Towards a DNA Sequencing Theory (Learning a String)

(Preliminary Version)

Ming Li
University of Waterloo

DNA: Death's Natural Alternative
— Globe & Mail, April 14, 1990

Abstract

In laboratories, DNA sequencing is (roughly) done by randomly sequencing large amount of relatively short fragments and then heuristically finding a shortest common superstring of the fragments.

We study mathematical frameworks, under plausible assumptions, suitable for massive automated DNA sequencing and for analyzing DNA sequencing algorithms. We model the DNA sequencing problem as learning a superstring from its randomly drawn substrings. Under certain restrictions, this may be viewed as learning a superstring in Valiant learning model and in this case we give an efficient algorithm for learning a superstring and a quantitative bound on how many samples suffice.

One major obstacle to our approach turns out to be a quite well-known open question on how to approximate the shortest common superstring of a set of strings, raised by a number of authors in the last ten years [6, 29, 30]. We give the first *provably* good algorithm which approximates the shortest superstring of length n by a superstring of length $O(n \log n)$.

[1]The author is supported by NSERC operating grants OGP-0036747 and OGP-0046506. Address: Computer Science Department, University of Waterloo, Ontario N2L 3G1, Canada. E-mail: mli@watmath.waterloo.edu

Key words: Machine Learning, Shortest Common Superstring, Approximation Algorithm, DNA sequencing.

1 Introduction

The fundamental quest to Life is to understand how it functions. Since the discovery of deoxyribonucleic acid (DNA) in the 1950's, the new field of molecular genetics has expanded at a rate that can be matched by no other fields except perhaps computer science. A DNA molecule, which holds the secrets of Life, consists of a sugar-phosphate backbone and, attached to it, a long sequence of four kinds of nucleotide bases.

At an abstract level, a single-stranded DNA molecule can be represented as a character string, over the set of nucleotides $\{A, C, G, T\}$. Such a character string ranges from a few thousand symbols long for a simple virus to 2×10^8 symbols for a fly and to 3×10^9 symbols for a human being. Determining this string for different molecules, or *sequencing* the molecules, is a crucial step towards understanding the biological functions of the molecules. Huge (in terms of money and man years) national projects of sequencing human genome are proposed or underway although, unfortunately, major technical problems remain.

With current laboratory methods, such as Maxam-Gilbert or Sanger procedure, it is by far impossible to sequence a long molecule (of 3×10^9 base pairs for human) as a whole. Each time, a randomly chosen fragment of only up to 500 base pairs can be sequenced. Various methods to sequence the long molecule are used. In general, biochemists "cut" millions of such (identical) molecules into pieces each typically containing about 200-500 nucleotides (characters) [4, 16]. For example, a biochemist can decide to cut the molecules whenever the substring GGACTT (via restriction enzymes) appears. She or he may apply many different cuttings using different substrings. After cutting, these fragments can be roughly "sorted" according to their "weight" ("length") in a not very precise fashion. A biochemist "samples" the fragments (of each length plus minus 50 base pairs) randomly. Then, say, Sanger procedure is applied to sequence the sampled fragment. A good technician can process, say, 7 to 14 such fragments a day manually. Sampling is expensive and slow in the sense that it is manual or, at the best, mechanical. The sampling is random in the sense that we have absolutely no idea where the fragment of 500 characters might come from the 3×10^9 characters long sequence. It is also well-known that long repetitions appear in a human genome.

From hundreds, sometimes millions, of these random fragments, a biochemist has to *assemble* the superstring representing the whole molecule. Programs, that try to approximate a shortest common superstring (whole molecule) of a given set of strings (fragments) are routinely used [23, 16]. However, it is not known whether this always works or why it works. It is reported in [23, 27, 16] that programs based on shortest common superstring algorithms[1] work very satisfactorily for real biological data. In [29], it is mentioned that "although there is no a priori reason to aim at a *shortest* superstring, this seems to be the most natural restriction that makes the program nontrivial." The goal of this paper is to initiate a study of mathematical foundations for above discussions and experiments. We will try to formulate the problem in different ways and provide solutions in certain situations. We face at least two major problems:

(a) Given a set of strings, how do we efficiently approximate the shortest common superstring (finding it is NP-complete)? This has been an open problem raised by Gallant, Maier and Storer [6], Turner [30], and Tarhio and Ukkonen [29] in the past ten years. The latter two papers contain two algorithms, based on the maximum overlapping method also used by biochemists, and a conjecture that these algorithms guarantee good performance with respect to optimal length.

