

ALGORITHMS FOR COMPUTATIONAL BIOLOGY: SEQUENCE ANALYSIS

**RISHI SAKET
VARUN GUPTA**



DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY DELHI

JULY, 2004

Algorithms for Computational Biology - Sequence Analysis

*A thesis submitted in partial fulfillment
of the requirements for the degree of*

Bachelor of Technology
in
Computer Science and Engineering
by

Rishi Saket

Varun Gupta



DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING,
INDIAN INSTITUTE OF TECHNOLOGY DELHI

Certificate

This is to certify that the thesis entitled *Algorithms for Computational Biology - Sequence Analysis*, which is being submitted by *Rishi Saket* and *Varun Gupta* in fulfillment of the requirements for the award of *Bachelor of Technology in Computer Science and Engineering* at the Indian Institute of Technology is a true record of the students' work carried out under my supervision and guidance.

The matter embodied in this dissertation, to the best of my knowledge, has not been submitted for the award of a degree or a diploma elsewhere.

Prof. S.N. Maheshwari

Department of Computer Science and Engineering

Indian Institute of Technology Delhi

New Delhi - 110016

July 2004

Acknowledgement

We would like to express our gratitude to our supervisor Prof. S.N. Maheshwari for his continued support and valued guidance at every step of our project. Without his timely advice it would have been impossible to make any inroads into the project. Working on this project has been an enriching as well as an enjoyable experience for both of us. Prof. Maheshwari provided us with initiation, encouragement and motivation at every stage without which the completion of this project would not have been possible.

We also take this opportunity to thank Prof. Alok Bhattacharya of Jawaharlal Nehru University for the insightful sessions of discussions and explanations which helped us clear doubts and understand many concepts.

Rishi Saket

Varun Gupta

Abstract

Approximate repeats in genomic sequences are important to identify regions of genomes with similar functionality and understand how they are preserved over time. Previous work on this topic, such as *RePuter* and *Vmatch*, was focussed on the fixed error bound version of this problem. The percentage error version has not been effectively tackled yet. Our work tries to give scheme for finding long approximate repeats, which are of greater biological importance, where the bound on the errors is a percentage of the size of the repeat. We made the fact that every approximate repeat can be decomposed into a chain of maximal repeats the basis of our scheme. It involves computation of maximal repeats as seeds and the extension of the seeds by a process of *stitching* together of maximal repeats. Suitable algorithms for fast edit distance calculation, required for the *stitching* process, were also integrated. For the purposes of evaluating the performance, the entire scheme was implemented and executed on sequences of various lengths and with varying error rates. Analysis of the results obtained showed that our scheme performed well, both in terms of the quality of approximate repeats obtained and time required for the computation. We also present a method to find approximate occurrences of a given pattern in a database sequence using a specified inter-character distance/similarity matrix.

Contents

List of Figures	vi
List of Tables	vii
1 Introduction	1
2 A New Scheme for Finding Approximate Repeats under the Percentage Error Model	5
2.1 Problem Statement	5
2.2 Preliminaries	6
2.2.1 Data Structures	6
2.2.2 Repeats	7
2.3 Motivation	8
2.4 Algorithmic Description	9
2.5 Issues	12
2.5.1 The <i>depth</i> parameter	12
2.5.2 Edit distance calculation	13

3	Experimental Results	19
3.1	Modules Developed	19
3.2	Experiments conducted	20
3.3	Results	20
3.4	Analysis of Results	21
4	The Generalized Hamming-Distance Problem and Fast Fourier Transform	25
4.1	Motivation	25
4.2	Notation and Problem Definition	26
4.3	Previous Work	27
4.4	Using FFT to Calculate MC'	28
4.5	$ \Sigma $ -Dimensional Embeddings	30
4.6	Low Dimensional Embeddings for Approximate Similarity	34
4.7	More Low-Rank Approximations of \mathbf{S}	37
5	Conclusions	41
	Bibliography	43

List of Figures

2.1	Breakup of approximate repeat into maximal repeats	8
2.2	Stitching of repeats	12
2.3	Four-Russians speedup	15
2.4	Offset Encoding for Four-Russians speedup	16
2.5	Timing Comparison for different dynamic programming modifications	17
3.1	Average running time graph for Table 3.3	23
4.1	Scree plot for an alphabet formed from 4-grams on an alphabet Σ ($ \Sigma = 4$)	36

List of Tables

3.1	Size of the sequences and the minimum seed size used	20
3.2	Experimental results obtained by varying the minimum seed size for each string at 15% error rate.	21
3.3	Experimental results obtained by varying the error rate. The number in brackets is the seed size used for each string. Number of repeats includes the seeds computed.	22
3.4	Breakup of average running time for 10% error bound	22
3.5	Results obtained by varying the <i>depth</i> parameter for chr4.fa with 15% error rate and minimum seed size 15. Here <i>d</i> is the value obtained by the formula in Section 2.5.1	24

1 Introduction

In the recent past there has been increasing interest in the applications of Computer Science in solving problems in gene sequencing and analysis. This has fuelled research in areas of computer science with applications in biology, and has led to the emergence of Computational Biology as a new field of research. Algorithms on strings and sequences are of importance in conducting genome sequencing and characterization.

A genome is a sequence of base pairs bonded together. Each region of the genome has a particular function which is determined by the base pairs consisting it, the neighbourhood region and the its position in the genome. A genomic string is the representation of a genome in the form of a string of characters. Each type of base pair is assigned a character from an alphabet. Typically the base pairs are of four types a, c, g and t. The length of genomes may range from a few hundreds to billions of base pairs.

Various algorithms and data structures on string matching had been developed earlier and it was later that their applications in biology were understood. The suffix tree data structure and linear time algorithms for its creation were developed [1]. However, with the size of the genomic strings in the order of millions of base pairs, the space and time complexity, and memory addressing issues became of utmost importance. The suffix array [2] data structure was developed to overcome these problems. Recently, linear time creation algorithms were developed for it [3, 4], along with data structures to simulate a suffix tree [5].

In sequencing of genes and classification of various regions of a gene, problems faced by the biologist are those of finding exact and approximate repeats in the genomic string. Approximate or degenerate repeats are useful to identify regions of a genome that have similar functionality such as protein synthesis. These repeats may have errors of the type replacements, insertions and deletions of base pairs. The problem is that of finding approximate repeats that satisfy a given upper bound on the number of errors. This bound may either be a fixed quantity or may be a percentage of the size of the approximate repeat.

The most significant and widely used software tools developed for approximate repeats are *RePuter* [10, 12] and the more recent *Vmatch* [11, 13]. As part of this project, we have made a study of these tools.

RePuter was developed by Stefan Kurtz's team at the University of Bielefeld. The kinds of repeats that it finds are *maximal repeats*, *maximal reversed repeats*, *maximal palindromic repeats* and *maximal complemented repeats*, which are all exact repeats. In addition, it also computes approximate repeats and contains options to specify a lower limit on the length of the repeats and an upper limit on the allowable edit distance.

RePuter uses the compact suffix tree proposed by Kurtz [9] to compute maximal repeats. The tree allows to compute the longest common prefix of two leaves in $O(1)$ time [14, 15]. The maximal repeats calculated are used as seeds. Along with the upper bound, e on errors, *RePuter* also takes as input a parameter l , which is a lower bound on the length of the approximate repeat. The minimum length of the longest exact maximal repeat which that can be present in any approximate repeat is given by $\lfloor l/(e+1) \rfloor$. This is the lower bound on the seed size i.e. the size of the exact maximal repeat pairs which are then computed and used as seeds. This lower bound also be set manually.

Each seed is an exact maximal repeat pair. *RePuter* extends each such pair on both sides using dynamic programming. The extension ends when the edit distance crosses the specified limit. *RePuter* incorporates the $O(1)$ retrieval of *longest common prefix* of two suffixes while doing dynamic programming. The calculated repeat satisfies the error bound, and is reported

if its length satisfies the specified lower bound. *RePuter* does not solve the percentage error problem. It uses a static upper bound e on the edit distance, and hence solves the fixed error problem. Since *RePuter* uses suffix trees as its basic data structure, it has the drawbacks associated with suffix trees on large genomic sequences. These are related to low spacial locality in memory addressing which makes access times longer for large sequences [2].

Vmatch is similar to *RePuter* in functionality except that it can also compute branching tandem repeats (along with those computed by *RePuter*). However, algorithmically it has significant differences. Instead of the suffix tree, it builds suffix array and the related data structures like the lcp-array. Also, it uses an *X-drop* algorithm [16] for dynamic programming for extending the seeds in finding approximate repeats. The *X-drop* algorithm is a greedy method for dynamic programming, which limits the computation to only a few diagonal elements. It has a worst case running time of $O(d_{max}L)$ where d_{max} is the maximum value of the distance between the strings and L is the maximum value of $|i - j|$ where i and j are the offsets from the starting positions of the two sequences that occur. Like *RePuter*, *Vmatch* also computes only approximate repeats with a fixed bound on errors. Moreover, the value of X for *X-drop* has no direct relation to the similarity of repeats reported.

As mentioned, *RePuter* and *Vmatch* attempt to extend one seed at a time using dynamic programming. However, this is possible only when a limit on the number of errors is given and it can be seen that the termination condition in the dynamic programming is when the error exceeds the fixed bound. Such a termination condition is not applicable to the percentage error problem, as the error bound is not static but a fraction of the length of the repeat. This makes extension by dynamic programming inappropriate for extension of seeds in the percentage error problem. Also, in case of *RePuter*, dynamic programming is inherently costly and is likely to be slow for very long repeats which are more likely to occur in the percentage error problem, while the *X-drop* algorithm used by *Vmatch* does not work for all scoring functions.

Contribution of the Project

During the course of this project we have closely interacted with Prof. Alok Bhattacharya

who is a biologist at the Jawaharlal Nehru University. He proposed the percentage error problem for approximate repeats. This problem was relevant to his research which focussed on sections of genomes that are preserved over time. We have designed a scheme that works for error rates. The scheme has been implemented and tested on substantially long genomic strings. The focus is on finding the longest repeats which are of special importance to biologists.

Chapter 2 gives the problem definition and an overview of the various algorithms and data structures studied and used in developing the scheme. Concepts like maximal and supermaximal repeats and algorithms for finding them are discussed. It also contains the motivation behind our scheme and discusses the details. In doing this we focus on various aspects like calculation of seeds, extension of seeds and calculation of edit distance between gaps.

Chapter 3 describes the experiments conducted using this scheme on different genomic strings. The results for varying string lengths and error rates are presented and analysed. Since the error rate problem has not been effectively tackled previously there are no set benchmarks. Empirical data show that our scheme runs fairly fast and finds longest approximate repeats of length in the thousands.

In Chapter 4 we define the *Generalized Hamming Distance* problem, originally motivated from the problem of finding approximate occurrences of a pattern in genomic sequences. Previous similar work, its extension to the problem of generalized hamming distance and limitation are discussed. Sections 4.4 to 4.7 outline our approach and how it overcomes the limitations of previous methods.

In conclusion we discuss the strengths and weaknesses of our scheme and mention problems that remain to be tackled in finding approximate repeats.

2 A New Scheme for Finding Approximate Repeats under the Percentage Error Model

2.1 Problem Statement

The percentage error problem of approximate repeats is formally stated as follows.

Given a string S and a value p , find two non-overlapping substrings R_1 and R_2 , of S , such that $e/l \leq p$ and l is maximized, where,

- $l = \max(|R_1|, |R_2|)$, and
- $e = \text{edit_distance}(R_1, R_2)$.

