CS 224 Fall 2011 HW 2, Due Thursday Oct. 13, 2011

The Unique Decipherability Problem

Basic Definitions: A *code* C is just a finite set of finite length strings, called *codewords*. For example, $C = \{aba, a, abb, ab, ba\}$ is a code, and aba is one of the codewords in C. A *message* M is a string created by concatenating strings from C, with repetitions of codewords allowed. So there is no bound on the length of a message. For example, in the code C given above, the string $abaabbababbaa$ is a message that can be created using the codewords in C.

A message M is called *uniquely decipherable (UD)* if there is only one way to write M as the concatenation of codewords in C. For example, if $C = \{aba, a, abb, ab, ba\}$ then the message $abbab$ is UD (it can only be written as abb concatenated with ab. However, the message aba is not UD, since it can be written as ab concatenated with a, or as a concatenated with ba.

A code C is called UD if and only if *every* message that can be created from C is UD. So, the code C in the example is not UD, but if we remove the strings ba and ab from C, I claim that the resulting code would be UD. Note that the definition of a code C being UD is over an infinite set of messages, so we cannot verify that C is UD by enumerating and examining all possible messages that could be created using C. Instead a convincing argument must be made that shows that a code is UD.

Warmup Problem (do it but don't hand it in): Give an ad hoc proof (i.e., a convincing argument) that the code $C = \{aba, a, abb\}$ is UD. Just make a logical argument, without trying to use any of the deeper material presented below.

1 The Unique Decipherability Problem - formal definition and efficient algorithms

Basic Definitions: A *code* C is just a finite set of finite length strings, called *codewords*. A *message* M is a string created by concatenating codewords from C, with repetitions allowed. So there is no bound on the length of a message. A message M is called *uniquely decipherable (UD)* if there is only one way to write M as the concatenation of strings in C. Writing M as

the concatenation of codewords in C is called a *parse* of M. For example, if $C = \{aba, a, abb, ab, ba\}$ then the message $abbab$ is UD, while the message aba is not UD. A code C is called UD if and only if *every* message that can be created from C is UD. So, the code C in the example is not UD, but if we remove the strings ba and ab from C, the resulting code would be UD. Note that the definition of a code being UD is over an infinite set of messages.

The **UD problem**: Given a code C, determine if C is UD.

At first, it may seem that the UD problem would be undecidable (it seems similar to the famous undecidable Post Correspondence Problem). But, in fact it has a polynomial-time solution, and it is even conjectured to have a linear-time solution. There are several algorithms for the UD problem (and variants of it), but they are all similar and can be thought of as a path problem on a graph.

2 A graph-based solution

Given C, we build a graph G with one node for each string that occurs as a suffix of any codeword in C, including a node for each complete codeword (the trivial suffix). Note that the same string might occur as a suffix of more than one codeword, but it is only represented by a single node in G. We label each node by the suffix (string) it represents. In graph G, there is a directed edge from node u to node v if there is a codeword c such that $c = uv$ (this is an L1, or Type 1, edge), or if $u = cv$ (this is an L2, or Type 2, edge). Figure 1 shows graph G for the code $C = \{ac, bc, bca, cb, bacb\}$.

Theorem: C is not UD if and only if there is a directed path in G with at least one edge, starting at a node representing a codeword, and ending at a node representing a codeword. The start and end nodes can be the same, but don't have to be.

CS 224 NOTE: You can read this proof if you want, but it is not needed for the HW problems that follow.

Proof: First we will prove that if the code is not UD, then there is such a directed path in G. Assuming C is not UD, there must be a message M that can be parsed in two different ways. Over all such M, choose one that is shortest. Let c_1 and c_1' denote the first codewords in the two parses of M.

