out of the array).

  Note that after doing so, the remaining credit attached to each element is now $\frac{3}{2} - 1 = \frac{1}{2}$.

  Since the OUTPUTANDREDUCE() operation effectively removes the last 2/3rd of the array (only fhe first third of the array remains), we now give to *each* element in the first third of the array the $2 \times \frac{1}{2}$ credits of *two* elements located in the removed part of the array.

  After doing this, we are back to a state where each element in the array has $\frac{1}{2} + 1 = \frac{3}{2}$ credit attached to it.

- **Then show that for some n: $T(n)/n > 2$.**

  Consider the following sequence of operations: first insert 27 elements, then print 27, print 9, print 3, and finally print 1.

  This is a sequence of $n = 27 + 4 = 31$ operations whose total cost $T(n) = 27 + 27 + 9 + 3 + 1 = 67$.

  So $T(n)/n = 67/31 > 2$.

Answer to Question 2.

a. Assume that the directed graph G is given by its adjacency matrix A, so $A[i,j] = 1$ if and only if G has an edge from node i to node j. To find out if the graph G has a supersource, the basic idea is to use each comparison "$A[i,j] = 0$?", which can be done in $O(1)$-time using matrix A, to *eliminate one of node i or j* as a possible candidate for being a supersource: if $A[i,j] = 0$ (for $i \neq j$), then node i cannot be supersource; and if $A[i,j] \neq 0$, then node j cannot be a supersource. So with $n-1$ questions, i.e., in $O(n)$-time, we can eliminate $n-1$ nodes, and we remain with only one possible supersource candidate, say node s. It is now easy to see whether s is indeed a supersource by checking whether the following two conditions hold: (a) $A[s,j] = 1$ for every $j \neq i$ (there is an edge from s to every other node), and (b) $A[j,s] = 0$ for every j (there are no edges into s). This check, which involves scanning a row and a column of A, takes $O(n)$-time. More precisely, the pseudo-code of the algorithm is:

$s \leftarrow 1$ {s is the current supersource candidate}
$j \leftarrow 2$ {j is another candidate }
while $j \leq n$ **do**
  if $A[s,j] = 0$ **then** $s \leftarrow j$ { if there is no edge from s to j, then j becomes the new supersource candidate}
  $j \leftarrow j+1$
{ At the exit of the above loop, *if* G has a supersource *then* it must be s }
{ We now check whether s is indeed a supersource.}
for $j \leftarrow 1$ **to** n
  if $A[s,j] = 0$ **and** $j \neq s$ **then print** "G does not have a supersource"; **stop**.
  if $A[j,s] = 1$ **then print** "G does not have a supersource"; **stop**.
print "G has a supersource and it is" s ; **stop**

Proof of the Algorithm: It is not difficult to prove that at the end of each iteration of the **while** loop the following invariant holds:[1]

Invariant: $[1 \leq s < j \leq n+1]$ **and** [**if** G has a supersource **and** it is among $\{1, 2, \ldots, j-1\}$ **then** it must be s].

At the exit of the **while** loop $j = n+1$. So, by the above loop invariant, if there is a supersource among $\{1, 2, \ldots, n\}$ then it must be s. The code that follows the **while** loop simply checks whether s is indeed a supersource (i.e., whether there is an edge from s to every other node but there is no edge into s).

b. The **while** loop above executes $n-1$ times, so it accesses the matrix A exactly $n-1$ times. The **for** loop executes n times, and each iteration accesses the matrix A twice, so it accesses the matrix $2n$ times. So the total number of accesses to A of this algorithm is $3n-1$.

c. Now assume that the graph G is given by its adjacency lists L. To check whether there is an edge from a node i to a node j now requires traversing the list $L[i]$ to see whether j is in this list. In the worst case this takes $\Theta(n)$ time. So the above algorithm now takes $\Theta(n^2)$ time in the worst-case.

A more efficient algorithm (for the case where G is given by its adjacency lists) is sketched below:

1. For each $i, 1 \leq i \leq n$, check whether the list $L[i]$ contains exactly $n-1$ nodes. If there is *exactly one* node s such that $L[s]$ contains exactly $n-1$ nodes, then s is a supersource candidate; otherwise print "G does not have a supersource" and stop.

   This step takes $\Theta(n+m)$ time in the worst-case.

2. If Part 1 above yielded a supersource candidate s, check that for all $i, 1 \leq i \leq n$, node s is *not* on list $L[i]$; if so print "G has a supersource and it is" s, else print "G does not have a supersource", and stop.

   This step also takes $\Theta(n+m)$ time in the worst-case.

Answer to Question 3.

a. We run a BFS starting from any node of G to create a BFS tree. Then we direct every tree edge from the parent to the child node. The non-tree edges are oriented arbitrarily. Because the running time of this algorithm

[1] By convention, the end of the 0-th iteration of the loop is the beginning of the first iteration of the loop.

is dominated by the running time of BFS, the algorithm runs in time $O(m + n)$. We claim that, after the edges are oriented, every node of G, except the root of the BFS tree has at least one edge going into it. Indeed, every node in the BFS tree except the root has a parent node, and the edge connecting them is oriented from the parent to the child node.

b. Since each $v \in V$ has at least one edge (u, v) going into it (for some $u \in V$) in the oriented graph G, and every directed edge goes into exactly one vertex, the oriented graph G must have at least n directed edges. Since each undirected edge of the original graph yields exactly one directed edge in the oriented graph, the undirected graph must hav at least n undirected edges as well.

c. We run BFS once, from an arbitrary vertex. Because the graph is connected and has at least n edges, while the BFS tree has $n - 1$ edges, there must be at least one non-tree edge; let us call this edge $e = (u, v)$. We direct this edge from v to u, and remove it from the graph to get a new graph $G' = (V, E - \{e\})$. Because G' still has all edges that were in the BFS tree, it is still connected. We run BFS starting from u, and orient all edges in the new BFS tree from parent to child node; we orient all other edges arbitrarily. By the analysis in the first subquestion, this leaves only vertex u possibly without any edge going into it. However, we oriented edge e so that it goes into u, and, thus, we have at least one edge going into every vertex. This proves the correctness of the algorithm. The running time is $O(m + n)$ because it is dominated by the two runs of BFS, each of them taking time $O(m + n)$.