(b) Even given good approximation algorithms for the superstrings, does that guarantee we can infer the DNA sequence correctly?

We will provide an answer for (a) by giving a provably good approximation algorithm for the shortest common superstring problem. Our algorithm outputs a superstring of length at most $n \log n$ where n is the optimal length. We will also partially answer (b). Under certain restrictions, we study how well the shortest superstring algorithm does and we provide a new "sequencing" algorithm under the Valiant learning model. We will also give quantitative bounds on the number of random fragments needed to be sequenced.

This paper studies Valiant learning in a domain where the sample space is of *polynomial size*. Since these concepts are trivially polynomially learnable, not much attention has been paid to them. In the past, researchers are concentrated on the "learnability" of concept classes whose sample spaces are, of course (otherwise the problem would be trivial), superpolynomial [31, 2, 12, 8, 26, 22, 11], although efficient sampling was studied for example in [9]. On the other hand, our problem has trivial algorithms that need high polynomial number of samples, but it also has non-trivial algorithms requiring low polynomial number of samples.

It worths noting that artificial intelligence methods [21] and string matching algorithms have been

[1] It was only conjectured that these algorithms approximate shortest superstrings.

extensively applied to DNA sequence analysis [5].

The paper is organized as follows: In the next section, we provide a solution to question (a). Our new approximation algorithm is also crucial in the following sections. In Section 3, we study possible ways of modeling our DNA sequencing problem. We will associate this problem with the Valiant learning model. We also state a stronger version of the Occam Razor theorem proved in [3], which is needed in Section 4. Then in Section 4 we give provably efficient learning algorithms for the DNA sequencing problem.

2 Approximating the Shortest Common Superstring

As we have discussed above, the shortest common superstring problem plays an important role in the current DNA sequencing practice. This section will present the first provably good algorithm for such problem.

2.1 The Shortest Common Superstring Problem

The *shortest common superstring* problem is: Given a finite set S of strings over some finite or infinite alphabet Σ, find a shortest string w such that each string x in S is a substring of w, *i.e.*, w can be written as uxv for some u and v. The decision version of this problem is known to be NP-complete [6, 7]. Due to its wide range of applications in many areas such as data compression and molecular biology [29, 30, 6], finding good approximation algorithms for the shortest common superstring problem has become an important subject of research. It is an open question to find a provably good approximation algorithm [6, 29, 30] for this problem. We provide a solution. The superstring constructed by our algorithm is at most $n \log n$ long where n is the optimal length.

There have been two independent results by Tarhio and Ukkonen [29] and Turner [30] both of which considered similar approximation algorithms. Both [29, 30] conjectured that their algorithms would return a superstring within the length of 2 times the optimal. But this Tarhio-Turner-Ukkonen conjecture and the question of finding an algorithm with non-trivial performance guarantee with respect to length remain open today. Papers [29, 30] did establish some performance guarantees with respect to so-called "overlapping" measure. Such measure basically measures the number of bits saved by the algorithm compared to plainly concatenating all the strings. It was shown that if the optimal solution saves m bits, then the approximate solutions in [29, 30] save at least $m/2$ bits. However, in general this has no implication with respect to the optimal length since this in the best case only says that the approximation procedure produces a superstring which is of half the total length of all strings. The basic algorithm of [29, 30, 28] is quite simple and elegant. It is as follows: Given a set of strings S. Choose a pair of strings from S with largest overlap, merge these two, and put the result back in S. Repeat this process until only one string, the superstring, is left in S. This algorithm has always produced a superstring of length within 2 times the optimal length in all, random or artificial, experiments so far.

We will give a different algorithm. Our algorithm guarantees that the solution is within a $\log n$ multiplicative factor of the optimal length n, although we believe that it achieves $2n$.

Notation: We usually use small letters for strings and capital letters for sets. If s is a string, then $|s|$ denotes its length, that is, number of characters in s. If S is a set, then $|S|$ denotes its cardinality, that is, number of elements in set S.

2.2 The Approximation Algorithm

Assume that we are given a set S of m strings over some finite or infinite alphabet Σ. Let t be an optimal superstring with $|t| = n$. So each string in S is a substring of t. Following [29, 30], we assume that no string in S is a substring of another since these strings can be easily deleted from S. Hence we can order the strings in S by the left ends of their first appearances in t, reading from left to right. We list them according to above ordering: $s_1, ..., s_m$. In the following we identify s_i with its first appearance in t.

The idea behind our algorithm is to merge large groups. Each time, we try to determine two strings such that merging them properly would make many others become substrings of the newly merged string.

