Solution to Problem 1 of HW 2.

Finding the L1 and L2 edges of the graph used in the UD problem, using a suffix array instead of a suffix tree. The basic approach is the same as when using a suffix tree, so first review that material.

To implement all of this in a suffix array instead of a suffix tree, concatenate all the code words together with a lex-lowest separator between each one. Build a suffix array for this list. Note that in the suffix array a suffix (i,j) is a prefix of a suffix (i',j') to its right in the suffix array, if and only if all of the lcp values between the positions for (i,j) and (i',j') in LCP are at least as large as the length of the (i,j) suffix. Hence, we can build the list L1 for (i,j) by just scanning to the right from (i,j), as long as the LCP values are that high; when we run into an entry where $j' = 1$ we recognize an L1 rule. However,that would not give us the desired O(nl) time bound.

Instead, we first scan the suffix array to remove duplicate suffixes. Note that duplicate suffices are found together in POS where the consecutive LCP values are equal to the suffix lengths. We leave only one copy of each duplicate, but keep a list of the removed suffixes, and a pointer from each removed suffix to the duplicate left in the suffix array. Why do we want that?

Then from a code word C of length $|C|$, scan left for $|C|$ places to find all the suffices in the array such that the min lcp value in the scan is greater or equal to the length of the suffix being examined. We only need to scan $|C|$ places because C can have at most that many preface, and we have already removed all duplicates (i.e., identical suffices from different code words. So this scan has time $O(nl)$ where l is the length of the longest code word, and there are n code words.

L2 is built similarly.

Solution to Problem 2 of HW 2, parsing a message created using a UD code.

Solution: Build a suffix tree for M in $O(m)$ time. Then use it to find all the locations in M of each of the n code words. The time for this is bounded by m plus the number of starting locations in M of each of the code words. That latter number is of course bounded by $n|M|$ and so the time to find all those locations is $O(m + n|M|)$. Now we build a graph as follows: There is one node for every position in M, and one node for position $|M| + 1$, and an edge from i to $i + k + 1$ if and only if there is a code word of length k that appears in M starting at position i. The graph can be built at the same time that the suffix tree is being used to find all starting locations of codewords

in M, and the time to build the graph is the same as the time to find all the occurrences. Now since M is UD, the graph should contain exactly one path from node 1 to node $|M|+1$, and that path specifies the unique parse of M.

Another approach that works is to do a branching search, building a search tree, trying to match a growing prefix of M to a concatenation of codewords. So each node in the search tree corresponds to some concatenation of codewords. When the search fails at some node, because the next character in M does not match the codeword being examined, iteratively back up to the preceding node, until a new match from that branch is possible. This branching search is clearly correct, and will find a parse of M (and assuming C is UD, it will be the only parse). But why doesn't this approach take exponential time? A naive analysis would give $O(n^{\frac{n|M|}{m}})$. In fact, the time bound is within the desired time bound for the following reason. Consider a position i in M such that the prefix of M ending at position i can be written as the concatenation of codewords. So that prefix corresponds to a path in the search tree ending at a node in the search tree. We associate that node with the index i. No other node in the search tree can also be associated with index i, for then there would be two different parses of some message (that prefix of M), contradicting the assumption that C is UD. Hence the search tree can have only $|M|$ branching nodes, and each node has branching factor at most n, so the tree has size $O(n|M|)$, and the desired bound holds.

Another approach that does not work was given by several people. A number of people gave an answer where they implicitly or explicitly assumed that because the code is UD, the parse of a message M could be found myopically, left to right. That is, if some concatenation of code words matched a prefix of M, than those codewords must be part of the correct, unique, parse of all of M. That leads to a simple algorithm using suffix trees (which in fact would run in $O(m+M)$ time), but the assumption is incorrect. For example $C = \{a, ab, ba\}$ and $M = abba$. The unique parse of M is ab,ba, but the codeword $\{a\}$ matches a prefix of M.

Solution to Problem 4 of HW 2

Below we refer to the original linear-time solution (that we studied in class) as Lukas's solution.

4a) Build a suffix tree T for all the K strings in the input set, and do a lexicographic depth-first search to order the leaves of the suffix tree. This also builds suffix array for all positions in all the K strings. Now scan the full suffix array left to right to pull out K suffix arrays, one for each of the K

strings in the input set. Now for each of these K suffix arrays, use the linear-time LCP algorithm to find the LCP array for each string in the input set. Note that for any input string, the LCA depths that Lukas's solution needs are precisely the LCP values now in the LCP array for that string. However, the LCA algorithm actually identifies the node in the suffix tree that is the least common ancestor of the neighboring elements in a suffix array, and the LCP array only gives the depth of that node. Moreover, Lukas's solution needs to know that LCA node in order to put the U values there. So, we still have the detail of how to find those LCA nodes in linear time without using a general LCA algorithm. Here is one solution.

