Report Number: WUCS-88-18

1988-06-01

DNA Restriction Mapping from Random-Clone Data

Authors: Gwangsoo Rhee

This report addresses the problem of constructing DNA restriction maps from random-close data produced by cutting the whole DNA structure with restriction enzyme and measuring possibly overlapping segments. Our approach to DNA mapping is based on the overlapping segments that occur between adjacent clones. The shortest common superstring problem (SCS) and the shortest common matching string problem (SCMS) are discussed as abstract computational models of DNA mapping. Since these string problems are NP-complete, we need efficient approximation algorithms to avoid excessive computational complexity. Some greedy algorithms to SCMS are presented along with performance data obtained through simulation.

DNA RESTRICTION MAPPING FROM RANDOM-CLONE DATA

Gwangsoo Rhee

WUCS-88-18

June 1988

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
Saint Louis, MO 63130-4899

Abstract

This report address the problem of constructing DNA restriction maps from random-clone data produced by cutting the whole DNA structure with restriction enzyme and measuring possibly overlapping segments. Our approach to DNA mapping is based on the overlapping segments that occur between adjacent clones. The shortest common superstring problem (SCS) and the shortest common matching string problem (SCMS) are discussed as abstract computational models of DNA mapping. Since these string problems are NP-complete, we need efficient approximation algorithms to avoid excessive computational complexity. Some greedy algorithms to SCMS are presented along with performance data obtained through simulation.

This work was supported by NIH grant RR01380.

DNA Restriction Mapping from Random-clone Data

Gwangsoo Rhee

1. Introduction

There is no substance as important as DNA. Because it carries within its structure the genetic information that determines the structures of proteins, it is the prime molecule of life. If we succeed in unlocking the information within DNA, we shall have taken a giant step toward eventually understanding the many complex sets of interconnected chemical reactions that cause fertilized eggs to develop into highly complex multicellular organisms.

The building blocks of the DNA polymer are nucleotides, which in turn consist of a phosphate group, a sugar ring group and either a purine or a pyrimidine base group. The two possible purines are adenine (A) and guanine (G); the two possible pyrimidines are cytosine (C) and thymine(T). The backbone of the DNA strand is formed by covalent bonds connecting alternating sugars and phosphates. The patterns present on a DNA strand are then the sequential arrangements of A, C, G and T along the strand. DNA isolated from cells is found to be an antiparallel double-stranded helix, with the alignment of the two strands mediated through hydrogen bonding between a purine or a pyrimidine on one strand and a pyrimidine or a purine on the other. Furthermore, this base paring is quite specific. A is always paired with T, and G is always paired with C. Thus, the base sequence on one strand completely determines the base sequence on the other, complementary strand. These A–T or G–C pairs are called *basepairs*. Now we can view a DNA molecule as a chain of basepairs.

The *DNA sequencing problem* is to determine a complete sequence of basepairs for a given DNA chain. There have been some biochemical techniques allowing sequencing of 100–500 basepair fragments of DNA [1,2,3]. But, most DNA molecules are much larger than this. For example, the size of DNA from the yeast *Saccharomyces* has been estimated to be 1.5×10^4 *kilobases* (kb), and a human DNA is as large as 3×10^6 kb. Hence it is not possible to apply the existing direct DNA sequencing methods to a whole DNA molecule.

Since the discovery of site-specific restriction nucleases in 1970, DNA *restriction enzymes* (RE) have become very useful in DNA sequencing. Restriction enzymes are chemicals that cut DNA strands at sites where specific base sequences, called *recognition sequences,* appear. For instance, *Eco* RI cuts at GAATTC. Using *Eco* RI, it is possible in principle to determine all the places where GAATTC appears. Places where such sequences appear are called *RE sites.* The fragments generated when a specific DNA is cut by one or more restriction enzymes are called *restriction fragments,* and the *restriction mapping problem* is to construct a *restriction map,* a picture showing the position of restriction fragments in the original DNA. Uses of restriction maps include: (1) a restriction map tells all the locations in a DNA strand where a specific base sequence appears; (2) when a complete base sequence for a restriction fragment is known, the restriction map tells where the fragmentary sequence should be located in the DNA, eventually leading to the complete base

sequence of the DNA; (3) in combination with other laboratory analyses, these maps can show the physical arrangements of closely linked genes.

The first restriction map was constructed in 1971 by Daniel Nathans [4]. Nathans used the *Hin* dII enzyme to cut the circular DNA of SV40 into 11 restriction fragments. The order in which these 11 fragments occurred in the SV40 DNA could be deduced by studying the patterns of fragments produced as the digestion proceeded to completion. The first cut broke the circular molecule into a linear structure that was then cut into progressively smaller fragments. By following the pattern of production of, first, the overlapping intermediate-sized fragments and, from them, the fragments of the complete digest, Nathans produced a restriction map that located the sites on the circular viral DNA that are cut by *Hin* dII. Repeating the experiment with other enzymes produced a more detailed map with many different restriction sites.

In general, the most useful enzymes are the one that have rare recognition sequences and that therefore produce small numbers of fragments that can be easily separated from one another. It is also easy to follow the patterns of producing fragments when there are only a small number of restriction sites, which allows ordering of fragments without excessive computation. Many commercially available DNA restriction enzymes have recognition sequences of six or fewer basepairs, and a very few of them have recognition sequences longer than eight basepairs. If we consider a DNA strand as a random sequence of four bases, a specific recognition sequence of length six will appear once at every 4096 ($= 4^6$) bp long sequence on the average. Since the reverse of a recognition sequence also forms a recognition sequence, the average restriction fragment size is reduced to 2048 bp. Furthermore many restriction enzymes have multiple recognition sequences. For example, *Hin* dII has four different recognition sequences: GTCAAC, GTCGAC, GTTAAC, and GTTGAC. Therefore the average restriction fragment length produced by *Hin* dII is 512 bp. If we use the enzyme *Hin* dII to cut a human DNA of length 3×10^9 bp, we'll obtain about 6×10^6 restriction fragments, neither the separation nor the analysis of which seems to be tractable. An enzyme with a unique recognition sequence of length eight will cut the human DNA into about 9×10^4 fragments, which makes reconstruction a lot more manageable, although still not easy.

For more insight, let's look at the actual steps of the experiment used to construct a DNA restriction map [5].

step 1. Start with a sample of multiple source DNA molecules and a restriction enzyme that cuts DNA into segments.

step 2. Add the restriction enzyme and limit reaction so that only about one site in ℓ (say 10) is cut. Take a random sample of n (say 5000) of these coarse segments(*clones*). Such a coarse segment contains ℓ restriction fragments on the average.

step 3. Separate and replicate using gene splicing techniques. This yields n clones, each containing many copies of a particular coarse segment.

step 4. Add restriction enzyme to each clone, cutting at all RE sites. Measure length of resulting fine segments (restriction fragments) by electrophoresis.

step 5. List the fragment lengths for each clone. Note that neither the order of clones nor the order of fragments within a clone is known from previous steps.

step 6. Use the data to infer positions of initial RE sites. Algorithms for the last step are the subject of this thesis.

