OREGON STATE UNIVERSITY COMPUTER SCIENCE DEPARTMENT

Algorithms for Constructing a Consensus Sequence

Paul Cull
Jim Holloway
Department of Computer Science
Oregon State University
Corvallis, OR 97331

91-20-1

Algorithms for Constructing a Consensus Sequence

Paul Cull
pc@cs.orst.edu

Jim Holloway
holloway@cs.orst.edu

Department of Computer Science
Oregon State University
Corvallis, OR 97331

Abstract

Biological and physical limitations require that DNA be sequenced in fragments. There are several approaches to obtain the appropriate sized fragments of DNA to sequence. The method of sequencing that we are interested in is loosely referred to as shotgun sequencing. Many copies of the genomic DNA to be sequenced are cleaved by one or more restriction endonucleases resulting in a multiset, $\mathcal{S}$, of DNA fragments that are not ordered. DNA fragments are essentially selected at random from this multset and sequenced. A consensus sequence is constructed by joining together fragments which overlap. (One hopes that the consensus sequence is very close to the original sequence.) Since errors occur reading the sequences, the overlaps must be approximate, not exact.

This process of reassembly is similar to the NP–complete shortest common superstring problem [GMS80]. To simplify the problem we make the following assumptions.

- An integer k can be supplied that defines the minimum acceptable overlap between two sequences.
- There is a unique alignment of the sequence fragments such that all suffix/prefix overlaps are of length k or greater.
- All suffix/prefix overlaps are exact (log inexact) matches.

We define the string consensus problem and give three algorithms to solve it. We then define the log inexact string consensus problem and give three algorithms to solve it. We believe that the log inexact string consensus problem is closer to the problem of constructing a consensus sequence from shotgun data that biochemists are trying to solve than the problems previous approximation algorithms for the shortest common superstring problem.

1 Introduction

Biological and physical limitations require that DNA be sequenced in fragments. Because of these limitations there are, from a computational perspective, two approaches used to obtain the appropriate sized fragments of DNA to sequence.

One class of methods for sequencing DNA is loosely termed ordered sequencing. These methods use an oligonucleotide primer to initiate the DNA sequencing at a known point and the sequencing reaction proceeds from this point. The leading several hundred bases of the DNA strand are sequenced and then removed using exonucleases exposing the next segment of the DNA to be sequenced. The process of sequencing several hundred bases and removing them is continued until the entire DNA sequence has been determined.

Another class of methods for sequencing DNA is loosely referred to as shotgun sequencing. Many identical copies of the DNA genome to be sequenced are cleaved by one or more restriction endonucleases. A restriction endonuclease will cleave the DNA sequence at each occurrence of a specific six to eight base subsequence (sonication can also be used to create unordered fragments). This results in a multiset of DNA fragments that are not ordered. DNA fragments are essentially selected at random from this multiset and sequenced. A consensus sequence that is believed to represent the original DNA sequence is assembled by finding overlaps between the DNA fragments that have been sequenced.

In this technical report we will present several algorithms that can be used to assemble the sequenced DNA fragments into a consensous sequence. In section 2 we will review the work that has been done on the shortest superstring problem. Section 3 introduces the perfect string consensus problem. Examining this problem will motivate some of the assumptions that we will make later in the paper. The string consensus problem is introduced along with three algorithms to solve it in section 4. Section 5 will define the inexact shotgun sequencing problem, a problem similar to the string consensus problem, but allowing some errors in the prefix/suffix matches. Section 5 will also present three algorithms to solve the inexact shotgun sequencing problem.

2 Previous Work

In this section we will examine the work that has been done on the process of reconstructing a consensus sequence from the individual sequence fragments. Several people have studied this problem from the biologist's perspective [CK82, Sta82, JJD86] and many

people have studied similar problems in computer science [Mai78, GMS80, PSTU83, TU88, Tur89, Ukk90, Li90, BJL+91].

In 1980 Gallant, Maier & Storer [GMS80] showed that the shortest common superstring problem is NP–complete. They first defined superstring,

> a *superstring* of a set of strings $\mathcal{S} = \{S_1, S_2, \ldots, S_n\}$ is a string S containing each S_i, $1 \leq i \leq n$, as a substring,

and then defined the shortest common superstring problem,

> Given a set of strings $\mathcal{S}$ and a positive integer l, does $\mathcal{S}$ have a superstring of length l?

With these definitions they are able to show that the shortest common superstring problem is NP–complete even when the alphabet is restricted to {0,1}. The NP–completeness result suggests that there is no polynomial time algorithm for this problem. Therefore, one should probably attack this problem in one of several ways:

1. Add assumptions about the substrings so that the problem with the extra assumptions is no longer NP–complete.

2. Show that the hard instances of the problem are rare, so that the problem can usually be solved quickly.

3. Instead of finding the superstring with the minimum length, find a superstring with a length that can be shown to be close to the minimum length.

In a paper by Peltola, Söderlund, Tarhio & Ukkonen [PSTU83] a heuristic for a generalized minimal length superstring problem is given. The problem is generalized by allowing errors in the string matching. They define superstring,

> a superstring of a set of strings $\mathcal{S} = \{S_1, S_2, \ldots, S_n\}$ is a string S containing each S_i, $1 \leq i \leq n$, as an **approximate** substring.

An approximate substring S_i of S with an error ratio δ is defined,

> a substring of S that can be transformed into S_i with at most $\delta|S_i|$ delete, insert, replace, and transpose operations.

The minimum number of delete, insert, and replace operations needed to transform one string into another is frequently called the minimum edit distance. They then define the generalized minimal length superstring problem,

> Given a set of strings $\mathcal{S}$, a positive integer k, and an error ratio δ, $0 < \delta < 1$, does $\mathcal{S}$ have a common superstring of length at most k?

No performance guarantees are given for the heuristic.

The heuristic has three basic steps. First a complete pairwise alignment graph is computed using the standard dynamic algorithm of Sellers [Sel74] and others. Each node of the graph represents some $S_i \in \mathcal{S}$. An edge exists between S_i and S_j iff the minimum edit distance is less than $\delta|S_j|$. The value assocated with the edge is the minimum edit distance between S_i and S_j. From this graph, a global alignment is computed by essentially computing the minimal spanning tree of the pairwise alignment graph. The minimal spanning tree provides an ordering of the strings and the optimal alignment of adjacent strings, but does not provide the optimal global alignment, or the optimal local alignment when more than two strings overlap. The final step in the heuristic computes the consensus string from the overlap graph and the minimal spanning tree. The running time of the heuristic is $O(\delta N^2)$ where N is $\sum_{S_i \in \mathcal{S}} |S_i|$, the sum of the lengths of the strings in $\mathcal{S}$. No performance guarantees are given for the heuristic.

Papers by Tarhio & Ukkonen [TU88] and Turner [Tur89] develop approximation algorithms that they conjecture will find an approximate shortest common superstring that is, at worst, twice the length of the actual shortest common superstring. Tarhio & Ukkonen [TU88] analyzed their algorithm in terms of the compression of the strings in $\mathcal{S}$ instead of the length of the shortest common superstring of $\mathcal{S}$. The compression is $N - c$, where c is the length of the approximate shortest common superstring that their algorithm computes. The main result of the paper is that $(N - c) \geq \frac{1}{2}(N - c_{min})$ where c_{min} is the length of the actual shortest common superstring. The running time of the algorithm presented by Tarhio & Ukkonen is $O(nN)$.

The algorithm computes the maximal overlap of all pairs of strings in $\mathcal{S}$. It then behaves in a greedy fashion to construct a short common superstring by selecting the longest overlap between two strings. When a string has been overlapped on both ends

it is removed from $\mathcal{S}$. The process of selecting the longest overlap continues until no strings remain in $\mathcal{S}$.

Turner [Tur89] presents an algorithm to approximate the shortest common superstring with the same performance guarantees as the algorithm of Tarhio & Ukkonen [TU88]. Turner uses suffix arrays to reduce the number of suffix/prefix comparisons that need to be done. The running time of the algorithm is $O(N \log N)$ or $O(N \log n)$, depending on whether direct indexing over the alphabet Σ is allowed.

Later, in a paper by Ukkonen [Ukk90], an $O(N)$ or $O(N \min(\log n, \log |\Sigma|))$ algorithm, depending on whether direct indexing over the alphabet Σ is allowed, to solve the approximate shortest common superstring problem is presented. This reduction in time is achieved by a clever use of the Aho–Corasick [AC75] string matching automaton. Again, this algorithm achieves the same compression ratio as the algorithms of Tarhio & Ukkonen and Turner.

The first approximation algorithm that approximated the shortest common superstring of $\mathcal{S}$ instead of the maximal compression of the strings in $\mathcal{S}$ was given by Li [Li90]. Li was able to give an algorithm to compute an approximate shortest common superstring of length $O(L \log L)$ where L is the length of the shortest common superstring. The algorithm is similar to the greedy algorithms given above by Tarhio & Ukkonen, Turner, and Ukkonen. It differs when the strings with the maximum overlap are joined, not only are the strings that were joined removed from the set of strings, but all substrings of the resulting joined string are removed from the set of strings. This results in the size of the set of strings decreasing fast enough to show the $O(L \log L)$ bound on the length of the approximate shortest common superstring. Although the running time of the algorithm is not considered, it is clearly polynomial in the size of the input.

Blum et. al.[BJL$^+$91] recently showed that the greedy algorithm of Tarhio & Ukkonen [TU88] and Turner [Tur89] does find superstrings that are, at worst, a multiplicative factor of four longer than the minimum length superstring.

3 The Perfect Match String Consensus Problem

Previous work on finding the shortest common substring problem has ignored the process of generating the fragment strings. In this section we will look at a problem similar to the shortest common superstring with some added information about how the fragment strings were generated. In section 4 we will make much more severe assumptions about how the fragment strings were generated.

3.1 Problem definition

We will use the symbol $\uplus$ for multiset union. Let there be k identical copies of the string $W \in \Sigma^n$, $W_1, W_2, \ldots, W_k$. Associated with each W_i, $1 \leq i \leq k$, is a multiset of substrings, $\mathcal{S}_i = \{T_1, T_2, \ldots, T_{l_i}\}$, such that $W_i = T_1 \cdot T_2 \cdots T_{l_i}$. Let $\mathcal{S} = \biguplus_{1 \leq i \leq k} \mathcal{S}_i$. The perfect string consensus problem is to find all W that could generate $\mathcal{S}$ given the multiset $\mathcal{S}$ and the integer k.

