Fast Computation of Supermaximal Repeats in DNA Sequences

Chen Na Lian

A Thesis

in

The Department

of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Computer Science at
Concordia University
Montréal, Québec, Canada

January 2009

Your file Votre référence
ISBN: 978-0-494-63238-3
Our file Notre référence
ISBN: 978-0-494-63238-3

Abstract

Fast Computation of Supermaximal Repeats in DNA Sequences

Chen Na Lian

Searching for repetitive structures in DNA sequences is a major problem in bioinformatics research. We propose a novel index structure, called Parent-of-Leaves (POL) index and an algorithm for finding supermaximal repeats (SMR) which uses the index. The index is derived from and designed to replace the more versatile, but considerably larger suffix tree index STTD64. The results of our experiments using 24 homo sapiens chromosomes indicate that SMR significantly outperforms the Vmatch tool, the best known software package. Using constructed POL index, SMR is 2 times faster than Vmatch in searching for supermaximal repeats of size at least 10 bases. SMR is 7 times faster for repeats of minimum length of 25 nucleotide bases, and about an order of magnitude faster for repeats of length at least 200 basis. We also studied the cost of constructing the POL index, and the number of times we need to run SMR in order for the cost to payoff. The results indicate that our proposed technique outperforms Vmatch after two runs on a particular sequence using the POL25 index which has minimum index length (MIL) of 25 nucleotides, 3 runs with POL10, 5 runs with POL100, and 10 runs with POL200. The storage requirements of various POL indexes are much less than the suffix tree index used, about 200 times smaller for POL200 and POL100, and 25 times smaller for POL25. POL10 requires the largest storage space, which is one quarter the size of the STTD64 index.

Acknowledgments

I would like to express my sincere appreciation to my supervisor Professor, Dr. Nematollaah Shiri, for his guidance, support, documents, and patience, for completing my thesis. Also, I am particularly thankful to Mihail Halachev, to whom I am very much indebted. This thesis would have never appeared without their help and encouragement.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

In this chapter, we first briefly review the history of molecular biology and bioinformatics, and consider some repetitive structures in human DNA and protein. We also discuss some popular bioinformatics applications in detecting repeats. The outline of this thesis appears at the end of this chapter.

1.1 Molecular biology: a general view

Since 1930s, numerous physicists and chemists have taken their interests in understanding life in its most fundamental level. Molecular biology, named by Warren Weaver of the Rockefeller Foundation in 1938, is one of the research fields that tries to explain the phenomena of life from the macromolecular viewpoint.

As described by William Astbury [Astbury, 1961], molecular biology is:

"... not so much a technique as an approach, an approach from the viewpoint of the so-called basic sciences with the leading idea of searching below the large-scale manifestations of classical biology for the corresponding molecular plan. It is concerned particularly with the forms of biological molecules and ..... is predominantly three-dimensional and structural - which does not mean, however, that it is merely a refinement of morphology - it must at the same time inquire into genesis and functions".

In particular, molecular biologists focus on two categories of macromolecules, one of which is nucleic acids. The most famous nucleic acid is deoxyribonucleic acid (DNA), which carries the genetic information in the cell and is capable of self-replication. It is a chain of 4 types of molecules, which are adenine (abbreviated A), cytosine (C), guanine (G) and thymine (T). They are packaged in units known as chromosomes. Some hereditary units in chromosomes that occupy specific locations and determine particular characteristics in an organism are called genes. A set of chromosomes or genes are called the genome, which is known as the blue-print of life. It's known that human haploid genome contains 3,000,000,000 DNA nucleotide pairs, divided among twenty two (22) pairs of autosomes and one pair of sex chromosomes. The other category of macromolecules is proteins. Proteins are fundamental components of all living cells and include many substances, such as enzymes, hormones, and antibodies, which are necessary for the proper functioning of an organism. Proteins are made of 20 amino acids, represented by letters [Nair, 2007].

1.2 Research in bioinformatics

In the past decade, information-heavy and computer driven research has been developed at a very fast pace. As the size of genetic information available is rapidly growing, molecular biologists need effective and efficient computational tools to store and retrieve such information from databases, to analyze the sequence patterns and to obtain the biological characteristic from the sequence. As a result, mathematical methods and computational techniques are strongly needed for the challenging computational tasks in biological research, such as constructing three-dimensional structure of the molecules from the sequence data.

It is obvious that performing tasks mentioned above manually is practically impossible. Researchers therefore resort to bioinformatics, which refers to the use of computer science and related technologies in solving problems of molecular biology such as modeling, analyzing, comparing, graphically displaying, storing, systemizing, searching, and ultimately distributing biological information. For example, some applications are developed to analyze DNA sequence data in order to locate genes.

Research in bioinformatics includes several aspects. A critical research area in bioinformatics is sequence analysis which uses computer programs to search the genomes of thousands of organisms, to align related DNA sequences, to assemble genome fragments, etc. One representative problem in this area is the assembly of high-quality genome sequences from fragmentary shotgun DNA sequencing [Weber et al., 1997] which is a method used for sequencing long DNA strands.

Genome annotation, which identifies the genes and other biological features in a DNA sequence, is another research area in bioinformatics. A number of software tools are developed for biologists to explore genomic annotations at many levels of detail in a graphical environment, such as the popular genome annotation viewer and editor, Apollo Genome Annotation Curation Tool [Lewis et al., 2002]. Bioinformatics also assists evolutionary biologists to trace the evolution of a large number of organisms by measuring changes in their DNA, as well as comparing entire genomes for the study of more complex evolutionary events, such as gene duplication, lateral gene transfer, etc.

Biological databases collect the species names, descriptions, distributions, genetic information, status and size of populations, habitat needs, and the methods that organism interacts with other species. Moreover, through biological databases, the entire DNA sequences, or genomes of endangered species can be preserved on computer and possibly reused in the future, even if that species is extinctive.

Besides our discussion above, there are other exciting and important research in bioinformatics, such as analysis of mutations in cancer [Cairns, 1998], prediction of protein structure [Zhang, 2008], and so on.

With the booming of computer technologies such as databases, graphical user interface(GUI) design, distributed object computing, storage area networks (SAN), data compression, network and communication and remote management, bioinformatics plays more important roles in biological research and science than ever.

1.3 Repetitive DNA sequences

Repetitive DNA sequence occurring in the genome is one of the most striking features of DNA, especially in higher-order organisms such as eukaryotes. For example, [McConkey, 1993] indicates that families of reiterated sequences account for about one third of the human genome. Besides their considerable quantity, the variety of repetitive structures in DNA sequences and their hypothesized biological functions are also intriguing. Some repeats are discovered to play important roles in mutation and evolution. For example, Alu repeats [Alkes et al., 2004] which are the most abundant mobile elements in the human genome, can cause mismatching in DNA duplication.

Since the role of most repetitive structures is mainly unknown, there are numerous difficulties in genome sequencing and analysis. For example, the presence of a small number of copies of repeats can confuse a sequence assembly algorithm, especially for whole genome shotgun sequencing [Weber et al., 1997]. Therefore, identification and characterization of repetitive structures are critical tasks in sequence assembly and genome analysis.

Generally, repetitive DNA sequences are divided into two types:

- tandem repeated DNA
- interspersed repetitive DNA

Tandem repeated DNA, known as satellite DNA, consists of large number of repeats of a short sequence. Satellite DNAs are classified into three groups based on their repeat lengths [Charlesworth et al., 1994], described as follow.