An example of the experimental process is shown in Figure 1, where each list of fragments is a clone containing those fragments. With a number of clones sufficient to provide redundant sampling

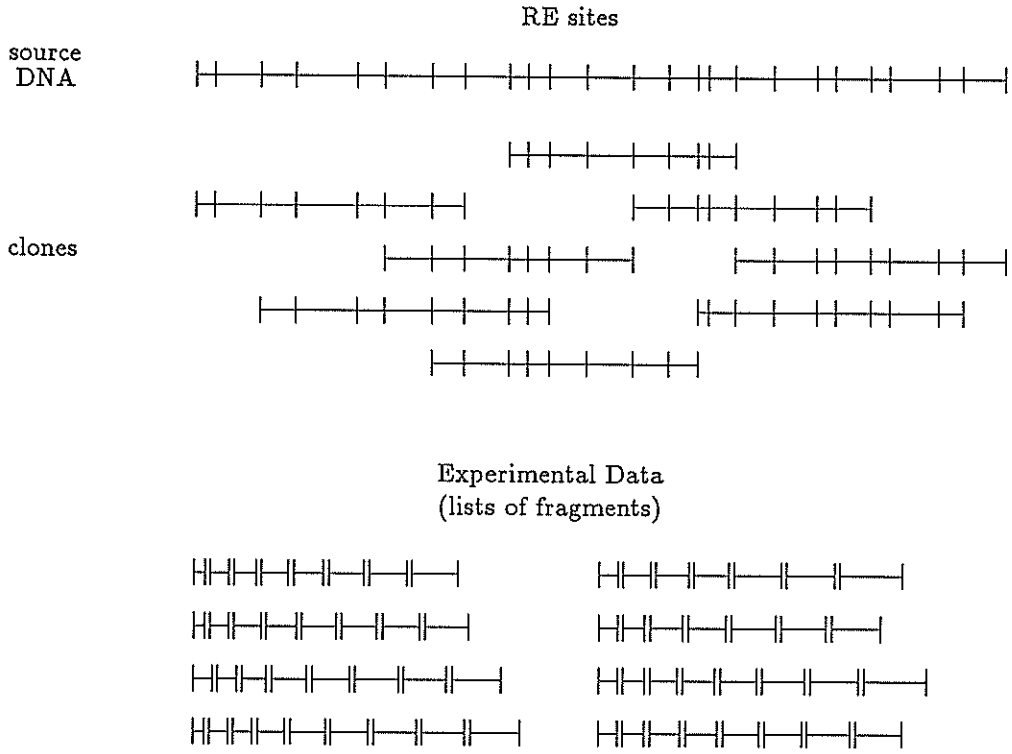

Figure 1: An example of experimental process

of most regions of the DNA, it is possible to extract mapping information from the unordered fragment-length lists using an algorithm that systematically imposes the requirement that the set of restriction fragments from a single clone should be contiguous.

Given a short DNA region and an enzyme cleaving the region at a few sites, it is easy to construct a restriction map. For example, consider a region that has only four recognition sites so that it can be broken into five fragments. Then the number of all the possible clones generated from that region is 15 ($= 1 + 2 + 3 + 4 + 5$). It won't be difficult to obtain all these clones, which provides sufficient information to sequence the fragments. In this case, it is also easy to doublecheck the fact that the given region has only four recognition sites, which was not known in advance. This checking can be done by comparing the length of the full region and the sum of the lengths of fragments. Since all the lengths from this experiment are subject to measurement error, this checking may not be done in the cases involving a large number of fragments. Moreover for large DNA molecules, obtaining all the possible restriction fragments is impractical.

The DNA restriction mapping problem can be abstracted to the shortest common matching problem (SCMS) and NP-completeness proofs of SCMS appear in [6] and also in [7]. The object of the problem is to find a minimum length string that matches every bag in the problem instance, where a bag is an unordered collection of symbols from a finite alphabet and the same symbol can appear more than once in a bag. A string is said to *match* a bag if the string contains some substring that has exactly the same symbols as the bag has. The DNA mapping problem is abstracted to

SCMS by viewing restriction fragment lengths as symbols and clones as bags. More details about SCMS are given in section 4.

There is a gap between the DNA mapping solution and the SCMS solution. The DNA mapping problem is to construct the original sequence of the restriction fragments from the experimental data, and SCMS, when applied to DNA mapping, is to find the shortest sequence that matches the data. Because of the NP-completeness of SCMS, there can be no efficient algorithm for SCMS unless P = NP. Even if we knew the shortest sequence, it isn't necessarily the original one, even if it is likely to be. The solution of SCMS, however, is useful in organizing the experimental data and planning experiments.

For regions spanning up to 50 kb, restriction maps are routinely constructed. In a few favorable cases, maps as long as 600 kb have been constructed at a resolution of a few kb [8,9,10], where fragments smaller than 0.5 kb are not visible by gel electrophoresis, but it has proven to be difficult to extend existing mapping methods beyond this range. Given that human DNA contains about 3×10^6 kb and that yeast DNA contains 1.5×10^4 kb, there is a gross disparity between the limits of those methods and the sequence complexity of DNA.

Direct DNA sequencing methods attempt to find complete basepair sequences of small DNA fragments and piece them together using restriction maps. The first substantial work on direct DNA sequencing methods was reported by Fred Sanger and A. R. Coulson in 1975; their method is now called the *plus-minus* method [1]. It is based on the elongation of DNA chains with DNA polymerase. With this technique the 5386-basepair sequence of the small DNA phage Φ X176 was determined [11]. An equally powerful method based on the chemical degradation that breaks a terminally labeled DNA molecule partially at each repetition of a base, was developed by Allan Maxam and Walter Gilbert in 1977 [2]. All the 5226 basepairs of SV40 DNA became quickly known in Gilbert's laboratory. Sanger later devised a third method for sequencing DNA, and again he used enzymatic rather than chemical techniques [3]. This method, which is based upon the incorporation of dideoxynucleotides as terminators of DNA chain elongation, has proved to be more efficient and accurate than any other method currently employed.

Another line of research on DNA sequencing, carried out by T. R. Gingeras [12], focuses on the overlaps between the fragmentary sequences to put fragments together so that a detailed restriction map may be avoided in the first place. This is accomplished by defining a region of DNA strand of manageable size (say, 1–5 kb) and then obtaining sequence information from that region in an arbitrary fashion. For instance, we can obtain a redundant set of fragmentary sequences, each 200–300 basepairs long, by cutting the region by many different restriction enzymes. The sequences are then searched for overlapping stretches and sequences combined until all data are accomodated within a final continuous string. The use of this method is restricted to assist in constructing long DNA sequences, or the results can be used for double-checking in combination with other methods. The *shortest common superstring problem* (SCS) is an abstract version of this method.

The object of SCS is to find the shortest possible string that contains every string in a given set as substrings. SCS has been given some attention for its application to data compression. The problem is shown to be NP-complete by Maier and Storer in [13]. In DNA sequencing applications, a fragmentary sequence can be equivalently represented by reversing it. The *reversible shortest common superstring problem* (RSCS) was originally introduced by Turner as an intermediate problem in the transformation from SCS to SCMS in the NP-completeness proof [7]. The object of the problem is very similar to that of SCS, except that each substring can be contained in the final superstring either in its original form or in the reversed form. The reversibility of RSCS makes it more suitable as an abstract version of Gingeras' method. Another use of RSCS algorithm is in DNA mapping if the order of restriction fragments within clones are determined by additional experiments on all the clones. But these additional experiments are too expensive with existing experimental

techniques unless we find a superior approximation algorithm for RSCS, the performance of which will compensate for the additional experiments.

2. Literature Review

In this section I am going to review some work on DNA sequencing and restriction mapping using computer algorithms.

Historically proteins and ribonucleic acids (RNA) got earlier attention for sequencing than DNA, because of their relative simplicity. A protein is a sequence of amino acids of which 20 kinds are known, and RNA is a sequence of purine and pyrimidine bases. The purine bases of RNA are the same as those of DNA, adenine and guanine, but the pyrimidine bases are cytosine and uracil (U). An algorithm for reconstructing protein and RNA sequences was suggested by Marvin B. Shapiro in [14]. This algorithm requires two kinds of input: (1) the number of occurrences of each base or each kind of amino acid, which also gives the overall length, and (2) the fragment data itself, that is, a set of fragmentary sequences with each fragment not longer than eight, where the constraint on the lengths are imposed by the data, not by the algorithm. Then the algorithm generates all the sequences meeting the conditions imposed by the data. The validity of this algorithm is shown only empirically, and its use is quite limited because it is hard to obtain the number of occurrences of each base in advance.

Mark Stefik conceived DNA mapping as an AI problem, and developed a rule-based program *GA1* using techniques similar to those used in the *DENDRAL* program [15]. His approach is basically the *generate-and-test* paradigm and uses the products of single and double digestion of DNA molecule with several different restriction enzymes. Initially there are exponentially many hypotheses about the possible sequences, and all the wrong answers are eliminated by examining the data, leaving a set (possibly singleton) of valid sequences. The combinatorial explosion in the number of hypothetical maps limits its use to very small DNA molecules. His real contribution to DNA mapping seems to be his efforts to deal with incorrect data such as missing segments, insufficient distinguishing power of fragments of similar sizes by the limited resolution of measurement, and extraneous fragments which are actually clones.

