CSE 4502/5717 Big Data Analytics. Fall 2020

Exam II Solutions

1. To begin with we bring BD elements of X and BD elements of Y from the disks into the core memory. We then start merging X and Y in the core memory. Whenever we find elements common between X and Y, we write them in an output buffer (of size BD). When the output buffer is full, we write the elements to the disks (one block per disk) and clear the buffer. When we run out of elements from X, we bring the next BD elements of X from the disks. When we run out of elements from Y, we do a parallel I/O to bring in the next BD elements from the disks.

   When we end up processing all the elements of X or Y or both (in the above manner), we stop.

   Clearly, we bring in all the elements of X and Y exactly once to the core memory, corresponding to a total of $2\frac{n}{DB}$ parallel read I/O operations. Also, we have to output (i.e., write) at most n elements in the disks. Thus it follows that the total number of parallel I/O operations is $O\left(\frac{n}{BD}\right)$.

2. Let S_1 and S_2 be any two sorted sequences with $|S_1| + |S_2| = N$. We'll first show that we can merge S_1 and S_2 in $O\left(\frac{N}{BD}\right)$ parallel I/O operations.

   To begin with we bring BD elements of S_1 and BD elements of S_2 from the disks into the core memory. We then start merging S_1 and S_2 in the core memory. Specifically, we compare the smallest element of S_1 and the smallest element of S_2, output the smaller of the two to an output buffer, and delete this element from its sequence. We continue this until we consume all the elements of either S_1 or S_2. At this point we output the remaining elements of the other sequence as such.

   The size of the output buffer is BD. When the output buffer is full, we write the elements to the disks (one block per disk) and clear the buffer. When we run out of elements from S_1, we bring the next BD elements of S_1 from the disks. When we run out of elements from S_2, we do a parallel I/O to bring in the next BD elements from the disks.

   Clearly, we bring in all the elements of S_1 and S_2 exactly once to the core memory, corresponding to a total of $\left\lceil \frac{N}{DB} \right\rceil$ parallel read I/O operations. Also, we have to output (i.e., write) at most N elements in the disks. Thus it follows that the total number of parallel I/O operations is $O\left(\frac{N}{BD}\right)$.

   Now consider the sequences $R_1, R_2, \ldots, R_\ell$. Consider a full binary tree with ℓ leaves. Leaf i has R_i, $1 \leq i \leq \ell$. Proceed from the leaves toward the root. At each node, merge the two sequences coming from the two children using the above algorithm. When the root completes its merging we get the merged sequence of interest.

   There are $\log \ell$ levels in the tree and in each level we spend $O\left(\frac{\ell n}{BD}\right)$ parallel I/O operations. Thus the total number of parallel I/O operations taken by the above algorithm is $O\left(\frac{\ell n}{BD} \log \ell\right)$.

3. Construct a suffix tree Q for S in $O(n)$ time. Followed by this, perform a post-order traversal of Q to label every internal node u of Q with an integer $c[u]$ such that $c[u]$ is the number of leaves in the subtree rooted at u.

   Now, perform one more traversal through Q to mark every node u such that $c[u] = 3$. Perform one more traversal of Q to identify the marked node whose string depth is the largest and output its path label.

   Clearly, the total run time of the algorithm is $O(n)$.

4. Construct a generalized suffix tree T on the given input strings in $O(M)$ time. Do a post-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.

   Keep an array $L[1 : k]$ all initialized to ∞. Do one more traversal of the tree T. Let N be a visited node whose label is $i \neq 0$. Let p be the parent of N. Note that the path label of p followed by the first character in the edge from p to N occurs only in S_i. If the string depth of p plus 1 is smaller than $L[i]$, then change $L[i]$ to the string depth of p plus 1.

   After the above traversal, $L[i]$ is the length of the signature for S_i, for $1 \leq i \leq k$. We can also keep track of the corresponding path labels to report the signatures.

   The total run time of the above algorithm is $O(M)$.

5. Let P be the pattern whose length is n. Note that there are only $(\sigma - 1)n$ strings P' of length n such that the Hamming distance between P and P' is 1. Each such P' is called a 1-neighbor of P. For example, let $\Sigma = \{a, b, c\}$ and $P = bcab$, then the only strings P' of length 4 such that the Hamming distance between P and P' is 1 will be $acab, ccab, baab, bbab, bcbb, bccb, bcaa$, and $bcac$.

   We first generate all the 1-neighbors of P. This can be done in $O(\sigma n)$ time. For every 1-neighbor P' of P we use the suffix array for T to identify all the occurrences of P' in T. We can also find all the occurrences of P in T. For each P' we spend $O(n + \log m)$ time. For P also we spend $O(n + \log m)$ time.

   Thus the total run time of the algorithm is $O(\sigma n(n + \log m))$.