Satellites are very highly repetitive with repeat lengths of one to several thousand base pairs (bp). They are typically organized as large (up to 100 million bp) clusters in the genome.

Minisatellites are moderately repetitive structures with medium-sized repeat lengths from 9 to 100 bp, but usually about 15 bp.

Microsatellites are also moderately repetitive of short (2-6 bp) repeats found in vertebrate, insect and plant genomes. The human genome contains at least 30,000 microsatellite loci located in euchromatin [Zheng et al., 2003].

The main functions of satellite DNAs are still unknown, but some biologists are convinced that certain satellite DNA has some vital functions such as malfunctioning in mutation.

Repetitive DNA that is interspersed throughout all eukaryotic genomes, is generally divided into two classes:

- SINEs
- LINEs

SINEs stand for Short Interspersed Nuclear Elements. Alu repeat [Alkes et al., 2004] is one of the classic examples of SINEs. The Alu repeats occur about 300,000 times in the human genome and account for as much as 5% of the DNA of human and other mammalian genomes [Gusfield, 1997]. LINEs which stand for Long Interspersed Nuclear Elements are not as common in the human genome as SINEs. But as they are much larger, they make up more of the total DNA. While there are about 1.5 million SINEs making up about 13% of the genome sequence, the 870,000 or so copies of LINEs constitute more than 20% of human DNA [Gregory, 2008].

1.4 Maximal repeats and supermaximal repeats

A *maximal repeat* in a sequence S is a substring that occurs at least twice in S, and that cannot be further extended to the left and/or right without destroying it being a repeat. For example, consider the DNA sequence:

$$S = \underline{\mathrm{A}\mathbf{ACGT}\mathrm{C}}\mathrm{G}\mathbf{ACGT}\mathrm{T}\underline{\mathrm{A}\mathbf{ACGT}\mathrm{C}}.$$

This sequence includes two maximal repeats: ACGT, which occurs three times (shown in boldface), and AACGTC, which occurs twice (shown with underlines).

A *supermaximal repeat* is a maximal repeat that never occurs as a substring of any other maximal repeat. In the above example, the sequence AACGTC is a supermaximal repeat, but ACGT is not, since ACGT occurs as a substring of AACGTC.

Searching for maximal repeats and supermaximal repeats is a basic analysis task which biologists often perform for finding repeated patterns in a new DNA sequence. With the exponential rate at which new DNA sequences are being acquired, we need more efficient techniques to find repetitive structures. Some effective search techniques use indexes to speed up the search process. The STTD64 index scheme [Halachev et al., 2007] is one of search techniques for indexing large sequences efficiently in bioinformatics applications. Using this indexing technique to support repeats search tasks are considered a way to improve searching performance. Our supermaximal repeats search technique is based on STTD64 index system.

As discussed above, there are many more maximal repeats than supermaximal repeats, and hence maximal repeats are less considered in finding repeats. Supermaximal repeats can filter out the abundant relatively shorter repeats that mostly occur by chance and do not carry structural or functional information. Therefore, providing a required minimum length of the discovered supermaximal repeats helps in collecting useful information, and improving the search performance.

In this thesis, we thus focus on providing a fast SuperMaximal Repeat (SMR) algorithm and its supporting index technique called Parent-of-Leaves (POL), which is derived from the STTD64 indexing scheme.

1.5 Related work

1.5.1 Bio-sequence repeat search tools

There are several software tools developed for finding repeats in genomic sequences, including REPuter [Kurtz et al., 1999], RepeatMatch [Delcher et al., 1999], RepeatMasker [Smit et al., 2008], MaskerAid [Bedell et al., 2000].

REPuter is a popular software tool for computing various kinds of repeats, including supermaximal repeats. It provides efficient and complete detection of various types of repeats with an evaluation of significance and interactive visualization [Kurtz et al., 1999]. The search engine REPfind of REPuter uses an efficient and compact suffix trees implemented in improved linked list to locate exact repeats in linear space and time. It has been estimated in [Kurtz, 1999] that this time-critical task can be done in linear time for sequences up to the size of the human genome. The output of the search engine REPfind is displayed in the form of a repeat graph by the interactive visualization program REPvis. More running time and space cost of

REPuter are reported in [Kurtz et al., 2000]. An online version of REPuter providing some basic functionality is available from the Bielefeld Bioinformatics web server (http://bibiserv.techfak.uni-bielefeld.de/reputer/).

RepeatMatch is another highly efficient computational tool that can find all exact repeats in genome sequences. This tool is also based on suffix trees, but does not support supermaximal repeat search.

RepeatMasker is a program that sifts DNA sequences for interspersed repeats and masks low complexity DNA sequence. The program outputs a detailed annotation of the repeats. On average, almost 50% of a human genomic DNA sequence currently will be masked by the program [Smit et al., 2008].

RepeatMasker performs string comparisons by the program *cross_match*, which implements the Smith-Waterman-Gotoh algorithm [Gotoh, 1982] efficiently. RepeatMasker is another approach for detecting repeats, which identifies repeats by performing exact or approximate string match of the sequence data against known repetitive patterns previously stored in its database. The interspersed repeat databases screened by RepeatMasker are based on the Repbase Update database which is copyrighted by the Genetic Information Research Institute (G.I.R.I.) [Jurka et al., 2005]. The Repbase Update database contains annotation of most repeats with respect to divergence level, affiliation, etc.

MaskerAid is an implementation of the same approach of RepearMasker. It is a drop-in accelerator that increases the speed of RepeatMasker about 30 times while maintaining sensitivity. Both of these tools find already known repeats in a given sequence, which is different from the problem we address in this thesis, i.e., finding supermaximal repeats in a sequence without any prior knowledge.

Vmatch [Kurtz, 2000] searches for supermaximal repeats using enhanced suffix array (ESA) index structure [Abouelhoda et al., 2004]. Vmatch can process very large DNA sequences. It is claimed in [Kurtz, 2000] that the 32-bit version of Vmatch can process up to 400 million symbols, if enough memory is available. For large server class machines (e.g. SUN-Sparc/Solaris, Intel Xeon/Linux, Compaq-Alpha/Tru64), Vmatch is available as a 64-bit version, enabling gigabytes of sequences to be processed.

Vmatch preprocesses sequences to create index structures which are stored as a collection of several files. The index efficiently represents all substrings of the preprocessed sequences. Different matching tasks require different parts of the index, but only the required parts of the index are accessed during the matching process.

Vmatch can process not only DNA or protein sequences, but also sequences over any user defined alphabet of up to 250 symbols. Vmatch fully implements the concept of symbol mappings, denoting alphabet transformations. It allows a multitude of different matching tasks to be solved using the index, such as maximal repeats, branching tandem repeats, supermaximal repeats, maximal substring matches, and so on. The solutions for maximal and supermaximal repeat search included in Vmatch subsume the ones in REPuter.

There are more than 20 completed or ongoing projects which are using Vmatch. For example, GenomeThreader, which computes gene structure predictions, uses the matching capabilities of Vmatch to efficiently map the reference sequence to a genomic sequence [Gremme et al., 2005]. The KPATH system [Slezak et al., 2003], developed at the Lawrence Livermore National Laboratories, uses Vmatch to detect unique substrings in large collection of DNA sequences.