3.2 How many distinct W exist?

For the remainder of this section, we will assume that we started with two copies of W, that is $k = 2$. There are two simple situations that will result in W not being unique. The first results when any two of the W's are cleaved at the same position. W is not unique if, for any h, i, and j, $j \neq |W|$

$$\boxed{w_i w_{i+1} \cdots w_j} \in \mathcal{S} \text{ and}$$
$$\boxed{w_h w_{h+1} \cdots w_j} \in \mathcal{S}$$

since

$$W = \boxed{w_1 \cdots w_{i-1}}\boxed{w_i \cdots w_j}\boxed{w_{j+1} \cdots w_{|W|}} = \boxed{w_1 \cdots w_{h-1}}\boxed{w_h \cdots w_j}\boxed{w_{j+1} \cdots w_{|W|}}$$

and

$$W = \boxed{w_{j+1} \cdots w_{|W|}}\boxed{w_1 \cdots w_{h-1}}\boxed{w_h \cdots w_j} = \boxed{w_{j+1} \cdots w_{|W|}}\boxed{w_1 \cdots w_{i-1}}\boxed{w_i \cdots w_j}$$

The second situation that will result in W not being unique can be described as follows. If we have seven distinct strings,

$$R1, R2, R3, R4, R5, R6, R7 \in \mathcal{S}$$

$$R1, R2, R3, R4, R5 \in \mathcal{S}_1, \text{ and}$$
$$R6, R7 \in \mathcal{S}_2$$

such that

$$\begin{aligned}
\mathsf{suffix}(R1) \cdot R2 \cdot \mathsf{prefix}(R3) &= R6 \\
\mathsf{suffix}(R4) \cdot \mathsf{prefix}(R5) &= R7 \\
\mathsf{suffix}(R1) &= \mathsf{suffix}(R4) \\
\mathsf{prefix}(R3) &= \mathsf{prefix}(R5)
\end{aligned}$$

then there must be more than one arrangement of the R's to construct W.

$$W = \ldots R1 \cdot R2 \cdot R3 \ldots R4 \cdot R5 \ldots = \ldots .R6 \ldots\ldots\ldots .R7 \ldots$$

and

$$W = \ldots R1 \cdot R3 \ldots R4 \cdot R2 \cdot R5 \ldots = \ldots R7 \ldots R6 \ldots$$

The first case is actually a special case of the second. This can be seen by letting prefix $(R5) = R3$ and noting that both $R3$ and $R6$ must end at the same position in W.

In a similar way we can create a multiset of n strings that can be arranged to form $n!$ distinct W's where, for $k = 2$, $|W| = \frac{n^2}{2} + \frac{7n}{2} + 6$. Similar arguments can be made for $k > 2$. Let the multiset of strings $\mathcal{X}_n$ be defined as

$$\begin{aligned}
\mathcal{X}_0 &= \{\mathrm{xyyx,\ xy,\ yxxy,\ xy}\} \\
\mathcal{X}_n &= \mathcal{X}_{n-1} \uplus \{\mathrm{yx}^{n+1}\mathrm{y},\ \mathrm{xyyx},\ \mathrm{x}^{n-1}\}
\end{aligned}$$

The length of W will be

$$\begin{aligned}
|W| &= \frac{\sum_{X \in \mathcal{X}_n} |X|}{k} \\
&= \frac{n^2}{2} + \frac{7n}{2} + 6
\end{aligned}$$

There are $n!$ distinct arrangements of the strings in $\mathcal{X}_n$ that form W. The form of these arrangements will be:

$$W = \mathrm{xyyx} \cdot \mathrm{x}^{h_1} \cdot \mathrm{xyyx} \cdot \mathrm{x}^{h_2} \cdots \mathrm{xyyx} \cdot \mathrm{x}^{h_n} \cdot \mathrm{xy} = \mathrm{xy} \cdot \mathrm{yxx}^{h_1}\mathrm{xy} \cdot \mathrm{yxx}^{h_2}\mathrm{xy} \cdots \mathrm{yxx}^{h_n}\mathrm{xy}$$

where $h_1, h_2, \ldots, h_n$ are distinct integers, $0 \leq h_i < n$. The order of the n pairs x^{h_i} and $\mathrm{yxx}^{h_i}\mathrm{xy}$ in the construction of W is arbitrary, so there are $n!$ distinct strings that can be W. Notice that in no case are the two constructions of W cleaved at the same position (case 1).

3.3 Algorithm to find W

Since it is possible that so many different W exist, if we must produce all possible W then the best we can do is to search the entire space. Given two strings, V and W, $i = |V|$, $j = |W|$, $i \leq j$, such that $v_1 = w_1$, $v_2 = w_2, \cdots, v_i = w_i$, we will define the extension of V to W to be the string $w_{i+1} \cdots w_j$. We can build candidate strings, W_1 and W_2, from strings in $\mathcal{S}$ by choosing some string $S_1 \in \mathcal{S}$ and letting it be a prefix of W_1. We must then remove S_1 from $\mathcal{S}$ so that it will not be reused in the construction of W.

$$\begin{aligned} W_1 &= \boxed{S_1} \\ W_2 &= \end{aligned}$$

The extension of W_2 to W_1 is just S_1. Next we compute S',

$$\begin{aligned} S' = \{S | S \in \mathcal{S} \land (\mathsf{prefix}(S) = \mathsf{extension}(W_1, W_2) \lor \\ S = \mathsf{prefix}(\mathsf{extension}(W_1, W_2)))\} \end{aligned}$$

We next pick some $S_2 \in \mathcal{S}'$ and make it a prefix of W_2. We now have

$$\begin{aligned} W_1 &= \boxed{S_1} \\ W_2 &= \boxed{\quad S_2 \quad} \end{aligned}$$

The process of computing the extension of W_1 to W_2 and finding the multiset of strings that could be used to extend the shorter of W_1 and W_2 is repeated as long as it is possible to do so. When it becomes impossible to continue and $\mathcal{S}$ is not empty, we must backtrack and pick some new S_i from some $\mathcal{S}'$ and try again.

The algorithm in Figure 1 assumes that $k = 2$, although the same ideas will work for any k. Let E be the extension of the candidate strings for W_1 and W_2. Let $\mathcal{U}$ be the submultiset of $\mathcal{S}$ whose elements have not been used in the construction of W_1 or W_2.

```
find-W (E, 𝒰, 𝒫, Results)
if (𝒰 = Λ) ∧ (E = ε)
     Print reverse (Results)
else
     if 𝒫 = Λ backtrack
     else
          for each nextP ∈ 𝒫
               Results ← nextP ⊎ Results
               newE ← extension (E, nextP)
               newP ← {S|S ∈ 𝒮 ∧ prefixq (newE, S)}
               find-W (newE, 𝒰 - nextP, newP, Results)
```

Figure 1: Algorithm to compute W

Let $\mathcal{P}$ be the submultiset of $\mathcal{U}$ whose elements have prefixes that exactly match E. The function prefixq will be true if either argument is a prefix of the other. This algorithm has been implemented in lisp and has run on various computers.

4 The String Consensus Problem

The perfect string consensus problem is not a good abstraction of the problem biochemists are faced with when they need to produce a consensus sequence from fragment sequences. In the perfect string consensus problem, every segment of each of the k copies of W must be in $\mathcal{S}$. We believe the string consensus problem, defined below, is a better abstraction of the problem biochemists are trying to solve.

4.1 Introduction

Finding the minimum length superstring of a set of strings seems to require us to look at many of the possible alignments of the strings in the set. When molecular biologists try to solve the similar problem of aligning their sequence fragments into a contiguous sequence, they assume that matches of a length greater than some constant are "significant". In this section will use this assumption to construct an algorithm to build a contiguous sequence. This assumption will allow us to find alignments that are "good enough" and not require us to search the entire space of alignments. We will see that, in these algorithms, the run time is directly related to the compression.

We will use calligraphic letters such as $\mathcal{S}$ to denote multisets of strings. Capital

letters late in the alphabet will be used to indicate strings, while lower case letters will be used for characters of a string such as $S = s_1 s_2 \ldots s_n$. The strings are composed of characters from the alphabet Σ, $\sigma = |\Sigma|$. We will let N be the sum of the lengths of the strings in $\mathcal{S}$, $N = \Sigma_{S_i \in \mathcal{S}} |S_i|$. Let C be the compression, $C = N - |S_{consensus}|$. Let suffix (T, j) be the suffix of T that has length j, similarly for prefix (T, j).

In this section we will present two new algorithms to solve the consensus string problem. The first uses the ideas of Rabin–Karp [KR87] string matching. The second algorithm sorts the strings before building a consensus string. The ideas of Knuth, Morris & Pratt [KMP77], and Boyer & Moore [BM77] were considered, but it is not clear to us how to construct a small finite automata that would match strings with errors.

4.2 Assumptions

By making the following assumptions we will define the string consensus problem.

- An integer k can be supplied that defines the minimum acceptable overlap between two strings.
- There is a unique alignment of the strings in $\mathcal{S}$ such that all suffix/prefix overlaps are of length k or greater.
- All suffix/prefix overlaps are exact matches.

4.3 The problem definition

We are given a multiset of strings, $\mathcal{S} = \{S_1, S_2, \ldots, S_n\}$, and an integer k. We make the following assumptions about $\mathcal{S}$ and k

1. S_i is not a substring of S_j for $1 \leq i,\ j \leq n,\ \ i \neq j$.
2. An ordering, H, of the strings in $\mathcal{S}$ exists such that

$$\forall_{1 \leq i < n}\, i\ \ \exists_{j \geq k}\, j\ \ \mathsf{suffix}(S_{H_i}, j) = \mathsf{prefix}(S_{H_{i+1}}, j).$$

The problem is, given the multiset $\mathcal{S}$ and the integer k, find the ordering H.

4.4 Naive algorithm

A naive algorithm to solve the string consensus problem is presented in Figure 2. The length k prefixes and suffixes of each pair of strings S_1, $S_2 \in \mathcal{S}$ are compared. If a prefix and suffix match, the strings S_1 and S_2 are removed from $\mathcal{S}$ and the string that results when S_1 and S_2 are joined is added to $\mathcal{S}$. Note that when a string is added to $\mathcal{S}$ it will be used in future prefix suffix comparisons. When no more prefix suffix matches of length k exist in $\mathcal{S}$ and $|\mathcal{S}| > 1$, k is incremented and the prefix suffix matching of the strings in $\mathcal{S}$ is repeated.

We will assign a comparison to the string S if a character in a suffix of S is compared to a character in the prefix of some T. Each iteration of the outer for loop will contribute one character of compression. The inner loop executes at most n times for each iteration of the outer for loop. The if statements in the inner for loop will take at most $O(N)$ time. The algorithm in Figure 2 has a worst case running time of $O(nCN)$.

In the average case we expect that almost all prefix suffix comparisons either match, or disagree after looking at a small constant number of characters. The expected time for the if statement in the inner for loop is constant. The expected time for the algorithm in Figure 2 is $O(nC)$.

The algorithm in Figure 2 has been implemented in C on a Sun 3/260. Figure 3 shows the results of running the program on strings of length 507. The length of the overlaps between strings was between 180 and 200 characters. The number of strings was varied between 25 and 500 and the amount of CPU time used to compute the consensus string is plotted. Since the length of the strings was held constant we expect the time to grow as $O(n^2)$. Figure 4 shows the square root of the running time plotted against the number of strings.

4.5 Rabin–Karp type algorithm

Given two strings S and T, $i = |S|, j = |T|, i < j$, the Rabin–Karp algorithm for string searching computes a hash value for the shorter string, S. A hash value for each length i substring of T is computed and compared with the hash value for S. By cleverly choosing the hashing function, the hash value for the substring of T ending

```
consensus_naive (S)

k ← minimum match length acceptable - 1
while |S| > 1
    k ← k + 1
    for each S_a ∈ S
        for each S_b ∈ S
            if suffix (S_a, k) = prefix (S_b, k)
                remove S_a and S_b from S
                add (join (S_a, S_b)) to S
            if suffix (S_b, k) = prefix (S_a, k)
                remove S_a and S_b from S
                add (join (S_b, S_a)) to S
```

Figure 2: Naive algorithm for the string consensus problem

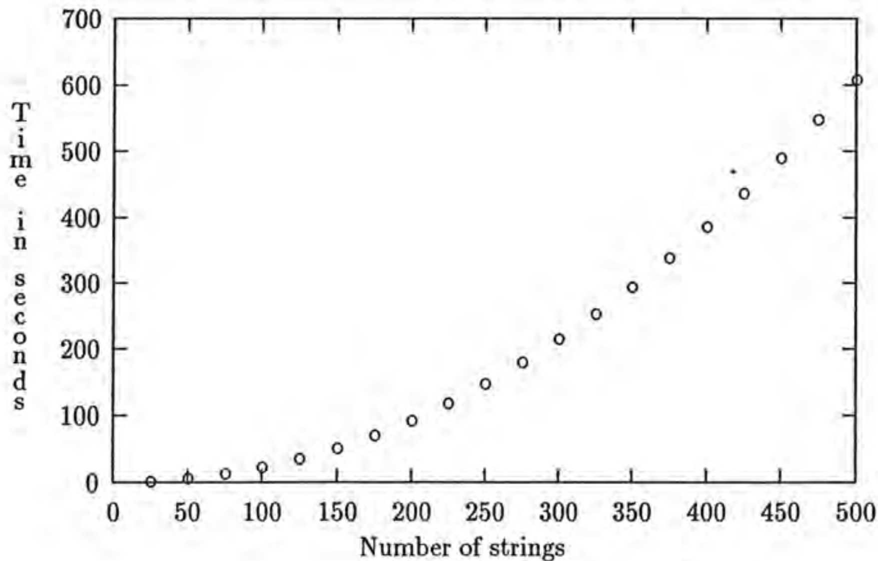

Figure 3: Running time for naive algorithm.