Here $\text{edit_distance}(R_1, R_2)$ is the minimum number of insertion, deletion and replacement operations to transform R_1 to R_2 . Clearly, the problem is solvable in polynomial time as it can be solved by considering all pairs of substrings of S , which are $O(N^2)$ (where N is the length of S) in number, and finding their edit distance which takes a $O(m \times n)$ (where m and n are lengths of the substrings) of time in computation for each pair. Such a scheme

would take $O(N^4)$ time for computation. With genomic sequences of lengths in millions, such computation time would be infeasible.

The fixed error version of this problem is the same, except that p denotes the limit on the number of errors, and the condition to be satisfied is $e \leq p$.

2.2 Preliminaries

2.2.1 Data Structures

The two main data structures used are suffix trees and suffix arrays [8]. Here we limit ourselves to giving only their definition.

We use the convention of representing a sequence in the form of an array $String[0..N-1]$, where N is the length of the string.

Suffix Tree: Given a string, its suffix tree is a compacted trie built on all the suffixes of the string. The suffix tree requires linear space as the substring on each edge is stored in the form of two pointers to positions in the array of the string denoting the substring. Algorithms have been developed to create the suffix tree in $O(N)$ time [1].

Suffix Array: Given a string, its suffix array is the array of pointers to its suffixes sorted in lexicographical order. While introducing this data structure, Manber and Myers [2] gave an $O(N \log N)$ construction algorithm for the suffix array without constructing a suffix tree, where N is the length of the string. Recently, algorithms have been proposed to construct the suffix array in $O(N)$ time [3] [4].

While algorithms are known for query string matching that run in optimal time using a suffix tree, an additional data structure, the *lcp-array* is needed along with the suffix array to achieve optimality.

Lcp-array: The Lcp-array stores the lengths of the *longest common prefix* of adjacent suffixes in the suffix array. $Lcparray[i] = lcp(String_{Suf[i-1]}, String_{Suf[i]})$, where $String_k$ is the

suffix starting at position k in the string and Suf is the suffix array.

The suffix array, along with the lcparray, can be used to answer exact string search queries using a suffix array in $O(m + z)$ time where m is the size and z is the number of occurrences of the query string [6].

2.2.2 Repeats

There are various notions of exact repeats that occur in a string such as maximal, supermaximal, tandem and branching tandem repeats. Here, we concern ourselves with maximal and supermaximal repeats.

Maximal Repeat A *maximal repeat* is, as the name suggests, a repeat that cannot be extended further. In a given string, if there occur two non-overlapping occurrences of a substring S of the form aSb and cSd , then if $a \neq c$ then this repeat of S is said to be *left maximal*, while if $b \neq d$ then it is known as *right maximal*. A repeat, which is both *left maximal* and *right maximal* is a *maximal repeat*. A maximal repeat is represented by $(S, i, j, |S|)$, where S is the repeated string and i and j are the starting positions of the left and right instances of S .

Supermaximal Repeat A *supermaximal repeat* is a maximal repeat which does not occur as a substring of any maximal repeat.

Optimal algorithms are known which find maximal repeats in optimal $O(N + z)$ (where N is the length of the sequence and z is the number of maximal repeats) time using suffix tree [8] or suffix array and lcp-array [5]. It can be seen that the number of maximal repeats in a string are $O(N^2)$ where N is the length of the string.

The repeats described above are exact repeats, i.e. the string S repeats without any transformation. However, in genomes the repeated substring very often undergoes mutations and the repeated instances may be dissimilar to some extent. These repeats are called *approximate* or *degenerate* repeats. An approximate repeat is represented by (i, j, l, r, e) , where i, j are starting indices of the left and right instances of the repeat; l, r are lengths of the left and

right instances and e is the edit distances of the instances. These are the class of repeats whose identification is the focus of this project.

2.3 Motivation

The main reasoning behind our scheme is the fact that any *non-extendable* approximate repeat can be broken into a series of exact maximal repeats such that left instance of each maximal repeat is present in left instance of the approximate repeat and the right instance of each maximal repeat is present in the right instance of the approximate repeat. Also, the maximal repeat instances are present in the same order in both instances of the approximate repeat. Two adjacent instances of different maximal repeats, present in the same instance of the approximate repeat, may either overlap or may have a gap of errors (inserted, deleted or replaced characters). An example is shown in Figure 2.1, where the maximal repeats A , B , C and D constitute the approximate repeat. Gaps are present between left and right occurrences of A and B , and B and C , while the left and right occurrences of C and D overlap. This observation implies

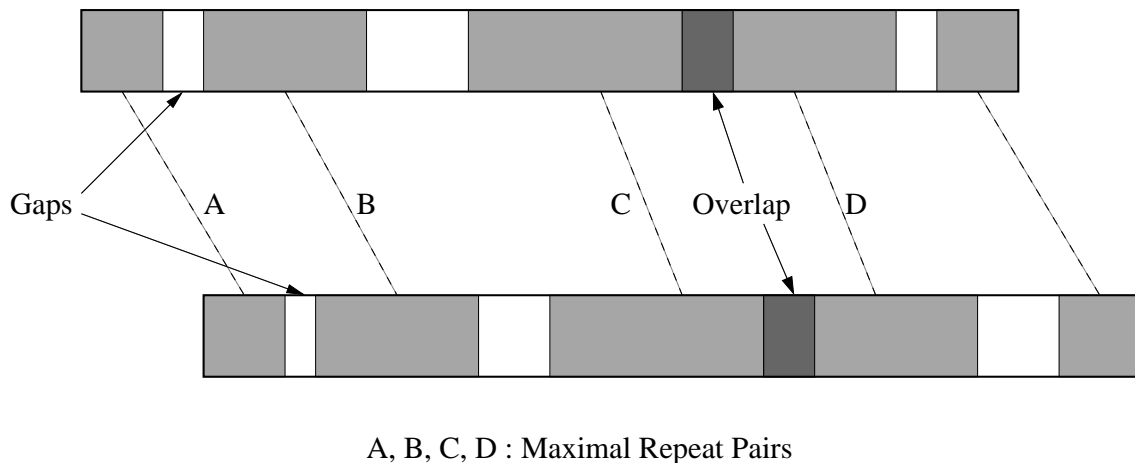


Figure 2.1: Breakup of approximate repeat into maximal repeats

that an approximate repeat can be obtained by *stitching* together its constituent maximal repeats. The *stitching* starts by taking a maximal repeat (seed) and the extending it by combining

(*stitching*) it with other maximal repeats. This forms the basis for our scheme. However, there are still some issues to be resolved.

It can be seen that in any decomposition of an approximate repeat into its complete chain of constituent maximal repeats, the maximal repeat pairs are the alignments and the gaps are the edits. In theory, one can compute an approximate repeat just by stitching together *all* of its constituent maximal repeats *without* calculating the edit distance of the gaps. However, to do this, one has to compute all the maximal repeats of the sequence, and extend each one as a seed. But the number of maximal repeats are $O(N^2)$, and so for very long strings calculating them and storing them is highly time and space consuming. Hence there have to be some tradeoffs.

One can avoid computing all the maximal repeats for stitching, if one can compute the error in the gaps between maximal repeats while stitching. It can also be seen that the smaller maximal repeats are generally far more numerous than the larger ones. Also one expects that an approximate repeat will consist of a number of large maximal repeats. Thus, it is much more convenient to compute only maximal repeats that are large. This can be done optimally (by slightly modifying the original algorithm [5]) in $O(N + z)$ time, where N is the length of the string and z are the number of repeats.

2.4 Algorithmic Description

The previous section gave an overview of the motivation and thinking involved in developing the scheme for approximate repeats. This section contains the detailed description of the computation.

The steps involved in computing the approximate repeats are as follows.

1. *Computation of maximal repeats*: The first step involves computing the maximal repeats greater than a certain length h which is determined by the length of the string. This also determines the precision of the algorithm. It is also limited by the computational

capacity and memory at hand. We implemented this computation using suffix arrays. The steps involved were,

- a) *Creation of the suffix array and the lcp-array.* For this the $O(N)$ algorithm [3] was implemented. It computes the lcp-array along with the suffix array.
- b) *Calculating maximal repeats.* This was done by applying the *process* function given by Abouelhoda for a bottom up processing of lcp-intervals using the suffix array and the lcp-array [5]. The minimum length h was used bound the upward traversal.

The maximal repeats obtained are stored in a list, say *max_List*.

2. *Stitching of repeats.* The obtained list *max_List* is sorted in the ascending order of the starting index of the left occurrence and then the right occurrence (i.e. on the pair ' (i, j) ' where $(i_1, j_1) < (i_2, j_2)$ if $i_1 < i_2$, or $i_1 = i_2$ and $j_1 < j_2$). In doing so, each maximal repeat is considered as an approximate repeat with zero errors and is stored in the approximate repeat format of $(i, j, l, r, e, isnew)$, where *isnew* a boolean initially set to *true*. Next the following algorithm is followed for stitching.

- 1 Initialize the list *new_List* of repeats and a parameter *depth*.
- 2 For the k th repeat $(i_k, j_k, l_k, r_k, e_k, isnew_k)$ in *max_list* where k goes from 1 to $sizeof(max_list)$. Do the following,
 - 2.1 If $isnew_k = false$, this implies that this seed has been extended in a previous iteration. Increment k and goto 1.
 - 2.2 Make a copy of the k th repeat. Let $(i, j, l, r, e) = (i_k, j_k, l_k, r_k, e_k)$. Set boolean variable $is_extended = false$. The value of $is_extended$ determines whether extension of the current seed takes place in this iteration. Set $index = k$. Do the following,
 - 2.2.1 Increment $index$ by 1 to get the next repeat to be looked at for stitching.
 - 2.2.2 If $index > sizeof(max_List)$, no more stitching can occur. Goto 2.2.

- 2.2.3 If $i_{index} - i - l > depth$ or $j_{index} - j - r > depth$, the *indexth* repeat falls outside the search *depth*. No repeat after *indexth* repeat in *max_List* can be inside the *depth*, due to the order in which they are stored in *max_List*. No more stitching can occur. Goto 2.2.
- 2.2.4 Calculate the error obtained by stitching (i, j, l, r, e) and $(i_{index}, j_{index}, l_{index}, r_{index}, e_{index})$. For this the error in the gap or overlap has to be calculated.
- If there is a gap between (i, j, l, r, e) and $(i_{index}, j_{index}, l_{index}, r_{index}, e_{index})$, the edit distance of the gap needs to be calculated using the any of the edit distance function discussed in Section 2.5.2.
 - If there is an overlap between the repeats, they are stitched only if e_{index} is 0, i.e. the *indexth* repeat is an exact maximal repeat. In this case, the overlap error is the *shift* in the starting positions of the two occurrences of the *indexth* repeat. $shift = ||i + l - i_{index}| - |j + r - j_{index}||$. If the *indexth* repeat is an approximate repeat, goto 2.2.1.

If repeat obtained by stitching (i, j, l, r, e) and $(i_{index}, j_{index}, l_{index}, r_{index}, e_{index})$ does not exceed the error rate merge them into (i, j, l, r, e) with updated values. Set $is_extended = true$ and $isnew_k = false$. Goto 2.2.1.

- 2.3 If $is_extended = true$, then extension has taken place and a new repeat found. Insert $(i, j, l, r, e, 'true')$ (which is the new repeat found) into *new_List*.
- 3 If *new_List* is not empty merge *max_List* and *new_List* according to the order given above in *Stitching of repeats* and goto 1, else end.