After evaluating the above software tools for repeats detection, we choose Vmatch as our experimental benchmark to compare our work with, for the following reasons. First, as mentioned in Vmatch homepage [Kurtz, 2000], Vmatch subsumes REPuter and has better space utilization and faster search performance comparing to REPuter.

Further, Vmatch is a general software tool for solving various search problems in large-scale sequence data, where supermaximal repeats search is just one of the functionalities it provides. Since our on-going FASST project also aims at providing a unified underlying index structure for various types of search tasks in large biological sequences, we consider the supermaximal repeat search comparison to Vmatch as yet another opportunity to evaluate and compare the two competing multipurpose alternatives.

1.5.2 Various index data structures

Recently, suffix trees (ST) and suffix arrays (SA) received considerable interest from research community as data structures suitable for indexing large DNA sequences. Each suffix is a string starting at a certain position in the sequence and ending at the end of the sequence. Suffix trees are introduced in the next chapter. Suffix array is simply an array containing all the pointers to the sequence suffixes sorted in lexicographical order. Searching a string can be performed by binary search using the suffix array [Manber et al., 1993].

A major drawback of suffix trees and suffix arrays index structures is their considerably large size, especially evident for ST. For a sequence of n symbols, suffix arrays require $4n$ bits for storing each symbol [Manber et al., 1993], while suffix trees require $8.5n$ bits [Giegerich et al., 1997].

In order to overcome the space problem, several compressed suffix arrays and suffix trees representations [Grossi et al., 2005, Ferragina et al., 2000, Niko et al., 2007] are proposed. For example, FM-index [Ferragina et al., 2000] is based upon the Burrows-Wheeler compression algorithm [Burrows et al., 1994] and the suffix array data structure. The major advantage of the compressed index representation is their smaller size, which makes it possible to fit entirely in the main memory available on regular desktop computers. However, this gain in space requirements comes at the cost of less efficient search support. As discussed in [Hon et al., 2004], compressed suffix array

[Grossi et al., 2005] and FM-index are much slower than suffix tree and suffix array for exact match search.

1.6 Organization of the thesis

In this thesis, we first review STTD64 (Suffix Tree Top Down 64 bits), proposed in [Halachev et al., 2007], which is the foundation of our FASST project. Then, we review a well-known supermaximal repeats search algorithm in [Gusfield, 1997], which uses a suffix tree index structure. Next, we demonstrate our supermaximal repeats search algorithm (SMR) and technique. We propose a novel parent-of-leaves (POL) index structure, which is derived from and replaces the STTD64 index for searching supermaximal repeats.

This thesis focuses on development of a novel Parent of Leave (POL) index and an efficient algorithm for finding supermaximal repeats (SMR) which uses POL. In our experiments and results, we assume the STTD64 and Vmatch indexes are already built and available, and hence we do not consider their construction cost in our measured figures. The time and space requirements of constructions of STTD64 and Vmatch indexes are studied and compared in [Halachev et al., 2007].

We conduct numerous experiments using real-life biological data to evaluate the performance of the proposed supermaximal repeats search algorithm (SMR). We study the cost both in terms of construction time and storage space of the proposed POL index. We then compare the search time performances of SMR with Vmatch under different situations. Furthermore, we study the number of supermaximal repeats and its impact on performance according to different minimum repeat lengths. Finally, we evaluate the POL construction cost and SMR performance for searching supermaximal repeats in synthetic DNA sequences.

The outline of this thesis is as follows. Suffix trees are discussed in Chapter 2, where we review the STTD64 suffix tree indexing technique. In Chapter 3, we review basic concepts of repeats as well as the Gusfield's supermaximal repeat algorithm. In Chapter 4, we propose our POL index structure, followed by the description of POL construction algorithm and corresponding SMR algorithm. Chapter 5 evaluates the POL index construction cost and compares performance of SMR algorithm with Vmatch in different perspectives. The SMR application is developed and incorporated as part of the FASST project at http://sepehr.cs.concordia.ca/. We also developed the web-based interface to this search tool, which is presented in Chapter 6. Chapter 7 draws conclusions and outlines future work.

Chapter 2

Suffix Trees

In this chapter we review the suffix tree(ST) data structure and its construction algorithm. We present several ST representations, discuss their advantages and shortcomings, and explain the STTD64 representation used as a basis in this work.

2.1 History and applications of suffix trees

The first linear-time suffix tree construction algorithm was proposed by [Wenior, 1973]. A few years later, McCreight proposed a more space efficient algorithm [McCreight, 1976]. In 1995, Ukkonen developed a conceptually different linear-time on-line suffix tree construction algorithm [Ukkonen, 1995], which is easier to implement and allows for easier proof of bounds.

A suffix tree is a versatile data structure which supports efficient solutions for many problems on strings (sequences of characters). One of the typical problems is exact string matching, which for a pattern sequence P, finds the matching patterns in $O(m + k)$ time, where m is the size of pattern sequence P and k is the number of occurrences of P in T. Another problem solved efficiently by suffix trees is Longest Common Substring problem, which is to find the longest string (or strings) that is a substring (or are substrings) of two or more strings. The longest common substrings of a set of strings can be found by building a generalized suffix tree for the set of strings, and then finding the deepest internal nodes which has leaf nodes from all the strings in the subtree below it [Gusfield, 1997].

Bioinformatics applications based on suffix trees are often used for searching for patterns in DNA or protein sequences. For example, REPuter which searches for maximal repeats in complete genomes [Kurtz et al., 1999], is based on suffix trees. Another popular software tool based on suffix trees is MUMmer [Delcher et al., 1999], which is a system that supports fast alignment of entire genomes. Another use of suffix trees is data clustering used in some search engines, e.g. [Zamir et al., 1998]. Recently, suffix trees have been used in data compression which is the process of encoding information using fewer bits (or other information-bearing units) than a normal representation through the use of specific encoding schemes. Sadakane [Sadakane, 2007] proposed a compressed suffix trees with full functionality of suffix trees.

2.2 Basic definitions

In this section, we review some definitions which are taken from [Gusfield, 1997].

Definition Given an input sequence S of size n characters, a *suffix tree* ST is a rooted directed tree with exactly n leaves numbered from 1 to n. Each internal node, other than the root, has at least two children and each edge is labeled with a nonempty substring of S. No two edges out of a node can have edge-labels beginning with the same character [Gusfield, 1997].

For any leaf node i, the concatenation of the edge-labels on the path from the root to node i exactly spells out the substring of S starting from position i. *i.e.*, $S[i..n]$.

For any node v in a suffix tree, the *string-depth* of v (or *depth* of v, for short) is the number of characters of the labels on the path from root to the parent of v.

For example, Figure 2.1 shows the graphical representation of ST for $S = banana$. In this figure, the path from root to the leaf node numbered 2 spells out the string $s = anana$, which starts at position 2 of S, while the *depth* of node 2 is 3.

