An Efficient Algorithm for the All Pairs Suffix-Prefix Problem

Dan Gusfield * *Gad M. Landau* † *Baruch Schieber* ‡

Abstract

For a pair of strings (S_1, S_2), define the suffix-prefix match of (S_1, S_2) to be the longest suffix of string S_1 that matches a prefix of string S_2. The following problem is considered in this paper. Given a collection of strings $S_1, S_2, \ldots, S_k$ of total length m, find the suffix-prefix match for each of the $k(k-1)$ ordered pairs of strings. We present an algorithm that solves the problem in $O(m+k^2)$ time, for any fixed alphabet. Since the size of the input is $O(m)$ and the size of the output is $O(k^2)$ this solution is optimal.

1. Introduction

For a pair of strings (S_1, S_2), define the suffix-prefix match of (S_1, S_2) to be the longest suffix of string S_1 that (exactly) matches a prefix of string S_2. We consider the **All Pairs Suffix-Prefix Problem** defined as follows. Given a collection of strings $S_1, S_2, \ldots, S_k$ of total length m, find the suffix-prefix match for each of the $k(k-1)$ ordered pairs of strings.

*Computer Science Division, University of California at Davis, Davis, CA 95616, Tel. (916) 752-7131, Email: gusfield@cs.ucdavis.edu. Partially supported by Dept. of Energy grant DE-FG03-90ER60999, and NSF grant CCR-8803704.

†Dept. of Computer Science, Polytechnic University, 333 Jay Street, Brooklyn, NY 11201, Tel. (718) 260-3154, Email: landau@pucs2.poly.edu. Partially supported by NSF grant CCR-8908286 and the New York State Science and Technology Foundation, Center for Advanced Technology in Telecommunications, Polytechnic University, Brooklyn, NY.

‡IBM - Research Division, T.J. Watson Research Center, P.O. Box 218, Yorktown, NY 10598, Tel. (914) 945-1169, Email: sbar@watson.ibm.com.

R. Capocelli et al. (eds.), *Sequences II*

Using a variant of the well-known KMP string matching algorithm [KMP77] one can find the suffix-prefix match of a single pair (S_i, S_j) in $O(m_i + m_j)$ time, where m_i and m_j are the respective lengths of the strings. So overall, this approach leads to an $O(km)$ time solution. We present an algorithm that solves the problem in $O(m + k^2)$ time, for any fixed alphabet. Since the size of the input is $O(m)$ and the size of the output is $O(k^2)$ this solution is optimal.

The motivation for the all pairs suffix-prefix problem comes from two related sources, sequencing and mapping DNA [Les88]. Sequencing means getting a complete linear listing of the nucleotides in the string, and mapping means finding the linear order or exact positions of certain significant features in the string. Sequencing is essentially the extreme special case of mapping. In either case, long strings of DNA must first be cut into smaller strings since only relatively small strings can be sequenced as a single piece. Known ways of cutting up longer strings of DNA result in the pieces being randomly permuted. Hence, after obtaining and sequencing or mapping each of the smaller pieces one has the problem of determining the correct order of the small pieces. The most common approach to solving this problem is to first make many copies of the DNA, and then cut up each copy with a different cutter so that pieces obtained from one cutter overlap pieces obtained from another. Myriad choices of how to cut are available; some are chemical, some or enzymatic and some are physical (radiation or vibration). Then, each piece is sequenced or mapped, and in the case of mapping a string is created to indicate the order and type of the mapped features, where an alphabet has been created to represent the features.

Given the entire set of sequenced or mapped pieces, the problem of assembling the proper DNA string has been *modeled* as the *shortest common superstring problem:* find the shortest string which contains each of the other strings as a contiguous substring.

There are several approaches to solve this problem. In one approach [TU88, KM89, Tur89, Ukk90, BJLTY91, KM91] the problem is modeled as a maximum length Hamilton tour problem as follows. First, each string that is totally contained in another string is removed (finding these strings in linear time is easy - a simple byproduct of building the suffix tree of the given set of strings, as explained later). Let $\mathcal{S} = \{S_1, \ldots, S_k\}$ be the set of remaining strings. Define a complete weighted graph G_S with k vertices, where each vertex i in G_S corresponds to string S_i. Define the weight of each edge (i, j) to be the length of the suffix-prefix match of (S_i, S_j). Observe that the *maximum* length Hamilton tour in this graph gives the shortest superstring. Any approximations within

factor $c < 1$ of the optimal Hamilton tour results in a superstring with *compression* size (i.e., number of symbols "saved") of at most most c times the optimal compression. Such approximations and other practical methods based on the Hamilton tour approach has given good results [KM89, KM91]. In a slightly different approach [BJLTY91] model the problem as a minimum length Hamilton tour problem. They consider the same graph G_S, but define the weight of each edge (i, j) to be the length of string S_i minus the length of the suffix-prefix match of (S_i, S_j). Observe that the *minimum* length Hamilton tour in this weighted graph gives the shortest superstring. Any approximations within factor $c > 1$ of the optimal Hamilton tour results in a superstring whose length is at most c times the optimal. Clearly, the all pair suffix-prefix problem is the first step in these methods.