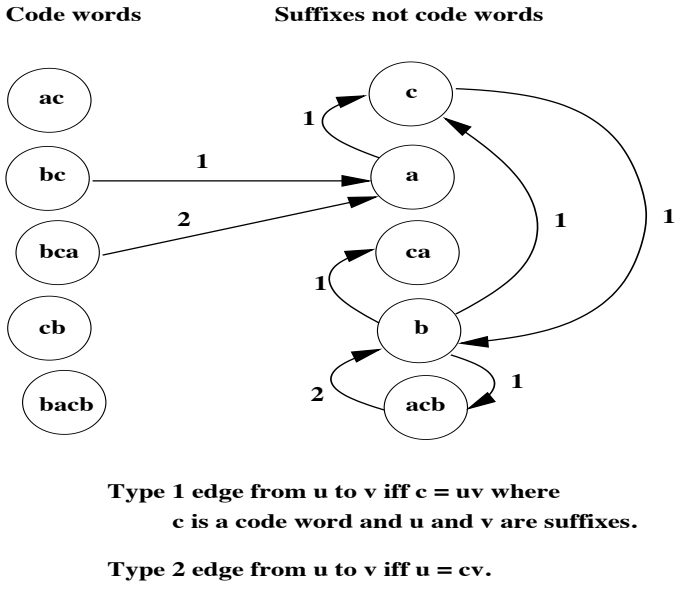

Figure 1: A code and its graph G.

Note that those two codewords cannot have the same length, or else they would be identical, and if we removed that codeword from the head of M, we would have shorter message that is also not UD, a contradiction.

We will show the existence of the claimed directed path P in G by building it constructively while examining the two different parses of M, going left to right in M, adding one additional codeword at a time. We examine the two parses starting with the empty parse (no codewords added yet) and then add in the longer of the two codewords, say c'_1. At that point, one parse has length $|c'_1| > 0$ and the other has length 0. Naturally, one parse is called the "longer" parse, and the other is called the "shorter" parse. The substring consisting of the part of M contained in the longer parse, but not contained in the shorter parse, is called the "overhang". So, at this point, the overhang consists of c'_1, which is (trivially) the suffix of a codeword.

In general, we extend the examination of the two parses by adding in the next codeword, call it cc, of the (currently) *shorter* parse (see Figure 2). There are three cases to consider: either the length of cc is strictly greater than the length of the current overhang; or the length of cc is strictly less than the length of the current overhang; or cc and the current overhang have

the same length. See Figure 2. In the first case, the addition of cc extends the currently shorter parse to the right of the end of the currently longer parse (in which case the extended parse becomes the longer parse); in the second case, it only extends the shorter parse to a point strictly to the left of the currently longer parse (in which case the extended parse remains the shorter parse); or it extends the parse exactly to the end of the longer parse. In the third case, the two parses each spell out the same complete string with two different parses, and so that string must be M. Otherwise, M was not a shortest string with two different parses. Hence, this third case can only happen once in the examination of the two parses. In either of the first two cases, the overhang changes. The overhangs before and after the addition of cc will be called the "old" and "new" overhangs. It is easy to see (established more formally by induction, below) that each overhang generated in this way is a suffix of a codeword, possibly an entire codeword. Finally, adding in cc extends the shorter parse to be the same length of the longer parse, at which point the two different parses of M are complete.

Now we want to describe how the desired path P is built up as the two parses of M are being extended.

Codeword c'_1 is the first codeword added into the parses, so the first overhang is just c'_1 itself. Note, trivially, that c'_1 is a suffix of a codeword. Next, codeword c_1 is added into the parses, since it is the first codeword of the shorter parse (which has length zero before c_1 is added in). Now $|c'_1| > |c_1|$, so c_1 is a prefix of c'_1; let v denote the *suffix* of c'_1 starting at position $|c_1| + 1$ of c'_1. Therefore $c'_1 = c_1 v$. But c_1 is a suffix of a codeword (namely, itself), and c'_1 is a codeword, so c'_1 and c_1 demonstrate the existence of a Type 2 edge in G from the node representing c'_1 to the node representing suffix v. So path P begins with an edge from the node for codeword c'_1 to the node for suffix v, which is the new overhang at this point. Note that the new overhang, v, is the suffix of a codeword. (c'_1 and c_1 also demonstrate the existence of a type 1 edge going to v. Which one?)