The stitching process is depicted in Fig 2.2. For the extension of the repeat *A*, search starts from the end of its left occurrence L_A and if a repeat, say *B* is found such that its left occurrence, L_B begins within *depth* distance of end of L_A , it is checked whether the right occurrence of *B*, R_B also begins within *depth* distance of end of R_A . If so, then the error of the gap or overlap (as the case may be) between them is calculated and they are

stitched together if the error in the combined repeat does not exceed the error bound.

The repeats that are obtained in the *max_list* are written in a file in the appropriate format. There are several issues involved in the stitching process such as the value of *depth* and the problem of calculating the edit distance of the gaps. These issues will be discussed in detail in the next sections.

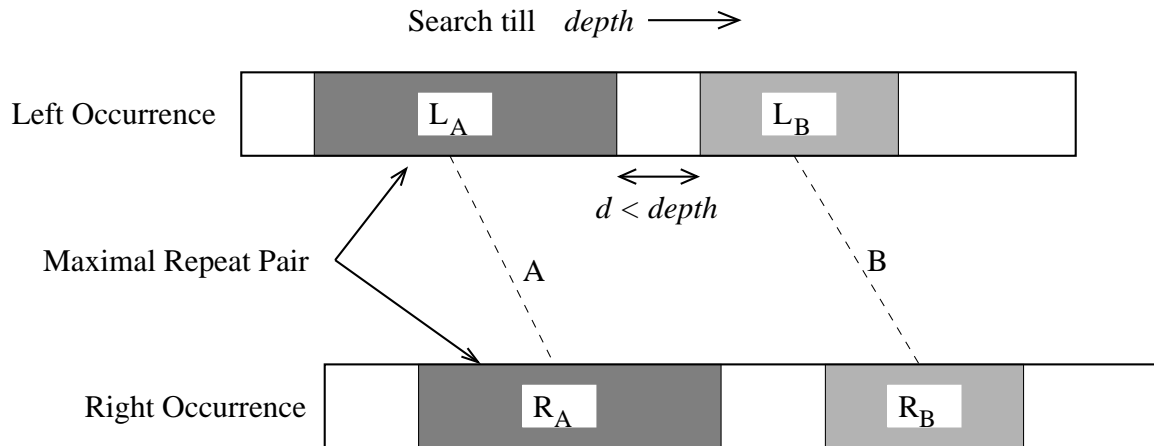


Figure 2.2: Stitching of repeats

2.5 Issues

The previous section gives the steps followed in computing the approximate repeats. As mentioned there are crucial issues involved in the process which need to be resolved for implementing the method. In this section we describe the issues, their relevance and the strategies adopted to resolve them.

2.5.1 The *depth* parameter

The *depth* parameter is basically the region of search towards the right of a repeat that the search for a stitchable repeat is performed. The *depth* parameter determines the likelihood of

a legitimate approximate repeat being missed by the algorithm. The larger the depth, the less likelihood of a repeat being missed, and greater the time taken by the algorithm.

Another question that arises is whether *depth* be statically fixed or be dependent on the length of the repeat. It seems reasonable to say that *depth* should reflect the capacity of the repeat to *absorb* errors. Also, there must be a lower bound on *depth* to keep a large enough value even for small repeats and repeats with large number of errors. We decided on the following formula to fix *depth* at each iteration,

$$depth = \max(min_depth, e' - e) \text{ where } e' = 2 \times p \times \min(r, l) \text{ and } min_depth \text{ is a fixed parameter.}$$

This formula ensures that repeats with a small number of errors will have a large depth and even small repeats will have a minimum depth.

2.5.2 Edit distance calculation

As has been mentioned earlier, the maximal repeats are not being calculated for stitching. This implies that during stitching, the edit distance between the gap between the left occurrences and the gap between the right occurrences of the repeats to be stitched has to be calculated. One must note, however, that the occurrences may overlap, and in that case the edit distance is just the displacement or shift. The latter is generally a rare occurrence so it is very important to formulate a suitable strategy for calculating the edit distance.

A useful observation made on the sequences we worked with is that there are very few insertion-deletions (*indels*) as compared to replacements between the degenerate copies and that the gaps are of comparable sizes. Therefore, when aligning the two gaps we need to do local searches for similarities. One approach to align the gaps can be to hash *d*-grams of the two gaps into a hash table with $|\Sigma|^d$ buckets. Pair the entries from the two gaps which are within some threshold distance of each other, and then chain these pairs. Another approach

considered was, to create a new string $S_g = gap_1 \cdot \# \cdot gap_2$. Then find the maximal repeat pairs in this string such that one lies in gap_1 and the other in gap_2 . Align the gaps starting with the longest pair which satisfies the threshold distance. This will leave very small gaps for which dynamic programming can be done.

Though the above approaches imply time complexities almost linear in gap lengths and therefore seem attractive in comparison to dynamic programming, the constants are very high. Instead, by tweaking the *vanilla* dynamic programming algorithm we can speed it 4 to 5 times. One way is based on the aforementioned observation. By putting a bound on indels, we constrain the number of diagonals to be explored in the dynamic programming table. Another optimization is the *Four-Russians speedup* ([17], [21]) which uses a precomputed lookup table to make the evaluation of dynamic programming table faster.

Dynamic Programming with bound on indel-rate

Let the length of the two gaps be l_1 and l_2 , ($l_2 \geq l_1$). If we put a bound on the indel rate, idr , we are essentially restricting the $|\#insertions - \#deletions|$ between any prefix p_1 of gap_1 and p_2 of gap_2 to $|l_2 - l_1| + 2 * idr * l_1$. Therefore, we only need to create a dynamic programming table of size $(l_1 + 1) \times (|l_2 - l_1| + 2 * idr * l_1 + 1)$ and fill the cells (i, j) where

$$i = 1 \text{ to } l_1, \quad j = \max\{0, idr * l_1 - i + 1\} \text{ to } \min\{|l_2 - l_1| + 2 * idr * l_1 + 1, l_2 - i + idr * l_1 + 1\}$$

after appropriate initialization. It is obvious that this method will give a considerable speedup if $|l_2 - l_1|$ is as small as possible. Also, as mentioned, the gaps are of comparable sizes (where the gaps are not of comparable sizes and the indels are large, any heuristics is likely to fail). Therefore, this is a very useful method of finding the edit distance between the gaps. The larger the value of idr , more accurate is the edit distance obtained to actual edit distance. (The obtained edit distances are always larger or equal to the actual edit distance.)

The Four-Russians speedup

The origin of this speedup lies in a paper concerning boolean matrix multiplication [17] and was adapted to *unweighted* edit distance computation by Masek and Paterson [21],[22]. An important assumption used for this algorithm is that the alphabet size, $|\Sigma|$, is constant. Below, we describe use of the Four-Russians speedup for edit distance calculation as explained by Gusfield [8].

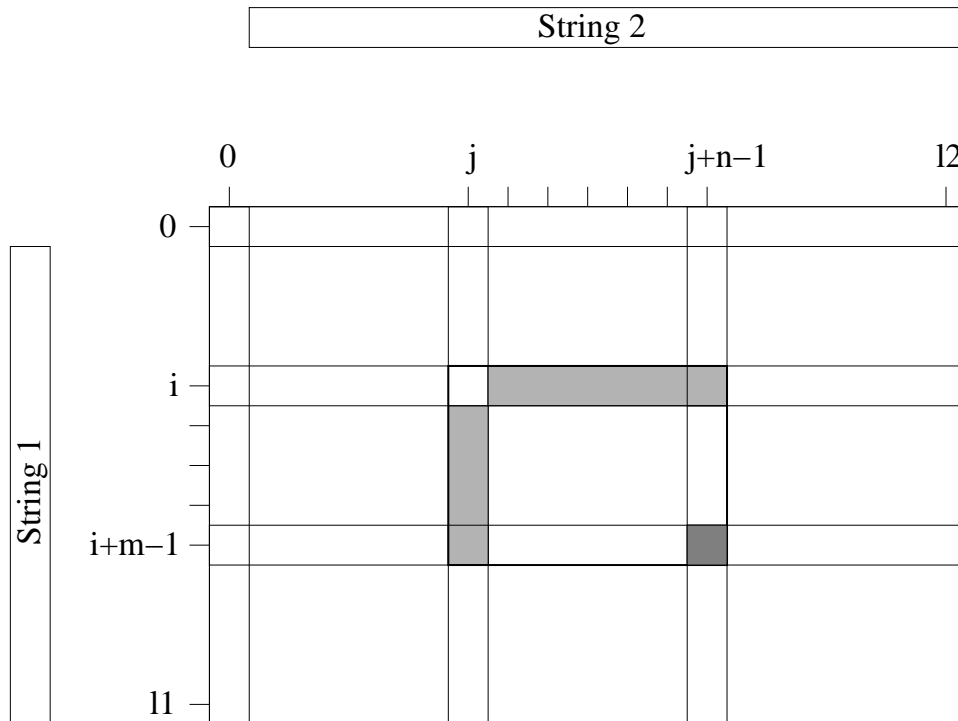


Figure 2.3: Four-Russians speedup

Let us break our dynamic programming table into overlapping blocks of size $m \times n$ (Fig. 2.3). To calculate the value in cell $(i + m - 1, j + n - 1)$, we only need the entries of the first row and column of the highlighted block and the substrings $String_1[i + 1 \dots i + m - 1]$ and $String_2[j + 1 \dots j + n - 1]$. If we had a lookup table that allowed us to obtain the value in the bottom-right cell on the basis of these then we can get the edit distance of these two substrings in $O(m + n)$ time (time to compute the index) instead of $O(mn)$. To compute the

edit distance of the complete strings we also need lookup tables that will give the entries in the last row and column of the blocks so that starting from the top-left block, moving in rowwise or columnwise manner, we can calculate the bottom-right cell of all the blocks. However, since the strings can be arbitrarily long, the possible values of the first row and column are not finite. To overcome this, offset encoding is used. First, we can subtract any constant from all the entries of a dynamic programming block and it will still be a correct dynamic programming table. Set the top-left entry as 0. Now no entry can be smaller than $-\max\{m-1, n-1\}$ or larger than $\max\{m-1, n-1\}$. This gives us a bound on the possible values of the first row and column as $(2n-1)^{n-1}$ and $(2m-1)^{m-1}$ respectively. This is still an over-estimation. We can also prove that difference between the value of any two adjacent cells (along row, column or diagonal) can be atmost 1. Therefore by encoding the elements in the row as the offset from its left neighbour and in the column as the offset from its top neighbour (figure 2.4) the possible combinations for rows and columns comes down to 3^{n-1} and 3^{m-1} . The possible substrings of length $m+n-2$ are $|\Sigma|^{m+n-2}$. Therefore the maximum index can be $(3|\Sigma|)^{m+n-2}$. After calculating these values for all the blocks, follow any chain of blocks from $(0,0)$ to (l_1, l_2) and add the values of the bottomright cell obtained from the lookup table (with the topleft entry as 0) to obtain the edit distance between $String_1$ and $String_2$.