William R. Pearson made a slight improvement over GA1 [16]. His initial hypotheses about the possible maps are formed using the products of single digestion and the products of double digestion are used only to eliminate the wrong answers. The running time of this method is much faster than GA1, but is still exponential. Durand and Bregegere used *backtracking* to construct a map from one end to the other by adding a fragment of single digestion in accordance with double digestion data at each step [17]. The worst-case time complexity of this method is still exponential.

Maynard V. Olson presented a *random-clone strategy* for restriction mapping in his recent paper [19], and applied the method to construct a set of local maps each of which shows the structure of a segment of the global restriction map of total nuclear DNA from the yeast *Saccharomyces,* whose size had been estimated to be 1.5×10^4 kb. The data collection involves picking a redundant set of clones at random and measuring the sizes of the restriction fragments generated by a digestion of a clone with gel electrophoresis. With extensive uses of pairwise comparisons between clones, the clones with the longest overlap are added to each map unit if they do not conflict with the existing structure of the map. Frequent backtracking is required because the correct pairing scheme at any given step is not always the one with the most matches. The sizes of local maps are about 20–100 kb, and they are expected to serve as building blocks for the construction of a continuous global map in the future.

There hasn't been any serious work on the approximation algorithm for either SCS or SCMS, except the analysis of a greedy algorithm for SCS given by Turner in [18]. He showed that the

algorithm using the greedy heuristic of pairwise overlap between strings produces solutions that are always within a factor of two of optimum with respect to the overlap measure. He also described an implementation of the algorithm with worst-case running time $O(m \log n)$ for small alphabets and $O(m \log m)$ for large alphabets, where m is the sum of the lengths of all the strings in the set and n is the number of strings.

3. SCS and RSCS

This section provides formal definitions for the shortest common superstring problem (SCS) and reversible shortest common superstring problem (RSCS). The basic definitions and notation for SCS are taken from [18] and extended for RSCS.

Let $s_1 = a_1 \cdots a_p$ and $s_2 = b_1 \cdots b_q$ be strings over some finite alphabet Σ. We say that s_1 is a *substring* of s_2 if there is an integer $i \in [0, q-p]$ such that $a_j = b_{i+j}$ for $1 \leq j \leq p$. We also say in this case that s_2 is a *superstring* of s_1.

A set of strings is said to be *substring free* if no string in the set is a substring of any other. We will generally limit our attention to substring free sets without loss of generality, because any set of strings has a unique substring free subset which has the same solutions as the original set both in SCS and in RSCS.

If s is a string, $|s|$ denotes the number of symbols in s. If S is a set of strings, $|S|$ denotes the cardinality of S and $\| S \|$ denotes $\sum_{x \in S} | x |$.

3.1. SCS

An instance of SCS is a set of strings $S = (s_1, \ldots, s_n)$ over a finite alphabet Σ. The object of the problem is to find a minimum length string that is a superstring of every $s_i \in S$. We let $\phi^*(S)$ denote the length of a minimum length superstring.

When Σ consists of only one symbol, SCS has a trivial solution, the longest string from the given set. There is also a linear time and space algorithm to find the solution for a set of strings of length less than or equal to two [20]. In all the other cases, SCS remains NP-complete.

We have presented the problem in the conventional way, with the object being to minimize the solution length. It is useful to consider an alternative viewpoint as well. One can view the object of the problem as being to find an ordering of the strings that maximizes the amount of overlap between consecutive strings. To make this precise we need a few definitions.

Let $s_1 = a_1 \cdots a_p$ and $s_2 = b_1 \cdots b_q$ be strings. We define

$$\psi(s_1, s_2) = \max \{k \geq 0 \mid a_{p-k+i} = b_i, 1 \leq i \leq k\}.$$

If $\psi(s_1, s_2) = k$ then $s_1 \circ s_2$ is defined to be the string $a_1 \cdots a_p b_{k+1} \cdots b_q$. We note that if s_1, s_2, s_3 are strings, none of which is a substring of another, then $s_1 \circ (s_2 \circ s_3) = (s_1 \circ s_2) \circ s_3$; that is, the overlapping operation is associative for substring free sets. Consequently, in this case we may write $s_1 \circ s_2 \circ \cdots \circ s_n$ with no ambiguity.

Let π be a permutation on $\{1, \ldots, n\}$. We will usually write π_i for $\pi(i)$. We define

$$\psi_\pi(s_1, \ldots, s_n) = \sum_{i=1}^{n-1} \psi(s_{\pi_i}, s_{\pi_{i+1}})$$

and

$$\phi_\pi(s_1, \ldots, s_n) = | s_{\pi_1} \circ \cdots \circ s_{\pi_n} |.$$

Note that for any instance $S = (s_1, \ldots, s_n)$ of SCS,

$$\phi_\pi(S) = \| S \| - \psi_\pi(S).$$

In particular, we define

$$\phi^*(S) = \| S \| - \psi^*(S) \text{ where } \psi^*(S) = \max_\pi \psi_\pi(S).$$

Hence, we can view the object of the SCS problem as being to find a mapping π that maximizes ψ_π.

Let A be an algorithm for SCS which given a set of strings $S = \{s_1, \ldots, s_n\}$ produces a mapping $\pi = \pi_A(S)$. We define $\psi_A(S) = \psi_\pi(S)$ and $\phi_A(S) = \phi_\pi(S)$.

As SCS is NP-complete, we are interested in approximation algorithms. One particularly simple algorithm, called the greedy algorithm, can be stated as follows. Given a non-empty set of strings S, repeat the following step until S contains just one string.

> Select a pair of strings $s_1, s_2 \in S$ that maximizes $\psi(s_1, s_2)$. Remove s_1 and s_2 from S, replacing them with $s_1 \circ s_2$.

We refer to this algorithm as SGREEDY. It has been shown [18] that in the worst case, the sum of overlaps between consecutive strings is at least as large as half the sum of overlaps in the optimal solution ($\psi^*(S) \leq 2\psi_{\text{SGREEDY}}(S)$). The performance measure used here is the *overlap measure.* The worst-case performance by the conventional *length measure* of the greedy algorithm is not known except that it has been shown not to be better than a factor of two, twice the shortest length.

3.2. RSCS

Let $s = a_1 \cdots a_p$ be a string over some finite alphabet Σ. Then $rev(s)$ is defined to be the reverse of s, that is, $a_p \cdots a_1$. An instance of RSCS is same as that of SCS, a set of strings $S = \{s_1, \ldots, s_n\}$ over a finite alphabet Σ. The object in this case is to find a minimum length string that is a superstring either of s or of $rev(s)$ for all $s \in S$. We need some additional definitions for the discussion of RSCS.

Suppose a set of strings $S = \{s_1, \ldots, s_n\}$ for which $\{s_1, \ldots, s_n, rev(s_1), \ldots, rev(s_n)\}$ is substring-free, is given as an instance of RSCS. Let $\sigma : \{1, \ldots, n\} \rightarrow \{\pm 1, \ldots, \pm n\}$ be a mapping with restriction $\sigma(i) = i$ or $-i$ for every integer i. With a permutation π on $\{1, \ldots, n\}$, we can define

$$\psi_{\sigma\pi}(S) = \sum_{i=1}^{n-1} \psi(s_{\sigma\pi_i}, s_{\sigma\pi_{i+1}}) \text{ where } \sigma\pi_i = \sigma(\pi(i)) \text{ and } s_{-i} = rev(s_i).$$

Then the objective of RSCS can be viewed as being to find a composite mapping $\sigma\pi$ that maximizes $\psi_{\sigma\pi}(S)$ and let's call the maximum value $\tilde{\psi}^*(S)$.

Let A be an algorithm for RSCS which given a set of strings $S = \{s_1, \ldots, s_n\}$ produces a composite mapping $\sigma\pi = \sigma\pi_A(S)$. We define $\tilde{\psi}_A(S) = \psi_{\sigma\pi}(S)$.

RSCS was shown to be NP-complete in [7]. A greedy strategy very similar to SGREEDY can also be applied to RSCS. For a pair of strings s_1 and s_2, we define $\tilde{\psi}(s_1, s_2)$ to be max $\{\psi(x, y) \mid x = s_1$ or $s_{-1}, y = s_2$ or $s_{-2}\}$, and $s_1 \tilde{\circ} s_2$ is defined to be one of $s_1 \circ s_2, s_1 \circ s_{-2}, s_{-1} \circ s_2, s_{-1} \circ s_{-2}$ which gives the maximum overlap $\tilde{\psi}(s_1, s_2)$. Ties are broken arbitrarily. Then the greedy algorithm for RSCS can be stated as follows. Given a non-empty set of strings S, repeat the following step until S contains just one string.