Each leaf represents a suffix in a specific string, and there is an LCP value for each leaf which does not correspond to the lexicographically least suffix (leftmost leaf) in its associated string. We record the LCP value at that leaf. Suppose the re corded LCP value at leaf i is q, meaning that the LCA of i and pred(i) is at depth q on the path from the root of the suffix tree to leaf i. The dept h here means the number of characters on the path. The algorithm will keep a vector V of size n, initialized to all zero entries. The algorithm will do a depth-first-traversal of the suffix tree, and whenever a leaf is encountered with LCP value of q, entry V(q) will get incremented by one. Then whenever the depth-first-traversal backs up from an internal node v of depth q, it sets U(v) = V(q), and then sets V(q) to zero. This clearly take O(n) time over the entire depth-first-traversal.

The rest of the algorithm is the same as in Lukas's solution.

4b) Given the solution to 4a) this part is trivial since we only used the suffix tree to obtain the suffix array in linear time, and that can be done without need for a suffix tree.

Solution to Problem 5 of HW 2:

Use a suffix array and the LCP array to find the longest common substring of two strings in linear time.

Answer: Again, it is assumed that the time bound should be as good as for suffix trees, so in linear time. For the two strings A and B, build a suffix array SA and LCP array for A\$B. Then look for adjacent entries in SA which represent suffixes from different strings (A followed by B, or B followed by A), and over all of such pairs, find the one with the largest LCP value. The point is that if the length of the longest common substring in A and B is k, then there should be two adjacent entries in SA with LCP value of k, where one comes from A and the other comes from B. Note that the largest

LCP value might be larger than k, but that corresponds to a longer common substring in a single string. Note also, that it is not necessarilly true that if suffix i in A and suffix j in B have common length k, which is the length of the longest common substring, then i and j will be adjacent in SA. But, there will be such an adjacent pair.

Solution to Problem 6 of HW 2: Give a short sketch of how to construct a suffix tree for a string S from a suffix array for S and the LCP array for S. The algorithm should run in O(n) time.

Answer: Several people said that each suffix could be added into the tree in constant time. I don't see that, but I could be missing something. The analysis below gives $O(n)$ time amortized over the entire algorithm, but not constant time for each suffix.

We will process the suffix array SA in order, from $i = 1$ to n, adding in the suffix $SA[i]$ to the suffix tree. Consider adding in $SA[i+1]$ after $SA[i]$ has been added in. We know exactly how many characters these two suffixes agree on. That number, say h, is given in the LCP array (either LCP[i] or LCP[i+1] depending on ones convention - my notes and the other notes were not consistent - but no matter for the conceptual algorithm - this isn't a programming course). So we can walk up the growing tree from the leaf from $SA[i]$ until the string depth from the root is exactly h. At that point we spawn off a new edge and leaf node labeled $SA[i+1]$. If the h point was inside an edge, we also break that edge by inserting a new node and writing the correct edge labels on the two incident edges (removing the old edge label). We walk by moving from node to node, knowning how long $SA[i]$ is, and how many characters are on each edge. The algorithm is clearly correct, but it does not look like it runs in linear time because for any j the walk up the tree may visit many nodes. So we need some argument to bound the time for the walks.

Consider again the walk from $SA[i]$ to the point where it branches to $SA[i+1]$, at a node u (which might be created during the insertion of $SA[i+1]$. That walk might have traversed many nodes. The key point is that none of those nodes will ever be visited again, while inserting any $SA[j]$ for $j > i+1$. To see this, note that since $SA[i]$ is lexicographically less than $SA[i+i]$ and the two strings agree down to node u, the first character on the edge from u on the path to $SA[i]$ must be lexicographically less than the first character on the edge from u to $SA[i+1]$. So in the final suffix tree, every leaf reached from u by taking the edge on the path to $SA[i]$ must represent a suffix that

is lexicographically less than $SA[i+1]$. But after $SA[i+1]$ is added to the tree, every walk up the tree is from a leaf representing a suffix that is lexicographicall greater than $SA[i+1]$, so no walk up will intersect the path from u to leaf $SA[i]$. Then since, no internal node is ever passes twice in the whole algorithm, the time to build the suffix tree is $O(n)$.