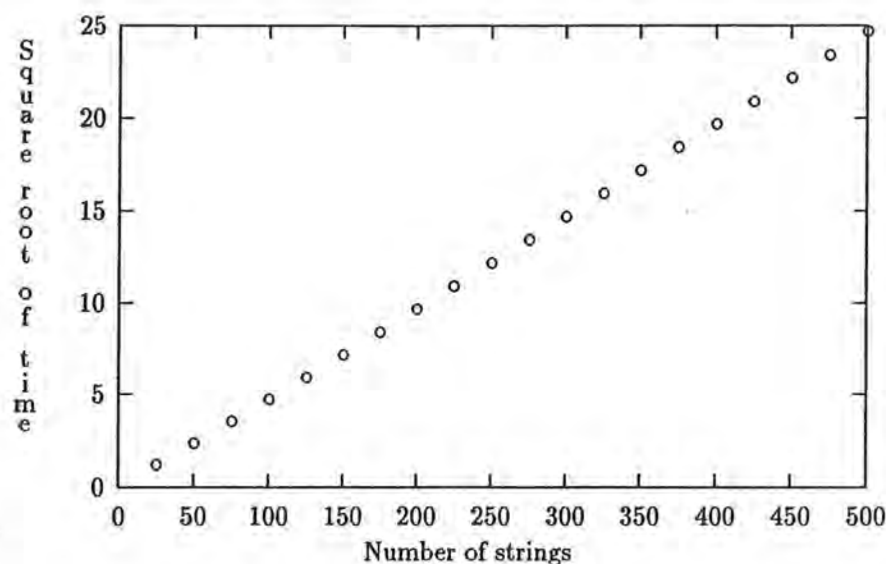

Figure 4: Square root of running time for naive algorithm.

at position h can be computed from the hash value for the substring of T ending at position $h-1$ and the character t_h. The hash function Rabin and Karp [KR87] chose was $\mathsf{hash}(m) = m \bmod p$ where p is a large prime and m is an integer representation of the string $T_{h-i+1} \ldots T_h$. With this in mind, it is easy to compute the hash value of the length i substring of T ending at position h

$$\mathsf{hash}(h) = ((\mathsf{hash}(h-1) - \mathsf{index}(t_{h-i}) \cdot \sigma^{i-1}) \cdot \sigma + \mathsf{index}(t_h)) \bmod p$$

Associated with each $c \in \Sigma$ is a unique integer l, $0 \leq l < \sigma$. The function $\mathsf{index}(c)$ will return the integer associated with c.

The probability of two strings drawn randomly from Σ^i having the same hash value is shown by Gonnet & Baeza-Yates [GBY90] to be

$$\frac{1}{p} + O\left(\frac{1}{\sigma^i}\right).$$

With the appropriate choices of p and k, the frequency of collisions will be small.

The Rabin–Karp algorithm has two properties that make it particularly well suited to the consensus string problem.

- The hash value for the prefix and suffix of a string can be computed incrementally. Given the hash value for the length i prefix of a string, the hash value for the length $i+1$ prefix can be computed with a constant number of operations.

- The hash value of a substring can be incrementally computed equally well from the right or the left end of the substring.

Figure 5 gives an algorithm based on the ideas of Rabin and Karp for the string consensus problem. Since the strings we are comparing are always prefixes (suffixes), we do not need to subtract the value of the leading (trailing) character of the string. The jth forward hash value of the string S (fhv[S]) is the hash value of the string $s_1 s_2 \ldots s_j$. The function we will use to compute the jth fhv[S] from the $(j-1)^{st}$ fhv[S] is

$$\text{fhv}[S] = (\text{fhv}[S] \cdot \sigma + \mathsf{index}(s_j)) \bmod p.$$

Let $i = |S| - j + 1$. The jth backward hash value of the string S (bhv[S]) is the hash value of the string $s_i s_{i+1} s_{|S|}$. The function we will use to compute the jth bhv[S] from the $(j-1)^{th}$ bhv[S] is

$$\text{bhv}[S] = ((\mathsf{index}(s_i) \cdot \sigma^{j-1}) + \text{bhv}[S]) \bmod p.$$

The initial length k forward and backward hash values are computed for each $S \in \mathcal{S}$. Each fhv[S] and bhv[S] is added to a binary tree. Whenever the hash value being added to the tree matches a hash value already in the tree, the associated strings are compared. If the prefix of one matches the suffix of the other, the strings are removed from $\mathcal{S}$ and joined, the resulting string is added back to $\mathcal{S}$. When all of the possible prefixes and suffixes of length k have been joined, the value of k is incremented, the search tree is cleared and the hash values for the new value of k are computed.

If we assume that no collisions occur, the number of operations used by the Rabin–Karp type algorithm for the consensus string problem is

$$O(C \cdot \log |n|).$$

The lines of the inner for loop will be executed once for each position of compression in the final consensus sequence. The "if (fhv[S] $\notin$ hashtree) **add** (fhv[s], hashtree)" and "if (bhv[S] $\notin$ hashtree) **add** (bhv[s], hashtree)" lines take $O(\log n)$ time to execute and each of the other lines of the inner loop will take constant time, therefore the running time of the algorithm in Figure 5 will be $O(C \cdot \log |n|)$.

The hash tree is set up as a binary tree. Associated with each node is a hash value and a linked list of all strings that have the hash value associated with the node.

```
consensus_RK (𝒮)

dm ← 1
k ← minimum suffix–prefix length
compute initial values of fhv and bhv
while |𝒮| > 1
     hashtree ← nil
     for each S ∈ 𝒮
          j ← |S| - k + 1
          fhv[S] ← (fhv[S] * σ + index (s_k)) mod p
          bhv[S] ← (bhv[S] + (index (s_j)·dm)) mod p
          if (fhv[S] ∉ hashtree) add (fhv[S], hashtree)
          else join (S, matched string from hash tree)
          if (bhv[S] ∉ hashtree) add (bhv[S], hashtree)
          else join (S, matched string from hash tree)
     dm ← (dm·σ) mod p
     k ← k + 1
```

Figure 5: Rabin–Karp type algorithm for the string consensus problem

When a string has the same hash value as a node, a linear search is performed on the strings associated with the node and if a match is found, it is returned. If no match is found, the string is added to the linked list. In the extraordinary case where each string hashes to the same value, the algorithm will perform just as the naive algorithm.

The algorithm in Figure 5 has been implemented in C on a Sun 3/260. Figure 6 shows the results of running the program on strings of length 507. The length of the overlaps between strings was between 180 and 200 characters. The number of strings was varied between 25 and 1000 and the amount of CPU time used to compute the consensus string is plotted. Figure 7 show the time divided by the log of the number of strings as the number of strings was varied.

4.6 Algorithm based on sorting

In the naive algorithm a great deal of time is spent searching for a string in $\mathcal{S}$ with a particular prefix. If the list of prefixes were sorted, the time needed to search $\mathcal{S}$ for a string with a particular suffix could be significantly reduced. In this section we will use a trie to speed the search.

By sorting the strings using a bucket sort and keeping track of the positions of the buckets within the sorted list, we can find a prefix of length i using $O(i)$ operations.

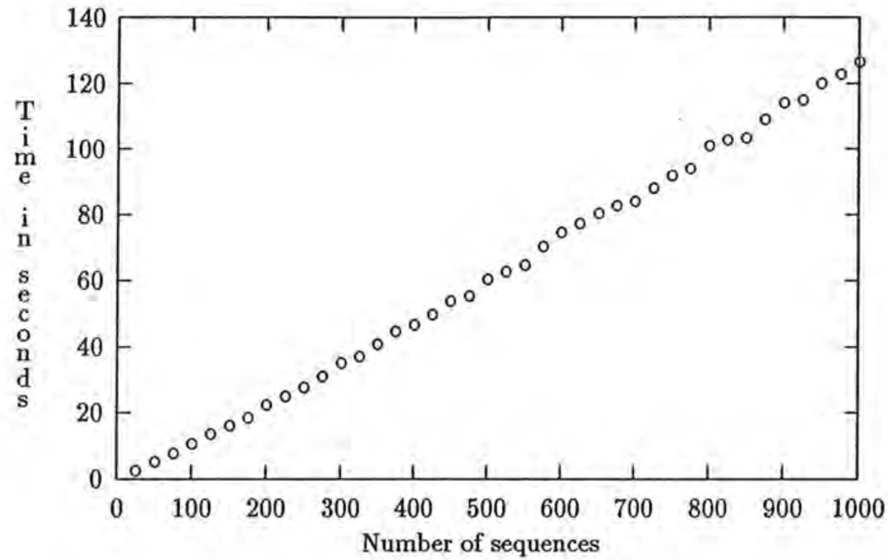

Figure 6: Running time for RK algorithm.

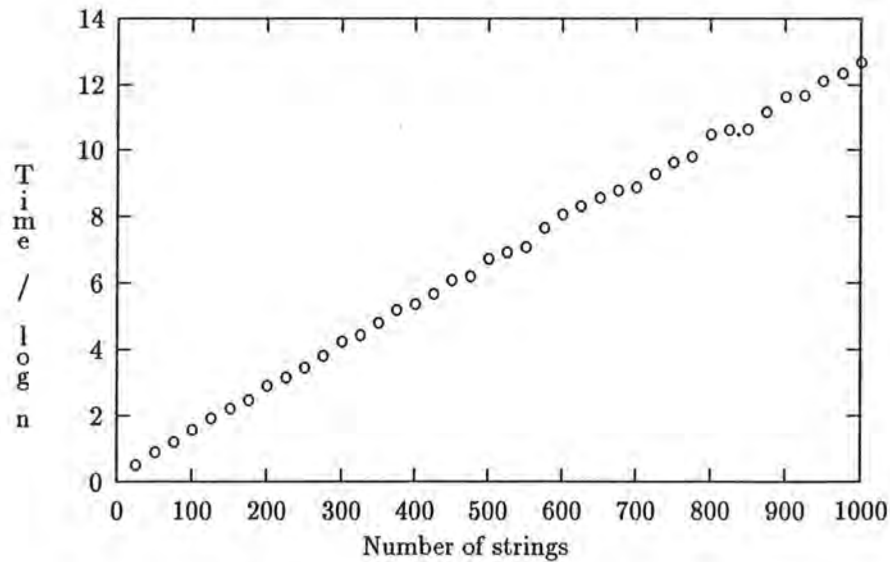

Figure 7: Running time / log (n) for RK algorithm.

```
consensus_trie (𝒮)

trie ← sort_seqs (𝒮, 0)
k ← minimum match length acceptable
while |𝒮| > 1
     for each S ∈ 𝒮
          R ← search (S, trie, k)
          if R ≠ Λ
               T ← join (R, S)
               add T to 𝒮
               remove R and S from 𝒮
     k ← k + 1
print one remaining entry in 𝒮
```

Figure 8: Algorithm to compute consensus sequence using a trie

During the standard bucket sort of the strings we will build a trie [Knu73] where a node at depth i represents a bucket containing all strings in $\mathcal{S}$ with a particular prefix of length i. The children of a depth i node represent the strings in $\mathcal{S}$ with prefixes of length $i+1$ where the strings associated with the parent node and the strings associated with the child node agree in positions 1 through i.