> Select a pair of strings $s_1, s_2 \in S$ that maximizes $\tilde{\psi}(s_1, s_2)$. Remove s_1 and s_2 from S, replacing them with $s_1 \tilde{o} s_2$.

We refer to this algorithm as RGREEDY. We are going to show that RGREEDY also has the same worst-case performance by the overlap measure, a factor of two, as SGREEDY. In other words, the performance ratio, $\tilde{\psi}^*(S)/\tilde{\psi}_{\text{RGREEDY}}(S)$, is less than or equal to two for a given set S of strings.

Consider the following set of strings

$$S = \{a(bc)^k b, c(bc)^k, (bc)^k bd\}$$

for which the optimal solution $a(bc)^{k+1}bd$ gives the total overlap $\tilde{\psi}^*(S) = 4k$ and a RGREEDY solution $a(bc)^k bdc(bc)^k$ gives $\tilde{\psi}_{\text{RGREEDY}}(S) = 2k + 1$. As k becomes large, the performance ratio approaches two. This example, together with Theorem 3.1 below, tells us that the worst-case performance of RGREEDY by the overlap measure is strictly a factor of two.

LEMMA 3.1. *Let w, x, y, and z be strings. If $\psi(w, y) = \max\{\psi(w, y), \psi(w, z), \psi(x, y), \psi(x, z)\}$ then $\psi(w, y) + \psi(x, z) \geq \psi(w, z) + \psi(x, y)$.*

The proof of this lemma appears in [7], and the lemma was used to prove $\psi^*(S) \leq 2\psi_{SGREEDY}(S)$ for SCS.

THEOREM 3.1. *Let S be any set of strings. $\tilde{\psi}^*(S) \leq 2\tilde{\psi}_{\text{RGREEDY}}(S)$.*

proof. Let s_i, s_j be the first pair of strings chosen from S by RGREEDY, and let S' be the new set obtained by replacing s_i, s_j with $s_i \tilde{o} s_j$. We show that $\tilde{\psi}^*(S) \leq 2\tilde{\psi}(s_i, s_j) + \tilde{\psi}^*(S')$. This, in turn, implies the theorem.

We define some notation for representing paired strings. Let $S_{eg} = \{s_1, s_2, s_3\}$. Then a pair $\langle s_1, s_2 \rangle$ denotes a *consecutive pair* in the solution string, where the tail part of s_1 is overlapped by the head part of s_2. Note that $\langle s_1, s_2 \rangle$ is equivalent to $\langle s_{-2}, s_{-1} \rangle$. We say that, for a given set of strings S, a consecutive pair $\langle x, y \rangle$ with $x, y \in S$ is *permissible* in the intermediate set S', in which some strings in S were already combined to yield new strings, if the strings x and y can be consecutively paired without destroying the new combined strings in S'. For S_{eg}, the set of permissible pairs in S_{eg} is $\{\langle x, y \rangle \mid x = s_i \text{ or } s_{-i}, \text{ and } y = s_j \text{ or } s_{-j}, \text{ and } i \neq j\}$. Let $S'_{eg} = \{\langle s_1, s_{-3} \rangle, s_2\}$. Then the set of permissible pairs in S'_{eg} becomes $\{\langle s_{-1}, s_2 \rangle, \langle s_{-1}, s_{-2} \rangle, \langle s_2, s_3 \rangle, \langle s_{-2}, s_3 \rangle\}$.

Let X be an optimal RSCS solution for S and let w, y be the strings such that $w = s_i$ or s_{-i}, $y = s_j$ or s_{-j}, $w \circ y = s_i \tilde{o} s_j$. By the definition of the greedy algorithm, $\psi(w, y) \geq \max\{\psi(s, t) \mid s, t$ are a *consecutive pair* in $X\}$. At most three consecutive pairs in X are not permissible in S'. If at most two are not permissible, then clearly $\tilde{\psi}^*(S) \leq 2\tilde{\psi}(s_i, s_j) + \tilde{\psi}^*(S')$ as desired.

If three consecutive pairs become impermissible then one must have the form $\langle w, z \rangle$ (equivalently, $\langle rev(z), rev(w) \rangle$) with $z \neq y$ (but possibly $z = rev(y)$), another the form $\langle x, y \rangle$ (equivalently, $\langle rev(y), rev(x) \rangle$) with $x \neq w$ (but possibly $x = rev(w)$) and the third one, any pair lying on X between y and w, say p. This means that the pair $\langle x, z \rangle$ can be consecutively paired in an optimal RSCS solution for S'. Clearly,

$$\tilde{\psi}^*(S') \geq \tilde{\psi}^*(S) + \psi(x, z) - \psi(p) - \psi(x, y) - \psi(w, z).$$

Then

$$\begin{aligned} \tilde{\psi}^*(S') + 2\psi(w, y) \quad & \geq \tilde{\psi}^*(S) + (\psi(w, y) + \psi(x, z) - \psi(x, y) - \psi(w, z)) \\ & \geq \tilde{\psi}^*(S) \qquad \text{by Lemma 3.1,} \end{aligned}$$

which completes the proof. □

4. SCMS

A *bag* $b = \langle a_1, \ldots, a_h \rangle$ is an unordered collection of symbols from some alphabet Σ in which the same symbol can appear more than once. If $s = a_1 \cdots a_h$ is a string, then $\langle s \rangle$ denotes the bag $\langle a_1, \ldots, a_h \rangle$. Then a bag $\langle s \rangle$ can be said to represent the set of strings obtained by permuting all the symbols in s. For example, the bag $\langle$abc$\rangle$ represents the strings abc, acb, bac, bca, cab, and cba.

We say that a bag b *matches* a string s if s contains a substring s' such that $\langle s' \rangle = b$. We also say that s matches b or that s is a *matching string* of b. For example, the string debcabf is a matching string of the bag $\langle$a, b, b, c$\rangle$.

An instance of SCMS is a set of bags $B = \{b_1, \ldots, b_n\}$ over a finite alphabet Σ. The object of the problem is to find a minimum length string that matches every bag in B. For example, the string bfgiakhfdegiach is a minimum length solution for the SCMS instance $B = \{\langle$aceghi$\rangle, \langle$abfgik$\rangle, \langle$adfhki$\rangle, \langle$defghi$\rangle, \langle$afghik$\rangle\}$.

A *partially ordered bag* (pob) is an ordered sequence of bags. We generally write a pob as a sequence of strings separated by |. A pob $b_1 \mid \cdots \mid b_n$ is interpreted as representing the collection of strings

$$\{s_1 \cdots s_n \mid \langle s_i \rangle = b_i, 1 \leq i \leq n\} \cup \{s_n \cdots s_1 \mid \langle s_i \rangle = b_i, 1 \leq i \leq n\},$$

where each string $s_1 \cdots s_n$ is said to be a *matching string* of the pob $b_1 | \cdots | b_n$. For example ab | cd | e, the pob produced by the combination of two bags abcd and cde, represents the set of strings {abcde, abdce, bacde, badce, ecdab, edcab, ecdba, edcba}.

As SCMS has been shown to be NP-complete, we need efficient approximation algorithms for SCMS to avoid excessive computational complexity. The basic structure of our approximate approach is the greedy heuristic which can be outlined as follows:

> Repeatedly combine two bags with the strongest *evidence of consecutiveness* either until all the bags are combined into a single pob or until bag-pairs with evidence of consecutiveness above a chosen threshold are exhausted.

The second terminating condition in the above algorithm is not necessary for SCMS algorithms, but is useful in DNA mapping applications.

A variety of different greedy algorithms can be proposed, depending on how we evaluate the evidence of consecutiveness and also on the strategies for refining and/or restructuring the pob to improve the solution. We have implemented more than ten different greedy algorithms and we present three of them in this report.

4.1. Greedy Algorithm for SCMS using Overlap between Pobs

Let $S = \{b_1, \ldots, b_n\}$ be an instance of SCMS. Then a greedy algorithm can be stated as follows. Repeat the following step until S contains a single pob.