As described above, if one suffix of S matches a prefix of another suffix of S,

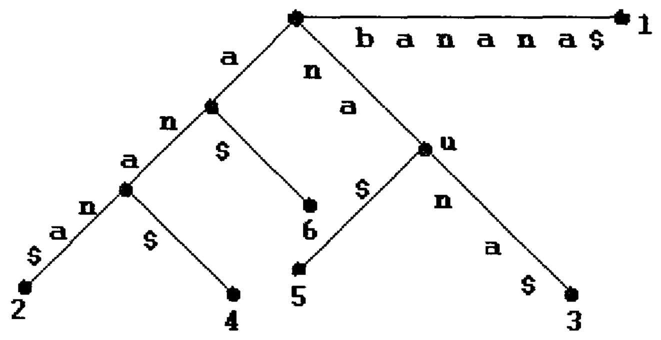


Figure 2.1: Suffix tree for string $S = banana$

the first suffix would not end at a leaf according to the definition of suffix tree. For example, in string *banana* in the figure, suffix *na* is a prefix of *nana*, so the path spelling out *na* would not end at a leaf. To solve this problem, we assume that last character in a string appears nowhere else in the string. That is, no suffix is a prefix of another suffix. To achieve this in practice, we add a termination character at the end of string. In this thesis, we use \$ as the termination symbol and extend every string with this symbol, even if the symbol is not explicitly shown.

2.3 Suffix trees representations

There are a number of suffix trees representations, including level-compressed Patricia tree [Andersson, 1995], write only top-down suffix tree (wotd) [Giegerich et al., 2003], suffix binary search tree [Irving et al., 2003].

Level-compressed Patricia tree is a compact representation of suffix tree that combines path compressed and level compressed techniques. At each internal node, an index indicates the character used for branching at the node. With this additional information available at each node, we can remove all internal nodes with an empty subtree. This path-compressed binary tree is called a Patricia tree [Morrison, 1968]. Level compression can be used to reduce the size of the Patricia tree. That is, each internal node of degree two that has an empty subtree is removed, and at each internal node we use an index that indicates the number of bits skipped.

The write only top-down suffix trees (wotd) [Giegerich et al., 2003] is another suffix tree representation in which each node is 32 bits. It requires $8.5n$ bytes on average which is much larger than the input sequence. To index very large sequences, suffix trees need either large memory or require disk based construction algorithms.

In this section, we first introduce wotd as proposed in [Giegerich et al., 2003]. Then, we describe STTD64, which is an extension of wotd that overcomes some of its limitations.

2.3.1 wotd representation

To illustrate the structure of suffix trees, let us consider the following sample sequence $S = AGAGAGC\$$, where \$ is used as a terminal symbol to ensure no suffix is a prefix of another suffix. A graphical representation of a ST for sequence S is shown in Figure 2.2, in which the numbers in squares indicate the order in which the ST nodes are evaluated and recorded. The number below each leaf node shows the starting position of the suffix of S represented by this leaf node, and this suffix is encoded by the edge-labels on the path from the root to this leaf node.

Next, we introduce some concepts taken from [Giegerich et al., 2003].

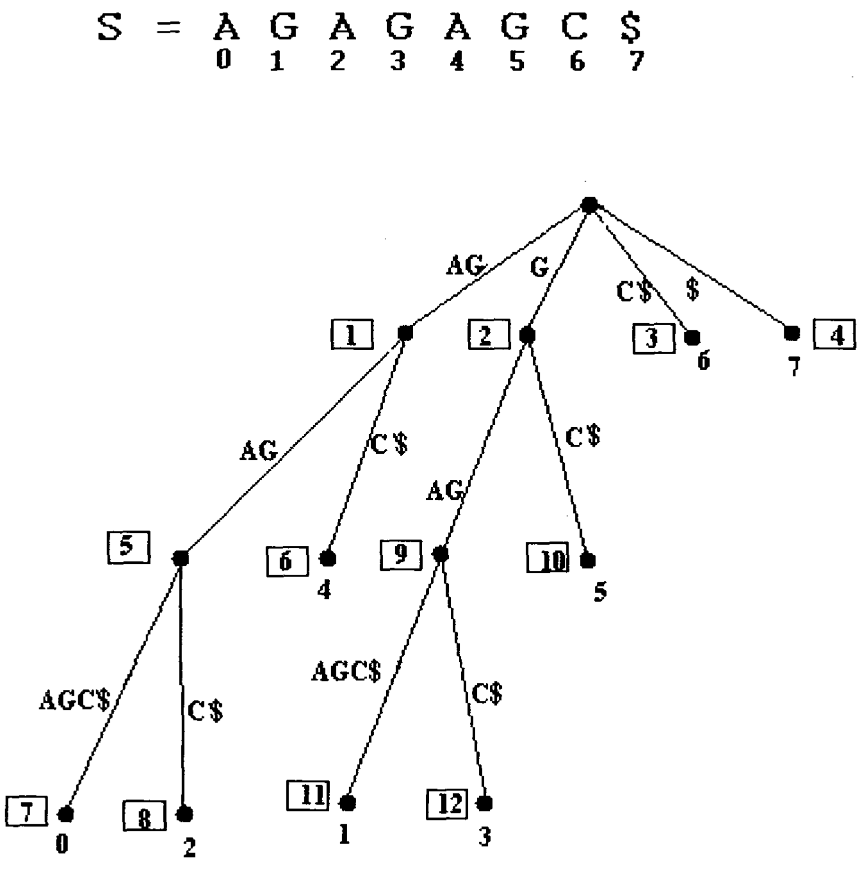


Figure 2.2: Graphical Suffix tree (ST) for the sample sequence S = AGAGAGC$

Concept 1: For a leaf node s in a suffix tree (ST), the leaf set of s, denoted $l(s)$, contains the position i in sequence S where the string starting from $S[i]$, is denoted by the edge-labels from root to s.

For example, for leaf node 12 in Figure 2.2, we have $l(12) = 3$.

Concept 2: For a branch node u in a ST, the leaf set of u is defined as the set of the leaf sets of the children of u, *i.e.*, $l(u) = \{l(s) | s \textit{ is a leaf node in the subtree rooted at } u\}$.

For instance, the leaf set of node 9 in Figure 2.2 would be $l(9) = \{l(11), l(12)\} = \{1, 3\}$.

Concept 3: For a node v, its *left pointer*, denoted $lp(v)$, is defined as minimum value of $l(v)$ plus the number of characters on the path from the root to the parent of v.

For example, $lp(9) = \min l(9) + 1 = 1 + 1 = 2$.

In wotd representation, an internal node u occupies two adjacent elements. The first element contains the lp value of u and two additional bits, called the *rightmost bit* and *leaf bit*. If the rightmost bit is set to 1, it indicates that node u is the rightmost child of a branch node. For instance, in Figure 2.2, node 6 is the rightmost child of node 1. Therefore, its rightmost bit is 1. A *leaf bit* 1 indicates the node is a leaf. For each internal node, its leaf bit is always 0. The second element of an internal node u stores a pointer to its first child. The pointer points to the address in wotd index where the first child of u is stored. A leaf node in ST occupies one element in wotd index, where stores the same information as the first element of a branching node(*i.e.* lp value, rightmost bit, and leaf bit).