At the general step, codeword cc is added into the shorter parse, changing the overhang. We use u to denote the current overhang before cc is added. We assert, inductively, that u is the suffix of a codeword, and that there is a directed path in G from the node for c'_1 to the node for u. We will see that the new overhang, call it v, is also a suffix of a codeword, and that it demonstrates the existence in G of an edge from the node for suffix u to the

node for suffix v. There are three cases, depending on the lengths of cc and u.

Case 1: If $|cc| > |u|$, then $cc = uv$, establishing the existence of the directed, Type 1, edge (u, v) in G. Note that again the new overhang, v, is a suffix of a codeword, e.g. cc. This also establishes the existence of a directed path from the node for c'_1 to the node for v.

Case 2: If $|cc| < |u|$, then $u = ccv$, establishing the existence of the directed, Type 2, edge (u, v) in G. It also establishes that the new overhang, v, is a suffix of a codeword, because inductively u is a suffix of a codeword. This also establishes the existence of a directed path from the node for c'_1 to the node for v.

Case 3: If $|cc| = |u|$, then the two parses of M are complete, and this case can happen only once. So up to this point, all of the other additions are instances of Case 1 or Case 2, and since u is the current overhang (before cc is added), inductively, there is a directed path from the node for c'_1 to the node for u. But $|cc| = |u|$ and both are suffices of M, so $cc = u$, and hence u is a codeword. Therefore the path from c'_1 to u is the path P, claimed to exist in the statement of the theorem.

This proves one direction of the theorem. The proof of the other direction is very similar. We assume the existence of a directed path P, of length at least one, that goes from a node for a codeword to a node for a codeword, and use P to form a message M and two different parses of M.

Let P be the shortest path (of non-zero length) in G from a node representing a codeword to a node representing a codeword. By following the edges of P we will build up a message M and two different parses of M, showing that C is not UD. The first edge e_1 of P goes from a node, say c_1, representing the codeword c_1 to a node v representing a suffix of some codeword. Abusing notation somewhat, we will use the name of the node to refer to the string it represents. So we will talk about suffix v and node c_1, in addition to node v and codeword c_1.

If edge e_1 is a Type 1 edge, then it must be that there is a codeword c'_1 where $c'_1 = c_1 v$. If edge e_1 is a Type 2 edge, then it must be that there is a codeword c'_1 where $c_1 = c'_1 v$. In either case, the two parses begin with the codewords c_1 and c'_1 respectively, and v is the overhang of the two parses. It cannot be the $c_1 = c'_1$, for then P would not be the shortest path from a codeword to a codeword. Hence one of the two parses is the shorter parse

and the other is the longer parse.

The next edge e_2 in P goes from v to some node w. If e_2 is a Type 1 edge, then $c = vw$ for some codeword c, and if e_2 is a Type 2 edge, then $v = cw$ for some codeword c. In either case, add codeword c to the shorter parse, creating overhand w. This argument is general for each successive edge in P, meaning that each edge identifies a codeword that is added to the (currently) shorter parse.

Finally, consider the last edge $e = (u, v)$ in P. Suffix u is the overhang in the two parses being constructed, before edge e is traversed. Since e is the last edge in P, v must be a codeword, say c'. If e is Type 1, then $c = uv = uc'$ for some codeword c. In this case, add codeword c to the shorter parse and codeword c' to the longer parse.

If e is Type 2, then $u = cv = cc'$ for some codeword c. In this case, add the string cc' to the shorter parse.

In both cases, the end result is two different parses of the same length, where each parse is a concatenation of codewords, and the two parses spell out the same string, denoted M. Hence M is not UD, and C is not UD.

3 How to build the L1 and L2 edges for the UD graph in $O(nm)$ time

CS 224 NOTE: You need to read this part for the homework.

