DNA Sequencing and String Learning *

Tao Jiang†
McMaster University

Ming Li‡
University of Waterloo

Abstract

In laboratories, the majority of large-scale DNA sequencing is done following the *shot-gun* strategy, which is to randomly sequence large amount of relatively short fragments and then heuristically find a shortest common superstring of the fragments [26].

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 string from its randomly drawn substrings. Under certain restrictions, this may be viewed as string learning in Valiant's distribution-free learning model and in this case we give an efficient learning algorithm and a quantitative bound on how many examples suffice.

One major obstacle to our approach turns out to be a quite well-known open question on how to approximate a shortest common superstring of a set of strings, raised by a number of authors in the last ten years [9, 29, 30]. We give the first *provably* good algorithm which approximates a shortest superstring of length n by a superstring of length $O(n \log n)$. The algorithm works equally well even in the presence of negative examples, *i.e.*, when merging of some strings is prohibited.

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

---

*Some of the results presented in this paper appeared in the *Proceedings of the 31st IEEE Symposium on Foundations of Computer Science*, 125-134, 1990 [21].

†Supported in part by NSERC Operating Grant OGP0046613. Address: Department of Computer Science, McMaster University, Hamilton, Ont. L8S 4K1, Canada. E-mail: jiang@maccs.mcmaster.ca

‡Supported in part by the NSERC Operating Grants OGP0036747 and OGP0046506. Address: Department of Computer Science, University of Waterloo, Waterloo, Ont. N2L 3G1, Canada. E-mail: mli@watmath.waterloo.edu

1 Introduction

The fundamental quest to Life is to understand how it functions. Since the discovery of the structure 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. The entire living organism is encoded in the nucleic acid DNA. Determining this string for different molecules, or *sequencing* the molecules, is a crucial step towards understanding the biological functions of the molecules. In fact, today no problem in biochemistry can be studied in isolation from its genetic background. Huge (in terms of money and man years) national projects of sequencing human genome are proposed or underway although, unfortunately, major technical problems remain unresolved.

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) directly as a whole. Each time, a rather 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 first break a very long molecule to reasonably shorter ones (of 50,000 base pairs say). Then they "cut" millions of such (identical) molecules into pieces each typically containing about 200-500 nucleotides (characters or base pairs) [7, 20]. For example, a biochemist can decide to cut the molecules whenever the substring GGACTT (via restriction enzymes) appears. One may apply many different cuttings using different substrings (enzymes). 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, for example, 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. 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 a large amount (perhaps hundreds) of these random fragments, a biochemist has to *assemble* the superstring representing the whole molecule. This is usually referred to as the *shot-*

gun method. Programs, that try to approximate a shortest common superstring of a given set of strings (fragments) are routinely used [24, 20]. However, it is not known whether this always works or why it works. It is reported in [24, 27, 20] that programs based on shortest common superstring algorithms [1] work very satisfactorily for real biological data. In [29], it is observed 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 a shortest common superstring (finding it is NP-complete)? This has been an open problem raised by Gallant, Maier and Storer [9], 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) Given a good approximation algorithm for the shortest superstrings, how do we infer DNA sequences 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's distribution-free learning model. We will also give quantitative bounds on the number of random fragments that need to be sampled.

This paper studies Valiant's 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), super-polynomial [31, 3, 25, 12], although efficient sampling was studied for example in [11]. On the other hand, our problem has trivial algorithms that need high polynomial number of examples, but it also has non-trivial algorithms requiring low polynomial number of examples.

It is worth noting that artificial intelligence methods [23] and string matching algorithms have been extensively applied to DNA sequence analysis [8].

The paper is organized as follows: In the next section, we provide a solution to question (a).

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

Our new approximation algorithm is also crucial in the following sections. We also introduce a new concept: k-approximate common superstrings. This concept can be very useful in dealing with inaccurate data. In Section 3, we study possible ways of modeling the DNA sequencing problem. We will relate this problem to the Valiant's learning model. We also state a stronger version of the Occam Razor theorem proved in [4], which is needed in Section 4. Then in Section 4 we give provably efficient learning algorithms for the DNA sequencing problem. We also present a model for DNA sequencing in the presence of inaccurate samples. We discuss the limitation of our approach and open problems in section 5.