For two strings s and s', we use the word *merge* to mean that we put s and s' (and nothing else) together, possibly utilizing some overlaps they have in common, to construct a superstring of the two. In general there may be more than one way to merge s and s'. There may be two optimal ways, and many other non-optimal ways. For example, if $s = 010$ and $s' = 00200$, we can merge them with s in front optimally as $m_1(s, s') = 0100200$ or with s' in front optimally as $m_2(s, s') = 0020010$; We can also merge them non-optimally as $m_3(s, s') = 01000200$ or $m_4(s, s') = 00200010$. These are all possible ways of merging s, s'. For each merge function m, we write $m(s, s')$ to denote the resulting superstring. There are at most $2\min\{|s|, |s'|\}$ ways of merging s and s'. We now present our algorithm.

Group-Merge: Input: $S = \{s_1, ..., s_m\}$.

(0) Let $T := \emptyset$.

(1) Find $s, s' \in S$ such that

$$cost(s, s') = \min_m \frac{|m(s, s')|}{weight(m(s, s'))}$$

is minimized where

$$weight(m(s, s')) = \sum_{a \in A} |a|,$$

where A is the set of strings in S that are substrings of $m(s, s')$.

(2) Merge s, s' to $m(s, s')$ as defined in Step (1). $T := T \cup \{m(s, s')\}$. Delete set A from S.

(3) If $|S| > 0$ then go to (1).

(4) If $|T| > 1$ then $S := T$ and go to (1), else return the only string in T as superstring.

In the above algorithm we could put the result $m(s, s')$ back into S, and hence get the following algorithm.

Group-Merge 1: Input: $S = \{s_1, ..., s_m\}$.

(0) Let $weight(s_i) = |s_i|$ for each string $s_i \in S$.

(1) Find $s, s' \in S$ such that

$$cost(s, s') = \min_m \frac{|m(s, s')|}{weight(m(s, s'))}$$

is minimized where,

$$weight(m(s, s')) = \sum_{a \in A} weight(a)$$

where A is the set of strings in S that are substrings of $m(s, s')$.

(2) Merge s, s' to $m(s, s')$ as defined in Step (1). Delete set A from S. Add $m(s, s')$ to S.

(3) If $|S| > 1$ then go to (1), else the only string left in S is the superstring.

In order to keep the analysis simple, we only analyze steps (0) - (3) of algorithm **Group-Merge**.

Theorem 1 *Given a set of strings S, if the length of optimal superstring is n, then algorithm* **Group-Merge** *produces a superstring of length $O(n \log n)$.*

Proof: As discussed above, we can assume that in S no string is a substring of another and all strings are ordered by their first appearance in the shortest superstring. Let this order be $s_1, s_2, ..., s_m$. We separate S into groups: The first group G_1 contains $s_1, ..., s_i$ where i, $i \leq m$, is the largest index such that (the first appearances of) s_1 and s_i overlap in t; The second group contains $s_{i+1}, ..., s_j$ where j, $i + 1 \leq j \leq m$, is the largest index such that s_j overlaps with s_{i+1} in t; And so on. In general if s_k is the last element in G_l, then G_{l+1} contains $s_{k+1}, ..., s_p$ where p, $k + 1 \leq p \leq m$, is the largest index such that s_{k+1} overlaps with s_p in t. See Figure 1.

Assume that there are g groups: $G_1, ..., G_g$. For G_i let b_i and t_i be the first (bottom) and last (top) string in G_i, according to our ordering, respectively. That is, b_i and t_i sandwich the rest of strings in G_i and for some optimal m_i, every string in G_i is a substring of $m_i(b_i, t_i)$.

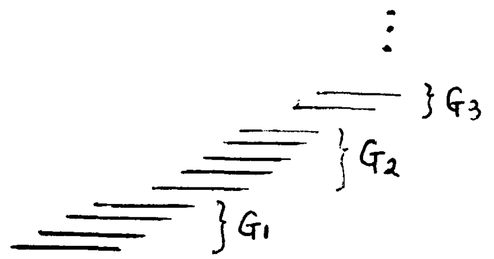

Figure 1: Grouping of Substrings

Lemma 2 *$\Sigma_{i=1}^{g}|m_i(b_i,t_i)| \leq 2n$, where n is the length of shortest superstring t for S.*

Proof: This can easily be seen geometrically: put all the groups $G_1, ..., G_g$ back to their original positions in the optimal arrangement (which gives the shortest superstring t). Then strings in G_i overlap with nothing except strings in G_{i-1} and G_{i+1} (and of course in G_i itself), for $i = 2, ..., g-1$. Thus counting the optimal solution for each G_i separately at most doubles the length of the optimal solution. □