Add the symbol \$ to the end of each codeword and build a generalized suffix tree T of all the codewords. Let m represent the total length of all the codewords, so T has $O(m) = O(nl)$ nodes and edges, where n is the number of codewords, and l is the length of the longest codeword. Build a two-D array D indexed by codeword ID and suffix position. Entry $D(i, j)$ points to the leaf in the suffix tree that represents the j'th suffix of codeword i. Also initiate two lists L_1 and L_2 at each leaf.

Do a dfs of T. When a node v is encountered with an edge e branching from v to a leaf labeled (i, j), and edge e only contains the label \$, the path to v spells out a complete suffix j of some codeword i. In that case, stack the pair (i, j). Note that a leaf might be labeled with more than one (i, j) pair, but we need only stack one of these. Essentially, we only need one pointer to the leaf - so alternately, we could put a pointer to the leaf rather than to the

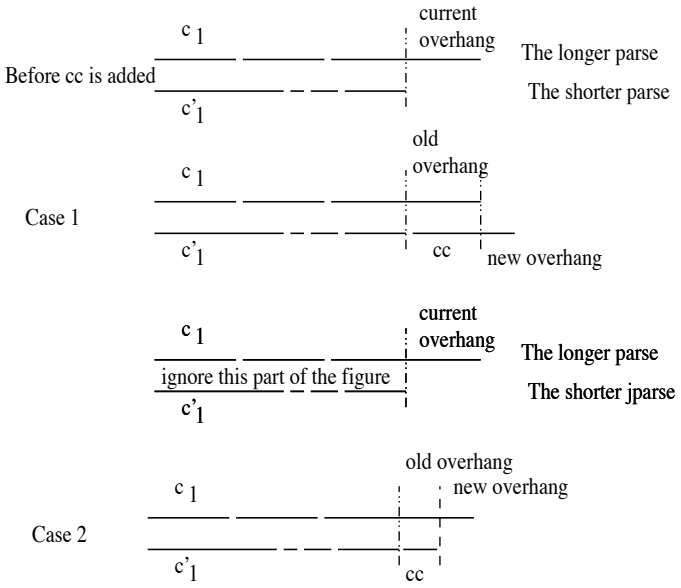

Figure 2: Examining the two parses of M, and adding in the new codeword cc to the shorter parse. The first two cases. Each horizontal line segment represents a codeword. There is some garbage in this figure that I couldn't remove, and I was too lazy to redo the figure.

(i, j) pair. When a leaf is reached in the dfs labeled by a complete codeword $(k, 1)$ for some codeword k, scan through the stack. For each (i, j) on the stack, add k to the list L_1 at the leaf labeled by (i, j). What this indicates is that the *string* represented by the leaf labeled (i, j) is a prefix of codeword k. Every suffix (considered as a string) in T that is a prefix of some codeword will be found in this way, so that when the dfs is done, each leaf labeled by (i, j) (and possibly other labels) has an explicit list L_1 showing the set of codewords that suffix (i, j) is a prefix of. Note that a codeword k of length d has only d prefixes and hence can be added to at most d of these L_1 lists. So all the L_1 lists can be accumulated in $O(m) = O(nl)$ time, and can have at most $O(m)$ total entries.

Now do a second dfs of T. When a node v is encountered with an edge e branching from v to a leaf labeled $(k, 1)$, and edge e only contains the label \$, the path to v spells out the complete codeword k. In that case, stack the pair $(k, 1)$. When a leaf is reached in the dfs labeled by (i, j) (and possibly other pairs), scan the stack. For each $(k, 1)$ on the stack, add k to the list L_2 at the leaf labeled (i, j). What this indicates is that the codeword k is a prefix of the suffix specified by the path to the leaf labeled (i, j). Every such relation between codeword and suffix is found in this way, so that when the dfs is done, each leaf has an explicit list L_2 showing the set of codewords that are prefixes of string represented at the leaf labeled (i, j). The size of the stack can be at most the minimum of l and n, and there are only m leaves, so the time for building all the L_2 lists is $O(ml) = O(nl^2)$, and also $O(nm) = O(n^2 l)$, so the time is $O(mt)$ where t is the minimum of n and l. This will typically be l. Note also that the size of any L_2 list is at most the minimum of n and l.