Figure 2.3 shows the wotd representation of the ST for sequence S, where the first row is node number of ST used for illustration purpose only; the number in the second row indicates the order of elements in the wotd index, and the third row is the content stored in the index file. In this figure, superscript 'R' indicates a rightmost

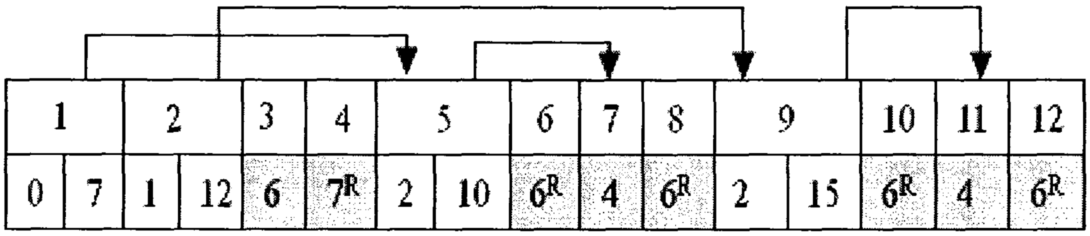


Figure 2.3: wotd representation for S = AGAGAGC\$

bit, and a grey cell indicates a leaf bit. For example, in Figure 2.3, the internal node 2 is stored in the third and fourth elements, *i.e.*, wotd[2] and wotd[3]. Its first child is node 9 (see Figure 2.2). Hence, the value in third element allocated for node 2 points to the position 12, which is the first of the two elements storing node 9.

As already illustrated, 2 bits are reserved for the rightmost bit and leaf bit in wotd representation. In a 32-bit system, an element in wotd structure occupies 32 bits. Only 30 bits are available for storing the lp value. That is, the sequence we can index using this structure is limited to $2^{30} - 1$ bits, *i.e.*, about 1 billion characters. This limitation is a bottleneck of using wotd to index very large sequences.

To overcome this limitation, [Halachev et al., 2007] propose an alternative ST representation, called STTD64, presented in next section.

2.3.2 STTD64 representation

In this section, we present STTD64 representation proposed in [Halachev et al., 2007]. This index shares some common properties with wotd [Giegerich et al., 2003]. First, they both use a top-down traversal manner. Second, they use pointers pointing to the first child of branch (internal) nodes. Finally, the rightmost bit and leaf bit are used in both ST representations.

Next, we illustrate the differences between the two ST structures. First, every node in STTD64 is 64 bits record, no matter if it is an internal node or a leaf node, while an internal node in wotd occupies 64 bits, and a leaf node occupies 32 bits. Secondly, the size of each element in STTD64 is 64 bits, whereas it is 32 bits in wotd as the names indicate. Figures 2.4 and 2.5 depict the structure of internal node and leaf node in STTD64, respectively.

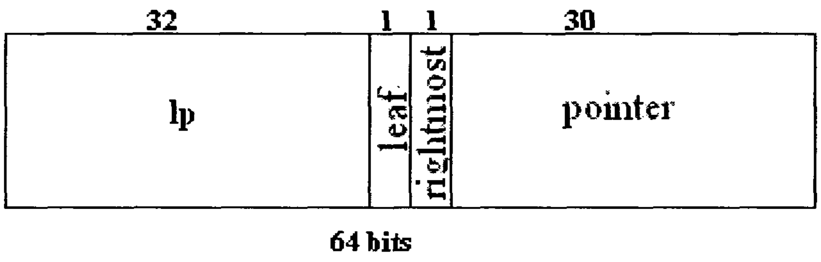


Figure 2.4: Branch node in STTD64 representation

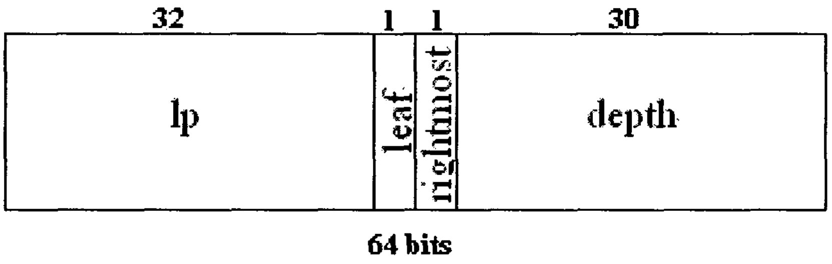
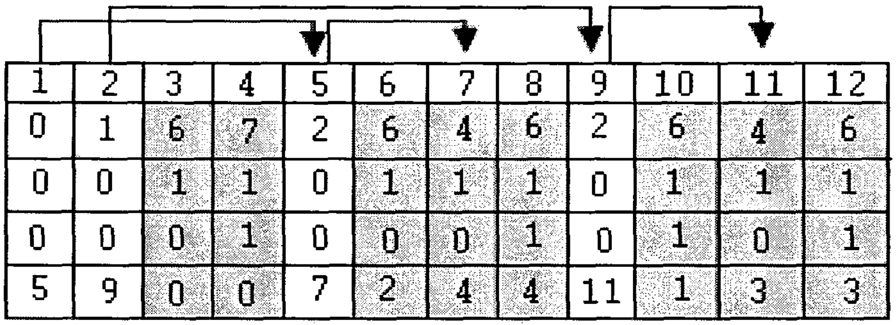


Figure 2.5: Leaf node in STTD64 representation

For both branch nodes and leaf nodes, the first 32 bits store *lp* value, bits 33 and 34 record leaf bit and rightmost bit, respectively. The last 30 bits for a branch node are available for a pointer to its first child. In a leaf node, the last 30 bits record its *depth* value. The *depth* of a leaf node *s* is defined as the number of characters on the path from the root to the parent of s. For example, the *depth* of node 8 in Figure 2.2 is 4, labeled by four characters $|$ *AGAG* $|$) on the path.

For our running sequence S = AGAGAGC\$, Figure 2.6 shows its STTD64 index



1	2	3	4	5	6	7	8	9	10	11	12
0	1	6	7	2	6	4	6	2	6	4	6
0	0	1	1	0	1	1	1	0	1	1	1
0	0	0	1	0	0	0	1	0	1	0	1
5	9	0	0	7	2	4	4	11	1	3	3

Figure 2.6: STTD64 representation of sequence S = AGAGAGC\$

representation. The numbers on the top (used for illustration purpose only) indicate the node number in the suffix tree of Figure 2.2. The 32 bits *lp* values are shown in the second row and the following two rows indicate leaf bit and rightmost bit, respectively. The pointer/depth values are shown in the last row. For clarity, leaf nodes are shown in gray and branch node pointers are illustrated by the arrows above the table.

As illustrated above, the second difference from wotd is that STTD64 records the depth values for leaf nodes, which leads to efficient indexing. We will further explain this in the next section when we describe our algorithm. Another advantage of STTD64 is its capability to index sequences of up to 4 GB, i.e., 4 times larger than wotd.

On the other hand, STTD64 needs more storage space than wotd. For a given sequence of size n symbols, there are exactly n leaf nodes, and at most n internal nodes. In the worst case, STTD64 occupies $16n$ bytes per symbol, and on average, the space required is $13n$ as shown in [Halachev et al., 2007], while wotd occupies $12n$ bytes in the worst case and $8.5n$ bytes on average.

Chapter 3

Maximal Repeats and Supermaximal Repeats

In this chapter, we review some concepts and techniques related to maximal and supermaximal repeats. We then introduce existing algorithms for finding maximal repeats and supermaximal repeats which use suffix tree index. Finally, we analyze the supermaximal repeat search algorithm, and discuss their limitations.