The function sort_seqs in Figure 9 will recursively sort the sequences and build the trie. When $|\mathcal{S}| > 1$, the strings in $\mathcal{S}$ will be sorted by the position indicated by the variable "column". The strings are then divided into at most σ groups, one group for each distinct character appearing at position "column" in some $S \in \mathcal{S}$. A node in the trie is created for each of the non-empty groups of strings. Each of these nodes is a child of the node representing $\mathcal{S}$. The function sort_seqs is then called recursively for each of the groups of strings.

The function sort_by_position will order the strings passed to it by the characters in the position passed to it. The function sort_by_position returns a node of the trie that points to the beginning of σ buckets along with the size of each bucket. The function sort_by_position takes time proportional to the number of strings being sorted. The time to sort the strings and construct the trie is proportional to N, the sum of the length of the strings, since each character of the strings in $\mathcal{S}$ is compared at most once.

The algorithm used to search the trie is given in Figure 10. It is essentially just a σ-ary tree search for the length j suffix of the string S. It does assume that you can index by the characters in Σ in constant time. If it is not possible to index by the

```
sort_seqs (S, column)
/* column is the position in the strings that the strings are to be ordered by */
    root ← nil
    if |S| > 1
        root ← sort_by_position (S, column)
        for each i, such that 0 ≤ i < σ
            S_i ← root.bucket_pos_i
            root.child_i ← sort_seqs (S_i, column + 1)
    return root
```

Figure 9: Algorithm to build trie and sort strings

```
search (S, trie, j)

/* Traverse trie */

    while (trie ≠ Λ) ∧ (j > 0)
        prev ← trie
        c ← s_{|S|-j}
        trie ← trie.next_c
        j ← j - 1
    j ← j + 1
    T ← prev.bucket_pos_c
    c_s ← s_{|S|-j}
    h ← 0
    c_t ← t_{h-j}

/* Compare strings */

    while (c_s ≠ c_t) ∧ (j > 1)
        j ← j - 1
        h ← h + 1
        c_s ← s_{|S|-j}
        c_t ← t_h
    if (c_s = c_t) ∧ (j = 1) return T
    else return nil
```

Figure 10: Algorithm to search a trie for a string

characters in Σ then an $O(\log|\Sigma|)$ search would have to be used to index the buckets and trie pointers. The worst case time required to determine if a prefix of length l exists in the $|S|$ strings is $O(l)$. This can be seen by noting that in the search procedure, at most one comparison is done at each character position and only characters in the prefix are compared.

The algorithm to find the consensus sequence using the sorted list of strings and the trie is given in Figure 8. For a given value of k, the suffix of length k of each string is searched for in the sorted list of prefixes. If a match is found the strings are removed from $\mathcal{S}$, joined, and the result of the join is added back to $\mathcal{S}$. When all of the the suffixes of length k have been searched for, k is incremented and the process is repeated.

Each iteration of the inner loop will result in one character of compression. In the worst case, the time to build the consensus string is

$$O(\sigma nC).$$

This situation arises when the the branching factor on the trie is nearly one. When the branching factor is closer to σ, the expected time to build the consensus string is

$$O(C\log n).$$

The algorithm in Figures 8, 9 and 10 has been implemented in C on a Sun 3/260. Figure 11 shows the results of running the program on strings of length 507. The length of the overlaps between strings was between 180 and 200 characters. The number of strings was varied between 250 and 10000 and the amount of CPU time used to compute the consensus string is plotted. Figure 12 shows the time divided by the log of the number of strings as the number of strings was varied.

5 Inexact Consensus Problem

The process of sequencing DNA and RNA is not perfect and mistakes are occasionally made in processing the gels, reading the gels and entering the data into a database. In this section, we will not assume that all suffix/prefix overlaps match perfectly, but that there are at most $\log(v)$ positions where the suffix and prefix do not match (v is the length of the overlap between the two strings).

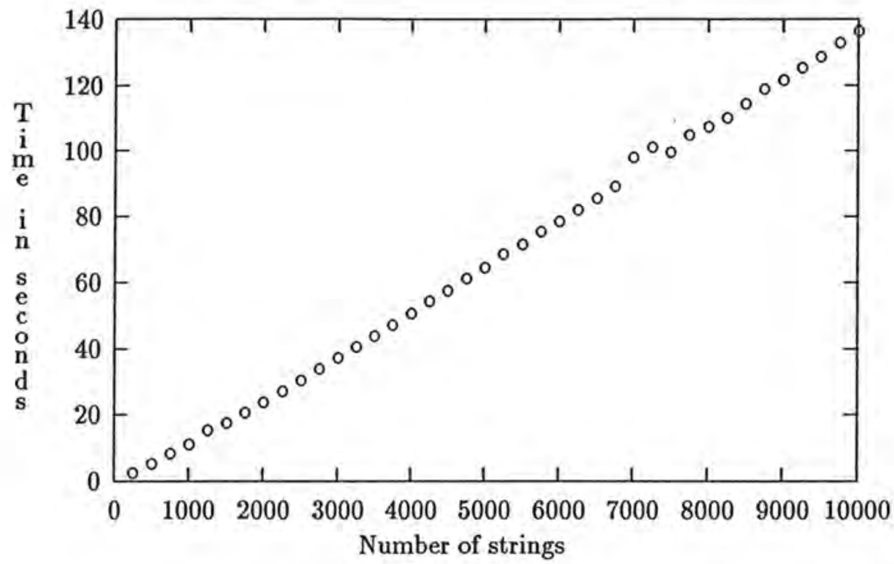

Figure 11: Running time for sort algorithm.

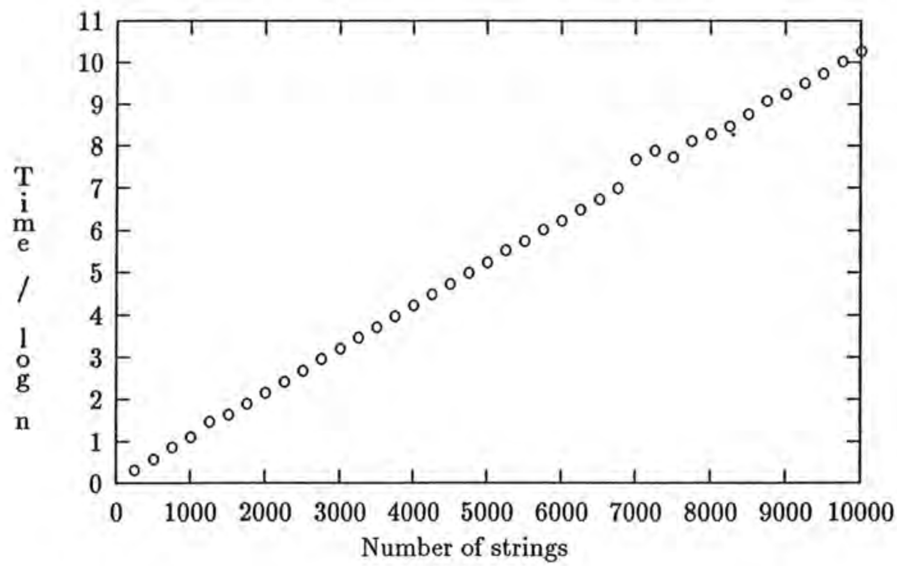

Figure 12: Running time / log (n) for sort algorithm.

We will discuss three algorithms to solve the log inexact string consensus problem. The first is a naive algorithm that we present for the purpose of comparison. The second algorithm is based on the Rabin–Karp string matching algorithm. The final algorithm that we will present is based loosely on the sorting algorithm presented in section 4.6.

The algorithm based loosely on the Rabin–Karp string matching algorithm will, with high probability, return the correct solution in

$$O(C|\mathcal{S}|(\log\log^2 p) + |\mathcal{S}|\log p) \text{ time.}$$

The worst case time bound for the Rabin–Karp based log inexact string matching algorithm is

$$O(nC\log p)$$

where p is some large prime, usually chosen to fit in a small number of words of memory.

The second algorithm, based very loosely on the sorting algorithm presented in section 4.6, will return a correct answer in

$$O\left(\frac{N^3\log k}{k}\right)$$

worst case time and

$$O\left(\frac{nN\log k}{|\Sigma|^{\frac{k}{\log k}}} + N\right)$$

expected time.

The large table required by the Rabin Karp algorithm makes it useful only for consensus sequences with very short prefix/suffix overlaps. The sorting based algorithm for the log inexact consensus sequence problem seems to be practical for a variety of problems.

We will say that the two length n string S and T "log inexact match" or $S \approx T$ if

$$\sum_{i=0}^{n-1}(s_i \oplus t_i) \leq \log n$$

where

$$s_i \oplus t_i = \begin{cases} 0 & \text{if } s_i = t_i \\ 1 & \text{if } s_i \neq t_i \end{cases}.$$

5.1 Assumptions

We will make the following assumptions and then define the log inexact string consensus problem.

- An integer k can be supplied that defines the minimum acceptable overlap between two strings.
- There is a unique alignment of the strings in $\mathcal{S}$ such that all suffix/prefix overlaps are of length k or greater.
- All suffix/prefix overlaps are log inexact matches.

5.2 Problem definition

We are given a multiset of strings, $\mathcal{S} = \{S_1, S_2, \ldots, S_n\}$, and an integer k. We make the following assumptions about $\mathcal{S}$ and k

1. S_i is not a substring of S_j for $1 \leq i,\, j \leq n,\ \ i \neq j$.
2. An ordering, H, of the strings in $\mathcal{S}$ exists such that

$$\forall_{1 \leq i < n}\, i\ \exists_{j \geq k}\, j\ \ \mathsf{suffix}(S_{H_i}, j) \approx \mathsf{prefix}(S_{H_{i+1}}, j).$$

The problem is, given the multiset $\mathcal{S}$ and the integer k, find the ordering H.

5.3 Naive Log Inexact Algorithm

The naive algorithm presented in Figure 13 will solve the log inexact consensus sequence problem. The algorithm simply tries all possible length k prefix/suffix log inexact matches, merges the log inexact matches that are found and increments k. This process is iterated until the only string remaining in $\mathcal{S}$ is the single consensus string.

Theorem 1 *The algorithm presented in Figure 13 will use*

$$O(nCN)$$

time in the worst case to compute the log inexact consensus string.

```
naive (S, k)
    while |S| > 1
        for each S_i ∈ S
            for each S_j ∈ {S − S_i}
                if verify (S_i, S_j, k)
                    add (join (S_i, S_j), S)
                    remove (S_i, S)
                    remove (S_j, S)
        k ← k + 1
```

Figure 13: Naive algorithm for log inexact consensus string problem

Proof. Each iteration of the outer for loop increases the compression by one. The time for verify(S_i, S_j, k) will be at most min($|S_i|, |S_j|$) since we can use a simple character by character comparison of the strings to determine if they log inexact match. The time to complete one iteration of the outer for loop will be no more than nN. So the time to compute the log inexact consensus sequence will be $O(nCN)$. ∎

Theorem 2 *The algorithm presented in Figure 13 will use*

$$O(nC \log N)$$

time in the average case to compute the log inexact consensus string.