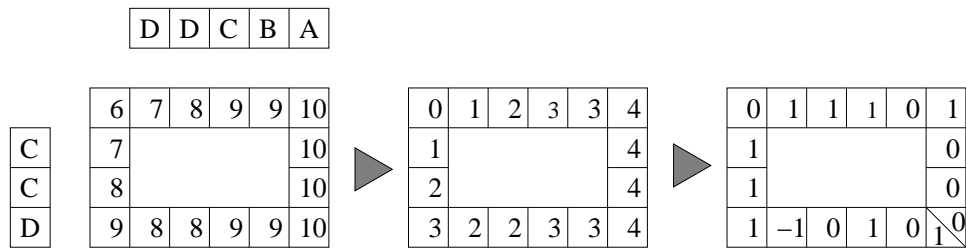


Figure 2.4: Offset Encoding for Four-Russians speedup

In our implementation, we have chosen a block size of 4×4 . By using the datatype short for the values of right and left columns (the value of the bottom-right cell can be obtained from these), size of the lookup table becomes $(3 * 4)^{4+4-2} * 2 * size(short) \approx 12MB$. The above description assumed that the table can be fully covered by the $m \times n$ blocks. This will not

always be the case. For a 4×4 block, we can always obtain the edit distance if the lengths are like $l_1 = 4k_1 - c$, $l_2 = 4k_2 - c$, by appending any substring (same for both) of length c to the two strings. When this is not the case, we can either additionally create lookup table for 2×4 blocks to cover the remaining part, or do complete dynamic programming for the remaining part. However, even without doing all this we can get an edit distance which can be atmost 2 off from the correct edit distance by just using the 4×4 blocks.

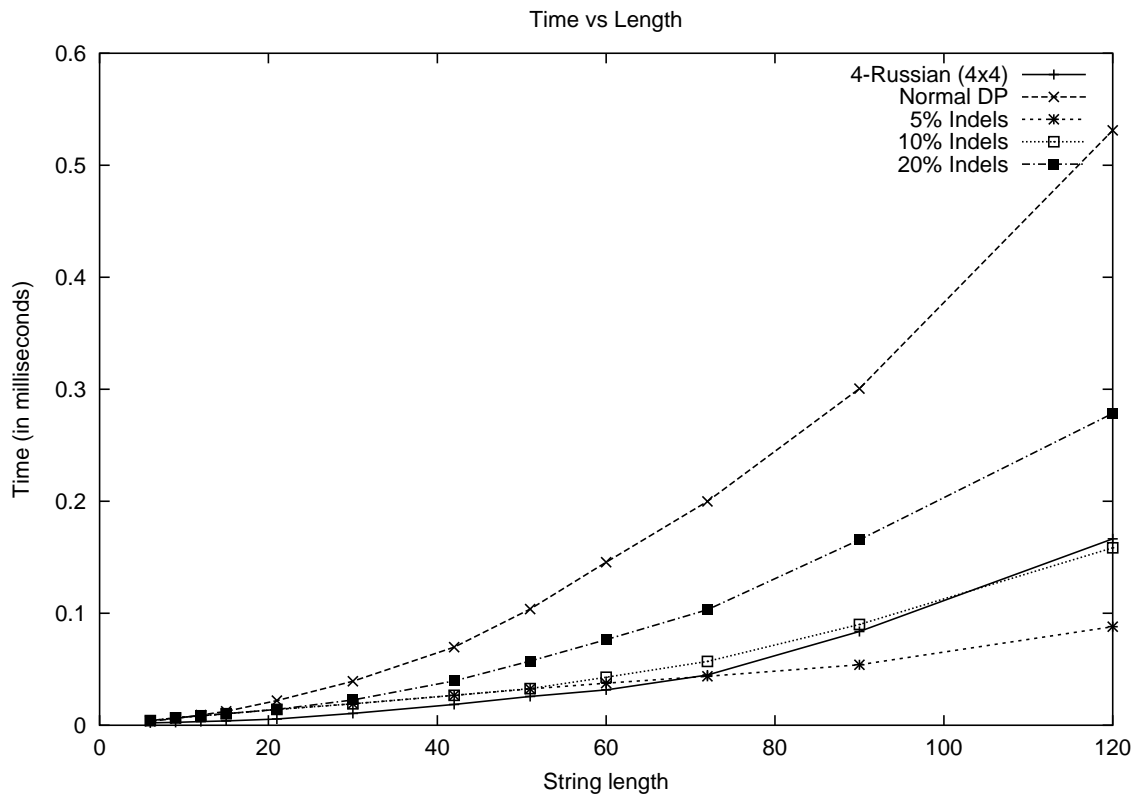


Figure 2.5: Timing Comparison for different dynamic programming modifications

A comparison of the times of the different methods of dynamic programming is shown in Figure 2.5 (for this figure length of both the strings was taken same, as shown on X-axis). 10-15% indel rates have been found to give sufficiently accurate edit distances. As compared to the times with these rates, the Four-Russians speedup is faster. Also its running time does not increase if the difference in the gap lengths is large (because of the $|l_2 - l_1|$ term in the size of

the dynamic programming table, for equal values of $l_1 l_2$ the cells in the table increase as $|l_2 - l_1|$ increases). By combining the two approaches we can get substantial speedups for longer gaps. The current implementation only uses the Four-Russians speedup as described above. By taking precautions in implementation, the running times can be further brought down. The dynamic programming table should be created only once and re-used for subsequent calls. A new table should be created only when the size of existing table is insufficient. In the four-russians speedup, while calculating the indices for the lookup table, it is better to use bit-shifts than multiplications. This can be easily achieved for DNA sequences where the alphabet size is 4. Our implementation (currently only for DNA sequences) uses bit shifts to calculate the indices.

3 Experimental Results

For the purpose of checking the performance of the scheme various experiments were conducted. The results were interpreted in terms of the quality of the output and the time taken for computation. In this chapter we provide a detailed description and analysis of the experimental work undertaken.

3.1 Modules Developed

The first task was to code the various algorithms. The coding was done in C++ and on a Linux platform. The various modules implemented were,

- *Extension Algorithm*: The algorithm developed for extension of maximal repeats for finding approximate repeats was coded.
- *Edit Distance calculation*: The methods for calculating the edit distance of the gaps were implemented.

In addition to these, modules for linear time computation of suffix arrays and maximal repeats were developed. However, the *repsfind* [10] module was used to calculate the maximal repeats for large strings as it was more efficient than the suffix array implementation. Also, the *repvis* [10] module was used to graphically display the results.

3.2 Experiments conducted

The scheme was tested on sequences ranging from 4000 to 5 million base pairs in length. The program was executed for various error bounds ranging from 5% to 30%. Also, the minimum size of seeds computed was varied for each string (keeping the error rate fixed) and the results tabulated. Experiments were also conducted to check the effect of changing the *depth* parameter by taking 0.5 times and 2 times the value obtained by the formula presented in Section 2.5.1. The *min_depth* parameter is fixed to 20 in the experiments. The sequences and their lengths are given below.

Sequence name	amoeba1.txt [20]	contig2.txt [20]	chr7.fa [19]	chr15.fa [19]	chr4.fa [19]	batchseq.cgi [20]
No. of Base Pairs	4016	52500	1109128	1109475	1557447	5301960

Table 3.1: Size of the sequences and the minimum seed size used

3.3 Results

The results obtained were tabulated. Table 3.2 gives the results obtained by varying the minimum seed size for each sequence, keeping the error rate fixed. The results include the number of repeats found, length of the longest repeat found, and the time taken for execution. Table 3.3 gives the results obtained by varying the error rate keeping the minimum seed size fixed for each sequence. It gives the number of approximate repeats found and the length of the longest repeat found. Figure 3.1 is a plot of the timings obtained in this experiment. Table 3.4 contains the breakup of the running time for each sequence at 10% error rate, in terms of the time required to calculate the seeds and the time required for stitching. Table 3.5 has the data obtained for chr4.fa sequence for 15% error rate and minimum seed size 15, by varying the *depth* parameter. The results were obtained on a Intel Pentium IV processor based PC running Linux with 256 MB of RAM.

amoeba1.txt					
Min. seed size	3	4	5	10	15
No. of approx. repeats	14594	3403	902	0	0
Longest repeat length	125	85	36	20	20
Time taken (sec)	18.44	2.93	0.62	0.02	0.01
contig2.txt					
Min. seed size	5	6	8	14	20
No. of approx. repeats	127739	25188	858	6	1
Longest repeat length	370	537	481	90	43
Time taken (sec)	269.23	43.83	2.97	0.13	0.11
chr7.fa					
Min. seed size	10	12	14	20	25
No. of approx. repeats	9149	2593	1049	322	144
Longest repeat length	5799	4542	4542	4928	4673
Time taken (sec)	37.96	7.28	4.55	4.13	4.05
chr15.fa					
Min. seed size	10	12	14	20	25
No. of approx. repeats	6696	1952	777	189	79
Longest repeat length	6539	5894	5894	5894	5894
Time taken (sec)	36.79	6.97	4.51	4.11	4.04
chr4.fa					
Min. seed size	11	13	15	20	25
No. of approx. repeats	8037	2982	1706	996	549
Longest repeat length	11500	11500	11500	11473	11473
Time taken (sec)	27.12	9.93	8.04	6.96	6.10
batchseq.cgi					
Min. seed size	15	17	18	24	30
No. of approx. repeats	1500	1147	1030	637	516
Longest repeat length	5782	5753	5711	5711	5711
Time taken (sec)	24.76	22.75	22.45	22.38	22.19

Table 3.2: Experimental results obtained by varying the minimum seed size for each string at 15% error rate.

3.4 Analysis of Results

The data shown in Table 3.2 is along expected lines. The number of approximate found decreases as the minimum seed size is increased, due to the fact that the number of seeds decreases. Also, the running time decreases because less time is required for the computation of

Error Bound	5%	10%	15%	20%	30%
amoeba1.txt(5)					
No. of repeats found	15609	16091	16511	17284	19616
Longest repeat length	20	32	36	52	90
contig2.txt(8)					
No. of repeats found	95748	96266	96581	97267	99511
Longest repeat length	68	435	481	556	663
chr7.fa(14)					
No. of repeats found	18033	18462	18746	19026	19500
Longest repeat length	3479	4542	4542	4542	5655
chr15.fa(14)					
No. of repeats found	14646	14960	15180	15403	15685
Longest repeat length	5894	5894	5894	5894	5894
chr4.fa(15)					
No. of repeats found	15655	16166	16438	16693	17128
Longest repeat length	7067	7067	11500	11500	11500
batchseq.cgi(18)					
No. of repeats found	5634	5790	5871	5921	6012
Longest repeat length	5287	5387	5711	5782	6142

Table 3.3: Experimental results obtained by varying the error rate. The number in brackets is the seed size used for each string. Number of repeats includes the seeds computed.

Sequence name	amoeba1.txt	contig2.txt	chr7.fa	chr15.fa	chr4.fa	batchseq.cgi
Time for Calculating maximal repeats(sec)	0.02	0.3	3.0	3.1	5.0	21.0
Time for Stitching(sec)	0.6	2.6	1.5	1.4	1.3	1.4

Table 3.4: Breakup of average running time for 10% error bound

the seeds as they decrease in number, and the time required for extension also decreases due to the fact that there are lesser number of seeds to be extended. Generally we observe that the length of the longest repeat found also decreases with increase in minimum seed size. However, there are some exceptions. These occur due to the fact that stitching a long repeat with a smaller repeat may result in a repeat with large number of errors. This results in decreased *depth* parameter, and the repeat may not extend further.