3.1 Maximal repeats

As initial illustration of maximal repeats was given in 1.4. Here we give a more formal definitions, taken from [Gusfield, 1997].

Definition 1: A *maximal pair* of strings in a sequence S is defined as a pair of identical substrings α and β of S such that the character to the immediate left(right) of α is different from the character to the immediate left(right) of β [Gusfield, 1997](P.143).

That is, extending α and β in either direction would destroy the equality of the two strings. A maximal pair is represented by the triple $(p1, p2, l)$, where $p1$ and $p2$ are the starting positions of the two substrings α and β, and l is their length. For a string S, we use $\Re(S)$ to denote the set of all triples describing maximal pairs in S.

For example, consider the string $S = wr\mathbf{axy}ttrv\mathbf{ax}ebvg\mathbf{axy}$, which includes three occurrences of substring ax. The first and second occurrences are represented by maximal pairs as (3,10,2), and the second and third occurrences are represented as (10,16,2). However, the first and third occurrences of ax do not form a maximal pair since their immediate right characters are the same. Hence, the two occurrences of axy form a new maximal pair (3,16,3). Also the definition of maximal pairs allows the two strings to overlap each other. For instance, string $xxyxxyxx$ has a maximal pair (1,4,5) whose representative substring is $xxyxx$. Generally, in this thesis, we assume the immediate left (right) of the first (last) character of a string differs from any other characters in this string.

In some cases, the full set of maximal pairs $\Re(S)$ is explicitly found and presented. Note that in some situations, $\Re(S)$ may be too enormous to be displayed or used. Therefore, a more compact representation of maximal pairs is provided below.

Definition 2: A *maximal repeat* α is a substring of S that occurs in a maximal pair in S. That is, α is a *maximal repeat* in S if there is a triple $(p1, p2, |\alpha|) \in \Re(S)$ and α occurs in S starting at position $p1$ and $p2$. We use $\Re'(S)$ to denote the set of maximal repeats in S [Gusfield, 1997](P.143).

In our above example string S, substrings ax and axy are both maximal repeats. Note that the number of maximal repeats $|\Re'(S)|$ is less than or equal to the number of maximal pairs $|\Re(S)|$, and is generally much smaller since a string is represented only once no matter how many times it participates in maximal pairs in S.

3.2 Supermaximal repeats

Maximal repeats form one type of repetitive structures. However, in some applications, they are not a desired repetitive structure. For example, in string $\mathbf{x}\alpha\mathbf{y}sd\alpha k\mathbf{x}\alpha\mathbf{y}$, $x\alpha y$ and α are both maximal repeats, but $x\alpha y$ includes α. In this case, it may not be desired to report α as a repetitive structure, since $x\alpha y$ may be more informative. This leads to the concept of supermaximal repeats defined as follow.

Definition 1: A *supermaximal repeat* is a maximal repeat that never occurs as a substring of any other maximal repeat [Gusfield, 1997](P.144).

From our example above, $x\alpha y$ is a supermaximal repeat but α is not, since it is a substring of $x\alpha y$.

Another repetitive structure is *near-supermaximal repeats* defined as follow.

Definition 2: A substring α of S is a *near-supermaximal repeat* if α is a maximal repeat in S that occurs at least once in a location where it is not contained in another maximal repeat. Such an occurrence of α is said to *witness* the near-supermaximality of α [Gusfield, 1997](P.146).

For example, in string $x\alpha ysd\alpha kx\alpha y$, substring α is not a supermaximal repeat, but a near-supermaximal repeat. The second occurrence of α witnesses the fact.

According to the above definition, a supermaximal repeat α is a maximal repeat in which every occurrence of α is a witness to its near-supermaximality [Gusfield, 1997].

3.3 Finding maximal repeats using suffix trees

Before discussing an algorithm to find supermaximal repeats, we first illustrate an algorithm of finding maximal repeats by using suffix tree which is taken from [Gusfield, 1997]. Finding maximal repeats is a simpler problem and forms a basis for finding supermaximal repeats.

Let ST be a suffix tree for string S. If a string α is a maximal repeat in S then α is the path-label of an intended node v in ST. To see this, we review the definition of maximal repeats. If α is a maximal repeat, there must be at least two occurrences of α in S where the character to the immediate right of the first occurrence differs from that of the second occurrence. According to the definition of suffix trees, no two edges out of a node can have edge-labels beginning with the same character. Therefore, α is a path-label of a node v in ST.

From the above discussion, we concluded that to find maximal repeats we only need to consider substrings (i.e.,path-labels) that end at nodes of the suffix tree ST. But what kind of specific nodes are representatives of maximal repeats?

Before answering this question, we need to introduce some concepts first.

Concept 1: For each position i in string S, character $S[i-1]$ is called the *left character* of i. The *left character of a leaf* is the left character of the suffix position represented by that leaf [Gusfield, 1997](P.144).

Concept 2: A node v of a suffix tree ST is called *left diverse* if at least two leaves in the subtree at v have different left characters [Gusfield, 1997](P.144).

By definition, a leaf cannot be left diverse. If a node is left diverse, all its ancestors in the tree are also left diverse. Then a theorem comes out.

Theorem: *Let S be a string and ST be a suffix tree. A string α labeling the path to a node v in ST is a maximal repeat if and only if v is left diverse* [Gusfield, 1997](P.144).

For example, we suppose a node v is left diverse. That means there are at least two substrings $x\alpha$ and $y\alpha$ of S. Then assume first that $x\alpha$ is followed by character p. If the second substring is followed by any character other than p, then α is a maximal repeat. In another case, if the second substring $y\alpha$ is also followed by p. That is, the two occurrences are $x\alpha p$ and $y\alpha p$. By definition of suffix trees, a branching node v must have at least two children. Hence, there must be a substring αq in S for some character q other than p. If the occurrence of αq is preceded by character x, then $x\alpha q$ forms a maximal pair with $y\alpha p$. And if it is preceded by character y, then $y\alpha q$ forms a maximal pair with $x\alpha p$. In either case, α is a maximal repeat. The details of proof are described in [Gusfield, 1997].

3.4 Finding supermaximal repeats using suffix trees

In this section, we introduce an algorithm to compute supermaximal repeats in linear time proposed in [Gusfield, 1997]. The proposed algorithm uses a suffix tree ST of string S to search for the supermaximal repeats in S.

The following theorem described in [Gusfield, 1997] forms a basis for computing supermaximal repeats, to which we refer as Gusfield's algorithm.

Theorem: *A left diverse internal node v in a suffix tree represents a supermaximal repeat α if and only if all children of v are leaves, and each has a distinct left character.*

To discuss this theorem, we assume a node v in ST corresponds to a maximal repeat α, and v has two children w and u. Let $L(w)$ denote some (but not all) occurrences of α in S which are located in the subtree of ST rooted at w.

We consider two possibilities of node w. First, suppose w is an internal node in ST, and substring r is the label of edge (v, w). Every element in $L(w)$ identifies an occurrence of αr. Since w is an internal node, $|L(w)| > 1$, and αr is the prefix of a maximal repeat. Therefore, all the occurrences of α specified by $L(w)$ are involved in a maximal repeat that begins with αr. Hence, α is not a supermaximal repeat.

Secondly, suppose w is a leaf node. Let i be the starting position of the substring corresponding to the leaf w and x be the left character of leaf w. In this case, we consider node u. if u is an internal node, as we discussed above, α is not a supermaximal repeat.