Proof. Each iteration of the outer for loop increases the compression by one. The expected time for a verify call that fails is $O(\log N)$ since we only need to find $\log k$ errors and k may be as large as the longest string in $\mathcal{S}$. The total time spent in successful calls to verify is $O(C)$. Since the outer for loop is iterated once for each character of compression, the expected time is $O(nC \log N)$. ∎

If the overlap between adjacent strings in the log inexact consensus sequence and string length are both held constant, the worst case performance of the algorithm is $O(n^3)$ and the expected case is $O(n^2 \log n)$.

The naive algorithm was implemented in C and run on a Sun 3/260. Figure 14 shows the running time as the number of strings is varied from 4 to 120. Each string was 100 characters long and had an overlap length of 50 with its neighbor in the log inexact consensus sequence. The minimum overlap accepted (k) was 24 characters. Figure 15 shows the sqrt of the running times presented in Figure 14.

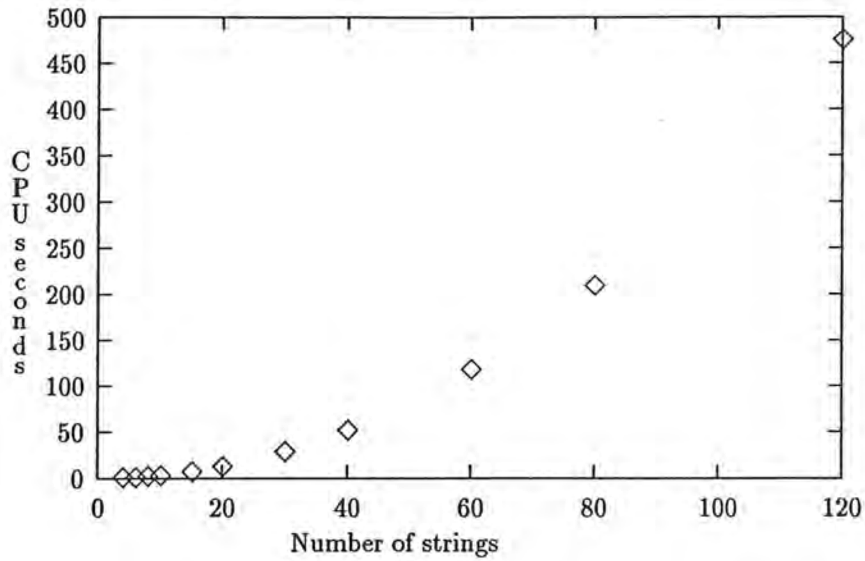

Figure 14: Running time of naive algorithm

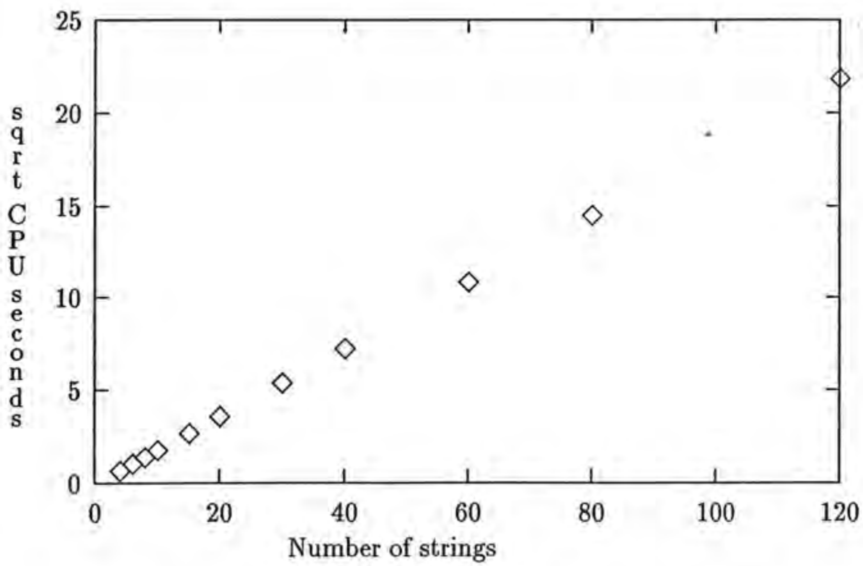

Figure 15: Square root running time of naive algorithm

```
enum-diffs (XOR-value, x, y, position)
    if bitposition (position, XOR-value) is set
        enum-diffs (XOR-value - 2^position, x + 2^position, y, position + 1)
        enum-diffs (XOR-value - 2^position, x, y + 2^position, position + 1)
    else if XOR-value > 0
        enum-diffs (XOR-value, x, y, position + 1)
    else if x > y
        print (x - y)
```

Figure 16: Algorithm to convert XOR value to differences

5.4 Rabin Karp Log Inexact Algorithm

We can use the ideas developed in section 4.5 and extend the definition of string equality to build an algorithm that will solve the log inexact consensus string problem. Let the function F have the domain of all $S \in \Sigma^*$ and the range of non-negative integers. $F(S)$ will be the base σ integer representation of the string S. Given two strings S and T, from Σ^q we let

$$\begin{aligned} F(S) &= \mathsf{index}(s_1) \cdot \sigma^{q-1} + \mathsf{index}(s_2) \cdot \sigma^{q-2} + \cdots + \mathsf{index}(s_q) \\ F(T) &= \mathsf{index}(t_1) \cdot \sigma^{q-1} + \mathsf{index}(t_2) \cdot \sigma^{q-2} + \cdots + \mathsf{index}(t_q). \end{aligned}$$

When there is exactly one position i such that $s_i \neq t_i$, $F(S)$ and $F(T)$ will differ by $d\sigma^{q-i}$, $0 \leq d < \sigma$. If $s_i \neq t_i$ and $s_j = t_j$ for all j, $0 \leq j < i$, $i < j \leq q$, then

$$F(S) = F(T) + (\mathsf{index}(s_i) - \mathsf{index}(t_i))\sigma^{q-i}$$

If the two strings S and T differ in positions i_1 and i_2 then $F(S)$ and $F(T)$ will differ by some additive combination of $d_1\sigma^{q-i_1}$ and $d_2\sigma^{q-i_2}$, $1 \leq d_1, d_2 \leq \sigma$, and so on for larger numbers of differences.

Before the log inexact consensus string is computed, a table of differences between $F(S)$ and $F(T)$ for all $S \approx T$ must be computed. A list of integers, XORlist, is generated. The binary representation of each integer in the list has at most $\log p$ ones, each one representing the position of a mismatch between S and T. Since the mod operator can be distributed over addition, but mod can not be distributed over XOR, the XORlist must be converted into a list of differences. For each integer in the XOR list, there may be several different values of S and T that will generate $F(S)$ XOR $F(T)$. The table of differences is the set of all positive differences between $F(S)$ and

$F(T)$ where $F(S)$ XOR $F(T)$ is in the XOR list. The algorithm in Figure 16 will return the set of all possible values $F(S)-F(T)$ such that XORvalue = $F(S)$ XOR $F(T)$. The enum–diffs algorithm in Figure 16 is initially called as

$$\text{enum}-\text{diffs}(F(S)\ \text{XOR}\ F(T), 0, 0, 0).$$

The RK log inexact algorithm presented in Figure 17 will find an ordering of the strings in $\mathcal{S}$ that solves the log inexact consensus string problem. The algorithm first computes the forward and backward hash values for each $S \in \mathcal{S}$ for the length k prefixes and suffixes (fhv[S] and bhv[S] respectively). The while loop will proceed as long as two conditions hold, there is more than one string in $\mathcal{S}$ and the length of the prefixes being examined is less than $\log p$ times some constant. Each iteration of the while loop looks for prefix/suffix matches of length one greater than the previous iteration and starts at k, the minimal acceptable prefix/suffix match length. The two for loops simply select pairs of strings to be examined. The forward and backward hash values are computed for strings being compared and then the difference of the two hash values is computed. The errorlist is searched for the difference and if the difference is found, it is verified that the two strings do have a prefix/suffix log inexact match of length at least k. Once the prefix/suffix log inexact match has been verified, the strings are removed from $\mathcal{S}$, joined and the joined string is returned to $\mathcal{S}$.

5.5 Reliability

The algorithm in Figure 17 relies on the values $F(S) - F(T)$, $S \approx T$, to be sparsely distributed between 0 and $p-1$. To estimate the running time of this algorithms we need to know how frequently $S \not\approx T$ and $F(S) - F(T) \in$ errorlist.

We will show that the number of error values when $S_1 \approx S_2$ is small compared to the complete range of error values. Therefore, when $S_1 \not\approx S_2$, it is unlikely that $F(S_1) - F(S_2) \in$ errorlist.

Lemma 1 *Given q pairs of strings, $<S_1, S_2>$, where S_1 and S_2 are randomly selected from $\Sigma^{\log q}$ such that $S_1 \not\approx S_2$, we expect $\log\log q \cdot \log q^{\log\log q}$ of the q pairs to have $F(S_1) - F(S_2) \in$ errorlist.*

```
match (𝒮, k, errorlist)
for each S ∈ 𝒮
    fhv[S] ← index(s_1)·σ^(k-1) + index(s_2)·σ^(k-2) + ··· + index(s_k)   (mod p)
    bhv[S] ← index(s_{|S|-k+1})·σ^(k-1) + index(s_{|S|-k+2})·σ^(k-2) + ··· + index(s_{|S|})   (mod p)
dm ← 1
while (|𝒮| > 1) ∧ (k < c·log p)
    k ← k + 1
    for each S ∈ 𝒮
        fhv[S] ← fhv[S] * σ + index(s_k)   (mod p)
        for each T ∈ (𝒮 − S)
            bhv[T] ← bhv[T] + index(t_{|T|-k}) * dm   (mod p)
            diff ← |fhv[S] − bhv[T]|
            if diff ∈ errorlist
                if verify (S, T, k)
                    add (join (S,T), 𝒮)
                    remove (S, 𝒮)
                    remove (T, 𝒮)
    dm ← dm * σ   (mod p)
```

Figure 17: RK based algorithm for log inexact match problem

Proof. We know that

$$\binom{x}{\log x} = \frac{x\cdot(x-1)\cdot\dots\cdot(x-\log x+1)}{\log x\cdot(\log x-1)\cdot\dots\cdot 2} \le x^{\log x}$$

so the number of error values in the error list is at most

$$\sum_{i=0}^{\log\log q}\binom{\log q}{i} \le \sum_{i=0}^{\log\log q}\binom{\log q}{\log\log q} \le \log\log q\cdot\log q^{\log\log q}.$$

Since we are assuming that the hashing function is randomly distributing the hash values from $0\dots q-1$ the number of times that $F(S_1)-F(S_2)\in$ errorlist will be about the same as the number of values in errorlist. ∎

5.5.1 Running time

The log inexact RK algorithm in Figure 17 will not build a tree of prefix values as the exact RK algorithm (See Figure 5) does, but must scan the entire list of prefix values sequentially since strings that are nearly the same may have very different hash values.