2 Approximating a 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 approximation algorithm for the 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 (*i.e.*, given S and an integer k, is the shortest common superstring of S less than k?) is known to be NP-complete [9, 10]. Due to its wide range of applications in many areas such as data compression and molecular biology [9, 29, 30], finding good approximation algorithms for the shortest common superstring problem has become an important subject of research. It was an open question to find a provably good approximation algorithm [9, 29, 30] for this problem. We provide the first solution to this problem. 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 remains 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, put the result back in S, and delete the two original strings. 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.

A Historical Note. Half a year after our $O(n \log n)$ solution, Blum *et al.* [2] obtained $O(n)$ approximation bounds for several greedy algorithms. However, in our proposed DNA sequencing model we will need to consider a more general version of the superstring problem, *i.e.*, we are also given some *forbidden strings* and the superstring is required not to contain any forbidden string as a substring. (See Section 4.1.) It will be shown in Section 4.1 that the algorithm Group-Merge to be presented below can handle the forbidden strings easily. So far *only* Group-Merge can achieve $O(n \log n)$ approximation for this generalized problem. It is still unclear if an $O(n)$ approximation algorithm may exist. In a separate paper [15], we have shown that the greedy algorithms do not achieve $O(n)$ and in fact, the forbidden strings may cause a greedy algorithm to produce a superstring of length $\Theta(n^{1.5})$. Thus, the results in [2] *do not* supersede the approximation algorithm in this paper.

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 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 that 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 subset 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,

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

```
                                         ...
                                   *************
                                      ...
                               *************
                            *************                (G3)
                     *************
                        ...
                 *************
              *************                              (G2)
       *************
          ...
   *************
*************                                            (G1)
```

Figure 1: Grouping of Strings.

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 a shortest common superstring t. 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 $m_i(b_i, t_i)$.

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,

$$\sum_{i=1}^{g} L_i \le \Theta(\log l) \sum_{i=1}^{g} |m_i(b_i, t_i)|$$
$$\le \Theta(\log l) 2n = \Theta(\log l) n,$$

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

Remark 1. Samir Khuller suggested that we can replace the weight function by just counting the number of strings "sandwiched" by the two merging strings. This still guarantees the $n \log n$ upper bound. Since we can show that **Group-Merge** may actually produce a superstring of length $\Omega(n \log n)$, this simplification will not worsen the worst case performance of the algorithm. It is an interesting question how this will affect the algorithm's average performance. The last part of our analysis uses ideas from [5] for the set cover problem.

Remark 2. Recently we have learned that the $O(n \log n)$ bound can also be obtained using an $O(n \log n)$ approximation method for the traveling salesman's problem on directed graphs with triangle inequality. However, that method again cannot handle forbidden strings and thus *does not* work for the generalized superstring problem which is required in our DNA sequencing model.

2.3 A Lower Bound for Group-Merge

We show that **Group-Merge** may actually produce a superstring of length $\Omega(n \log n)$. For simplicity, assume $\Sigma = \{a, b, c\}$. The construction can be easily modified to work for a binary alphabet. Let $m = 4^k$ be an arbitrary power of 4 and the set S of given strings be $\bigcup_{i=0}^{k-1} S_i \cup V$, where each S_i and V are given below:

$$S_i = \{a^j b^{m-j} \mid m/2^{2i+1} \le j < m/2^{2i}\}$$

$$V = \{cb^{i-1}ca^{4^i - i - 1} \mid 1 \le i \le k\}$$

A simple calculation shows that the optimal superstring length n for S is $\frac{7m}{3} + O(\log^2 m)$. A possible arrangement of the strings in the superstring achieving $\frac{7m}{3} + O(k^2) = \frac{7m}{3} + O(\log^2 m)$ is illustrated in Figure 2.

