CSCI 3104-300 Summer 2019 Problem Set 6
Hoffman, Ryan
01/11

1. (15 points) Give an $O(VE)$-time algorithm for computing the transitive closure of a directed graph $G = (V, E)$. Compute its asymptotic running time.

Solution:

Run a single-source shortest path algorithm from each of the *(*V) vertices in the graph. Here, we use *(*BFS) $|V|$ times, giving us $O(VE)$-time. In the worst-case we iterate through the entire set of edges. BFS has $O(V + E)$-time.

I REALLY wanted to write pseudocode for this problem but I can't figure out how to do it correctly. . . But here is my attempt:

Algorithm 1

```
transClosure($G$):
Initialization of matrix
for each vertex $u \in V$ do:
    run $BFS(u)$
    for each vertex $v$ do:
        add element to the matrix
    return the matrix
```

2. (15 points) Grog –master of pictures– needs your help to compute the in- and out-degrees of all vertices in a directed multigraph G. However, he is not sure how to represent the graph so that the calculation is most efficient. For each of the three possible representations, express your answers in asymptotic notation (the only notation Grog understands), in terms of V and E, and justify your claim.

   (a) An *adjacency matrix* representation. Assume the size of the matrix is known.

   Solution:

   Instead of storing a 1 or a 0 in the adjacency matrix, it would be best to store the key to a Hash table in which the corresponding edge list can be found. In this way, we still have the matrix of size $|V|$by $|V|$ which requires $O(V^2)$-time.

   (b) An *edge list* representation. Assume vertices have arbitrary labels.

   Solution:

   From our notes, since an edge list only stores the set of edges in E, the set of vertices will also need to be traversed. This gives us $O(VE)$-time complexity.

   (c) An *adjacency list* representation. Assume the vector's length is known.

   Solution:

   The adjacency list representation stores a vector of length n vertices. Each element (vertex) points to a linked list which we then take the size as the out-degree for that vertex. To perform a linear search we know that the time complexity will be $O(V)$.

3. (40 points) Consider a valleyed array $A[1, 2, \ldots, n]$ with the property that the subarray $A[1 \ldots i]$ has the property that $A[j] > A[j+1]$ for $1 \leq j < i$, and the subarray $A[i \ldots n]$ has the property that $A[j] < A[j+1]$ for $i \leq j < n$. For example,
$A = [16, 15, 10, 9, 7, 3, 6, 8, 17, 23]$ is a valleyed array.

   (a) Write a recursive algorithm that takes asymptotically sub-linear time to find the minimum element of A.

   Solution:

   A *sub-linear*-time algorithm would be something like $log(n)$ or n^c for $c < 1$. I don't know how to do all the fancy stuff like making a recurrence relation but I know that to get this to be sub-linear we need to simply take, as input, less than n items. Now, since the valleyed array we are given is basically just two sorted arrays (decreasing and increasing), we can find the minimum element in the middle of the list.

   (b) Prove that your algorithm is correct. (Hint: prove that your algorithm's correctness follows from the correctness of another correct algorithm we already know.)

   Solution:

   Similar to *Quicksort*, which has $O(nlogn)$ running time, the above would simply use less than n elements giving us our expected sub-linear running time.

   (c) Now consider the multi-valleyed generalization, in which the array contains k valleys, i.e., it contains k subarrays, each of which is itself a valleyed array. Let $k = 2$ and prove that your algorithm can fail on such an input.

   Solution:

   Well, my algorithm fails because it divides the array into two pieces essentially. Therefore we don't need any information about the second half of the array. Now though, we are looking at a wave array I believe, with multiple peaks and valleys.

   (d) Suppose that $k = 2$ and we can guarantee that neither valley is closer than $n/4$ positions to the middle of the array, and that the "joining point" of the two singly valleyed subarrays lays in the middle half of the array. Now write an algorithm that returns the minimum element of A in sublinear time. Prove that your algorithm is correct, give a recurrence relation for its running time, and solve for its asymptotic behavior.

   Solution:

   Like I stated above, I can't "prove" that my algorithm is correct or give a recurrence relation for its running time and solve for its asymptotic behavior but, I know it still holds true because it makes sense to me that if we start in the middle, the closest valley is $n/4$ positions away.