Lemma 3 *For each G_i, after deleting an arbitrary number of strings from G_i, among the strings left, we can still merge a pair of strings in G_i so that all other strings are substrings of this merge. Furthermore the resulting merge is a substring of $m_i(b_i,t_i)$.*

Proof: Consider the original optimal arrangement and ordering of strings $s_1, ..., s_m$ in t. All strings in G_i overlap at a single position. So after deleting some strings, we can still properly merge the first (bottom) and last (top) strings among the leftover strings in G_i to obtain a superstring such that all the leftover strings in G_i are substrings of this superstring. Obviously this superstring is also a substring of $m_i(b_i,t_i)$. □

For a set of strings A, let $||A|| = \sum_{a\in A}|a|$. Let G_i^r be the set of strings remaining in G_i before the rth iteration. Let S_r be the set of strings cancelled at rth iteration of **Group-Merge**. Let b_r, t_r be the first and last strings in S_r, according to our ordering, respectively. And let m_r be the merge function used in the rth iteration to combine b_r and t_r. Let there be a total of k iterations executed by **Group-Merge**. Now the length L_j we used to merge strings in G_j can be measured as follows:

$$L_j \leq \sum_{r=1}^{k} \frac{|m_r(b_r,t_r)|}{||S_r||}||G_j^r - G_j^{r+1}||$$

$$\leq \sum_{r=1}^{k'} \frac{|m_j(b_j,t_j)|}{||G_j^r||}||G_j^r - G_j^{r+1}||$$

(where $k' \leq k$ indicates the first step $G_j^{k'+1}$ becomes empty)

$$= |m_j(b_j,t_j)|\sum_{r=1}^{k'} \frac{1}{||G_j^r||}||G_j^r - G_j^{r+1}||$$

$$\leq |m_j(b_j,t_j)|H(||G_j||)$$

(where $H(m) = \sum_{i=1}^{m}\frac{1}{i} = \Theta(\log m)$)

$$= |m_j(b_j,t_j)|\Theta(\log||G_j||).$$

Hence the total length we use to merge all $G_1, ..., G_g$ is, by Lemma 2,

$$\sum_{i=1}^{g} L_i \leq \Theta(\log l)\sum_{i=1}^{g}|m_i(b_i,t_i)|$$

$$\leq \Theta(\log l)2n = \Theta(\log l)n,$$

where $l = \max_i ||G_i||$. But $O(\log n) = O(\log l)$ since n is (polynomially) larger than the number of strings in any G_i and (polynomially) larger than the length of longest string in any G_i. Therefore the algorithm will output a superstring of length at most $O(n\log n)$. □

Remark. We conjecture that our algorithm always outputs a superstring of length no more than $2n$ (and possibly even less) where n is the optimal length. Vijay Vazirani constructed an example for which our algorithm outputs a superstring of length $1.3n$. Samir Khuller suggested that we can replace the weight function by just counting the number of strings "sandwiched" by the two merging strings. Although this algorithm still guarantees the $n\log n$ performance, but a counter example shows that the output sometimes exceeds $2n$ in length.

3 Modeling DNA Sequencing Problem

The task of this section is to properly put our DNA sequencing problem into a suitable mathematical framework. In the following, we safely assume that the expected sampling length l is much larger than $\log n$ where n is the length of the DNA sequence[2].

It may also be reasonable to assume that in order to sequence a human genome of 3×10^9 long, at about 500 characters per fragment, sequencing process must be fully automated without human intervention. We are interested in fast and *simple* sequencing methods that can be easily automated. Another safe assumption we will make is that sampling will be uniformly random. This is more or less the reality or achievable. We consider two models.

3.1 Recovering It Precisely

If we can assume that a DNA sequence s is random and substrings from S_l are uniformly sampled, where S_l contains s's substrings of length l and substrings of length less than l that are prefixes or suffixes of s, then s can be identified with *no error* with high probability.