Now consider the operations of **Group-Merge** on S. Observe that

$$cost(cb^{k-1}ca^{m-k-1}, a^{m/2}b^{m/2}) = \frac{3m/2 + k}{m(m/2+1)} = \frac{3}{m+2} + \frac{2k}{m(m+2)}$$

```
                                                 aabbbb            ...  bb
                                                aaabbb            ...   b      (Sk-1)
                                             aaaaaaaabb          ...   b
                                                                ...
                                        a     ...     ab     ... bb
                                       aa     ...     ab    ...  b             (S1)
                                   a       ...        ab...b
                                  aa       ...        ab..b
                                         ...
                                 a       ...          abb
                                aa       ...          ab                       (S0)
                           cb..bcaa     ...        a
                         ...
                  cbcaaaaa
              ccaa                                                        (V)
```

Figure 2: An optimal arrangement of the strings.

is the minimal cost overall pairs of strings. Thus, **Group-Merge** can merge $cb^{k-1}ca^{m-k-1}$ and $a^{m/2}b^{m/2}$ in the first step and obtain a string $cb^{k-1}ca^{m-1}b^{m/2}$. The new string covers all strings in S_0. Similarly, after the first merge,

$$cost(cb^{k-2}ca^{m/4-k}, a^{m/8}b^{7m/8}) = \frac{9m/8+k-1}{m(m/8+1)} = \frac{9}{m+8} + \frac{8(k-1)}{m(m+8)}$$

is the minimal cost and thus **Group-Merge** can merge $cb^{k-2}ca^{m/4-k}$ with $a^{m/8}b^{7m/8}$ to obtain $cb^{k-2}ca^{m/4-1}b^{7m/8}$, covering all strings in S_1, and so on. Since the new strings obtained are not mergable with each other, the final superstring produced by **Group-Merge** has length at least $km = \frac{m\log m}{2} = \Omega(n\log n)$.

The above example also proves that $\Theta(n\log n)$ is a tight bound for **Group-Merge 1**. To see this, we only need to observe that each new string will not be considered by **Group-Merge 1** in the subsequent merges because it is "too long".

2.4 Introducing Errors into a Superstring

Just like people may tolerate a small number errors in string matching, sometimes we may be satisfied with a superstring containing some errors. As in [19], here we have three types of errors when matching string x with y: (i) A character of x corresponds to a different character of y (*i.e.*, a *mismatch*); (ii) A character of x corresponds to "no character" in y; (iii) A character of y corresponds to "no character" in x. We consider the following *shortest k-approximate common superstring problem*: Given a finite set S of strings, find a shortest string w such that each string x in S matches some substring of w with at most k errors.

For a given set S of strings, an optimal k-approximate superstring can be significantly shorter than an optimal superstring, even for small k. *E.g.*, let $S = \{10^i1 \mid 1 \leq i \leq m\}$. Then 10^m is an optimal 1-approximate superstring for S. It is easy to see that the optimal superstring length for S is $(m+1)(m+2)/2$. Generally, for each k, there is a set S_k such that if n is the optimal superstring length for S_k and n' is the optimal k-approximate superstring length for S_k, then

$$n \approx \binom{n'}{k} n' = \frac{n'^2(n'-1)...(n'-k+1)}{k!}$$

Thus, shortest k-approximate superstrings can be useful in data compression. In this paper we are more interested in another application: DNA sequencing with inaccurate samples. For many reasons, the samples obtained using the Sanger or other procedures often contain errors [27]. The errors can be an incorrect character, missing character, or an extra character, as described above. With the presence of these errors, if we insist on finding a shortest common superstring containing no error, then the sequence (superstring) obtained can be much longer than the true underlying sequence and thus will not even be close to the true sequence. However, if we relax the requirement and allow the superstring to have some errors when "necessary", then we might have a better chance to obtain the true sequence (or something close to it). Of course, sometimes it is rather hard to decide during a sequencing process what errors are "necessary" (or "obvious") and what are not. Usually this is judged based on the technician's experience and auxiliary knowledge about the true sequence. Such "experience" and "knowledge" on what *cannot* be a subsequence may be regarded as negative examples. Since in reality irrelevant segments of a DNA sequence do not have much overlap, there is a reason to take those errors that can significantly increase the overlap (thus reduce the superstring length) as "necessary" errors. (This is actually what some technicians do in practice.) This coincides with finding a shortest k-approximate common superstring.