Another approach for solving the problem is the greedy approach [TU88, Ukk90, BJLTY91]; substrings are built up by greedily joining the two strings (original or derived) with maximum suffix-prefix match. Although this approach does not require solving the all pairs suffix-prefix problem, the all-pairs result leads to an efficient implementation of this method, and generalizes the kinds of operations used in [TU88, Ukk90, BJLTY91].

The main data structure used in our algorithm is the *suffix tree* of a set of strings. To make the paper self contained we recollect the definition of suffix trees in Section 2, and describe our algorithm in Section 3.

2. Suffix trees

Let $C = c_1 c_2 \ldots c_n$ be a string of n characters, each taken from some fixed size alphabet Σ. Add to C a special symbol \$ (c_{n+1}) that occurs nowhere else in C. The suffix tree associated with C is the trie (digital search tree) with $n+1$ leaves and at most n internal vertices such that: (1) each edge is labeled with a substring of C, (2) no two sibling edges have the same (nonempty) prefix, (3) each leaf is labeled with a distinct position of C and (4) the concatenation of the labels on the path from the root to leaf i describes the suffix of C starting at position i. (See Fig. 1 for an example.)

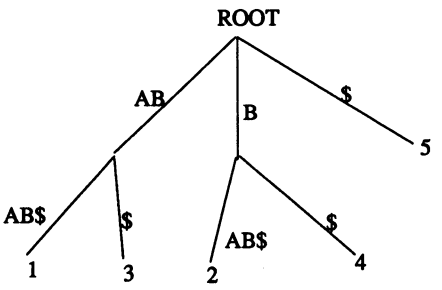

Figure 1. The Suffix Tree of the string ABAB$

Weiner [Wei73] and McCreight [McC76] presented linear time algorithms to compute a suffix tree, Apostolico et al. [AILSV88] compute a suffix tree in parallel.

Given a set of strings $\mathcal{S} = \{S_1, S_2, \ldots, S_k\}$, define the *generalized suffix tree* of $\mathcal{S}$ to be the tree given by "superimposing" the suffix trees of all strings in $\mathcal{S}$, identifying edges with the same labels. In other words, the *generalized suffix tree* of $\mathcal{S}$ is the trie consisting of suffixes of all strings in $\mathcal{S}$. The generalized suffix tree can be computed in the following three steps.

STEP 1. Concatenate the strings in $\mathcal{S}$ into one long string $C = S_1\$_1S_2\$_2\cdots\$_{k-1}S_k$, where $\$_i$ $(1 \leq i \leq k-1)$ is a special symbol that occurs nowhere else in C.

STEP 2. Compute the suffix tree of C.

STEP 3. Extract the generalized suffix tree of $\mathcal{S}$ from the suffix tree of C by changing the labels of some edges in the tree as follows. Let (u, v) be an edge of the suffix tree whose label contains (at least) one of the special symbols $\$_i$. Consider the concatenation of the labels on the path from the root to v. Since the resulting string contains (at least) one special symbol, it occurs only once in C, and hence, vertex v must be a leaf. Change the label of (u, v) to be the prefix of its original label up to (and including) the first occurrence of a special symbol, and change the label of v accordingly. It is not difficult to see, that the part of the suffix tree of C that remains connected to the root is the generalized suffix tree of $\mathcal{S}$.

Implementation Note. Computing the generalized suffix tree by Weiner's algorithm would

take $O(m \log k)$ time. However, since each of the special symbols $\$_i (1 \leq i \leq k-1)$ appears only once in C one may add some minor changes to the algorithm in order to achieve linear time. We leave the details to the interested reader.

3. The algorithm

We start by constructing the generalized suffix tree (ST) for the given set of k strings. Consider the path from the root of ST to each internal vertex v. It defines a string $\texttt{Str}(v)$ whose length is $\texttt{Len}(v)$. (Note that $\texttt{Len}(v)$ gives the number of characters on the path to v, and not the number of edges on this path.) As we construct this tree we build up a list $L(v)$ for each internal vertex v. List $L(v)$ holds the index i if and only if the suffix of length $\texttt{Len}(v)$ of the string S_i is equal to $\texttt{Str}(v)$. That is, the vertex v is the vertex just before a leaf representing a suffix from string S_i. This step can be done in linear time for any fixed alphabet using the algorithms of [Wei73, McC76].