> Select a pair of pobs $p_1, p_2 \in S$ that have a maximum overlap and combine them yielding a new pob p_3. Remove p_1 and p_2 from S and add p_3.

We refer to this algorithm as MGREEDY-P and Figure 2 illustrates an example of MGREEDY-P application.

The greedy heuristic used in MGREEDY-P is based on the size of overlap between pairs of pobs. The description of MGREEDY-P is very similar to those of SGREEDY and RGREEDY, respectively for SCS and RSCS. But, its implementation is more complex.

Let's compare SGREEDY and MGREEDY-P. Both algorithms requires a repeated selection of a pair of elements (strings in SCS and pobs in SCMS) that have a maximum overlap. In SGREEDY, an algorithm with at worst $O(\ell^2)$-time for finding the size of overlap between a pair of strings is quite straightforward, where ℓ is the length of the shorter string. Furthermore, once the sizes of overlap are determined, they remains effective and useful throughout the entire execution of the algorithm as long as we trace each original string.

If the bags are pre-sorted, then the size of overlap between bags can be easily found in a linear time of the size of the smaller bag. But, this overlap becomes invalid as soon as one of the bags is merged into a pob. Consider the bags $b_1 = \langle \text{aaccc} \rangle$, $b_2 = \langle \text{bbccc} \rangle$, $b_3 = \langle \text{abcc} \rangle$. Let o_{ij} be the overlap between b_i and b_j, then $o_{12} = o_{13} = o_{23} = 3$. Suppose that MGREEDY-P combined b_1 and b_2 to yield aa | ccc | bb, call this b_4. Then $o_{34} = 1$ and we can see that the overlaps o_{13} and o_{23} are not valid any more even though the bags b_1 and b_2 are at the sides of the new pob b_4. Therefore we can say that every time a new pob is constructed we have to recompute the overlaps between this new pob and the remaining pobs. What is worse, a brute-force algorithm for finding the size

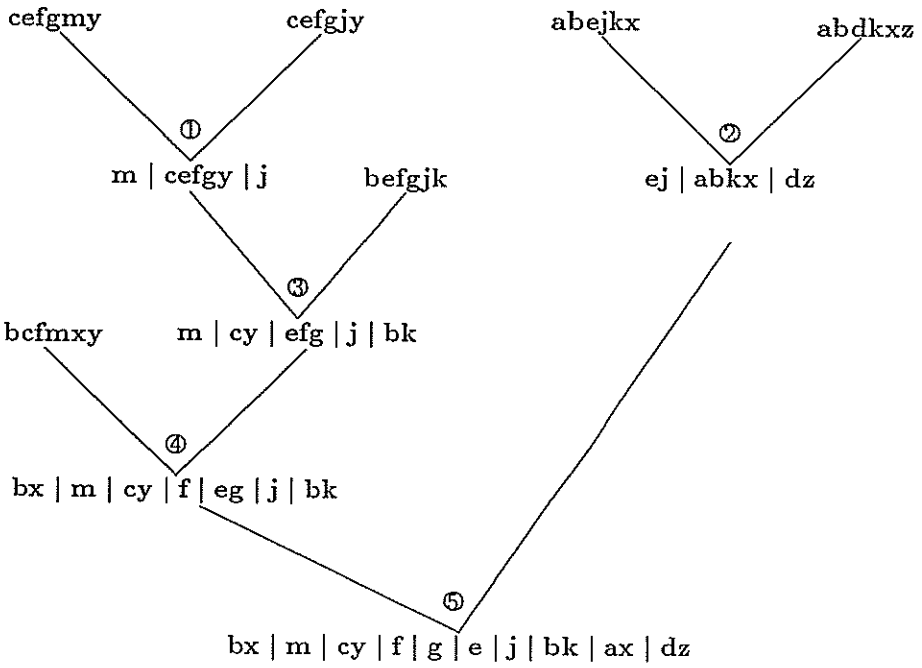

Figure 2: An example of MGREEDY-P application

of overlap between pobs can be as complex as $O(\ell 2^{\ell})$ in its running time, where ℓ is the size of the smaller pob.

There are four different ways in which two pobs can be overlapped. Let $p = b_1 \mid \cdots \mid b_n$ be a pob. Then $revp(p)$ is a reversed pob, that is, $b_n|\cdots|b_1$. Let $p_1 = \mathrm{ab|cde|fg}$ and $p_2 = \mathrm{aefg|cc|bfg}$ be pobs. Then we can think of the following overlap

p_1: ab| cde| fg
p_2: e fga| cc| bfg

In the above overlap, the pob p_1 precedes the pob p_2 and the order of bags in each pob is not reversed. We define $\omega(p_1, p_2)$ to be such an overlap. Hence $\omega(p_1, p_2) = \mathrm{e|fg}$. Now we can define the four possible overlaps between two pobs.

$$\omega_1(p_1,p_2) = \omega(p_1,p_2),$$
$$\omega_2(p_1,p_2) = \omega(p_1,revp(p_2)),$$
$$\omega_3(p_1,p_2) = \omega(revp(p_1),p_2),$$
$$\omega_4(p_1,p_2) = \omega(revp(p_1),revp(p_2)).$$

In the above example, $\omega_1(p_1,p_2) = \mathrm{e|fg}$, $\omega_2(p_1,p_2) = \mathrm{fg}$, $\omega_3(p_1,p_2) = \mathrm{a}$, $\omega_4(p_1,p_2) = \mathrm{b}$.

Before describing the algorithm to compute ω, we introduce a few notations. Let B be a bag over some alphabet Σ. Then $Occ_B(x)$ is defined to be the number of occurrences of the symbol x in the bag B, and $size(B)$ is defined to be $\sum_{x\in\Sigma} Occ_B(x)$. Let $P = b_1 \mid \cdots \mid b_n$ be a pob, then $size(P) = \sum_{1\le i\le n} size(b_i)$. Let A and B be bags. Then $A = B$ if and only if $\forall x \in \Sigma(Occ_A(x) = Occ_B(x))$. And $A \sqsubset B$ if and only if $A \neq B$ and $\forall x \in \Sigma(Occ_A(x) \le Occ_B(x))$. $A \sqsubseteq B$ if and only if $A = B$ or $A \sqsubset B$. $A - B$ is defined by $\forall x \in \Sigma(Occ_{A-B}(x) = \max(0, Occ_A(x) - Occ_B(x)))$, and $A + B$ by $\forall x \in \Sigma(Occ_{A+B}(x) = Occ_A(x) + Occ_B(x))$. A string s is said to be a *prefix of the pob* P if s is a prefix of a matching string of P. Again a pob Q is said to be a *prefix* of the pob P if a matching string of Q is a prefix of P.

Figure 3 describes an $O(\ell^2)$-time algorithm MAXMATCH to compute ω, where ℓ is again the size of the smaller pob.

THEOREM 4.1. *The algorithm MAXMATCH correctly computes* ω.

proof. The function $PREMATCH(S = s_1|\cdots|s_m, T = t_1|\cdots|t_n)$ is to check that the pob S is a prefix of the pob T, and to return the bag of extra symbols in s_x which is the first bag such that $s_1|\cdots|s_x$ is not a prefix of T. The returned bag will be $NULL$ when S is a prefix of T. More formally, $PREMATCH$ is a quite straightforward implementation of the function μ defined by

$$\mu(S,T) = (s_1 + \cdots + s_x) - (t_1 + \cdots + t_y),$$
where $s_1 + \cdots + s_{x-1}$ is a prefix of T,
$s_1 + \cdots + s_x$ is not a prefix of T,
$t_1 + \cdots + t_{y-1}$ is a prefix of $s_1 + \cdots + s_x$,
$t_1 + \cdots + t_y$ is not a prefix of $s_1 + \cdots + s_x$.

Let $S = s_1|\cdots|s_m$, $T = t_1|\cdots|t_n$, $\omega(S,T) = s'_a|s_{a+1}|\cdots|s_m$, $\omega' = \mathrm{MAXMATCH}(S,T) = s'_b|s_{b+1}|\cdots|s_m$ with $s'_a \sqsubseteq s_a$ and $s'_b \sqsubseteq s_b$. It is clear from the definitions of ω and μ that $|\omega'| \le |\omega(S,T)|$. Hence we have only to show $\omega(S,T) \sqsubseteq \omega'$.