All this was preparatory to building the graph discussed above, or simulating the search in such a graph. The starting suffices needed by the UD algorithm are specified by the L_1 lists of any leaf $(i, 1)$. Suppose k is in the L_1 list of $(i, 1)$. That means that codeword i is a prefix of codeword k, and the suffix created from these two codewords is the suffix of codeword k starting at position w, where w is one plus the length of codeword i. That suffix is represented at the leaf (k, w) of T. We mark that leaf as being a suffix that has been found, and put (k, w) in a queue of suffices to be processed.

When we process a suffix (k, w) we simply scan through its L_1 and its L_2 lists. If i is in its L_2 list, then codeword i is a prefix of suffix (k, w), and the next suffix generated is identified by (k, z), where z is the length of

suffix (k, w) minus the length of codeword i. If the leaf labeled by (k, z) is already marked, then we do nothing. Otherwise, we mark leaf labeled (k, z) and place (k, z) in the queue of suffices to be processed. Note that over the entire algorithm, the time needed to do the scans of L_2 lists is bounded by the total size of the L_2 lists, which is $O(mt)$.

If i is in the L_1 list of (k, w), then (k, w) identifies a prefix of i, and the new suffix generated is (i, y), where y is one plus the length of suffix (k, w). Over the entire algorithm, the time needed to do the scans over L_1 is $O(m)$, since that is the total length of the L_1 lists.

HOMEWORK PROBLEM 1: Show how to replace the use of a suffix tree with the use of a suffix array in the above discussion. Do you need the LCP array for this?

HOMEWORK PROBLEM 2: Suppose that the code C is UD and contains n codewords of total length m. Let M be a message created from C, i.e., M is the concatenation of code words (with possible repetition) from C. Since C is UD, there is only one way to create M from codewords in C. Give an algorithm that can take in M and C and find the unique parse of M into codewords in C, in $O(m + n|M|)$ time. Hint: Suffix trees or arrays. If you can, explain how to do it with a suffix array.

The next problems do not relate to unique decipherability.

HOMEWORK PROBLEM 3.a. The KS suffix array algorithm we saw in class divides the suffixes into those that start at positions 1,2 mod 3 and those that start at positions 0 mod 3. Can you find another way to split the suffixes that also achieves a linear time to construct the suffix array? Briefly explain how the algorithm would work in that case, emphasizing any parts of the KS algorithm that change significantly. Do the complete time analysis.

HOMEWORK PROBLEM 3b. Given you answer to part a, is the original division in the KS algorithm superior to yours and perhaps all others? In what way superior?

HOMEWORK PROBLEM 4.

This question relates to material we will discuss on thursday Oct. 6. The linear-time algorithm we discussed in class to build the table of k v. l(k) for a set of strings, builds a suffix tree for the strings, and also uses a general linear-time preprocesing /constant-time query, least common ancestor algorithm. The suffix tree takes more space than a suffix array, and the general LCA algorithms are quite complex. That solution was obtained before it was

known how to construct suffix arrays and compute depths (LCP) in linear time. Given that, we have two opportunities (or problems).

4a) Still using a suffix tree for the set of strings, find a method to avoid using a general LCA algorithm, and still get the overall linear-time solution to the common substrings problem. I believe I see a way to do that using suffix arrays, but I haven't written it down, so I might be wrong.

4b) Replace the suffix tree in the solution to problem 4a) with a suffix array. I have not yet tried this, so I don't know if it is possible or easy or hard.

HOMEWORK PROBLEM 5.

The input to the *longest common substring problem* is a pair of strings S and S'. The problem is to find the longest substring (consecutive run of characters) that is in both S and S'. This has an easy linear-time solution using a suffix tree.

Can you see how to solve the longest common substring problem using a suffix array and the LCP array in linear time. I think it is doable, but don't know if this is easy or hard.

HOMEWORK PROBLEM 6.

Give a quick sketch (a few sentences should suffice) to explain how to build a suffix tree for a string S from the suffix array and the LCP array for S, in linear time.