Just to simplify calculation, we consider a fixed l, rather than for all l (less than 500). This should not change the result. We provide a rough calculation. Let $K(s) \geq |s| = n$, where $K(s)$ is the Kolmogorov complexity of s [3]. Divide s into $\frac{n}{l/2} = \frac{2n}{l}$ consecutive blocks. Let E_i denote the event where the ith block is not fully covered by any sample. Let $E = \cup E_i$. At each random sampling, $Pr(E_i) \leq \frac{n-l/2}{n}$. Then after $O(\frac{n \log n}{l})$ samples, $Pr(E) \leq n^{-O(1)}$. So all positions of s are covered with probability $1 - n^{-O(1)}$. By the same calculation, the probability of existing some substring which does not overlap for more than $l/2$ characters with some other substring in the sample is also less than $n^{-O(1)}$. Since s is random, there is no repetitions in s of length greater than $l/2$ if $l > 4 \log n$ [18], we thus can precisely recover s with probability $1 - n^{-O(1)}$, after $O(\frac{n \log n}{l})$ samples.

Of course, assuming s being random is dangerous since it is known that human DNA molecules are not very random and actually contain many (pretty long) repetitions (and redundancies). Above calculation may be meaningful only for lower lifes when the repetitions mostly are less than $l/2$.

3.2 Learning It Approximately

An alternative approach, which we will take in the rest of this paper, is to make *no assumptions* about the sequence s, although we will lose some power of our theory. We appeal to Valiant learning model. Our learned DNA sequence will only be good with high probability for short queries about several hundred characters long. Although this only partially captures reality, we are able to fully characterize the world we have captured. This is still meaningful since certain biological functions are encoded by just a few hundred base pairs. One purpose of DNA sequencing is to find the functionality of the genes, there is really no need to insist on recovering the original sequence precisely, especially when this is impossible.

We first describe the distribution independent model of learning introduced by Valiant [31]. More discussion and justification of this model can be found in [2, 12, 14, 32, 33]. [15] has been a useful source for me. We assume that the learning algorithm of A has available a black box called EXAMPLES(A), with two buttons labeled POS and NEG. If POS (NEG) is pushed, a positive (negative) example is generated according to some fixed but unknown probability distribution D^+ (D^-). We assume nothing about the distributions D^+ and D^-, except that $\sum_{s \in POS} D^+(s) =$

[2] By current technology l is about 500, whereas $n \leq 3 \times 10^9$ to be interesting.

[3] We refer the reader to [18] for an introduction of Kolmogorov complexity. Here we give an intuitive definition: Kolmogorov complexity of a binary string x conditional to y, written as $K(x|y)$, is the length of the shortest Pascal program, encoded in binary in a standard way, that starts with input y, prints x and halts. $K(x) = K(x|\epsilon)$.

1 and $\sum_{s\in NEG} D^-(s) = 1$ (i.e., $\sum_{s\in POS} D^-(s) = 0$ and $\sum_{s\in NEG} D^+(s) = 0$). For discrete domains, the Valiant learnability can be defined as follows.

Definition 1 *Let C and C' be concept classes. C is polynomially learnable from examples (by C') if there is an (randomized) algorithm A with access to POS and NEG which, taking inputs $0 < \epsilon, \delta < 1$, for any $c \in C$ and D^+, D^-, halts in polynomial($size(c), \frac{1}{\delta}, \frac{1}{\epsilon}$) time and outputs a concept $c' \in C'$ that with probability greater than $1 - \delta$ satisfies*

$$\sum_{c'(s)=0} D^+(s) < \epsilon$$

and

$$\sum_{c'(s)=1} D^-(s) < \epsilon.$$

We say that A is a learning algorithm of C.

In the following definition, we model (a subset of) DNA sequencing problem as the *string learning problem* under the Valiant learning. Notice that the Valiant model allows sampling under arbitrary distribution, whereas in our case uniform sampling is sufficient. Therefore we are really studying a more general problem of learning a string.

Definition 2 *String Learning Problem. The concept class C is the set of strings (DNA-molecules to be sequenced) over the 4 letter alphabet $\{A, C, G, T\}$. The positive examples for each concept (i.e. string) c of length n are its substrings of length no more than n; The negative examples*[4] *are strings that are not substrings of c. Sampling is done at random according to some unknown distributions for positive and negative examples, respectively.*

We strengthen a well-known result of Blumer, Ehrenfeucht, Haussler, and Warmuth [2, 3] in the following theorem. We have replaced their requirement that $size(c') \le n^\alpha m^\beta$ by that of $K(c'|c) \le n^\alpha m^\beta$. The same proof of [3] still works.

[4]In DNA sequencing practice, there do appear to be negative examples, due to biological restrictions on what cannot be combined.