As explained earlier, the percentage error problem has not been tackled previously. Hence,

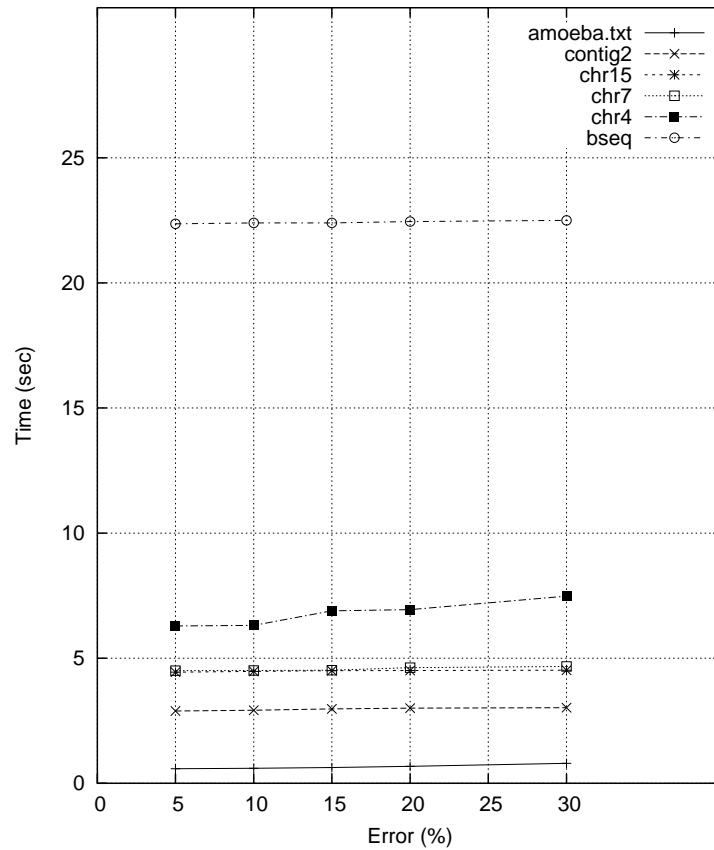


Figure 3.1: Average running time graph for Table 3.3

we did not have any benchmark on which to base the analysis of the results. However, Table 3.3 gives us some idea of the performance of the scheme. As expected, the number of repeats found rises with the increase in the error bound. It is also seen that the increase in the number of repeats is not very substantial and is marginal in some cases. This can be attributed to the fact that most long repeats that are biologically significant do not contain large number of errors. This is also supported by the observation that the length of the longest repeat found too does not change by a large amount as the error rate increases.

To check for the quality of long repeats that were obtained, we manually checked the portions of strings that gave the longest repeat using *repfind* and *repvis* for the existence of long repeats. Our results were corroborated by the results obtained by this manual checking.

Value of <i>depth</i>	$d/2$	d	$2 \times d$
No. of approx. repeats	1530	2982	1784
Length of longest repeat	7067	11500	11500
Time taken(sec)	6.55	8.04	10.23

Table 3.5: Results obtained by varying the *depth* parameter for chr4.fa with 15% error rate and minimum seed size 15. Here d is the value obtained by the formula in Section 2.5.1

The timing results are interesting as the running time of the program shows almost no variation with the increase in the error bound. This is due to the fact that there is only a small increase in the number of repeats calculated with increase in the error bound. Also, for long sequences the most time consuming step of the scheme is the computation of the maximal repeats. The actual time taken for stitching is relatively small for such sequences. This can be seen in Table 3.4 which shows the breakup of the running time for 10% error bound.

This also suggests that the time taken is also linked to the number of maximal repeats. This is also a reason why it is important to limit the number of seeds in long sequences. Taking into account the size of the sequences the running time of the scheme, especially the stitching process, appears to be satisfactory.

Expectedly, the number of repeats found increases as the *depth* parameter is increased as shown in Table 3.5. The size of the longest repeat increases and then remains constant. This depends to a large extent on the sequence. Also, the time for execution increases as the *depth* increases as the amount of search and stitching done increases.

4 The Generalized Hamming-Distance Problem and Fast Fourier Transform

4.1 Motivation

In Chapter 2 we analysed the problem of finding approximate repeats. An equally important problem in computational genomics is of finding approximate occurrences of a pattern P in a database sequence T . A lot of work has been done on this problem, [23, 24, 25, 26, 27, 28, 29] to cite a few. One approach to solve this problem is: for every position i in P and T , group q consecutive characters $P[i..i+q-1]$ (equivalently, $T[i..i+q-1]$) and make this the new alphabet ${}^n\Sigma = \{{}^n\sigma_1, \dots, {}^n\sigma_{|\Sigma|^q}\}$ (Σ being the original alphabet). In this alphabet the characters do not simply match or mismatch. Instead, we can define a distance function for this new alphabet, $distance({}^n\sigma_i, {}^n\sigma_j)$, which can, for example, be the edit distance of the q length sequences ${}^n\sigma_i$ and ${}^n\sigma_j$ represent. Now we need to find the occurrences of the modified pattern in the modified string so that their distance (a kind of hamming distance on ${}^n\Sigma$) is below a threshold t .

It still remains to be seen whether this approach has any chance of solving the approximate matching problem, but it gives rise to another interesting problem. Given pairwise distances between characters of an alphabet, to find the distance between the pattern P and

the $|P|$ length substring $T[i \dots i + |P| - 1]$, for every alignment position i in T . In addition to the above application, non- $\{0,1\}$ distances occur elsewhere too. In biological sequences, the probability of substitution of one unit (base pair/amino acid) by another is not the same for all pairs. For DNA sequences, transitions ($A \leftrightarrow G$, $T \leftrightarrow C$) occur more often than transversions (purine \leftrightarrow pyrimidine) [30]. Some other DNA substitution models are [31, 32, 33, 34, 35]. PAM [37, 36] and BLOSUM ([38]) matrices are the more commonly used substitution matrices for proteins. By allowing the inter-character distances in $[0, 1]$ such alphabet can be handled. In addition, it provides a more elegant and flexible way of dealing with wild cards by augmenting the alphabet with characters representing the wild cards and setting the distances between the wild card and the characters it matches as 0. In the next few sections we formalize the problem and discuss previous similar work and their limitation. The remaining chapter gives the details of our approach.

4.2 Notation and Problem Definition

Denote the alphabet set by $\Sigma = \{\sigma_1, \sigma_2 \dots \sigma_{|\Sigma|}\}$. The distance matrix \mathbf{D} , is a $|\Sigma| \times |\Sigma|$ matrix where $D(i, j) = d_{ij} = \text{distance}(\sigma_i, \sigma_j)$. The similarity matrix \mathbf{S} is also a $|\Sigma| \times |\Sigma|$ matrix where $\mathbf{S}(i, j) = s_{ij} = 1 - d_{ij}$.

Definition 4.1 The generalized hamming distance between two strings $S_1[0 \dots l - 1]$ and $S_2[0 \dots l - 1]$ is defined as

$$d_g(S_1, S_2) = \sum_{i=0}^{l-1} \text{distance}(S_1(i), S_2(i)).$$

However, instead of finding the dissimilarity it is convenient to find the similarity

$$s_g(S_1, S_2) = \sum_{i=0}^{l-1} \{1 - \text{distance}(S_1(i), S_2(i))\}.$$

The problem can be formally stated as calculation of the real-valued array MC^l where $MC^l(i) =$

$s_g(T[i \dots i + |P| - 1], P)$, the similarity score when the leftmost character of P is aligned with the i th character of T .

4.3 Previous Work

Let us take the simple case of 0/1 distances corresponding to matches/mismatches. Then the above problem reduces to finding the number of matches between the two sequences at all possible alignments. Gusfield [8] calls it the *Match-Count Problem*. Solution for this problem requiring $O(|T| \log |T|)$ arithmetic operations was developed by Fischer and Paterson [39] and by Felsenstein, Sawyer and Kochin [40]. The method is based on Fast Fourier Transform and is described below.

We can break the above problem as a sum of $|\Sigma|$ subproblems. The i th subproblem is to calculate the number of matches due to σ_i in the two sequences. This can be solved by creating two new sequences T_i and P_i where

$$T_i[j] = \begin{cases} 1 & \text{if } T[j] = \sigma_i \\ 0 & \text{otherwise.} \end{cases}$$

and

$$P_i[j] = \begin{cases} 1 & \text{if } P[j] = \sigma_i \\ 0 & \text{otherwise.} \end{cases}$$

Now if we convolve P_i and T_i (using FFT), we get the the number of matches of σ_i for all possible alignments of T and P . If we solve all the $|\Sigma|$ subproblems and add their result, we get the matches due to all the characters for all possible alignments. Felsenstein et al. [40] also propose a solution to deal with missing characters. For example, if we just know that the character at position j is a *purine*, then we set that position to 0.5 for the subproblems for A and G and 0 for C and T. Gusfield presents a method to deal with wildcards by setting 1 for that position in T_{i_1}, \dots, T_{i_m} where $\sigma_{i_1}, \dots, \sigma_{i_m}$ are the characters it can match. If both the strings have wild cards then the similarities will be over estimated (for example when two *match-all*

wild cards are against each other, the similarity for that position would be calculated as $|\Sigma|$ instead of 1) and another step is required to correct this . However, he does not say how this can be done if there are more than one kind of wild cards.

We will now describe the method to solve the arbitrary-distance version of the Match-Count problem with the above intuition. Again break the problem into $|\Sigma|$ subproblems (here every kind of wildcard/missing character is treated as a character in Σ). Form the sequence T_i by replacing occurrences of σ_i by 1 and every other character by 0 as before. The sequence P_i , however, is formed by setting $P_i[j] = s_{ik}$ when $P[j] = \sigma_k$. Solution of the i th subproblem gives the similarity due to σ_i in T and all characters in P for all alignments. The sum of the solutions of all $|\Sigma|$ subproblems give the similarity due to all characters in T and P for all alignments.

The number of operations can be reduced to $O(|T| \log |P|)$ (as described in the next section), but there is also an implicit $|\Sigma|$ factor in the complexity because FFT of $2|\Sigma|$ sequences has to be taken. If $|\Sigma|$ is large, finding the similarities for all alignments might become expensive and one may want to first get a rough estimate of the similarities at a lower cost and then refine that estimate. By the above idea of using one sequence per character in the alphabet, it is not possible to achieve the lowering of cost. We describe an alternate way to look at the procedure of calculation of similarity by embedding the alphabet in $|\Sigma|$ dimensional space. The cost reduce the number of dimensions retained for FFT step thereby reducing the cost but introducing errors. We then propose methods to embed the alphabet so that these errors are minimised.

Other work that builds on the approach of Fischer and Paterson [39] and Felsentein et al. [40] is found in [41], [42], [43] and [44].

4.4 Using FFT to Calculate MC'

First we review the use of Fast Fourier Transform for convolution. Let A and B be sequences over \mathbb{C} (the set of complex numbers) of length n with U and V as their discrete Fourier trans-

forms. Convolution of A and B , denoted $A * B$, is a sequence C , where

$$C(i) = \sum_{j=0}^{n-1} A(j)B(i+j)$$

with indices taken modulo n . C has discrete Fourier transform Z where $Z(j) = U(j)V^*(j)$, V^* indicating complex conjugate of V . If complex conjugate is not taken then reverse of B will be convolved.

Now, let A and B be sequences over \mathbb{C}^d , that is, each element of A and B is a d -dimensional vector. We can divide A and B into d pairs of sequences (A_i, B_i) , $i = 1, \dots, d$. Let $C_i = A_i * B_i$ and $C = \sum_{i=1}^d C_i$. C is again a sequence over \mathbb{C} which can alternately be written as

$$C(i) = \sum_{j=0}^{n-1} A(j) \cdot B(i+j),$$

dot indicating inner product of two vectors.