When j, the for loop control variable of *MAXMATCH*, is less than a, *PREMATCH* may not return *NULL*, and the iteration continues until $j = a$. We claim that $u = \mu(s_a|s_{a+1}|\cdots|s_m,\ T) \sqsubseteq s_a - s'_a$ as long as $s'_a \sqsubseteq s_a$. In other words, the new bag $s_j = s_j - u$ is still a superset of s'_a.

Suppose $u = \mu(s_a|\cdots|s_m,\ T)$ is not *NULL*, then $u = (s_a + \cdots + s_x) - (t_1 + \cdots + t_y)$ for some x, y. $|t_1 + \cdots + t_y| > |s'_a + s_{a+1} + \cdots + s_x|$ and hence $t_1|\cdots|t_y \sqsupset s'_a|s_{a+1}|\cdots|s_x$, otherwise $t_1 + \cdots + t_y \sqsubset s_a + \cdots + s_x$ and $t_1|\cdots|t_y$ becomes a prefix of $s_a|\cdots|s_x$. Therefore $u = (s_a + \cdots + s_x) - (t_1 + \cdots + t_y) = (s_a - s'_a) + (s'_a + s_{a+1} + \cdots + s_x) - (t_1 + \cdots + t_y) \sqsubseteq s_a - s'_a$, where the last inequality comes from the fact that $s'_a|s_{a+1}|\cdots|s_x$ is a prefix of $t_1|\cdots|t_y$.

Thus the bag s_j in the while loop of *MAXMATCH* becomes smaller, but remains a superset of s'_a, until it becomes s'_a. □

Let p_1, p_2 be pobs. Then we define $\omega^*(p_1, p_2)$ to be $\omega_i(p_1, p_2)$ such that $size(\omega_i(p_1, p_2)) \geq size(\omega_j(p_1, p_2))$ for all j between 1 and 4. When p_1 and p_2 are bags, $\omega^*(p_1, p_2)$ is uniquely defined since $\forall i(\omega_i(p_1, p_2) = \omega(p_1, p_2)$. When p_1 and/or p_2 are not bags, there can be ties about the overlap size among two or more ways of overlapping. We assume the ties are arbitrarily broken in computing ω^*. Now we define $p_1 \otimes p_2$ to be the new pob that results from the combination of p_1, p_2 such that a maximum overlap $\omega^*(p_1, p_2)$ is obtained. In the above example,

$$\omega^*(p_1, p_2) = \omega_1(p_1, p_2) = \mathrm{e} \mid \mathrm{fg}$$
and $$p_1 \otimes p_2 = \mathrm{ab} \mid \mathrm{cd} \mid \mathrm{e} \mid \mathrm{fg} \mid \mathrm{a} \mid \mathrm{cc} \mid \mathrm{bfg}.$$

With the algorithm MAXMATCH and the above definitions, MGREEDY-P can be implemented to have a running time $O(n^2\ell^2)$ where n is the number of bags and ℓ is the size of the largest bag in the input.

4.2. Greedy Algorithms for SCMS using Weighted Overlap between Bags

When a bag is by itself, it can take any sequence consistent with the bag. But, once it is combined into a pob, then the possible sequences are constrained by the structure of the pob. Consider the following example.

From the orginal string **abcdefagehij**, the following four bags are obtained: $b_1 = \langle$**abcdef**$\rangle$, $b_2 = \langle$**cdefag**$\rangle$, $b_3 = \langle$**fagehi**$\rangle$, and $b_4 = \langle$**ehij**$\rangle$. Then MGREEDY-P will combine the first two bags into **b | acdef | g**, and then b_3 will be added to get **b | cd | aef | g | hi**. When it comes to b_4, whose overlap with b_3 was originally **ehi**, it can use only **hi** as overlap, because the overlap **e** is hidden by the pob structure. This could be worse if we had more bags after b_4 and if some of the overlap **hi** are also hidden in another pob, preventing **hi** in b_3 and **hi** in b_4 from overlapping.

MGREEDY-P might have a good performance in obtaining shorter matching string, but it might also fail to recognize the consecutive bags in the original string. Hence we propose another greedy algorithm based on the original overlaps between bags, not the overlaps between intermediate pobs. Further improvements in performance were observed with the weighted overlaps (= overlap size / sum of two bag sizes). This heuristic gives the algorithm MGREEDY-B described below.

Let $S = \{b_1, \ldots, b_n\}$ be an instance of SCMS.

step 1. (Preprocessing) Compute $P = \{p \mid p = \langle b_i, b_j, c_{ij}\rangle$ and $1 \leq i < j \leq n$ and $c_{ij} = size(\omega(b_i, b_j))/(size(b_i) + size(b_j)) > c\}$, where c is a user-supplied threshold value.

step 2. Repeat the following step until P is empty.

```
function PREMATCH (pob S = s_1 | ··· | s_m, pob T = t_1 | ··· | t_n)

/* PREMATCH(S,T) returns μ(S,T), which is the set of extra   */
/*     symbols in S preventing S from being a prefix of T.   */

    if (m = 0) then return NULL fi
    if (s_1 = t_1)              then return PREMATCH( s_2 | ··· | s_m, t_2 | ··· | t_n)
    else if (s_1 ⊏ t_1)         then return PREMATCH(s_2 | ··· | s_m, t_1 − s_1 | t_2 | ··· | t_n)
    else if (s_1 ⊐ t_1)         then return PREMATCH(s_1 − t_1 | s_2 | ··· | s_m, t_2 | ··· | t_n)
    else                             return s_1 − t_1
    fi fi fi
end PREMATCH


procedure MAXMATCH (pob S = s_1 | ··· | s_m, pob T = t_1 | ··· | t_n)

/* MAXMATCH(S,T) prints ω(S,T), which is the maximum overlap  */
/*   between the pobs S and T.                                */

    int     j, k;
    bag     u;

    find the smallest integer k such that size(s_k | ··· | s_m) ≤ size(T);
    for j = k to m
        u = PREMATCH(s_j | ··· | s_m, t_1 | ··· | t_n);
        while(u is not NULL)
            if (u ⋢ s_j) then exit while;
            s_j = s_j − u;
            u = PREMATCH(s_j | ··· | s_m, t_1 | ··· | t_n);
        endwhile
        if (u is NULL) then print "s_j | ··· | s_m"; stop; fi
    endfor
    print "NULL"
end MAXMATCH
```

Figure 3: The algorithm to compute the overlap between two pobs

> Remove $p = \langle b_{i_1}, b_{i_2}, c_{i_1 i_2} \rangle$ that has a maximum $c_{i_1 i_2}$. If neither b_{i_1} nor b_{i_2} is in the middle of an existing pob, and if they are not in the same pob, then replace the two pobs containing those two bags by the new pob obtained by combining them.

When n is the number of bags and ℓ is the average size of bags, MGREEDY-P has the running time $O(n^2\ell^2)$ and needs the space $O(n^2)$ to store the table of overlaps between pairs of pobs. But, MGREEDY-B doesn't need to store such a large table and has the running time $O(n^2\ell + n\ell^2)$, where the first term is for the preprocessing — $O(\ell)$ to find the overlap between a pair of bags — and the second term is for the main algorithm, which calls MAXMATCH at most $n-1$ times.

4.3. Greedy Algorithms for SCMS using Weighted Overlap between Bags and Subset-Consecutiveness Constraint

Greedy algorithms tend to emphasize local optimizations, which sometimes becomes an obstacle in achieving the global optimum. MGREEDY-B is more localized than MGREEDY-P, for MGREEDY-B uses only two bags to evaluate its heuristic while MGREEDY-P uses two pobs each of which may involve more than one bag. We tried adding some context constraints to MGREEDY-B to enhance the evidence of consecutiveness to overcome its locality. The most successful addition so far is the *subset-consecutiveness constraint*, which requires the bag in the middle to be a subset of the union of the surrounding bags in every three consecutive bags. The algorithm thus obtained, called MGREEDY-BS, is basically the same as MGREEDY-B, but combining two pobs is done by the function SUBSET-CONSECUTIVENESS-CHECK described in Figure 4.