It is not obvious how the greedy algorithms in [30, 29] can be modified to work for the shortest k-approximate common superstring problem with good approximation guarantee. However, **Group-**

Merge can deal with errors easily. We only need generalize the definition of a merge of two strings to allow errors. Formally, for any strings s and s', a *k-approximate merge of s and s'* is any string x of length at most $|s|+|s'|$ such that s matches a prefix of x with at most k errors and s' matches a suffix of x with at most k errors. The modification to function $weight()$ is straightforward: Let S be a set, $s, s' \in S$, and $m(s, s')$ a k-approximate merge of s and s'. Then $weight(m(s, s')) = \sum_{a \in A} |a|$, where A is the largest subset of S such that $m(s, s')$ is a k-approximate superstring for A. Finally, in the definition of $cost(s, s')$, the minimum should be taken over all k-approximate merges of s and s'. To avoid accumulation of errors, our algorithm will only execute steps (0) - (3) of **Group-Merge**. Denote the modified **Group-Merge** by **k-Approximate Group-Merge**. Clearly the proof of Theorem 1 also works for **k-Approximate Group-Merge**. Thus we have

Theorem 4 *Given a set of strings S, if the optimal k-approximate superstring length is n, then algorithm* **k-Approximate Group-Merge** *produces a k-approximate superstring of length $O(n \log n)$.*

A straightforward implementation of **k-Approximate Group-Merge** requires $O(|S|^3 n^k)$ running time. This is because two strings of length $O(n)$ can have at most $O(n^k)$ different k-approximate merges. Obviously, the running time is not very practical, even for small k's. It is an interesting question how efficiently one can implement the algorithm. In particular, can the running time be made not exponential in k?

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 sample 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.

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

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 example. Let $E = \cup E_i$. At each random sampling, $Pr(E_i) \leq \frac{n-l/2}{n}$. Then after $O(\frac{n \log n}{l})$ examples, $Pr(E) \leq n^{-O(1)}$. 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 set of examples is also less than $n^{-O(1)}$. Since s is random, there is no repetition in s of length greater than $l/2$ if $l > 4 \log n$ [22], we thus can precisely recover s with probability $1 - n^{-O(1)}$, after taking $O(\frac{n \log n}{l})$ examples.

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 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 [3, 17, 32]. [18] has been a useful

[3] We refer the reader to [22] 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. $K(x) = K(x|\epsilon)$.

source for us. We assume that a learning algorithm A has available a black box called EXAMPLES, 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 $\sum_{s \in POS} D^+(s) = 1$ 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 $\mathcal{C}$ and $\mathcal{C}'$ be concept classes. $\mathcal{C}$ is polynomially learnable from examples by $\mathcal{C}'$ if there is an (randomized) algorithm A with access to POS and NEG which, taking inputs $0 < \epsilon, \delta < 1$, for any $c \in \mathcal{C}$ and D^+, D^-, halts in polynomial$(size(c), \frac{1}{\delta}, \frac{1}{\epsilon})$ time and outputs a hypothesis $c' \in \mathcal{C}'$ 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 $\mathcal{C}$.

In the above, $c'(s) = 1$ (or 0) means that s is a positive (or negative, resp.) example of concept 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 $\mathcal{C}$ 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 are its substrings; 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.*

[4] In a shot-gun sequencing practice, there do appear to be negative examples, due to biological restrictions on what cannot be combined. Also, negative examples are as crucial as positive examples in the emerging new sequencing technology - *sequencing by hybridization* (SBH) [6, 13, 26].