Convolution of d sequences actually gives the sum of the inner products of the vectors. What was being done in section 4.3 was: since for T there is 1 for σ_i in only T_i , the similarity due to σ_i can be obtained by just convolving T_i with P_i . We can use this knowledge to calculate MC' as follows: If we can find a d -dimensional vector representation X_i for each character σ_i in Σ such that $X_i \cdot X_j = s_{ij}$, then the convolution of sequences formed from T and P by replacing the characters by their vector representation gives the required similarity. Formally, replace the characters of T and P as above. Pad $|P|$ zero vectors at the end of T and $|T|$ zero vectors at the end of P . This eliminates illegal wraparounds of T and P . Fast radix-2 FFT algorithms are available when sequence lengths are power of two. Extend the $|T| + |P|$ length sequences obtained by more zero vectors to make length a power of two. Following the convention of section 4.3, let us split T into d sequences T_1, \dots, T_d . Do the same for P . Let U_i be the discrete Fourier transform of T_i and V_i be the transform of P_i . Calculate Z_i ($i = 1, \dots, d$) where $Z_i = U_i V_i^*$ (if matching in forward direction). Let $Z = \sum_{i=1}^d Z_i$. Taking the inverse Fourier transform of Z gives MC' . This algorithm has a running time of $O(|T| \log |T|)$.

A very simple windowing scheme can reduce the running time of above to $O(|T| \log |P|)$. For simplicity, assume $|P|$ is a power of two (extension to the general case is straightforward). Create windows, W_i , of size $2|P|$ from T by moving a mask of size $2|P|$ over T in steps of $|P|$. The i th window is

$$W_i = T[(i-1)|P| \dots (i+1)|P| - 1] \quad i = 1, \dots, \left\lceil \frac{|T|}{|P|} \right\rceil$$

Extend P by adding $|P|$ zero vectors (instead of $|T|$ zero vectors). Perform convolution of the windows with extended P as before to obtain $MC'_i = W_i * P$. Now $MC'_i[(i-1)|P| \dots i|P| - 1] = MC'_i[0 \dots |P| - 1]$. The running time is $O(\frac{|T|}{|P|} |P| \log |P|) = O(|T| \log |P|)$. The reason for choosing the above windowing scheme is that any $|P|$ length substring of T should be *completely covered* by atleast one window so that the similarity score for that alignment can be obtained. By choosing a window size of w , we can obtain scores for $(w - |P|)$ alignments from each window and $\frac{|T|}{w - |P|}$ windows need to be created.

Rest of the chapter explains different methods to obtain the vector representation of the alphabet and reducing the dimensionality of the embedding.

4.5 $|\Sigma|$ -Dimensional Embeddings

As mentioned before, our aim is to have a vector representation X_i for each character σ_i in Σ such that $X_i \cdot X_j = s_{ij}$. This can also be written as

$$\mathbf{X}^T \mathbf{X} = \mathbf{S}$$

where $\mathbf{X} \in \mathfrak{R}^{d \times |\Sigma|}$ whose i th column is X_i . Let us assume that such an \mathbf{X} exists. Then all X_i represent points on the surface of a d dimensional sphere of unity radius. One method to obtain \mathbf{X} can be to rotate that sphere to force X_1 to lie along the first axis, i.e. only the first component is non-zero. This fixes the first component of all the X_i 's as s_{1i} . Now fix X_2 so

that only the first two components are non-zero. This fixes the second component of all X_i 's. This way a $|\Sigma|$ dimensional embedding can be obtained (Algorithm *simple_embed*). This procedure can be visualised as reducing the degree of freedom of rotation of the sphere by one every iteration. The operations in this algorithm turn out to be identical to those in Cholesky Decomposition of a symmetric positive definite matrix. The time complexity of this algorithm is $O(|\Sigma|^3)$.

Algorithm *simple_embed*

```

Initialize  $X_i = \underline{0}$ 

for  $i = 1$  to  $|\Sigma|$ 
    for  $j = 1$  to  $i - 1$ 
         $X_i(j) = \left[ \frac{s_{ij} - X_i \cdot X_j}{X_j(j)} \right]$ 
    end
     $X_i(i) = \sqrt{1 - X_i \cdot X_i}$ 
end

```

Under some restrictions such an embedding indeed exists, as proved below.

Lemma 4.1 If the following conditions are satisfied

1. Self-similarity: $d_{ii} = 0 \quad \forall i$
2. Bounded distances: $0 \leq d_{ij} \leq 1 \quad \forall i, j$
3. Triangle Inequality: $d_{ij} \leq d_{ik} + d_{jk} \quad \forall i, j, k$

then an embedding \mathbf{X} satisfying $\mathbf{X}^T \mathbf{X} = \mathbf{S}$ can be found.

Proof: The proof is based on the construction of one such embedding. Let $D_{ij} = \sqrt{2d_{ij}}$.

Since d_{ij} satisfy triangle inequality, D_{ij} also satisfy triangle inequality.

$$(D_{ik} + D_{jk})^2 \geq D_{ik}^2 + D_{jk}^2 = \frac{d_{ik} + d_{jk}}{2} \geq d_{ij}/2 = D_{ij}^2.$$

In fact, D_{ij} satisfy a stronger constraint. Since $D_{ik}^2 + D_{jk}^2 \geq D_{ij}^2$, the triangles will have only acute angles. Given the inter-point distances between n points (satisfying triangle inequality), an $n - 1$ dimensional embedding satisfying these can be found (*Multidimensional Scaling* [45]). After finding this embedding, we find the circumcenter C of these points. If no more than m points lie on an $(m - 1)$ dimensional hyperplane ($m = 1, \dots, n - 1$), then such a point always exists (otherwise, the circumcenter may become a point at infinity). It can be shown that when this condition is not satisfied, at least one angle will be obtuse. Moreover, as mentioned above, all angles are acute and therefore the circumcenter can be found. (To be exact, more than m points can lie on a $m - 1$ dimensional plane with some angles being right angles, but in this case too the circumcenter can be found. For example, for a rectangle in a 2D plane). Translate the embedding obtained so that the circumcenter coincides with the origin. The points now lie on the surface of a $|\Sigma| - 1$ dimensional sphere of some radius R . Add a $|\Sigma|$ th dimension to all the vectors setting it as $\sqrt{1 - R^2}$. This does not change the inter-point distances. Now all the vectors satisfy $\|X_i\| = 1$ and $\|X_i - X_j\| = \sqrt{2d_{ij}}$. All that is left to prove is $X_i \cdot X_j = s_{ij}$. Using cosine formula,

$$\begin{aligned} X_i \cdot X_j &= \frac{\|X_i\|^2 + \|X_j\|^2 - \|X_i - X_j\|^2}{2} \\ &= 1 - d_{ij} \\ &= s_{ij} \end{aligned}$$

□

Example: Embeddings for a distance matrix, \mathbf{D} , using the methods described

$$\mathbf{D} = \begin{array}{|c|c|c|c|} \hline 0 & 0.4 & 0.7 & 0.8 \\ \hline 0.4 & 0 & 0.3 & 0.4 \\ \hline 0.7 & 0.3 & 0 & 0.4 \\ \hline 0.8 & 0.4 & 0.4 & 0 \\ \hline \end{array}$$

$$\mathbf{X} \text{ (using Algorithm } \textit{simple_embed}) = \begin{array}{|c|c|c|c|} \hline 1 & 0.6 & 0.3 & 0.2 \\ \hline 0 & 0.8 & 0.65 & 0.6 \\ \hline 0 & 0 & 0.698 & 0.215 \\ \hline 0 & 0 & 0 & 0.744 \\ \hline \end{array}$$

$$\mathbf{X} \text{ (using construction of proof of Lemma 4.1)} = \begin{array}{|c|c|c|c|} \hline 0.663 & -0.006 & -0.388 & -0.555 \\ \hline 0.000 & 0.217 & 0.542 & -0.325 \\ \hline 0.080 & 0.632 & 0.041 & 0.179 \\ \hline 0.744 & 0.744 & 0.744 & 0.744 \\ \hline \end{array}$$

There is a much simpler method to find the vector representations: if instead of having the same embedding for characters in the pattern sequence and the database string we embed the characters in one using \mathbf{X}_1 and in the other using \mathbf{X}_2 so that $\mathbf{X}_1^T \mathbf{X}_2 = \mathbf{S}$. Now setting \mathbf{X}_1 (or \mathbf{X}_2) to \mathbf{I} , the identity matrix, and \mathbf{X}_2 (or \mathbf{X}_1) to \mathbf{S} gives the desired result. This is identical to the generalization proposed in section 4.3. The advantage of this *asymmetric* embedding is that it does not require the distances to satisfy the triangle inequality or even that they be bounded or self-similar. Situations where this can be important are

- When the alphabet contains wild-cards
- Searching for complement of the pattern P

Example: Transformation of sequences using the asymmetric transformation described

String : ABDDDBCAACCDB

1	0	0	0	0	0	0	1	1	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0	0	1
0	0	0	0	0	0	1	0	0	1	1	0	0
0	0	1	1	1	0	0	0	0	0	0	1	0

Pattern: ABBBDC

1	0.6	0.6	0.2	0.3
0.6	1	1	0.6	0.7
0.3	0.7	0.7	0.6	1
0.2	0.6	0.6	1	0.6

4.6 Low Dimensional Embeddings for Approximate Similarity

Though the time complexity of the algorithm is mentioned as $O(|T| \log |P|)$, it actually is $O(|\Sigma| |T| \log |P|)$ because FFT of $2|\Sigma|$ sequences has to be taken for every window. It is worthwhile searching for methods which can reduce the $|\Sigma|$ factor while not changing the answers significantly. The aim of this and the next section is to find the best d ($< |\Sigma|$) dimensional representation of the alphabet.

When the distances satisfy the constraints of Lemma 4.1, the matrix of similarities \mathbf{S} has a structure similar to a correlation matrix. In such cases \mathbf{S} is a symmetric positive definite matrix (as also proved in Lemma 4.1, it can be written as $\mathbf{X}^T \mathbf{X}$). Therefore it can be factorized using Singular Value Decomposition as

$$\mathbf{S} = \mathbf{U} \mathbf{R} \mathbf{U}^T,$$

where \mathbf{U} is an orthogonal matrix whose columns are the eigenvectors of S and R is a diagonal matrix containing the singular values of S written in decreasing order so that $R(1,1)$ is the largest. Let \mathbf{U}_d be the first d columns of \mathbf{U} and \mathbf{R}_d be the d -by- d matrix formed from first d rows and columns of \mathbf{R} . The d -rank approximation of \mathbf{S} , \mathbf{S}_d , that minimises $\|\mathbf{S} - \mathbf{S}_d\|_2$ is

$$\mathbf{S}_d = \mathbf{U}_d \mathbf{R}_d \mathbf{U}_d^T = \mathbf{U}_d \sqrt{\mathbf{R}_d} \sqrt{\mathbf{R}_d} \mathbf{U}_d^T = \left[\sqrt{\mathbf{R}_d} \mathbf{U}_d^T \right]^T \left[\sqrt{\mathbf{R}_d} \mathbf{U}_d^T \right] = \mathbf{X}^T \mathbf{X}$$

Therefore the d dimensional embedding of alphabets is $\mathbf{X} = \left[\sqrt{\mathbf{R}_d} \mathbf{U}_d^T \right]$.