If we limit the number of mismatches to $O(\log v)$ then the number of error values will be at most $O(v^{\log v})$ since each error value will be a combination of $\log v$ or fewer

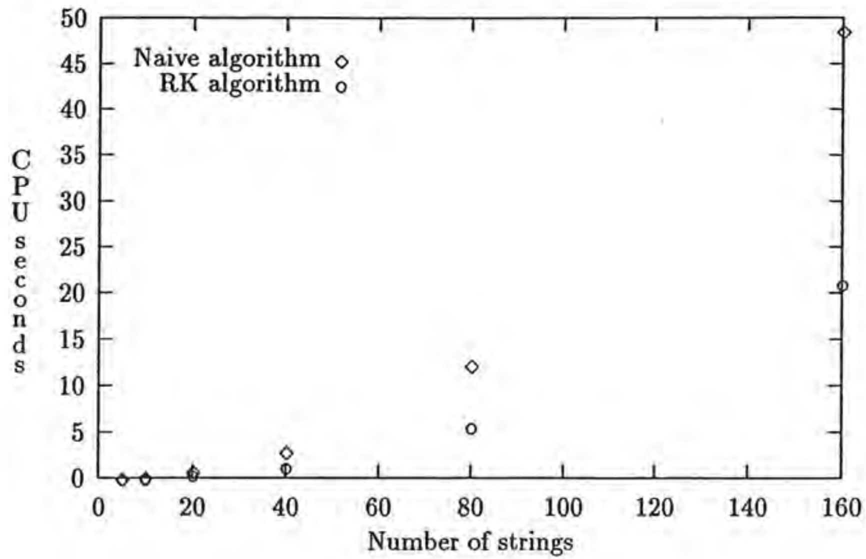

Figure 18: Running times for the naive and RK algorithms.

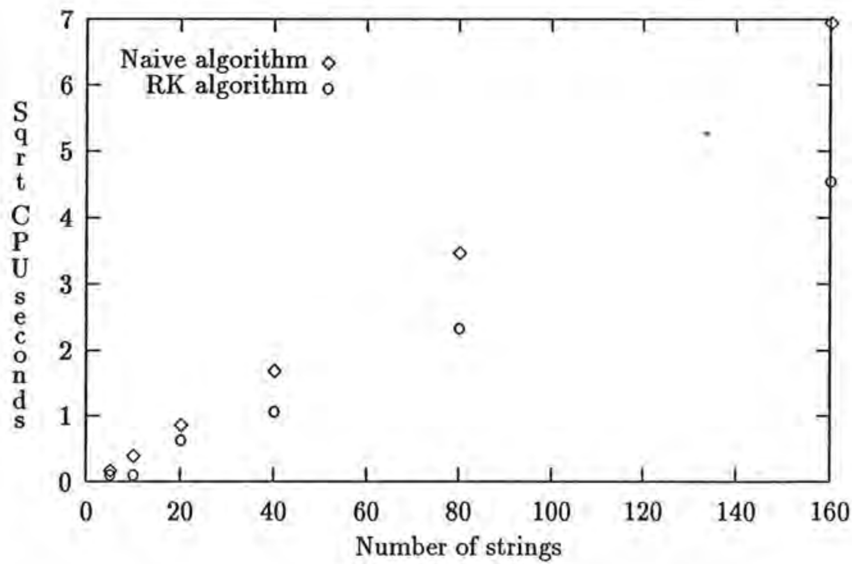

Figure 19: Square root of running times for the naive and RK algorithms.

powers of σ, $\{\sigma^0, \sigma^1, \ldots, \sigma^v, -\sigma^1, \ldots, -\sigma^v\}$ multiplied by some i, $i < \sigma$. These values can be precomputed and sorted so that the time to search the list of error values for a particular value will be $O(\log^2 v)$.

Theorem 3 *The algorithm presented in Figure 17 will use*

$$O(C|S|(\log\log^2 p) + |S|\log p)$$

expected time to compute the log inexact consensus string when v, the maximum overlap length between strings, is at most $\log p$.

Proof. The for loop that initializes the forward and backward hash values will execute in $O(nk)$. Each statement in the while loop will execute in constant time with the exceptions of the "if diff $\in$ errorlist" statement and the "if verify (S, T, k)" call. Since the number of values in the errorlist is $O(v^{\log v})$ and v is at most $\log p$, the "if diff $\in$ errorlist" statement will execute in $O((\log\log p)^2)$ time. Each iteration of the outer for loop will result in one character of compression. At most n searches of the errorlist will be done for each character of compression. Since there are C characters of compression, the running time without the calls to verify will be $O(nC(\log\log p)^2)$. By lemma 1 we see that if we let $v = \log p$ then we expect $\log\log p(\log p)^{\log\log p}$ incorrect inexact matches for every p pairs of strings that we look at. Since we will be looking at nC pairs of strings the expected number of incorrect calls to verify will be $O\left(nC\frac{(\log p)^{\log\log p}\log\log p}{p}\right)$. The expected time to discover that two strings do not log inexact match is $O(\log\log p)$ since there can be at most $\log\log p$ mismatches and we expect to find each mismatch by looking at a constant number of positions. The time contributed by incorrect calls to verify will be $O\left((\log\log p)^2\left(nC\frac{(\log p)^{\log\log p}}{p}\right)\right)$. Each of the $n-1$ correct calls to verify will take $O(\log p)$ time. The calls to verify will take $O\left((\log\log p)^2\left(nC\frac{(\log p)^{\log\log p}}{p}\right) + n\log p\right)$ time. Adding these times we get the running time of the RK inexact match algorithm.

$$O\left[(\log\log p)^2\left(nC\frac{(\log p)^{\log\log p}}{p}\right) + n\log p + nC(\log\log p)^2\right]$$

Since $\frac{(\log p)^{\log\log p}}{p} \leq 1$

$$n(\log\log p)^2 C \geq (\log\log p)^2\left(nC\frac{(\log p)^{\log\log p}}{p}\right)$$

leading to the running time of

$$O(C|\mathcal{S}|(\log\log^2 p) + |\mathcal{S}|\log p) \quad \blacksquare$$

Theorem 4 *The algorithm presented in Figure 17 will use*

$$O(nC\log p)$$

worst case time to compute the log inexact consensus string when v, the length of the overlap between strings, is at most $\log p$.

Proof. The time used to initialize the forward and backward hash values and the number of iterations of the for loops will be $O(nk)$. In the worst case, the "if diff $\in$ errorlist" statement will always be true and verify must always be called. Therefore there will be nC searches of the errorlist and calls to verify. Since each search of the errorlist takes $O\left((\log\log p)^2\right)$ time and each call to verify could, in the worst case, take $O(\log p)$ time, the worst case time is

$$O(nC(\log\log p)^2 + \log p[nC + n - 1]) \text{ or}$$
$$O(nC\log p).$$

The algorithm given in Figure 17 to solve the log inexact string consensus problem was implemented in C on a sun 3/260. The running time of this algorithm, as well as the running time of the naive algorithm, is shown in Figures 5.5.1. Figure 5.5.1 shows the square root of the running times.

5.5.2 Increased match length

We can increase the effective size of p by using the Chinese remainder theorem. Let p_1, p_2 ..., p_m be pairwise relatively prime positive integers, then the system of congruences

$$\begin{aligned} x &\equiv a_1 \pmod{p_1}, \\ x &\equiv a_2 \pmod{p_2}, \\ &\vdots \\ x &\equiv a_m \pmod{p_m}, \end{aligned}$$

```
match (𝒮, k, rprimes, errorlist)

for each S ∈ 𝒮
    for each p_i ∈ rprimes
        fhv[S, i] ← index(s_1) · σ^(k-1) + index(s_2) · σ^(k-2) + ··· + index(s_k)   (mod p_i)
        bhv[S, i] ← index(s_(|S|-k+1)) · σ^(k-1) + index(s_(|S|-k+2)) · σ^(k-2) + ··· + index(s_|S|)
                                                (mod p_i)
for each p_i ∈ rprimes dm[i] ← 1
while (|𝒮| > 1) ∧ (k < c · log p)
    k ← k + 1
    for each S ∈ 𝒮
        for each p_i ∈ rprimes
            fhv[S, i] ← fhv[S, i] * σ + index(s_k)   (mod p_i)
        for each T ∈ (𝒮 − S)
            for each p_i ∈ rprimes
                bhv[T, i] ← bhv[T, i] + index(t_(|T|-k)) * dm[i]   (mod p_i)
            for each p_i ∈ rprimes
                diff[i] ← |fhv[S, i] − bhv[T, i]|
            if diff ∈ errorlist
                if verify (S, T, k)
                    add (join (S, T), 𝒮)
                    remove (S, 𝒮)
                    remove (T, 𝒮)
    for each p_i ∈ rprimes
        dm[i] ← dm[i] * σ   (mod p_i)
```

Figure 20: Extended RK based algorithm for log inexact match problem

has a unique solution modulo $p = p_1 p_2 \cdots p_m$. [Ros84] With a few simple modifications to the algorithm in Figure 17 we can construct an algorithm that takes advantage of the Chinese remainder theorem. The extended RK inexact string matching algorithm is given in Figure 20.

The analysis of reliability does not change since the system of congruences has a unique solution modulo p. The running time of the extended RK inexact algorithm would increase by a factor of m.

5.5.3 Parallelism

There are at least two obvious ways to take advantage of parallelism for the algorithm in Figures 17 and 20. First, to increase the length of the prefix/suffix matches that we could confidently look at, we could, in parallel, compute the hash values modulo several primes instead of just one prime. Secondly, to increase the speed we could do almost all of the error list searching in parallel. We will look at the second of these ideas using several different models of parallel computation.

The following ideas are for the RK log inexact match algorithm in Figure 17. By using another factor of m processors the running time for the algorithm presented in Figure 20 will be the same as discussed below.

We can execute the "diff $\in$ errorlist" statement in constant time with $\log^2 v$ EREW[1] PRAM[2] processors if we are allowed to amortize the time. During each iteration of the inner loop, each of the $\log^2 v$ processors will be at a different position in the errorlist since a binary search on the errorlist is being done. There will be no time when two processors try to access the same position in the errorlist. If we can amortize the cost of the searches over the entire execution of the algorithm, the "diff $\in$ errorlist" statement can be done in constant time with $\log^2 v$ processors. By Brent's theorem [GR88] it is also true that the "diff $\in$ errorlist" statement can be done in $\log v$ time using $\log v$ processors.

The verify(S, T, k) statement can be done in $\log v$ time with $\frac{v}{\log v}$ EREW PRAM processors. Each processor, $P_i, 0 \leq i < \frac{v}{\log v}$, will count the number of mismatches

[1] Exclusive Read Exclusive Write

[2] Parallel Random Access Machine

```
log-inexact-suffix-array (S, k)
    S_tot ← S_1 ∘ γ ∘ S_2 ∘ γ ∘ ··· ∘ γ ∘ S_n
    build a suffix-array for S_tot
    for each string S_i ∈ S
        position[i] ← search-suffix-array (S_i, S_tot, k, suffix_array)
```

Figure 21: Algorithm to solve the log inexact match problem

between the substrings $s_{i \log v} \ldots s_{(i+1)\log v - 1}$ and $t_{i \log v} \ldots t_{(i+1)\log v - 1}$. The sum of the $\frac{v}{\log v}$ mismatch results can be computed in $O(\log v)$ time.

With $O\left(\frac{\log p}{\log \log p}\right)$ EREW PRAM processors we can solve the log inexact consensus sequence problem in $O(nC \log \log p)$ time using the algorithm in Figure 17. The time processor product is equal to the worst case time for the sequential version of this algorithm.

With a factor of n more EREW PRAM processors we can do the inner loop in $\log \log p$ time giving a running time of $O(C \log \log p)$ time using $n\left(\frac{\log p}{\log \log p}\right)$ EREW PRAM processors.

Using $O\left(n^2\left(\frac{\log p}{\log \log p}\right)\right)$ CREW[3] PRAM processors we could solve the log inexact consensus sequence problem in $O(n \log \log p)$ time. With this many processors we can do all of the n^2 string comparisons in parallel. The CREW PRAM processors are needed since n processors will be examining each $S \in \mathcal{S}$.