Suppose the pob S is a combination of the original bags $s_1, \ldots, s_m$ and the pob T is that of $t_1, \ldots, t_n$. When we combine S and T, we require that each consecutive triple of bags in the list $s_1, \ldots, s_m, t_1, \ldots, t_n$ meet the subset-consecutiveness constraint. If it meets the constraint, then we combine S, T in the listed order; but if it fails, then we try switching a pair of consecutive bags in the list s_{m-1}, s_m, t_1, t_2. The reason for this switching is based on the observations that most of the bags out of sequence in previous greedy algorithm simulations came from the neighborhood, and hence this switching helps to correct a wrong sequence to some extent.

The running time and storage requirement of MGREEDY-BS are same as those of MGREEDY-B.

4.4. Discussions on the Test Results from Simulation

Three algorithms MGREEDY-P, MGREEDY-B, MGREEDY-BS were implemented in C.

The criteria to evaluate the programs at this point is how much of the original bag sequence is recovered by each program, even though the ultimate goal is to recover the original symbol sequence. To make this clear, let's consider the example illustrated in Figure 5. In the example there are 10 bags numbered 0 through 9 from a string, and their original sequence is 0123456789. To recover this sequence, 9 right decisions (about selecting two bags to be combined) must be made. The steps 1 through 5 in the example reflect right decisions, but the steps 6 through 8 wrong decisions. The number of wrong decisions is the same as the number of consecutive pairs of bags in the final sequence which are not consecutive in the original sequence (35, 40, 07 in the example). Note that the decision to combine the bags 4 and 5 was right even though they appeared in a reverse order by subsequent wrong decisions. We define N_{rd} and N_{wd} to be the number of right decisions and wrong decisions, respectively. In the remainder of this subsection, the *performance* of a program means the ratio N_{wd}/N_{rd}.

```
function SUBSET-CONSECUTIVENESS-CHECK (pob S, pob T)

/* S is a combination of the original bags s_1, ..., s_m.                */
/* T is a combination of the original bags t_1, ..., t_n.                */
/* We assume m, n > 2, avoiding the description of trivial cases.        */
/* SUBSET-CONSECUTIVE(b_1, ..., b_k) returns YES                         */
/*          only if ∀i(1 < i < k)(b_i < b_{i-1} + b_{i+1}).              */
/* We are going to combine S, T to get s_1, ..., s_m, t_1, ..., t_n.     */

   if SUBSET-CONSECUTIVE(s_{m-1}, s_m, t_1, t_2) then combine S, T
   else if SUBSET-CONSECUTIVE(s_{m-3}, s_{m-2}, s_m, s_{m-1}, t_1, t_2)
                  /* s_{m-1}, s_m switched */
                then combine S, T using the bag order in params
   else if SUBSET-CONSECUTIVE(s_{m-1}, s_m, t_2, t_1, t_3, t_4)
                  /* t_1, t_2 switched     */
                then combine S, T using the bag order in params
   else if SUBSET-CONSECUTIVE(s_{m-2}, s_{m-1}, t_1, s_m, t_2, t_3)
                  /* s_m, t_1 switched     */
                then combine S, T using the bag order in params
   fi
end SUBSET-CONSECUTIVENESS-CHECK
```

Figure 4: The algorithm to combine two pobs with subset-consecutiveness checking

Original bag sequence = 0 1 2 3 4 5 6 7 8 9

step 0.	{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
step 1.	{0, 12, 3, 4, 5, 6, 7, 8, 9}
step 2.	{0, 123, 4, 5, 6, 7, 8, 9}
step 3.	{0, 123, 45, 6, 7, 8, 9}
step 4.	{0, 123, 45, 6, 78, 9}
step 5.	{0, 123, 45, 6, 789}
step 6.	{0, 12354, 6, 789}
step 7.	{123540, 6, 789}
step 8.	{123540789, 6}

Figure 5: An example of recovering bag sequences where $N_{wd} = 3$ and $N_{rd} = 5$

L (length of original string)	= 460
k (number of symbols)	= 60
m (maximum bag size)	= 25
n (number of bags)	= 100

bag size	number of bags	accumulated
25	12	12
24	9	21
23	13	34
22	12	46
21	14	60
20	5	65
19	10	75
18	11	86
17	6	92
16	3	95
15	2	97
14	1	98
9	1	99
8	1	100

sum of all bag sizes	= 2072
average bag size	= 20.72
standard deviation	= 3.284
overlap factor	= 4.50 (= sum of bag sizes $/L$)

Table 1: The statistics for an example of set of bags

For the purpose of testing these programs, we wrote a program to generate a set of bags randomly. The parameters required by this bag-generator are L: the length of original string, k: the alphabet size, m: the maximum bag size, and n: the number of bags. The bag size follows a geometric distribution, but all the bags whose size is less than 5 or greater than m are eliminated. The bags which are subbags of other bags are also eliminated (The bag A is said to be a *subbag* of the bag B if $A < B$, and the relation "$<$" was defined in the subsection 4.1). Note that n is the number of bags which survived such elimination processes. A typical set of statistics for the set of generated bags is shown in the Table 1.

The performance of the programs was expected to be different for the data generated with different parameters, and we attempted to identify the region of parameter values for each program in which the program performs well. But, we have found the performance of these programs are unstable. For example, the performance of MGREEDY-P was 2/91 and 12/81 for two sets of data generated by the same parameter values. This difference can be explained partly by variation of data even though their statistics are similar. But, a more serious reason is the lack of good strategy to break ties. Suppose we have the following three consecutive bags with the original sequence b_1, b_2, b_3 from the original string **abcedfgha**, where $b_1 = \langle$**abcdefg**$\rangle$, $b_2 = \langle$**bcdefgh**$\rangle$, $b_3 = \langle$**cdefgha**$\rangle$. Then $\omega(b_1, b_2) = \omega(b_1, b_3) = \omega(b_2, b_3) = 6$, and we have 2/3 and 1/3 as a chance of getting the

right sequence for MGREEDY-P and MGREEDY-B, respectively. This kind of situation may seem improbable, but it is very likely if the bags b_1 and b_3 have larger sizes than b_2. Thus we concluded that such a region identification wouldn't be reliable without very extensive tests involving large sets of data, a questionable investment at this point. Instead we decided to compare the results from the programs to evaluate the performances of those heuristics, and the effects of alphabet sizes and deviations of bag sizes.

The parameter set we used in the test was $\{\langle k, m, L, n\rangle \mid k = 40, 50, 60, 80, 100$ and $\langle m, L\rangle = \langle 15, 300\rangle, \langle 20, 380\rangle, \langle 25, 460\rangle$ and $n = 100\}$. The relationship between m and L keeps the overlap factor constant (between 3.8 and 4.6). Three different data sets are generated for each case, yielding 45 different data sets for each of three algorithms.

Table 2 summarizes the results from the simulation. We make some observations and notes regarding this table:

- The performance of MGREEDY-B is better than that of MGREEDY-P and this is not because we use the overlap between bags, but because we use the weighted overlap. In fact the performance wasn't better than that of MGREEDY-P when we used the unweighted overlap between bags in MGREEDY-B.
- The performance of MGREEDY-BS shows how powerful the subset-consecutiveness test is. Each data set contains 100 bags and it requires 99 right decisions to recover the original sequence. Therefore those 45 test cases require 4455 ($= 99 \times 45$) right decisions to recover all the original sequences. MGREEDY-B made 42 wrong decisions, 4395 right decisions, and gave up 18 ($= 4455 - 42 - 4395$) decisions, while MGREEDY-BS made 6 wrong decisions, 4408 right decisions, and gave up 41 decisions. In other words, MGREEDY-BS corrected 13 (= 4408 - 4395) decisions and discarded 23 (= 41 -18) decisions out of 42 wrong decisions made by MGREEDY-B.
- The performance of the programs in the cases with larger alphabet sizes is generally better. This is to be expected because larger alphabet sizes reduces the probability of chance matching.
- The average standard deviations of bag sizes for $n = 15, 20, 25$ was 2.0, 3.24, and 3.62, respectively. With smaller deviations the programs give better results, because with large deviation there are more small bags having smaller true overlap than the false overlap between larger bags.
- The performance of MGREEDY-BS doesn't seem to be affected by alphabet sizes and deviations of bag sizes, but this is because these parameters were in good range for MGREEDY-BS.

5. Closing Remarks

The primary goal of this work is to develop an efficient approximation algorithm with good experimental performance to recover the true fragment sequences of DNA from random-clone data. The programs described in section 4 showed good performance in recovering the bag (or clone) sequence, but didn't do well in recovering the complete symbol (or fragment) sequence.