If u is also a leaf, let j be the starting position of the substring corresponding to the leaf u. We discuss two cases. First, assume node u has left character x. Then $x\alpha$ occurs twice in S. Therefore, α is contained in a maximal repeat. Thus, α is not a supermaximal repeat. Second, assume u is preceded by any character but x, say y. Then α has different left characters at the positions i and j. Since w and u are both leaves, according to the definition of suffix trees, the first character labeled between v and w differs from that between v and u. That is, substrings α in positions i and j are followed by distinct characters. Therefore, the occurrences of α at i and j are involved in a supermaximal repeat, and hence α is the supermaximal repeat.

3.5 Computing supermaximal repeats

According to Gusfield theorem, we derived an algorithm named Gusfield algorithm. A pseudo code of Gusfield's algorithm is shown as Algorithm 1. As inputs it takes a sequence S to be searched and its suffix tree index ST. The algorithm returns the starting positions of all supermaximal repeats in S and their lengths.

The algorithm traverses the nodes in ST sequentially, and performs two major steps while traversing the ST index. In the first step, it examines ST branch nodes in ST, checking if a particular branch node v has only leaf node children (i.e., from steps

Algorithm 1 Gusfield's Algorithm (Sequence S, suffix tree index ST)

```
1: $v$ points to first $ST$ node
2: while there are unexamined $ST$ nodes do
3:   while $v$ is a branch do
4:     $v$ = the first child of $v$
5:   end while
6:   if $v$ is the first child then
7:     retrieve *position* of $v$ in $S$
8:     retrieve left character of *position* in $S$, i.e., $S[position - 1]$
9:     $v$ points to next node
10:    while $v$ is not the rightmost node do
11:      if $v$ is not a leaf then
12:        break from while loop
13:      else
14:        retrieve *position* of $v$ in $S$
15:        retrieve left character of *position* in $S$
16:      end if
17:    end while
18:    if all $v$'s children are leaves then
19:      compare left characters of all occurrences
20:      if all left characters are distinct then
21:        output corresponding length and positions of this repeat
22:      end if
23:    end if
24:    $v$ points to next node;
25:  end if
26: end while
```

3 to 17). If yes, the second step is executed, in which it compares the left characters of all children node of v, i.e., the left diverse check (i.e., step 20). If successful, the starting positions of the suffixes represent the starting positions of a supermaximal repeat, and are returned to the user, together with its length, which in fact is the depth of the v's leaf nodes. The time complexity of the algorithm is $O(n)$, where n is the number of nodes in the ST.

As shown in the pseudo code, Gusfield's algorithm has to traverse and examine all ST nodes. This results in a significant amount of disk I/Os to read into main memory the whole ST index from disk, which is an order of magnitude larger than the sequence size. To overcome this problem, we propose an auxiliary index and a novel algorithm in chapter 4.

Chapter 4

Our Proposed Technique for Computing Supermaximal Repeats

In the previous chapter, we discussed Gusfield's algorithm for finding supermaximal repeats (SMR), which returns the starting positions and the lengths of all supermaximal repeats in a sequence, by performing full sequential scan of its entire *ST* index. However, very often in practice, biologists are interested in repeats of size longer than a particular threshold value. For example, [Miki et al., 1980] studies different species for repeats whose sizes are longer than 200 base pairs. Gustfield's algorithm examines the entire *ST* index, which is about 13 times larger than the data sequence, cannot take advantage of this additional information on threshold size, and hence performs a constant and significant amount of disk I/O operations.

The supermaximal repeats search technique (SMR) that we propose here uses Gusfield's algorithm as a basis but extends it to a more efficient solution. It uses our proposed index structure, called Parent-Of-Leaf (POL), which is derived from and replaces the STTD64 index. The new POL index is considerably smaller than the STTD64 index. We organize and store the information in POL in such a way that the number of required disk I/O operations is much reduced, resulting in considerably shorter search time. Next, we present the structure of the POL index, followed by a description of its construction algorithm. We will then propose our SMR algorithm, which uses the POL index of a sequence to find the supermaximal repeats.

4.1 POL index structure and representation

As discussed earlier, each ST node v whose children are all leaf nodes, is a candidate of supermaximal repeats that has to be further examined. If all the suffixes represented by the leaves of v have distinct left characters (i.e., the nodes are left diverse), then a supermaximal repeat is found, which is the common prefix of all the leaf nodes (i.e., the characters on the path from the root to v).

Our POL index is a collection of records related to such candidate nodes v. Each record consists of two parts, the *header* and *data*, as shown in Figure 4.1.

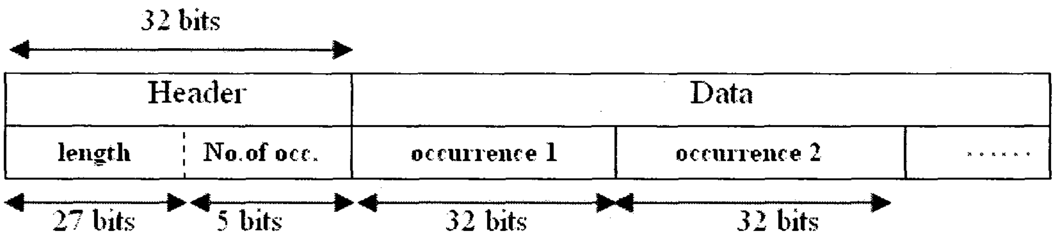


Figure 4.1: POL index representation

Header part: In the first 27 bits of the *header*, we store the number of characters on the path from the root of the ST index to node v, which represents the length of the potential supermaximal repeat. In the remaining 5 bits of the header, we record y, the number of the leaf children of v, which also indicates to the number of occurrences of the repeat in the input sequence S.

Data part: The *data* part of the record for candidate node v contains exactly y blocks, each of size 32 bits. In each block, we store the start location in sequence S at which the suffix represented by a particular leaf of v occurs.

The chosen sizes of the index fields allow for POL indexing of DNA and protein sequences of sizes up to 2^{32} characters (4GB), in which the length of the longest supermaximal repeat is at most 134 million characters. First, the 4 GB limit is due to the fact that in each data block, we have 32 bits available for recording a sequence location, i.e., the sequence size is limited to 2^{32}. Next, recall from the ST definition, that no two edges out of a node can have labels which start with the same character. Thus, the number of children of any ST node is bounded by the alphabet size, i.e., 5 for DNA data (A, C, G, T, and the terminal symbol $) and 21 for proteins. In order for POL index to be applicable for both DNA and protein data, we allocate 5 bits for the second part of the header in which we record the value y, i.e., the number of leaf children for each candidate node v. Thus, our POL index can handle sequences whose alphabet is of size at most 32 symbols. Last, the remaining 27 bits in the header part are used for recording the length of the repeat, which leads to the limitation on the supermaximal repeat length of 2^{27} characters, i.e., around 134 million nucleotides or amino acids.