Consider a string S_j, and focus on the path from the root of ST to the leaf $l(j)$ representing the first suffix of string j, i.e., the entire string S_j. The key observation is the following: Let v be a vertex on this path. If $i \in L(v)$ then a suffix of string S_i, of length $\texttt{Len}(v)$, matches a prefix of string S_j of the same length. For each index i, we want to record the deepest vertex v on the path to $l(j)$ (i.e., furthest from the root) such that $i \in L(v)$. Clearly, $\texttt{Str}(v)$ is then the suffix-prefix match of (S_i, S_j). It is easy to see that by one traversal from the root to $l(j)$ we can find the deepest vertices for all $1 \leq i \leq k$ $(i \neq j)$. However, performing this computation for each string S_j separately will result in running time of $O(km)$. In order to achieve linear time we traverse the tree only once.

Traverse the tree ST in Depth First Search (DFS) order ([Tar83]), maintaining k stacks, one for each string. During the traversal when a vertex v is reached in a forward edge traversal, push v onto the ith stack, for each $i \in L(v)$. When a leaf $l(j)$ (representing the entire string S_j) is reached, scan the k stacks and record for each index i the current top of the ith stack. It is not difficult to see that the top of stack i contains the vertex v for which $\texttt{Str}(v)$ is the suffix-prefix match of (S_i, S_j). If the ith stack is empty, then there is no overlap between a suffix of string S_i and a prefix of string S_j. When the DFS backs up past a vertex v, we pop the top of any stack whose index is in $L(v)$.

Complexity: The total number of indices in all the lists $L(v)$ is $O(m)$. The number of edges in ST is also $O(m)$. Each push or pop of a stack is associated with a leaf of ST,

and each leaf is associated with at most one pop and one push, hence traversing ST and updating the stacks takes $O(m)$ time. Recording of each of the $O(k^2)$ answers is done in $O(1)$ time per answer. Therefore, the total running time of the algorithm is $O(m + k^2)$ time.

Extensions. We note two extensions. Let $k' \leq k(k-1)$ be the number of ordered pairs of strings which have a non-zero length suffix-prefix match. By doubly linking the tops of each stack, all non-zero length suffix-prefix matches can be found in $O(m + k')$ time. Note that the order of the stacks in the linked list will vary, since a stack that goes from empty to non-empty must be linked at one of the ends of the list; hence we must also stack the name of the string associated with the stack.

At the other extreme, suppose we want to collect for every pair not just the longest suffix-prefix match, but all suffix-prefix matches. We modify the above solution so that when the tops of the stacks are scanned, we read out the entire contents of each scanned stack. This extension would be an initial step in producing a range of near-optimal superstrings, which is often of more use than just a single shortest superstring. If the output size is k^*, then the complexity for this problem is $O(m + k^*)$.

References

[AILSV88] A. Apostolico, C. Iliopoulos, G.M. Landau, B. Schieber, and U. Vishkin. Parallel construction of a suffix tree with applications. *Algorithmica*, 3:347–365, 1988.

[BJLTY91] A. Blum, T. Jiang, M. Li, J. Tromp, and M. Yanakakis. Linear approximation of shortest superstrings. In *Proc. of the 23rd ACM Symp. on Theory of Computing*, 328-336, 1991.

[KM89] J. Kececioglu and E. Myers. A procedural interface for a fragment assembly tool. Technical Report TR 89-5, University of Arizona, Computer Science Dept., April 1989.

[KM91] J. Kececioglu and E. Myers. A robust and automatic fragment assembly system, 1991. Manuscript.

[KMP77] D.E. Knuth, J.H. Morris, and V.R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6:323–350, 1977.

[Les88] A. Lesk, editor. *Computational Molecular Biology, Sources and Methods for Sequence Analysis.* Oxford University Press, Oxford, UK, 1988.

[McC76] E.M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23:262–272, 1976.

[Tar83] R.E. Tarjan. *Data Structures and Network Algorithms.* CBMS-NSF Regional Conference Series in Applied Math. SIAM, Philadelphia, PA, 1983.

[TU88] J. Tarhio and E. Ukkonen. A greedy approximation algorithm for constructing shortest common superstrings. *Theoretical Computer Science*, 57:131–145, 1988.

[Tur89] J. Turner. Approximation algorithms for the shortest common superstring problem. *Information and Computation*, 83(1):1–20, 1989.

[Ukk90] E. Ukkonen. A linear time algorithm for finding approximate shortest common superstrings. *Algorithmica*, 5:313–323, 1990.

[Wei73] P. Weiner. Linear pattern matching algorithm. In *Proc. 14th IEEE Symp. on Switching and Automata Theory*, pages 1–11, 1973.