CSE 4502/5717 Big Data Analytics. Fall 2022
Exam II Solutions

1. Consider the following algorithm:

    for every pair of vertices (a, b) in V **do**
    Bring the adjacency lists $A[a]$ and $A[b]$ of a and b, respectively, from the disk.
    if $A[a]$ and $A[b]$ have a common node c **then output** (a, b, c) and **quit**;

    Analysis: The **for** loop is run in the worst case $O(|V|^2)$ times. In each run of the **for** loop, we bring in $O(|V|)$ nodes from the disk. Thus each run of the **for** loop involves $O\left(\frac{|V|}{B}\right)$ I/O operations. Thus, the entire algorithm makes $O\left(\frac{|V|^3}{B}\right)$ I/O operations in the worst case.

2. We first merge R_1 and R_2 to get S_1. We also merge R_3 and R_4 to get S_2. Finally, we merge S_1 and S_2.

    We can merge R_1 with R_2 and merge R_3 with R_4 in one pass through the data. Consider the problem of merging R_1 and R_2. We start by bringing BD elements from R_1 and BD elements from R_2 into the core memory. We start merging them. We write the merged output into an output buffer of size BD (residing in the core memory). When the buffer is full, we write these BD elements across the disks in parallel and clear the buffer. When we run out of elements from any of the runs, we bring the next BD elements from that run. Clearly, we can merge R_1 and R_2 by bringing each element of these runs only once into the core memory. In the same way we can merge R_3 and R_4.

    Now we have two sorted sequences S_1 and S_2 of length $2M^2$ each. We can also merge them in a similar manner in one pass through the data.

3. Note that for this problem, $B = M^{0.75}$ and $D = M^{0.25}$. Here is an algorithm:

    (a) Form runs of length M each; There are $M^{0.25}$ runs that we have to merge. Let these runs be $X_1, X_2, \ldots, X_{M^{0.25}}$.
    (b) Unshuffle each run into $M^{0.25}$ parts. Let the parts of X_i be $X_i^1, X_i^2, \ldots, X_i^{M^{0.25}}$, for $1 \leq i \leq M^{0.25}$.
    (c) Recursively Merge $X_1^j, X_2^j, ..., X_{M^{0.25}}^j$ to get Y_j, for $1 \leq j \leq M^{0.25}$.
    (d) Shuffle $Y_1, Y_2, \ldots, Y_{M^{0.25}}$ to get Z.
    (e) Clean up the dirty sequence in Z.

    Analysis: Note that we have used LMM with $\ell = m = M^{0.25}$. Steps (a) and (b) take 1 pass together. Step (c) takes 1 pass.

    Assume that we have a memory of size $2M$. In this case we can clean up the dirty sequence while we are shuffling. Let Z be partitioned into blocks of size M each: $Z = Z_1, Z_2, \ldots$, where each block Z_i is of size $\ell m = \sqrt{M}$. Note that the dirty sequence can only span two successive blocks. Therefore, one way of cleaning the sequence Z is to: sort and merge Z_1 and Z_2; Z_2 and Z_3; etc. If we have $2M$ memory, we can do this cleaning as well as Step (d) in a total of one pass.

As a result, Steps (d) and (e) take 1 pass.

In summary, the total number of passes = 3.

4. We build a generalized suffix tree Q on all of the $(k+1)$ input strings. The time needed is $O(kn)$. Note that the size of the tree Q is $O(kn)$. For each node u in Q we associate a bit array $b^u[1:k+1]$. We start from the leaves and proceed towards the root as follows. If v is a leaf, and if it represents suffixes from S_i, for any i, $1 \leq i \leq k$, then set $b^v[i] = 1$ and if v does not represent any suffix of S_j, for any j, $1 \leq j \leq k$, then set $b^v[j] = 0$. If v has a label corresponding to a suffix of T, then set $b^v[k+1] = 1$ and if v does not have a label corresponding to a suffix of T then set $b^v[k+1] = 0$. If N is an internal node, then $b^N[1:k+1]$ is computed as the boolean OR of the bits arrays of its children. We spend $O(k)$ time at each node and hence the total time for computing the bit arrays for all the nodes of Q is $O(k^2n)$.

   After computing the bit arrays for the nodes of Q, traverse through Q to identify the node whose bit array has all ones in the first k positions and a zero in position $k+1$, and whose string depth is the largest. Output the path label of this node. This traversal also takes $O(k^2n)$ time. Thus the whole algorithm runs in $O(k^2n)$ time.

5. Construct a generalized suffix tree T on the given input strings in $O(M)$ time. Do an in-order traversal of T to label each node as follows. Any node N will get the label i (for some i, $1 \leq i \leq k$) if all the leaves in the subtree rooted at N correspond to suffixes from S_i. Any node N will get the label 0 if the leaves in the subtree rooted at N correspond to suffixes from at least two input strings. This labeling can be done in $O(M)$ time as well. For instance, consider a node N. If all the children of N have the same nonzero label i, then N gets the label i; else it gets the label 0.

   Do one more traversal of the tree T to look for a node N whose string depth is $\geq \ell$ and whose label is i for some $1 \leq i \leq k$. If there is such a node and if its string depth is ℓ, then output the path label of this node. If this node N has a string depth of $> \ell$: Let N' be the parent of N and let x be the path label of N'. Let y be the label of the edge from N' to N. Note that the string x concatenated with any (nonempty) prefix of y occurs only in S_i. If any of these strings is of length ℓ, then output that string.

   If there is no node in T whose string depth is $\geq \ell$ and whose label is i (for some $1 \leq i \leq k$), or if no unique substring of length ℓ can be found in the above traversal, then output "NO".

6. We sort the characters using the integer sort algorithm. Since the characters are integers in the range $[1, m^{10}]$, this sorting can be done in $O(m)$ time. Let the rank of t_i be r_i, for

$1 \leq i \leq m$. If $SA[1:m]$ is the suffix array for S, then set $SA[r_i] = i$, for $1 \leq i \leq m$. Clearly, the entire algorithm runs in $O(m)$ time.