We strengthen a well-known result of Blumer *et al.* [3, 4] in the following theorem. We have replaced their requirement that $size(c') \leq n^\alpha m^\beta$ by that of $K(c'|c) \leq n^\alpha m^\beta$. The same proof in [4] still works. Ignoring the numeric details, the essence of the following theorem says that compressing sample data into a *simple* hypothesis implies learning.

Theorem 5 *Let $\mathcal{C}$ and $\mathcal{C}'$ be concept classes. Let $c \in \mathcal{C}$ with $size(c) = n$. For $\alpha \geq 1$ and $0 \leq \beta < 1$, 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 $c' \in \mathcal{C}'$ that is consistent with the examples and satisfies $K(c'|c) \leq n^\alpha m^\beta$, where $K(c'|c)$ is the Kolmogorov complexity of c' conditional to c. Then A is a learning algorithm for $\mathcal{C}$ by $\mathcal{C}'$ 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 sample 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 $n \leq 3 \times 10^9$. 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 5. However, this is still not good enough. For example, we still do not want to sample say, $5000 \log 5000$, substrings to identify a virus DNA molecule of 5000 base pairs.

4 Learning a String Efficiently

4.1 Learning with both Positive and Negative Examples

Formally our concept class $\mathcal{C}$ is the set of strings over $\{A, C, G, T\}$. For each target concept (*i.e.*, string) c, the positive examples are substrings of c distributed according to D^+, which maybe uniform in the DNA applications. Negative examples are strings that are not substrings of c distributed according to D^-. Denote the D^+-expected sample length by l. In the following, we assume that the positive samples have a very small deviation (*i.e.*, $\ll l$) in length. This is consistent with real DNA sequencing practice. Let $\mathcal{C}'$ be the class containing all finite sets of strings over $\{A, C, G, T\}$. For any concept $c' \in \mathcal{C}'$, a positive example of c' is a string that is a substring of some string in c' and a negative example is a string that is not a substring of any string in c'.

Theorem 6 *$\mathcal{C}$ is learnable by $\mathcal{C}'$, with error probability ϵ, using only $O(\frac{n \log^2 n}{l\epsilon})$ samples.*

Proof. Given $O(\frac{n \log^2 n}{l\epsilon})$ positive and negative examples, if we output a concept c' such that $K(c'|c) \leq O(\frac{n \log^2 n}{l})$, then by Theorem 5, 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

$$c' = \{m(s, s') | m(s, s') \textit{ chosen in step } (1) \textit{ of } \mathbf{Group - Merge}\}.$$

So 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 each positive example is roughly l long. 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

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

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 5, our error probability is at most ϵ. $\square$

Remark 1. The greedy algorithms of Tarhio and Ukkonen [29] and Turner [30] do not seem to work when there are negative examples. 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 is not our original purpose, **Group-Merge** adapts to this situation perfectly.

Remark 2. The learning algorithm outputs a hypothesis that is in concept class $\mathcal{C}'$ (instead of $\mathcal{C}$). This is necessary since it has recently been shown in [14] that $\mathcal{C}$ is not learnable by $\mathcal{C}$.

It is conjectured that shortest common superstrings (even in the presence of negative examples) can be approximated within a factor of 2. The following corollary gives a bound on the number of examples needed for learning a string, assuming that the conjecture is true.

Corollary 7 *If we have an algorithm approximating a shortest common superstring of length n within length $2n$, then $\mathcal{C}$ is learnable, with error probability ϵ, using only $\frac{6n\log n}{l\epsilon}$ samples.*

4.2 Learning with only Positive Examples

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 8 *Assume a uniform distribution over the negative examples. $\mathcal{C}$ is learnable by $\mathcal{C}'$, with error probability ϵ, using only $O(\frac{n\log^2 n}{l\epsilon})$ positive samples.*[6]

Proof. By modifying the arguments of Theorem 5 (proof omitted) it is easily seen that our algorithm **Group-Merge** will still guarantee

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

[6]By the recent results in [2], the number of samples can be reduced to $\frac{12n\log n}{l\epsilon}$ using the maximum overlap algorithm [29, 30], or to $\frac{9n\log n}{l\epsilon}$ using a new greedy algorithm given in [2].