Theorem 4 *Let C and C' be concept classes. Let $c \in C$ with $size(c) = n$. For $\alpha \ge 1$ and $0 \le \beta < 1$, and let A be an algorithm that on input of $m/2$ positive examples of c drawn from D^+ and $m/2$ negative examples of c drawn from D^-, outputs a hypothesis $c' \in C'$ that is consistent with the examples and satisfies $K(c'|c) \le n^\alpha m^\beta$, where $K(c'|c)$ is the Kolmogorov complexity of c' conditional to c. Then A is a learning algorithm for C by C' for*

$$m = O(max(\frac{1}{\epsilon}\log\frac{1}{\delta}, (\frac{n^\alpha}{\epsilon})^{\frac{1}{1-\beta}})).$$

When $\beta = 0$, and $n > \log\frac{1}{\delta}$, we will use $m = O(\frac{n^\alpha}{\epsilon})$.

By the above theorem, we certainly can trivially learn, with error probability $1/n$, a string of length n by sampling $m = n^3$ examples: the output will be a set of substrings of total length at most n^2 since there are only n^2 substrings and we can merge them into only n substrings. Needless to say, sampling 5000^3 fragments to sequence a DNA-molecule is simply not acceptable. A more careful analysis can bring this down to $O(n\log n/\epsilon)$ which is still not satisfactory.[5] A theory must agree with practice. We will show that we need only about $\frac{n\log n \log(n/l)}{l\epsilon}$ fragments, where l is the D^+-expected sampling length. This is much closer to the laboratory practice. Notice that the divisor l is around 500 and hence is much more significant than a $\log n$ term for $n \le 3\times 10^9$. We conjecture that this can be improved to $\frac{6n\log n}{l\epsilon}$.

This algorithm will depend on a good approximation algorithm, given in Section 2, for finding a shortest superstring which guarantees an output not much longer than the shortest superstring for a given set of strings (fragments).

[5]Let n be the length of the string to be learned. Then a trivial algorithm would sample substrings and keep them, merging a and b only if a is a substring of b, as the concept. Thus, since there are at most n^n such concepts (although the total length may be n^2), we need only $O(n\log n/\epsilon)$ examples to learn with error probability ϵ by Theorem 4. However, this is still not good enough. For example, we still do not want to sample say, 5000log5000, substrings to identify a virus DNA molecule of 5000 base pairs.

4 Learning a String Efficiently

Formally our concept class C is a set of strings. Positive examples are substrings of a target string t distributed according to D^+, which maybe uniform in the DNA applicaitons. Negative examples are strings that are not substrings of t distributed according to D^-. C' is a class of sets of strings.

Theorem 5 *C is learnable (by C'), with error probability ϵ, using only $O(\frac{n\log^2 n}{l\epsilon})$ samples, where l is the D^+-expected sample length.*

Proof: Given $O(\frac{n\log^2 n}{l\epsilon})$ positive and negative examples, if we output a concept c' such that $K(c'|c) \le O(\frac{n\log^2 n}{l})$, then by Theorem 4, we have a learning algorithm.

We first change **Group-Merge** algorithm to also deal with negative examples: At step (1), we now look for a pair of s, s' such that $cost(s, s')$ is minimized under the condition that $m(s, s')$ must not contain a negative example as a substring. The learned concept is

$$c' = \{m(s,s')|m(s,s')\ \textit{chosen in step}\ (1)$$

$$\textit{of}\ \textbf{Group}-\textbf{Merge}\}.$$

So strings in c' may contain more than one string.

In order to show that the old analysis is still good, we only need to observe one fact: there is always a way to properly combine the first and last strings in each group G_i^r at the rth step such that they contain all strings in G_i^r as substrings and no negative examples as substrings. Hence the analysis of Theorem 1 still carries through.

Now we count how many c' are possible outputs of **Group-Merge**, given the fact that we draw examples using c. c' is constructed by less than $\frac{2n\log n}{l}$ iterations since the total length of c' is $n\log n$ and l is the D^+-expected length for a positive sample[6]. Each step, two possible substrings of c are combined in some way, and this has at most $n^{O(1)}$ choices. So altogether we have $n^{O(\frac{n\log n}{l})}$ potential c'. Therefore

$$K(c'|c) \le O(\frac{n\log^2 n}{l}).$$

Given a new string, we can decide whether it is a positive or negative example by testing whether it is a substring of some string in c'. By Theorem 4, our error probability is at most ϵ. □

Corollary 6 *If we have an algorithm approximating the shortest common superstring of length n within length $2n$, then C is learnable, with error probability ϵ, using only $\frac{6n\log n}{l\epsilon}$ samples, where l is the D^+-expected sample length.*