Again, we must consider the case when distances do not satisfy triangle inequality. Though \mathbf{S} may not always be positive definite but we can still write the Singular Value Decomposition

$$\mathbf{S} = \mathbf{U} \mathbf{R} \mathbf{V}^T,$$

where U and V are orthogonal matrices. Columns of U are eigenvectors of AA^T and columns of V are eigenvectors of $A^T A$. R is a diagonal matrix with entries in decreasing order of value (singular values are always positive). Let \mathbf{U}_d and \mathbf{V}_d be the first d columns of \mathbf{U} and \mathbf{V} respectively and \mathbf{R}_d be the first d -by- d block of \mathbf{R} as before. The d rank approximation of \mathbf{S} , \mathbf{S}_d , that minimises $\|\mathbf{S} - \mathbf{S}_d\|_2$ can be written as

$$\mathbf{S}_d = \mathbf{U}_d \mathbf{R}_d \mathbf{V}_d^T = \mathbf{U}_d \sqrt{\mathbf{R}_d} \sqrt{\mathbf{R}_d} \mathbf{V}_d^T = \left[\sqrt{\mathbf{R}_d} \mathbf{U}_d^T \right]^T \left[\sqrt{\mathbf{R}_d} \mathbf{V}_d^T \right] = \mathbf{X}_1^T \mathbf{X}_2$$

Therefore asymmetrically transforming one sequence using $\left[\sqrt{\mathbf{R}_d} \mathbf{U}_d^T \right]$ and the other using $\left[\sqrt{\mathbf{R}_d} \mathbf{V}_d^T \right]$ this case can be handled.

The time complexity of SVD for \mathbf{S} is $O(|\Sigma|^3)$, but this cost is incurred just once. The number of dimensions to be retained can be decided using a plot similar to the *scree-plot* used in Principal Component Analysis (essentially, a plot of the diagonal entries of \mathbf{R}). As an example, figure 4.1 shows the scree plot for the similarity matrix S of an alphabet whose characters are 4-grams of an alphabet of size 4 (as motivated in Section 4.1) and the distances between them are the edit distances of the sequences they represent. To determine the appropriate number of

dimensions, we look for an elbow in the scree plot. The number of dimensions is taken to be the point at which the remaining eigenvalues are relatively small and all about the same size [47]. In Figure 4.1 such an elbow occurs at $i = 17$, and therefore implies a dimensionality of $d = 16$. If the similarity matrix is an identity matrix then all the eigenvalues are equal ($= 1$) and no dimensionality reduction can be applied.

It is not possible to estimate the errors occurring due to the approximation because errors at different positions may cancel or add up. However, we can obtain a bound on the error in approximation. Let us say we use \mathbf{X}_1 to transform the pattern $P = \sigma_{i_1}\sigma_{i_2}\cdots\sigma_{i_{|P|}}$. Define the error matrix $\mathbf{E} = \mathbf{S} - \mathbf{X}_1^T \mathbf{X}_2$. Let e_{\min_i} and e_{\max_i} be the smallest and the largest values in the i th row of \mathbf{E} respectively. If the actual similarity is s , then the similarity obtained using \mathbf{S}_d lies in the interval $[s - \sum_{j=1}^{|P|} e_{\min_{i_j}}, s + \sum_{j=1}^{|P|} e_{\max_{i_j}}]$.

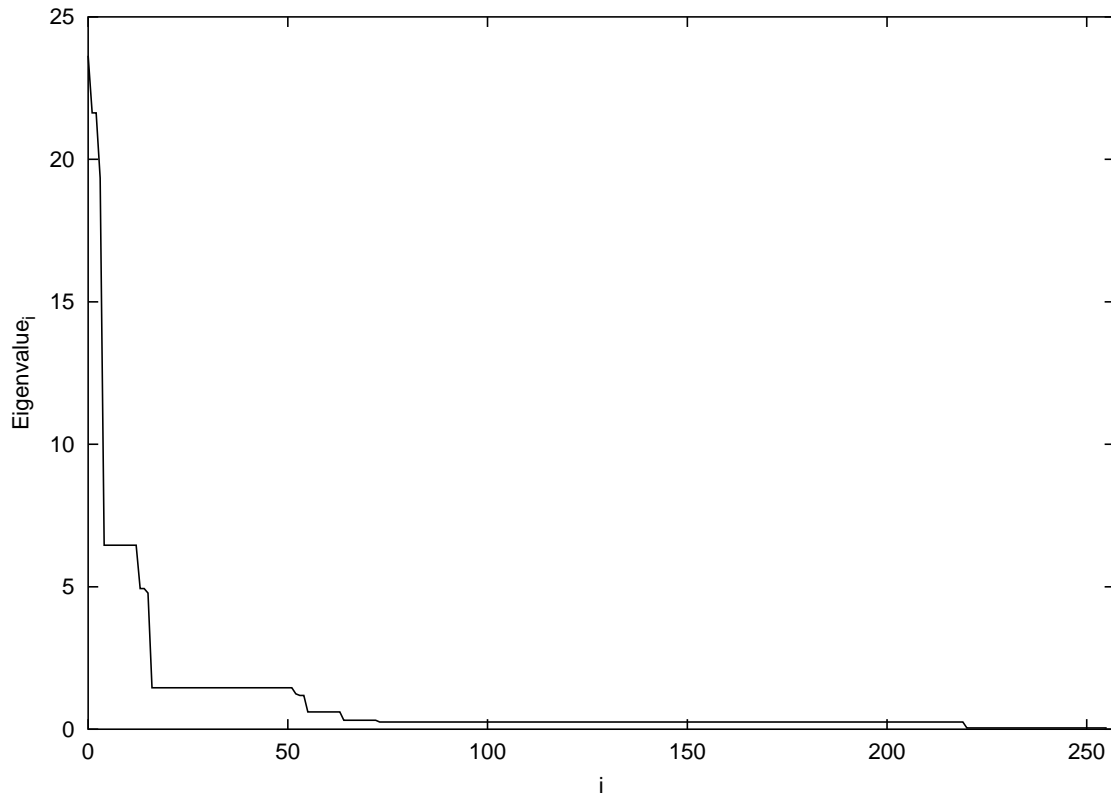


Figure 4.1: Scree plot for an alphabet formed from 4-grams on an alphabet Σ ($|\Sigma| = 4$)

4.7 More Low-Rank Approximations of \mathbf{S}

Any rank d matrix $\mathbf{A}_d \in \mathfrak{R}^{m \times n}$ can be written as $\mathbf{A}_d = \mathbf{U}\mathbf{V}^T$, where $\mathbf{U} \in \mathfrak{R}^{m \times d}$ and $\mathbf{V} \in \mathfrak{R}^{n \times d}$. Finding a low rank approximation can be thought of as minimization of some objective function $J(\mathbf{U}, \mathbf{V})$ over the pair (\mathbf{U}, \mathbf{V}) . The objective function considered in section 4.6 was

$$\begin{aligned} J_1(\mathbf{U}, \mathbf{V}) &= \sum_{i,j} (\mathbf{A}(i,j) - \mathbf{A}_d(i,j))^2 \\ &= \sum_{i,j} \left(\mathbf{A}(i,j) - \sum_k \mathbf{U}(i,k)\mathbf{V}(j,k) \right)^2, \end{aligned}$$

which seeks to minimize the sum of the squares of the difference. We can also consider an objective function which minimizes the sum of the absolute value of the difference.

$$\begin{aligned} J_2(\mathbf{U}, \mathbf{V}) &= \sum_{i,j} |\mathbf{A}(i,j) - \mathbf{A}_d(i,j)| \\ &= \sum_{i,j} \left| \mathbf{A}(i,j) - \sum_k \mathbf{U}(i,k)\mathbf{V}(j,k) \right|. \end{aligned}$$

The partial derivatives of J_2 are

$$\begin{aligned} \frac{\partial J_2}{\partial \mathbf{U}} &= \{ \text{sign}(\mathbf{U}\mathbf{V}^T - \mathbf{A}) \} \mathbf{V}, \text{ and} \\ \frac{\partial J_2}{\partial \mathbf{V}} &= \{ \text{sign}(\mathbf{V}\mathbf{U}^T - \mathbf{A}^T) \} \mathbf{U}. \end{aligned}$$

Since the objective function is not even differentiable (and hence the partial derivatives are discontinuous) it is very difficult to optimize this function and the convergence and optimality are not guaranteed. A few iterations of updating \mathbf{U}, \mathbf{V} using steepest descent and then updating the partial derivatives can give good approximation if the starting point is suitable. A sensible way to choose the starting point is the SVD-based d rank approximation. However, in practice minimizing J_2 does not always give any better results when calculating similarity than J_1 (in fact, very often the results degrade).

An objective function which actually does give better results is the weighted d rank approximation [46]. The objective function

$$\begin{aligned} J_3(\mathbf{U}, \mathbf{V}) &= \sum_{i,j} \mathbf{W}(i,j) (\mathbf{A}(i,j) - \mathbf{A}_d(i,j))^2 \\ &= \sum_{i,j} \mathbf{W}(i,j) \left(\mathbf{A}(i,j) - \sum_k \mathbf{U}(i,k) \mathbf{V}(j,k) \right)^2 \end{aligned}$$

minimizes the weighted Frobenius distance. Since the frequency of characters in the pattern or the string may not be uniform, but can be skewed, finding a low rank approximation on the basis of these frequencies gives lower errors. Here, even if the similarity matrix is identity, dimensionality reduction can be applied. (If the user wants to calculate the FFT and store it in the database for reusability, then this method cannot be used because of its dependence on pattern). Let the frequency of characters in the pattern be given by f_P and in the database string by f_T ($f_P, f_T \in \mathfrak{R}^{|\Sigma| \times 1}$). We can create the weight matrix \mathbf{W} as either $f_P f_T^T$ by using both the frequencies or as $f_P 1^T$ by using only the frequencies of the pattern. If \mathbf{S}_d is the rank d approximation of \mathbf{S} that optimizes J_3 then the embedding can be obtained by factoring \mathbf{S}_d as $\mathbf{X}_1^T \mathbf{X}_2$. But because of the way weights are specified, \mathbf{X}_1 has to be used for transforming P and \mathbf{X}_2 for T . Srebro and Jaakkola [46] give an efficient Expectation Maximisation based method for finding the weighted low rank approximations. However, when \mathbf{W} is of rank one (as in our case), an SVD based method can be used to minimize J_3 .

Factorize \mathbf{W} as product of two vectors w_1 and w_2

$$\mathbf{W} = w_1 w_2^T$$

Create a new matrix \mathbf{A}' whose (i,j) th element is given by

$$\mathbf{A}'(i,j) = \mathbf{A}(i,j) \sqrt{\mathbf{W}(i,j)}.$$

Now obtain the unweighted rank d approximation, \mathbf{X} , of \mathbf{A}' using SVD. The weighted rank d

approximation, \mathbf{A}_d^w , of \mathbf{A} is given by

$$\mathbf{A}_d^w = \left(\sqrt{\text{diag}(w_1)} \right)^+ \mathbf{X} \left(\sqrt{\text{diag}(w_2)} \right)^+,$$

where $^+$ indicates pseudo-inverse (for a square diagonal matrix it just involves taking the inverse of non-zero entries).

Proof: The matrix \mathbf{X} minimizes the objective function

$$\begin{aligned} f(\mathbf{X}) &= \sum_{ij} \left(\mathbf{A}(i, j) \sqrt{\mathbf{W}(i, j)} - \mathbf{X}(i, j) \right)^2 \\ &= \sum_{i, j} \mathbf{W}(i, j) \left(\mathbf{A}(i, j) - \frac{\mathbf{X}(i, j)}{\sqrt{\mathbf{W}(i, j)}} \right)^2 \end{aligned}$$

Therefore the weighted approximation of \mathbf{A} has (i, j) th element as $\mathbf{X}(i, j) / \sqrt{\mathbf{W}(i, j)}$. This matrix is given by $\left(\sqrt{\text{diag}(w_1)} \right)^+ \mathbf{X} \left(\sqrt{\text{diag}(w_2)} \right)^+$. \square

Here, for choosing the dimensionality, the scree plot of A' should be used.