The heuristics in the greedy class of approximation algorithms have a focus on determining consecutive clones. In order to get a good solution, we have to rearrange the fragments within each clone carefully. Therefore we need an efficient algorithm to get the shortest possible configuration for a group of consecutive clones by rearranging their fragments. Once this step is successful, the algorithm can be used to improve the results of the greedy heuristic because a wrongly matched

alphabet size	maximum bag size	N_{wd}/N_{rd} MGREEDY-P			MGREEDY-B			MGREEDY-BS		
40	15	2/92	2/89	5/83	1/97	0/99	0/99	0/98	0/99	0/99
	20	6/87	12/81	2/91	0/98	2/96	3/94	0/97	0/97	0/94
	25	9/82	5/88	4/87	2/97	2/97	0/99	0/99	0/99	0/99
50	15	1/92	0/97	1/89	0/99	1/98	0/99	0/99	0/99	0/98
	20	3/89	1/93	6/84	3/94	3/95	2/96	0/99	0/97	0/96
	25	3/89	6/87	4/90	2/97	3/95	0/99	0/98	2/96	0/98
60	15	1/90	2/94	2/92	2/97	0/99	2/97	0/99	0/97	1/98
	20	2/89	4/89	5/85	1/98	2/95	0/99	0/99	0/98	2/94
	25	5/87	6/88	1/92	0/99	0/99	1/98	0/98	0/99	0/98
80	15	2/90	1/94	0/92	0/99	0/99	0/99	1/98	0/99	0/99
	20	0/94	0/93	0/92	0/98	0/97	0/99	0/99	0/97	0/96
	25	2/93	2/91	4/87	0/99	2/97	2/97	0/98	0/99	0/99
100	15	1/90	2/92	0/93	2/97	0/99	0/99	0/99	0/99	0/99
	20	1/93	1/89	1/94	0/98	2/95	0/99	0/99	0/97	0/95
	25	1/96	2/92	4/89	0/99	0/99	2/97	0/97	0/99	0/99

Table 2 – (a). Performance of greedy algorithms for SCMS

algorithm	alphabet size 40	50	60	80	100	total
MGREEDY-P	47/780	25/810	28/806	12/826	13/828	125/4050
MGREEDY-B	10/876	14/872	8/881	4/884	6/882	42/4395
MGREEDY-BS	0/881	2/880	3/880	1/884	0/883	6/4408

Table 2 – (b). Summary of the dependence of performance on alphabet size

algorithm	maximum bag size 15	20	25	total
MGREEDY-P	22/1369	44/1343	59/1338	125/4050
MGREEDY-B	8/1476	18/1451	16/1468	42/4395
MGREEDY-BS	2/1479	2/1454	2/1475	6/4408

Table 2 – (c). Summary of the dependence of performance on maximum bag size

Table 2: Performance of greedy algorithms for SCMS and their summaries

clone with evidence of consecutiveness that is locally good is likely to fail to contribute to the global overlap.

The other problem with algorithms from the greedy class is their lack of ability to deal with subset clones. Before we apply such algorithms, we have to eliminate all the clones which are a subset of other clones. We can add these subset clones after the greedy process is complete, providing some refinement of the map. Alternatively, we can add subset clones whenever the intermediate map is ready to take them without causing representational ambiguity about the locations of these subset clones. The former approach is simpler, but the latter approach will provide some capability of error detection as the clone sequencing proceeds.

The next step in our research is to determine the performance of the algorithm by worst-case and probabilistic analyses. Probable performance will be confirmed experimentally by simulation. This performance analysis is important in two respects. First, it gives a reliability measure of the solution map as it is obtained from the approximation algorithm. Otherwise, we will have no knowledge of the accuracy of the solution map, since the true restriction map is not available for comparison. Second, we can determine regions for parameters, such as redundancy factors and average number of fragments within a clone, which guarantee good algorithm performance. This information can be used to optimize the data acquisition process.

The last and most important step of this work is to apply the algorithms to real DNA random-clone data to obtain a nearly error-free estimate of the true DNA restriction map. We have two problems that appear at this step. First, the identification of equal-length fragments from different clones must be able to proceed in spite of errors in the measurement of clone length. This will require that the sources of errors be identified and an appropriate error model established. Second, we cannot exclude the possibility of incorrect data such as missing fragments or extraneous fragments which actually appear in clones. Thus, a successful algorithm will have to detect incorrect data with good reliability.

References

[1] F. Sanger and A. R. Coulson. "A Rapid Method for Determining Sequences in DNA by Primed Synthesis with DNA Polymerase." *Journal of Molecular Biology,* 94: 441–448, 1975.

[2] Allan M. Maxam and Walter Gilbert. "A new method for sequencing DNA." *Proc. Natl. Acad. Sci. USA,* 74(2): 560–564, February 1977.

[3] F. Sanger, S. Nicklen, and A. R. Coulson. "DNA sequencing with chain-terminating inhibitors." *Proc. Natl. Acad. Sci. USA,* 74(12): 5463–5467, December 1977.

[4] Kathleen Danna and Daniel Nathans. "Specific Cleavage of Simian Virus 40 DNA by Restriction Endonuclease of Hemophilus Influenzae." *Proc. Natl. Acad. Sci. USA,* 68(12): 2913–2917, December 1971.

[5] Jonathan S. Turner. "The DNA mapping problem." *Unpublished notes,* 1985.

[6] Lawrence T. Kou. "Polynomial Complete Consecutive Information Retrieval Problems." *SIAM J. Computing,* 6(1): 67–75, March 1977.

[7] Jonathan S. Turner. "The Complexity of the Shortest Common Mathching Problem." *Washington University Technical Report* WUCS-86-9, Department of Computer Science, April 1986.

[8] J. P. Bouche, J. P. Gelugne, J. Louarn, and J. M. Louarn. "Physical Map of a 470×10^3 Base-pair Region Flanking the Terminus of DNA Replication in the *Escherichia coli* K12 Genome." *Journal of Molecular Biology,* 154: 21–32, 1982.

[9] W. Bender, P. Spierer, and D. S. Hogness. "Chromosomal Walking and Jumping to Isolate DNA from the *Ace* and *rosy* Loci and the Bithorax Complex in *Drosophila Melanogaster.*" *Journal of Molecular Biology,* 168: 17–33, 1983.

[10] M. Steinmetz, D. Stephan, and K. F. Lindahl. "Gene Organization and Recombinational Hotspots in the Murine Major Histocompatibility Complex." *Cell* 44: 895–904, 1986.

[11] F. Sanger, et al. "Nucleotide sequence of bacteriophage Φ X174 DNA." *Nature,* 265:687–695, February 1977.

[12] T. R. Gingeras, J. P. Milazzo, D. Sciaky and R. J. Roberts. "Computer programs for the assembly of DNA sequences." *Nucleic Acids Research,* 7(2): 529–545, 1979.

[13] David Maier and James A. Storer. "A Note on the Complexity of the Superstring Problem." *Princeton University Technical Report 233,* Department of Electrical Engineering and Computer Science, October 1977.

[14] Marvin B. Shapiro. "An algorithm for Reconstructing Protein and RNA sequences." *Journal of ACM,* 14(4): 720–731, October 1967.

[15] Mark Stefik. "Inferring DNA Structures from Segmentation Data." *Artificial Intelligence,* 11: 85–114, 1978.

[16] William R. Pearson. "Automatic construction of restriction site maps." *Nucleic Acids Research,* 10(1): 217–227, 1982.

[17] R. Durand and F. Bregegere. "An efficient program to construct restriction maps from experimental data with realistic error levels." *Nucleic Acids Research,* 12(1): 703–716, 1984.

[18] Jonathan S. Turner. "Approximation Algorithms for the Shortest Common Superstring Problem." *Washington University Technical Report WUCS-86-16,* Department of Computer Science, July 1986.

[19] Maynard V. Olson, et al. "Random-clone strategy for genomic restriction mapping in yeast." *Proc. Natl. Acad. Sci. USA,* 83: 7826–7830, October 1986.

[20] John Gallant, David Maier, and James A. Storer. "On Finding Minimal length Superstrings." *Journal of Computer and System Sciences,* 20: 50–58, 1980.