In the real life situation of DNA sequencing, there are many restrictions on what cannot be combined. These conditions may be regarded as negative examples. However, if one prefers to think that *no negative examples are given*, we can still reasonably assume that negative instances are more or less uniformly distributed. And then,

Corollary 7 *Assume a uniform distribution over the negative examples. C is learnable (by C'), with error probability ϵ, using only $O(\frac{n\log^2 n}{l\epsilon})$ positive examples.*

Proof: By modifying the arguments of Theorem 4 (proof omitted) it is easily seen that our algorithm **Group-Merge** will still guarantee

$$\sum_{c'(s)=0} D^+(s) \le \epsilon,$$

where $c'(s) = 0$ stands for s is not a substring of some string in the learned concept c'. On the negative side, because the output length is at most $O(n\log n)$, this will make at most $O((n\log n)^2)$ negative examples positive. Since we have assumed the uniform distribution, the error probability on the negative side is only, for not too small ϵ,

$$O(\frac{(n\log n)^2}{4^n}) < \epsilon.$$

[6]By statistics (Chernoff bounds), we know that with high probability, the average sample size l' for the sample we have drawn is close to l. In our case, the phrase "with high probability" can be absorbed in the error probability δ and the discrepancy between l' and l can be absorbed by some small constant.

□

Remark 1. The algorithms of Turner [30] and Tarhio and Ukkonen [29] do not seem to work when there are negative samples. This is because their algorithms merge a pair of strings each step; If merged wrongly at first, the negative examples could prohibit further mergings, hence ending up with a long superstring. On the other hand, although it was not our original purpose, algorithm **Group-Merge** adapts to this situation perfectly.

Remark 2. Our approach also fits the situation of inaccurate data. Such inaccurate data happen often during the Sanger procedure [27] and often obstruct us from obtaining the real underlying sequence. By allowing approximate matching, straightforward extension of our approach, combining ideas in [1, 13], would provide efficient and robust algorithms.

Remark 3. It seems that our new algorithm deals with the repetitions better than maximum overlap algorithm used by Turner, Tarhio, Ukkonen. It is believed that both algorithms have the 2 times optimal performance. Based on this conjecture, the Corollary 6 says that we need only $\frac{6n\log n}{l\epsilon}$ samples in order to achieve $(1-\epsilon)\times 100$ percent confidence. Several major open problems remain: Improve our $O(n\log n)$ bound (to $2n$). Prove the $2n$ bound or any non-trivial bound for the maximum overlap algorithm.

Remark 4. We have already indicated the limitation of our approach. In our model, we have assumed that future queries to the learned DNA sequence are of size of several hundred characters. Whether one can correctly answer, with high probability, queries about longer substrings of the learned DNA sequence is not clear. This should be an interesting subject of future research.

Remark 5. It is hoped that our theory and methods can be applied at a massive scale when the sampling and sequencing process are fully automated. The procedure we have described requires no human intervention. It only involves randomly sequencing short fragments and computing the shortest common superstring. Our theory guarantees the performance and we have bounded the number of samples required. It is foreseeable that this method may be useful in situations where we are interested only in finding out whether certain (not too long) character sequence (representing a function) appear in a given DNA molecule. If the future technology (like the PCR method) allows us to sample longer fragments, our method will be more useful since then we can guarantee to answer longer sequences with high confidence.

5 Acknowledgements

I gratefully thank Qiang Yang, Derick Wood, and Esko Ukkonen for bringing the literature on shortest common superstrings to my attention; Murray Sherk, John Tromp, and Paul Vitanyi for commenting on the manuscript; Deming Li, Ed Shipwash, Xiaohong Sun, and Baozhang Tang for teaching me biochemistry; David Haussler (who suggested me to calculate the probability of recovering the superstring precisely), Udi Manber, and a referee (COLT) for crucial criticisms and suggestions; Samir Khuller and Vijay Vazirani for enlightening discussions on the superstring algorithms, and NSERC for its financial support.

References

[1] D. Angluin and P.D. Laird. Learning From Noisy Examples. *Machine Learning 2(4) 343-370 1988.*

[2] A. Blumer, A. Ehrenfeucht, D. Haussler, and M. Warmuth. Learnability and Vapnik-Chervonenkis Dimension. *Journal of the ACM 35(4) 1989.*

[3] A. Blumer, A. Ehrenfeucht, D. Haussler, and M. Warmuth. Occam's Razor. *Information Processing Letters* 24 1987.