We could also have a weighted version of the sum of absolute value of difference.

$$\begin{aligned} J_4(\mathbf{U}, \mathbf{V}) &= \sum_{i, j} \mathbf{W}(i, j) |\mathbf{A}(i, j) - \mathbf{A}_d(i, j)| \\ &= \sum_{i, j} \mathbf{W}(i, j) \left| \mathbf{A}(i, j) - \sum_k \mathbf{U}(i, k) \mathbf{V}(j, k) \right| \end{aligned}$$

The partial derivatives in this case become

$$\begin{aligned} \frac{\partial J_4}{\partial \mathbf{U}} &= \{ \mathbf{W} \otimes \text{sign}(\mathbf{UV}^T - \mathbf{A}) \} \mathbf{V} \text{ and} \\ \frac{\partial J_4}{\partial \mathbf{V}} &= \{ \mathbf{W}^T \otimes \text{sign}(\mathbf{VU}^T - \mathbf{A}^T) \} \mathbf{U}, \end{aligned}$$

\otimes indicating element-wise multiplication. As in the case of J_2 , not only is it difficult to optimize this function, the similarities calculated using this are not guaranteed to be any better

than those calculated using the approximation obtained from J_3 .

Practical implementations show that weighted low rank approximations do give lower errors in similarity computation than unweighted, and therefore can work with smaller number of dimensions. However, as mentioned, the reusability of the Fourier transform of T for different patterns is lost. Optimising J_2 and J_4 can take a lot of iterations and does not necessarily improve the accuracy as compared to optimising J_1 and J_3 respectively (the differences in the similarities calculated is marginal).

5 Conclusions

In this project we have attempted to solve the problem of finding approximate repeats in a sequence with a given percentage bound on errors. For this purpose we made an indepth study of the existing work on approximate repeats, especially tools like *RePuter* and *Vmatch*. These tools were designed for the fixed error bound version of the problem, and did not solve the percentage error problem. It was observed that the approximate repeats could be decomposed into a chain of maximal repeats, and we made this observation the basis of a scheme that we have presented for tackling the percentage error problem.

We gave a detailed description of the scheme, in terms of the stages of computation and the various issues faced in designing it. The issues of minimum size of seed to limit the number of maximal repeats, the dynamic search depth and the edit distance calculation of the gaps have been described. As part of the experimental analysis, the scheme was coded and executed with different error bounds and sequences with lengths upto 5 million base pairs. The results show that the scheme is efficient and precise in finding the long approximate repeats which are of greater biological significance. An analysis of the running time of the program shows that the most time consuming stage of the scheme is the computation of the maximal repeats. This underlines the importance of setting the minimum length of the seeds appropriately. In totality, the running time appears satisfactory, considering the size of the sequences.

However, scope for improvement remains. Our current implementation handles only DNA sequences. Issues like the calculation of edit distance of gaps have to be dealt with in a more

systematic manner. One might be able to use the existing data structures to compute the edit distance more efficiently. Also, the minimum size of the seeds remains an issue which has to be analysed. The output also needs to be properly organized. The repeats may be classified according to their structure and position. Such classification would be of use to biologists. Another interesting problem would be to see whether the scheme proposed for approximate repeats can be adapted for the fixed error version of the problem. One can then compare the performance of this modified scheme with that of *RePuter*. This might give us an idea of the viability of extension using stitching of maximal repeats as compared to dynamic programming employed by packages like *RePuter*.

The *Generalized Hamming Distance* problem has been described and the possible applications mentioned. Though a part of the work was found to have been done earlier (and therefore, rediscovered), the approach to obtain similarities as the inner product of the vectors representing the characters enables one to use linear algebra techniques to reduce the cost of computation of similarities, and at the same time, keep the error as low as possible. By also taking into account the frequency of characters in the pattern, the errors can be further reduced.

Bibliography

- [1] Edward P. McCreight. *A Space Economical Suffix Tree Construction Algorithm*. Journal of the Association of Computing Machinery, pp. 263-272, Vol 23. No. 2, 1976.
- [2] Udi Manber and E.W. Myers. *Suffix Arrays: A New Method for On-line String Searches*. SIAM Journal of Computing, **22**(5):935-948, 1993.
- [3] Juha Karkkainen, Peter Sanders. *Simple Linear Work Suffix Array Construction*. ICALP 2003.
- [4] Pang Ko, Srinivas Aluru. *Space Efficient Linear Time Construction of Suffix Arrays*.
- [5] M.I. Abouelhoda, S. Kurtz, and E. Ohlebusch. *The Enhanced Suffix Array and its Applications to Genome Analysis*. Lecture Notes in Computer Science, Springer Verlag, 2002.
- [6] M.I. Abouelhoda, S. Kurtz, and E. Ohlebusch. *Optimal Exact Matching Based on Suffix Arrays*. Lecture Notes in Computer Science, Springer Verlag, 2002.
- [7] T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park. *Linear Time Longest Common Prefix in Suffix Arrays and its Application*. In Proceedings of the 12th Annual Symposium on Combinatorial Patterns and Matching, pp 181-192, LNCS 2089, Springer Verlag, 2001.

- [8] D. Gusfield. *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press, New York, 1997.
- [9] Stefan Kurtz. Reducing the Space Requirement of Suffix Trees. *Software Practice and Experience*, **29**(13):1149-1171, 1999.
- [10] <http://bibiserv.techfak.uni-bielefeld.de/reputer/>
- [11] <http://www.vmatch.de/>
- [12] S. Kurtz, E. Ohlebusch, C. Schleirmacher, J. Stoye, R. Geigerich. *Computation and Visualization of Degenerate Repeats in Complete Genomes*. In Proc. of The International Conference on Intelligent Systems for Molecular Biology, pp 228-238, 2000.
- [13] S. Kurtz, *The Vmatch large scale sequence analysis software - A Manual*. Center for Bioinformatics, Univ. of Hamburg, 2004.
- [14] D. Harel and R.E. Tarjan. *Fast algorithms for finding nearest common ancestors*. *SIAM Journal of Computing*, 13:338-355, 1984.
- [15] B. Schieber and U. Vishkin. *On finding lowest common ancestors: simplifications and parallelization*. *SIAM Journal of Computing*, 17:1253-1262, 1988.
- [16] Zheng Zhang, Scott Schwartz, Lukas Wagner and Webb Miller. *A Greedy Algorithm for Aligning DNA Sequences*. *Journal of Computational Biology*, Vol. 7, 2000.
- [17] V.L. Arlazarov, E. A. Dinic, M.A. Kronrod, and I.A. Faradzev. *On economic construction of the transitive closure of a directed graph*. *Dokl. Acad. Nauk SSSR*, 194:487-488, 1970.
- [18] S. Karlin, G. Ghandour, F. Ost, S. Tavaré, and L.J. Korn. *New approaches for computer analysis of nucleic acid sequences*. *Proc. Natl. Acad. Sci. USA*, 80:5660-5664, 1983.
- [19] University of California, Santa Cruz : Genome Browser. <http://genomes.ucsc.edu>

- [20] National Center for Biotechnology Information. <http://www.ncbi.nlm.nih.gov>.
- [21] W. J. Masek and M.S. Paterson. *A faster algorithm for computing string edit distances*. J. Comp. Sys. Sci., 20:18-31, 1980.
- [22] W. J. Masek and M.S. Paterson. *How to compute string edit-distances quickly*. In D. Sankoff and J. Kruskal, editors, *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*, pages 337-349, Addison Wesley, Reading, MA, 1983.
- [23] S. Altschul, W. Gish, W. Miller, E. Myers, and D.J. Lipman. *Basic local alignment search tool*. J. Mol. Biol., 215:403-410, 1990
- [24] S. Burkhardt, A. Crauser, P. Ferragina, H.P. Lenhof, E. Rivals, and M. Vingron. *q-gram based database searching using a suffix array (quasar)*. In RECOMM, pages 77-83, 1999.
- [25] E. Chavez and G. Navarro. *A metrix index for approximate string matching*. In LATIN, pages 181-195, 2002.
- [26] E. Giladi, M.G. Walker, J.Z. Wang, and W. Volkmuth. *Sst: an algorithm or finding near-exact sequence mathces in time proportional to the logarithm of the database size*. 18(6):873-877, 2002.
- [27] P. Jokinen and E. Ukkonen. *Two algorithms for approximate string matching in static texts*. Proc. MFCS'91, 19:240-248, 1991.
- [28] T. Kahveci and A.K. Singh. *Efficient index structures for string databases*. VLDB, pages 351-360, 2001.
- [29] G. Navarro, R.A. Baeza-Yates, E. Sutinen. and J. Tarhio. *Indexing methods for approximate string matching*. IEEE Data Engineering Bulletin, 24(4):19-27, 2001.

- [30] M. Kimura. *A simple model for estimating evolutionary rates of base substitutions through comparative studies of nucleotide sequences*. J. Mol. Evol., 16:111-120, 1980.
- [31] J. Felsenstein. *Evolutionary trees from DNA sequences: a maximum likelihood approach*. J. Mol. Evol., 17:368-376. 1981.
- [32] M. Hasegawa, H. Kishino, and T. Yano. *Dating of the human-age splitting by a molecular clock of mitochondrial DNA*. J. Mol. Evol., 22:160-174, 1985.
- [33] K. Tamura and M. Nei. *Estimation of the number of nucleotide substitutions in the control region of mitochondrial DNA in humans and chimpanzees*. Mol. Bio. Evol., 10:512-526, 1993.
- [34] M. Kimura. *Estimation of evolutionary distances between homologous nucleotide sequences*. Proc. Natl. Acad. Sci. USA, 78:454-458, 1981.
- [35] A. Zarkikh. *Estimation of evolutionary distances between nucleotide sequences*. J. Mol. Evol., 39:315-329, 1994.
- [36] R. Schwartz, M. Dayhoff, and B. Orcutt. *A model of evolutionary changes in proteins*. Atlas of Protein Sequence and Structure, 5:345-352, 1978.
- [37] M. Dayhoff and R. Schwartz. *Matrices for detecting distant relationship*. Atlas of Protein Sequences, pages 353-358, 1979.
- [38] S. Henikoff and J. D. Henikoff. *Amino acid substitution matrices from protein blocks*. Proceedings of the National Academy of Science USA, 89(22):10915-10919, November 1992.
- [39] M. Fischer and M. Paterson. *String-matching and other products*. In R.M. Karp, editor, *Complexity of Computation*, pages 113-125. SIAM-AMS proc., 1974.
- [40] J. Felsenstein, S. Sawyer, and R. Kochin. *An efficient method for matching nucleic acid sequences: inference and reliability*. Annu. Rev. Ecol. Syst., 14:313-333, 1983.

- [41] K. Abrahamson. *Generalized String Matching*. SIAM Journal of Computing, 16:1039-1051, 1987.
- [42] D. Benson. *Digital signal processing methods for biosequence comparison*. Nucleic Acids Research, 18:3001-3006, 1990.
- [43] D. Benson. *Fourier method for biosequence analysis*. Nucleic Acids Research, 18:6305-6310, 1991.
- [44] E. Cheever, G. Christian Overton, and D. Searls. *Fast Fourier Transform-based correlation of DNA sequence using complex plane encoding*. Comp. Appl. Biosciences, 7:143-154, 1991.
- [45] W.S. Torgerson. *Theory and methods of scaling*. John Wiley & Sons, New York, 1958.
- [46] N. Srebro and T. Jaakkola. *Weighted Low-Rank Approximations*. Proc. of the Twentieth International Conference on Machine Learning (ICML-2003), 720-727 ,2003.
- [47] R.A. Johnson and D.W. Wichern. *Applied multivariate statistical analysis 5ed*. Pearson Education, 2002.