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,

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

□

4.3 Coping with Inaccurate Samples

As mentioned earlier, samples obtained during a DNA sequencing procedure may contain noises, *i.e.*, wrongly identified characters. Usually the number of such incorrect characters in a single sample is bounded by a small constant, say k. Because of this inherent inaccuracy and some other technical reasons, in situations when a biochemist wants to test (*i.e.*, query) if an assembled DNA sequence contains some special fragment, k-approximate string matching (*i.e.*, string matching with at most k errors) may be used instead of exact string matching. One such example is clinical diagnosis using genetic kits. Here a probe (which can be a small fragment of some virus DNA molecule) is "matched" by some biochemical means (such as hybridization) against a patient's DNA molecule. If a "match" (actually, a homology) with possibly a small number of errors is found, then it is believed that the patient has been infected by the virus. Thus, since the samples may already contain incorrect characters, there does not appear to be a need to try to learn a string that contains the samples precisely. Maybe it makes more sense if we settle for a string that contains the samples approximately. Since future queries do not mind a few differences, such an "inaccurate" string can be used to answer queries involving fragments of reasonable sizes. This leads us to consider the following problem.

Definition 3 *k-Approximate String Learning Problem. Each concept is a string over the alphabet $\{A, C, G, T\}$. The positive examples of a concept c are strings that matches some substring of c with at most k errors; The rest of strings are the negative examples of c. Sampling is done at random according to some unknown distributions for positive and negative examples, respectively.*

Using algorithm **k-Approximate Group-Merge**, it follows from the proof of Theorem 6 that there is a learning algorithm for the k-approximate string learning problem which needs only $O(\frac{n \log^2 n}{l\epsilon})$ samples, where ϵ is the error probability and l is the D^+-expected sample length.

Let c' be a hypothesis produced by the algorithm after $O(\frac{n \log^2 n}{l\epsilon})$ samples, when learning a string c of length n. (Recall that c' is a set of strings.) Then for any given string s, we can decide if s is a positive example of c (*i.e.*, if s k-approximately matches a substring of c) by testing if s k-approximately matches a substring of some string in c'. By the definition of a learning algorithm, this "prediction" has an error probability at most ϵ if the string s was drawn according to the probability distribution of examples.

Remark. Another way of dealing with inaccurate data is to simply treat samples containing incorrect characters as malicious errors or classification errors [1, 16]. Although we think this approach deviates from the real practice, using techniques in [1, 16], it is straightforward to extend our algorithms to provide efficient and robust string learning algorithms. These algorithm perform well when the number of erroneous samples is small.

5 Concluding Remarks, Debatable Issues, and Open Questions

We have already indicated the serious limitation of our approach. In our model, we have assumed that future queries to the learned DNA sequence involve fragments of several hundred characters long. More precisely, the model requires that future queries are about fragments of length roughly l. (Thus we cannot answer queries involving shorter or longer fragments.) However, if we make the following (reasonable) special assumptions about positive examples, then it is possible to answer queries about shorter fragments: the positive examples involved in a process are distributed uniformly. Moreover, the future queries assume a positive example space different than in the sampling (learning) process. In the sampling process, the positive samples are all substrings (of the target concept) of length roughly l. In future queries, the positive examples are all substrings (of the target concept) of length l' or less, for some $l' \leq l$. (*I.e.*, substrings of length $\leq l'$ occur in future queries with equal probability.) The same result should hold if future queries are only concerned with fragments of length roughly l' instead. Note that, in the above no assumption is made about the negative examples (except that sampling and querying use the same negative example space, as in Valiant's model). 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.

One naive objection to this work might be that the biologists want "real" original DNA sequences, yet we do not construct the original sequences with our pac-learning model. For a trivial example, due to long repeated segments, the shortest superstring, even it is found, may *not* be the true DNA sequence. But this is precisely why pac-learning theory is needed since, theoretically speaking, precise reconstruction of an arbitrary original sequence is impossible if only short sampling is allowed and long repetitions exist. As we have proved, although the sequences we have learned may not be or even "look like" the original sequence, we do guarantee that we can answer the short queries correctly with high probability. And the short query may be a subsequence appear *anywhere* in the original sequence.