Since the hash values for prefixes and suffixes of length k are dependent on the hash values of the prefixes and suffixes of length $k-1$ the iterations of the while loop of the algorithm in Figure 17 can not easily be parallelized.

5.6 Suffix Array Based Log Inexact Algorithm

Two strings, S_i and S_j of length m, with at most $\log m$ positions that do not match must have a common substring S', $|S'| \geq \frac{m}{\log m}$. In this section we will develop an algorithm based on this simple observation to solve the log inexact consensus sequence problem.

5.7 Algorithm

[3] Concurrent Read Exclusive Write

```
search-suffix-array (S_i, S_tot, k, suffix_array)
    ptr ← 0
    blocksize ← ⌊k / log k⌋
    while ptr < |S_i|
        P = search (S_i [ptr] ... S_i [ptr + blocksize - 1], suffix_array)
        for each P_j ∈ P
            if verify (S_i, S_tot [P_j - ptr])
                return P_j - ptr
        ptr ← ptr + blocksize
    return nil
```

Figure 22: Algorithm to find log inexact matches using a suffix array

Figures 21 and 22 give an algorithm for the log inexact consensus sequence problem. Given the set of strings $\mathcal{S} = \{S_1, S_2, \ldots, S_n\}$ and an integer k that specifies the minimum length overlap, the algorithm will compute the log inexact consensus sequence. The string

$$S_{tot} = S_1 \circ \gamma \circ S_2 \circ \gamma \cdots \gamma \circ S_n,$$

where $\gamma \notin \Sigma$, is composed and a suffix array is created for S_{tot} (See [MM90]). Each string $S_i \in \mathcal{S}$ is positioned in S_{tot} so that a prefix of S_i log inexact matches a prefix of some suffix of S_{tot}. The log inexact match must be at least k characters in length and must not include the character γ. Knowing the position of each S_i in S_{tot} will allow us to easily construct a log inexact consensus sequence.

Positioning each $S_i \in S_{tot}$ is done using the algorithm in Figure 22. The string S_i is partitioned into $\frac{|S_i| \log k}{k}$ substrings, each of length $\frac{k}{\log k}$. While a log inexact match has not been found for S_i, the suffix array is searched for the substrings of S_i. For each exact match between a substring of S_i and a substring of S_{tot}, the the location of the exact match is used to verify that a log inexact match exist between a prefix of S_i and a suffix of some S_j in S_{tot}.

5.7.1 Worst case running time

Theorem 5 *The worst case running time of the algorithm in Figures 21 and 22 is* $O\left(\frac{N^3 \log k}{k}\right)$.

Proof. The worst case time to build the suffix array is $O(N \log N)$ [MM90]. The

function verify can easily be computed in time linear in the size of the strings using a simple character by character comparison. For each $S_i \in \mathcal{S}$ there will be at most $N\frac{|S_i|\log k}{k}$ calls to verify since there will be at most N substrings returned from the function search and at most $\frac{|S_i|\log k}{k}$ calls to search will be made. The total time spent in the function verify will be at most

$$N|S_1|\frac{|S_1|\log k}{k} + N|S_2|\frac{|S_2|\log k}{k} + \ldots + N|S_n|\frac{|S_n|\log k}{k}$$

$$= \frac{\log k}{k}\sum_{i=1}^{n} N|S_i|^2$$
$$= O\left(\frac{N^3\log k}{k}\right)$$

Searching the suffix array (calling the function search) for S_i takes $O(|S_i| + \log|S_{tot}|)$ [MM90] time. For each S_i there will be at most $\frac{|S_i|\log k}{k}$ calls to search so the total time spent in search will be at most

$$\sum_{i=1}^{n}\left[\left(\frac{|S_i|}{\log|S_i|} + \log|S_{tot}|\right)\frac{|S_i|\log k}{k}\right] = \frac{\log k}{k}\left(\sum_{i=1}^{n}\frac{|S_i|^2}{\log|S_i|} + \sum_{i=1}^{n}|S_i|\log N\right)$$
$$\leq O(\frac{N^3\log k}{k})$$

5.7.2 Expected running time

The expected running time is significantly better than the worst case running time. Before we give the expected running time we need to prove the following lemma that will be used to show that we can expect to look at a constant number of the $P_j's$ in Figure 22.

Lemma 2 *Given m urns and $m-1$ balls, each ball placed in a randomly selected urn, the expected number of empty urns after each ball has been placed in an urn is me^{-1} as m goes to infinity.*

Proof. Let P_i be the probability that urn U_i is empty.

$$P_i = \left(1 - \frac{1}{m}\right)^{m-1}$$

The expected number of empty urns is

$$\begin{aligned}\sum_{i=1}^{m} P_i &= m\left(1-\frac{1}{m}\right)^{m-1} \\ &= \frac{m}{1-\frac{1}{m}}\left(1-\frac{1}{m}\right)^{m} \\ \lim_{m\to\infty}\left(\frac{m}{1-\frac{1}{m}}\left(1-\frac{1}{m}\right)^{m}\right) &= me^{-1} \quad \blacksquare\end{aligned}$$

Theorem 6 *The expected running time of the algorithm in Figures 21 and 22 is*

$$O\left(\frac{nN\log k}{|\Sigma|^{\frac{k}{\log k}}}+N\right).$$

Proof. S_i is segmented into $\log k$ pieces of length $\frac{k}{\log k}$. We are likely to find a segment of S_i that exactly matches some substring of S_{tot} looking at a constant number of segments. This can be seen by lemma 2, let $m = \log k$, treat each segment of S_i as an urn, and each of the $\log k$ errors in the log inexact match as a ball. From lemma 2 we expect $\frac{\log k}{e}$ segments to contain none of the $\log k$ errors. Therefore, since the expected ratio of segments with no errors to total segments is a constant, we expect to look at a constant number of the segments to find a match with no errors.

We expect that search will return

$$\frac{N}{|\Sigma|^{\frac{k}{\log k}}}$$

substrings of S_{tot} since we are assuming that all length $\frac{k}{\log k}$ strings are equally likely to be substrings of S_{tot}. Each call to search will take $O\left(\frac{k}{\log k}\right)$ time. We expect each unsuccessful call to verify to take $\log k$ time since at each position of the potential match there is a $\frac{|\Sigma|-1}{|\Sigma|}$ chance that the characters do not match and we only need to find $\log k$ positions where the characters do not match. The total time spent in successful calls to the function verify will be less than $O(N)$. The time to build the suffix array is expected to be $O(N)$[MM90]. So, the expected time to solve the log inexact match problem using the algorithm in Figures 21 and 22 is

$$O\left(\frac{nN\log k}{|\Sigma|^{\frac{k}{\log k}}}+N\right) \quad \blacksquare$$

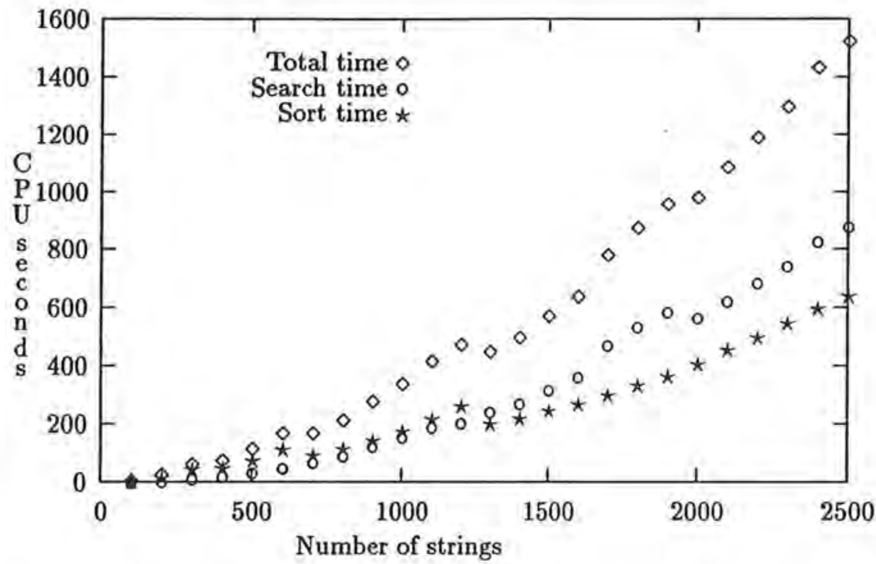

Figure 23: Running time of suffix array algorithm varying number of strings

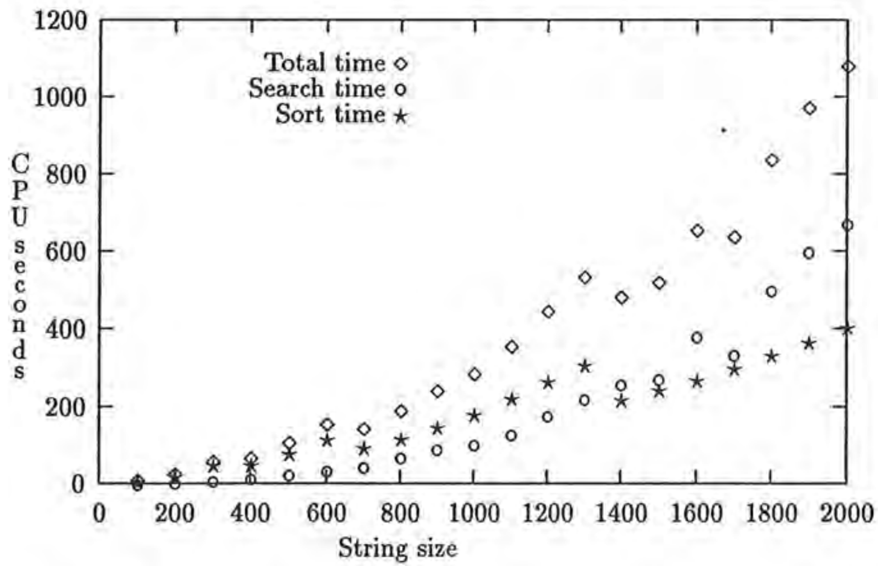

Figure 24: Running time of suffix array algorithm varying size of strings

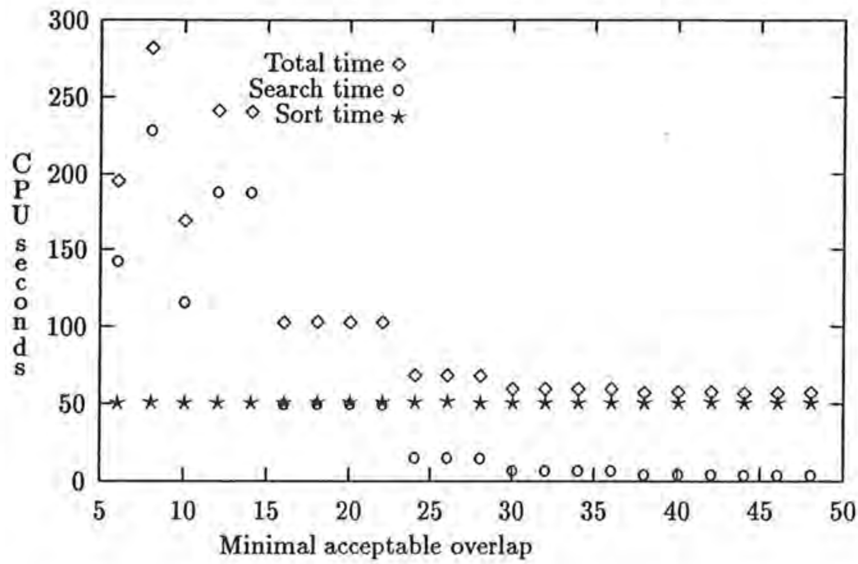

Figure 25: Running time of suffix array algorithm varying size of minimum overlap