[4] D. Freifelder. Molecular Biology. *Jones & Bartlett Publishers, Inc. 1983.*

[5] P. Freidland and L. Kedes. Discovering the secrets of DNA. *C. ACM 28(11) 1164-1186 1985.*

[6] J. Gallant, D. Maier, J. Storer. On finding minimal length superstring. *Journal of Computer and System Sciences 20 50-58 1980.*

[7] M. Garey and D. Johnson. Computers and Intractability. *Freeman, New York, 1979.*

[8] D. Haussler. Generalizing the PAC model: sample size bounds from metric dimension-based uniform convergence results. *30th IEEE Symp. on Foundation of Compt. Sci. 40-45 1989.*

[9] D. Haussler. Quantifying inductive bias: AI learning algorithms and Valiant's model. *Artificial Intelligence, 36(2) 177-221 1988.*

[10] D. Haussler, N. Littlestone, and M. Warmuth. Expected Mistake Bounds for On-Line Learning Algorithms. *29th IEEE Symp. on Foundation of Compt. Sci. 100-109 1989.*

[11] D. Helmbold, R. Sloan, and M. Warmuth. Learning nested differences of intersection-closed classes. *2nd Workshop on Computational Learning Theory, 41-56 1989.*

[12] M. Kearns, M. Li, L. Pitt, and L. G. Valiant. Recent Results on Boolean Concept Learning. *Proceedings of the 4th International Workshop on Machine Learning, 337-352, 1987*

[13] M. Kearns and M. Li. Learning in the Presence of Malicious Errors. *1988 ACM Symp. on Theory of Computing. Also Harvard TR-03-87.*

[14] M. Kearns, M. Li, L. Pitt, and L.G. Valiant. On the Learnability of Boolean Formulae. 19th *ACM Symposium on Theory of Computing* 285-295 1987.

[15] M. Kearns. The computational complexity of machine learning. *Ph.D. Thesis, Harvard University. Report TR-13-89, 1989.*

[16] A. Lesk (Edited). Computational Molecular Biology, Sources and Methods for Sequence Analysis. *Oxford University Press, 1988.*

[17] M. Li and P. Vitanyi. A theory of learning simple concepts under simple distributions. *30th IEEE Symp. on Foundations of Computer Science, 34-39, 1989.*

[18] M. Li and P. Vitanyi. Kolmogorov complexity and its applications. *Handbook of Theoretical Comput. Sci., J. van Leeuwen Ed. 1990*

[19] N. Littlestone. Learning Quickly When Irrelevant Attributes Abound: A New Linear Threshold Algorithm. *28th IEEE Symposium on the Foundations of Computer Science, 1987.*

[20] W. Maass and G. Turan. On the complexity of learning from counterexamples. *30th IEEE Foundations of Computer Science, 262-267 1989.*

[21] R. Michalski, J. Carbonell, T. Mitchell. *Machine Learning.* Morgan Kaufmann 1983.

[22] B.K. Natarajan. On Learning Boolean Functions. *ACM Symp. on Theory of Computing, 296-304 1987.*

[23] H. Peltola, H. Soderlund, J. Tarhio, and E. Ukkonen. Algorithms for some string matching problems arising in molecular genetics. *Information Processing 83 (Proc. IFIP Congress, 1983) 53-64.*

[24] L. Pitt and L.G. Valiant. Computational Limitations on Learning From Examples. *Journal of the ACM, 35(4) 965-984 1988.*

[25] L. Pitt and M. Warmuth. The minimum consistent DFA problem cannot be approximated within any polynomial. *ACM Symp. on Theory of Computing, 421-432 1989.*

[26] R. Rivest. Learning Decision-Lists. *Machine Learning, 2(3) 229-246 1987.*

[27] R. Staden. Automation of the computer handling of gel reading data produced by the shotgun method of DNA sequencing. *Nucleic Acids Research, 10(15) 4731-4751 1982.*

[28] J. Storer. *Data compression: methods and theory. Computer Science Press, 1988.*

[29] J. Tarhio and E. Ukkonen. A Greedy approximation algorithm for constructing shortest common superstrings. *Theoretical Computer Science 57 131-145 1988*

[30] J. Turner. Approximation algorithms for the shortest common superstring problem. *Information and Computation 83 1-20 1989.*

[31] L. G. Valiant. A Theory of the Learnable. *Comm. ACM* 27(11) 1134-1142 1984.

[32] L. G. Valiant. Learning Disjunctions of Conjunctions. In *Proceedings of the* 9th *IJCAI,* vol. 1 560-566 Los Angeles, CA. August, 1985.

[33] L. G. Valiant. Deductive Learning. *Phil. Trans. R. Soc. Lond.* A 312 441-446 1984.