Another objection might be that in practice the biologists do not do sequencing at one shot: they break the genome into subsequences of approximately 50,000 base pair each, then sequence each of these. As we have argued in the beginning, our theory is aimed at massive automated sequencing. If indeed the length of the queries to be made about some long sequence is within the boundary of our theory, then our theory provides a more reasonable method, suitable for automation, than the traditional way which tries to reconstruct the original sequence slowly and unnecessarily. It is hoped that our theory and methods can be applied on 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 a shortest common superstring. Our theory guarantees the performance and we have bounded the number of samples required. This method may be useful in situations where we are interested only in finding out whether certain (not too long) character sequence (representing some biological 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.

As mentioned earlier, Blum *et al.* [2] have recently shown that the maximum overlapping greedy algorithm studied by [29] and [30] produces a superstring of length at most $4n$, where n is the optimal length. Although this improves on our $O(n \log n)$ bound when there are only positive example, the greedy algorithm can perform much worse than Group-Merge in the general case when negative examples are also present.

6 Acknowledgements

We gratefully thank Qiang Yang, Derick Wood, and Esko Ukkonen for bringing the literature on shortest common superstrings to our attention; Murray Sherk, John Tromp, and Paul Vitanyi for commenting on the manuscript; Jun Han, Deming Li, Dan Lu, Ning Shen, Ed Shipwash, Xiaohong Sun, and Baozhang Tang for teaching us biochemistry and DNA sequencing procedures; David Haussler (who suggested us to calculate the probability of recovering the superstring precisely), Richard Beigel, Mark Fulk, Udi Manber and other colleagues for their criticisms and suggestions; and Samir Khuller and Vijay Vazirani for enlightening discussions on the superstring algorithms.

References

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

[2] A. Blum, T. Jiang, M. Li, J. Tromp, and M. Yannakakis, Linear Approximation of Shortest Superstrings, to be presented at *23rd ACM Symp. on Theory of Computing*, New Orleans, USA, May 6-8, 1991.

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

[4] A. Blumer, A. Ehrenfeucht, D. Haussler, and M. Warmuth. Occam's Razor. *Information Processing Letters 24, 377-380, 1987.*

[5] V. Chvatal. A greedy heuristic for the set covering problem. *Math. Op. Res. 4(3) 1979.*

[6] R. Drmanac and C. Crkvenjakov. Sequencing by hybridization (SBH) with oligonucleotide probes as an integral appraoch for the analysis of complex genomes. *International Journal of Genomic Research* 1-1, 1992, 59-79.

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

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

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

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

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

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

[13] R. Karp, Mapping the genome: some combinatorial problems arising in molecular biology, *ACM STOC'93*, pp. 278-285, 1993.

[14] T. Jiang and M. Li, On the Complexity of Learning Strings and Sequences. *Theoretical Computer Science*, 119(1993), 363-371

[15] T. Jiang and M. Li, Approximating Shortest Superstrings with Constraints. To appear in *Theoretical Computer Science.*

[16] M. Kearns and M. Li. Learning in the Presence of Malicious Errors. *SIAM J. Comput.*, 22:4(1993), 807-837.

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

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

[19] G. Landau and U. Vishkin. Efficient String Matching in the Presence of Errors. *26th IEEE Symp. on Foundations of Comp. Sci., 126-136, 1985.*

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

[21] M. Li. Towards a DNA sequencing theory. *31st IEEE Symp. on Foundations of Comp. Sci.*, 125-134, 1990.

[22] M. Li and P. Vitanyi. *An introduction to Kolmogorov complexity and its applications.* Springer-Verlag, 1993.

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

[24] 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.*

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

[26] L. Smith. The future of DNA sequencing. *Science* 262, 530-532, 1993.

[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. Deductive Learning. *Phil. Trans. R. Soc. Lond. A 312 441-446 1984.*