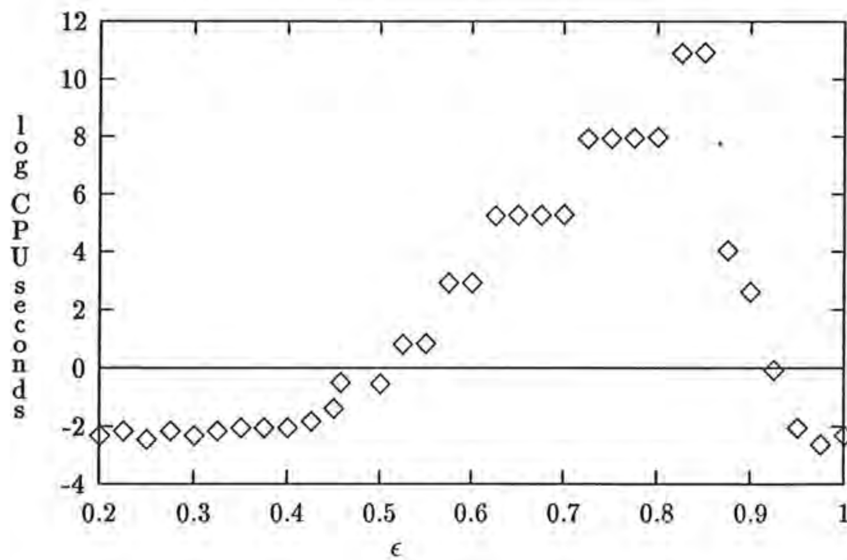

Figure 26: Log of running time of suffix array algorithm allowing n^ϵ errors

5.7.3 Implementation and Discussion

The algorithm discussed in section 5.6 has been implemented in C and run on a Sun 3/260. Figure 23 shows the running time of the algorithm presented in figures 21 and 22 as the number of strings is varied from 100 to 2500 while the length of the strings is 100, the minimum acceptable overlap is 24, and the overlap between adjacent strings is 50. Figure 24 shows the running time of the algorithm as the size of the strings is varied from 100 to 2000 characters while the number of strings is 100, the minimal acceptable overlap is 24, and the actual overlap is 50. Figure 25 shows the running time of the algorithm as the size of the minimum acceptable overlap is varied from 6 to 48. The number of strings is 200, the size of the strings is 200, and the actual overlap is 150.

Although the suffix array based log inexact consensus sequence algorithm was initially designed to construct a consensus sequence from sequence fragments, it can be used to align similar sequences. As an example of this, we have used the algorithm to align the following three sequences from GenBank [BB88].

1. *Saccharomyces cerevisiae* TATA-box factor (TFIID) gene, 5′ flank. This 1157 base pair DNA sequence was published by Schmidt et. al. [SKPB89] and has GenBank accession number M26403.

2. *S. cerevisiae* TATA-binding protein (TFIID) gene, complete cds. This 2439 base pair DNA sequence was published by Hahn et. al. [HBSG89] and has GenBank accession number M27135.

3. *S. cerevisiae* transcription initiation factor IID (TFIID). This 1140 base pair DNA sequence was published by Horikoshi et. al. [HWF+89] and has GenBank accession number X16860.

The sequences M26403 and M27135 can be aligned with 6 differences, the sequences M26403 and X16860 can be aligned with 7 differences and the sequences M27135 and X16860 can be aligned with 1 difference. Running the suffix array based log inexact algorithm uses 2.62 CPU seconds to build the suffix array and 0.08 seconds to align the sequences with a minimum acceptable overlap (the value k) of 24. The naive algorithm uses over 37 CPU seconds with the minimum acceptable overlap set to 24. To help the

naive algorithm we could remove the last 1000 base pairs of the sequence M27135 and set the minimum acceptable overlap to 900. With this help the naive algorithm still takes over 11 CPU seconds to compute the alignment.

5.7.4 Generalization

We have used $\log n$ errors in a match of length n simply because we used $\log n$ errors in the RK log inexact consensus sequence algorithm in section 5.4. Any function of n, $\text{fn}(n)$, such that $\text{fn}(n) \leq n$ could be used as the maximum number of errors allowed in a length n match. Following the same arguments that were used in sections 5.7.1 and 5.7.2 it can be shown that allowing $\text{fn}(n)$ errors in matches of length n, the algorithm in Figures 21 and 22 will use

$$O\left(\frac{N^3 \text{fn}(k)}{k}\right)$$

worst case time and

$$O\left(\frac{nN\text{fn}(k)}{|\Sigma|^{\frac{k}{\text{fn}(k)}}} + N\right)$$

expected time.

We allowed n^ϵ, $0 < \epsilon < 1$, errors for matches of length n. Figure 26 shows the log of the running time of the algorithm in Figure 21 and 22 when allowing n^ϵ errors in length n matches while varying ϵ. The data shown in Figure 26 is for 10 sequences, each 410 bases long with overlaps of 180 bases and a minimum acceptable overlap of 40 bases.

Figure 26 has a knee at about $\epsilon = 0.45$. As the number of errors allowed increases, the likelihood that some substring of S_{tot} will be falsely matched by the string

$$R = S_i[\text{ptr}] \ldots S_i[\text{ptr} + \text{blocksize} - 1]$$

(see Figure 22) increases. When the number of errors allowed is increased by one, the number of substrings in S_{tot} that falsely match R is expected to increase by a factor of σ. There is some value μ of ϵ where we expect there to be one substring in S_{tot} that falsely matches R. We expect the running time to be constant for $0 < \epsilon\mu$ since we expect one call, the correct call, to verify for each pair of strings that match. As ϵ grows above μ we expect false calls to verify. The number of false calls to verify will grow by

a multiplicative factor of σ for each additional error allowed in a match. This is seen in the exponential growth in running time as ϵ increases from $\epsilon = 0.45$ to $\epsilon = 0.85$ in Figure 26. Eventually, as ϵ grows, there will be so many errors allowed that nearly any pair of strings will match and the number of calls to verify will fall.

6 Conclusion

We have defined the consensus string problem and presented two new algorithms to solve it. We let

- n be the number strings in the multiset $\mathcal{S} = \{S_1, S_2, \ldots, S_n\}$
- $N = \sum_{i-1}^{n} |S_i|$
- C be the compressions, $C = N - |S_{consensus}|$
- k be the minimum acceptable overlap length between strings

The first algorithm using ideas of Rabin–Karp exact string matching, is expected to solve the problem in $O(C \log n)$ time. The second algorithm developed to solve the consensus string problem is also expected to run in $O(C \log n)$ time, but in practice runs faster than the Rabin–Karp type algorithm by a factor of about ten.

We also define a similar problem, the log inexact consensus sequence problem, and present two new algorithms to solve this problem. The first algorithm used an extension of the ideas based on Rabin–Karp string matching developed for the consensus string problem. This algorithm, although not practical in many situations, is useful when the prefix/suffix overlap is known to be small. The algorithm is also easily parallelizable.

The second algorithm uses suffix arrays as developed by [MM90] and is expected to run in $O\left(\frac{nN\frac{\log k}{k}}{|\Sigma|^{\overline{\log k}}} + N\right)$ time. This algorithm is generalized to allow not only $\log k$ errors but any reasonable function of k errors. Finally, we give an example of the use of this algorithm to align three nearly identical DNA sequences of the TFIID gene in yeast, although this was not the intended use of the algorithm.

These algorithms allow a few bases to be transformed but do not allow bases to be deleted. If, for example, a base is missed while reading a GC rich region (an occasional

problem) the algorithms presented in this paper will not work since part of the match will be offset by one position. One area of future research will be designing algorithms for this problem that allows a small number of deletions as well as the transformations.

References

[AC75] A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18:333–340, 1975.

[BB88] H. S. Bilofsky and C. Burks. The genbank genetic sequence data bank. *Nucleic Acids Research*, 16:1861–1863, 1988.

[BJL+91] A. Blum, T. Jiang, M. Li, J. Tromp, and M. Yannakakis. Linear approximation of shortest superstrings. In *Proceedings of the ACM Symposium on Theory of Computing*, pages 328–336, Baltimore, MD, 1991. ACM press.

[BM77] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Communications of the ACM*, 20:762–772, 1977.

[CK82] J. Clayton and L. Kedes. Gel, a DNA sequencing project management system. *Nucleic Acids Research*, 10:305–321, 1982.

[GBY90] G. H. Gonnet and R. A. Baeza-Yates. An analysis of the Karp–Rabin string matching algorithm. *Information Processing Letters*, 34:271–274, 1990.

[GMS80] J. Gallant, D. Maier, and J. Storer. On finding minimal length superstrings. *Journal of Computer and System Science*, 20:50–58, 1980.

[GR88] A. Gibbons and W. Rytter. *Efficient Parallel Algorithms*. Cambridge University Press, New York, 1988.

[HBSG89] S. Hahn, S. Buratowski, P. A. Sharp, and L. Guarente. Isolation of the gene encoding the yeast TATA binding protein TFIID: A gene identical to the SPT15 suppressor of ty element insertions. *Cell*, 58:1173–1181, 1989.

[HWF+89] M. Horikoshi, C. K. Wang, H. Fujii, J. A. Cromlish, P. A. Weil, and R. G. Roeder. Cloning and structure of a yeast gene encoding a general transcription initiation factor TFIID that binds to the TATA box. *Nature*, 341:299–303, 1989.

[JJD86] R. E. Johnston, J. M Mackenzie Jr., and W. G. Dougherty. Assembly of overlapping DNA sequences by a program written in BASIC for 64k CP/M and MS-DOS IBM–compatible microcomputers. *Nucleic Acids Research*, 14:517–527, 1986.

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

[Knu73] D. E. Knuth. *The art of computer programming: searching and sorting*, volume 3. Addison–Wesley, 1973.

[KR87] R. M. Karp and M. O. Rabin. Efficient randomized pattern–matching algorithms. *IBM Journal of Research and Development*, 32:249–260, 1987.

[Li90] M. Li. Towards a DNA sequencing theory. In *IEEE Symposium on the Foundations of Computer Science*, pages 125–134, 1990.

[Mai78] D. Maier. The complexity of some problems on subsequences and supersequences. *Journal of the Association for Computing Machinery*, 25:322–336, 1978.

[MM90] U. Manber and G. Myers. Suffix arrays: A new method for on–line string searches. In *Proceedings of the First Annual ACM–SIAM Symposium on Discrete Algorithms*, pages 319–327. SIAM, 1990.

[PSTU83] H. Peltola, H. Soderlund, J. Tarhio, and E. Ukkonen. Algorithms for some string matching problems arising in molecular genetics. In *Information Processing 83*, pages 53–64, 1983.

[Ros84] K. H. Rosen. *Elementary Number Theory and Its Applications.* Addison–Wesley, Reading, MA, 1984.

[Sel74] P. H. Sellers. An algorithm for the distance between two finite sequences. *Journal of Combinatorial Theory (A)*, 16:253–258, 1974.

[SKPB89] M. C. Schmidt, C. Kao, R. Pei, and A. J. Berk. Yeast TATA-box transcription factor gene. *Proceedings of the National Academy of Science*, 86:7785–7789, 1989.

[Sta82] R. Staden. Automating of the computer handling of gel reading data produced by shotgun method of DNA sequencing. *Nucleic Acids Research*, 10:4731–4751, 1982.

[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*, 83:1–20, 1989.

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