The POL index is implemented as an array of 32 bit blocks. Figure 4.2 shows

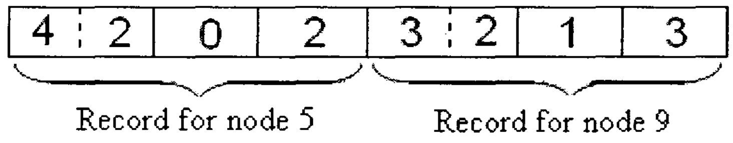


Figure 4.2: The POL Index for sequence $S = AGAGAGC\$$

the POL index for our example sequence S = AGAGAGC$. There are two *ST* nodes

(nodes 5 and 9 in Figure 2.2) satisfying POL index selection of the *ST*. The information about candidate nodes 5 and 9 is recorded respectively in the first 3 and the last 3 POL blocks of the index. The length of the repeat represented by node 5 is 4 nucleotides (i.e., size of "AGAG"), and node 5 has 2 leaf children. Thus, in the header of the record for node 5, we store values 4 and 2 (Figure 4.2). The next 2 blocks, as indicated by the last 5 bits of the header, are used for recording the starting locations in the sequence at which this repeat occurs. The children of node 5 are nodes 7 and 8, which represent suffixes starting at locations 0 and 2, which are stored in the second and third blocks of the POL index respectively. Similarly, since the length of the repeat "GAG" represented by node 9 is 3 nucleotides, and node 9 has 2 leaf children, we store values 3 and 2 in the header of the record for node 9 (Figure 4.2). The children of node 9 are nodes 11 and 12, which represent suffixes starting at locations 1 and 3, stored in the last 2 POL blocks.

To further improve the search performance for supermaximal repeats of length greater than a particular threshold value, we store the POL records in descending order with respect to the length of each potential supermaximal repeat (recorded in the first 27 bits of its header). Once the length of a particular supermaximal candidate becomes smaller than the threshold value, the SMR terminates without processing the remaining POL index.

4.2 POL index construction algorithm

In the previous chapter, we explained the general idea of the POL index. However, if we are to record all ST candidate nodes, this will result in large POL index sizes (comparable to the size of the STTD64 index), leading to high POL construction costs. Given that biologists are usually interested in supermaximal repeats greater than certain minimum length, we use this information in POL index creation. Therefore, our implementations of the POL construction algorithm and SMR search algorithm are flexible and allow for creating and using a desired POL index with minimum index length (MIL) that is relevant to the requirements of a particular application.

In our experiments, we consider 4 POL index structures with different MIL: POL10, POL25, POL100, and POL200. In POL10, we record all candidate nodes with repeat length of at least 10 nucleotides (i.e., MIL = 10). This POL index allows for improving the search time for supermaximal repeats of size at least 10 nucleotides. Similarly, in POL25, POL100, and POL200, we record all candidate nodes with length greater or equal to their MIL (i.e., 25, 100, 200) respectively, which will result in faster search for supermaximal repeats of at least 25, 100, and 200 nucleotides, respectively. In the next chapter, we study the construction cost of these 4 indexes, both in terms of construction time and storage space.

The POL index construction algorithm [Lian et al., 2008] is presented as Algorithm 2 . The algorithm takes as input:

1. The STTD64 index of the sequence to be searched for supermaximal repeats.

2. A user-defined minimum index length (MIL) of the candidate nodes that are to be recorded.

The output is an index, called POLMIL, which supports efficient search for supermaximal repeats of size at least MIL.

The construction algorithm traverses the STTD64 index sequentially, examining the leaf nodes. For a leaf node u, the algorithm compares its depth to MIL (Step 6) to eliminate ineligible nodes. Recall that the depth of a node is defined as the number of characters on the path from the root to the parent of the node. Thus, this step correctly identifies if a branch node v - the parent of u, meets the minimum length criterion. In steps 9 to 17, the algorithm checks if all siblings of u are leaf nodes. If

Algorithm 2 POL Construction Algorithm (STTD64 index ST, minimum index length MIL)

```
1: v points to first ST node
2: while there are unexamined ST nodes do
3:    while v is a branch node do
4:       v points to the first child of v
5:    end while
6:    while DEPTH(v) < MIL do
7:       v points to next node
8:    end while
9:    repeat
10:      if v is a leaf then
11:         leafcounter ++; allleaves = true
12:      else
13:         allleaves = false
14:      end if
15:      v points to next node
16:   until all children are examined || allleaves == false
17:   if allleaves is true then
18:      v.header.length = DEPTH(v)
19:      v.header.numofoccurrences = leafcounter
20:      for i = 0 to leafcounter do
21:         data[i] = LP(v) - DEPTH(v)
22:      end for
23:   end if
24:   v points to next node;
25: end while
26: sort records in descending order of v.header.length
27: write records to disk
```

this is the case, a POL record for this candidate v is created (steps 18 to step 24). In step 27, we sort the candidates records in descending order, to speed up the search algorithm (as described in previous section). Finally, after sorting the POL records, the construction algorithm writes sorted records into disk as the POLMIL index.

In this algorithm, there are two functions named LP() and DEPTH(). Function LP() is used to retrieve lp value from a STTD64 unit, and DEPTH() returns the depth value of a leaf node storing in suffix tree ST.

Next, we describe how to compute the starting position of a suffix represented by a leaf node. As mentioned in Chapter 2, the starting locations of the suffixes in S are not explicitly stored in the STTD64 representation (Figure 2.6), but rather calculated as follows. For each leaf node u, the starting position of the suffix represented by u is determined by subtracting the depth value of u from its lp value, since lp value is the starting position in S of the substring encoded from the root to u plus the depth of node u. In the STTD64 index structure, we store the depth values in leaf nodes directly. Therefore, the calculation of starting position becomes simple and efficient.

Consider the suffix tree in Figure 2.2 with $MIL = 2$. For node 7, which is a left-most leaf with *depth* > 2, the algorithm performs the while loop in step 10 to check if node 8 (the right sibling of node 7) is a leaf node. Since this is the case and node 8 is a rightmost child, the algorithm goes to Step 19 to create a record representing the branch node 5 - the parent of leaf nodes 7 and 8 (see Figure 4.2). The same steps are executed when node 11 is processed, which results in creating a record in the POL index representing node 9. Last, the candidate node records are sorted based on the repeat length in descending order, but in our example this is already the case. Figure 4.2 shows the final POL2 index for this example.

Assuming that the STTD64 index has already been created, the POL construction algorithm reads the entire STTD64 index in sequential order, which results in $O(n)$ constant time operations, where n is the number of nodes in STTD64. The sorting in Step 27 is done in time $O(r \log r)$, where r is the number of records in POL. Since r is much less than n, the overall time complexity of POL construction algorithm is $O(n \log n)$.

4.3 SMR algorithm

Our proposed SMR algorithm [Lian et al., 2008] is presented as Algorithm 3. It takes as input the sequence to be searched, its POL index, and a user-defined parameter, which indicates the requested minimum length of the supermaximal repeats. The output of the algorithm contains the starting positions in the sequence and the lengths of all supermaximal repeats satisfying minimum repeat length constraint.

The SMR algorithm first examines the minimum repeat length (MRL) parameter with the POL minimum index length (MIL). In case that MRL is less than the POL MIL, the SMR algorithm loads the STTD64 index instead of POL index and runs Gusfield's algorithm. Otherwise, SMR loads the POL index and compares the length of each candidate to the *min_len* value (Step 7) starting from beginning of the index. If the current record represents a candidate with a length at least equal to *min_len*, the algorithm reads the repeat occurrence positions from the data blocks of this record (steps 8 to 12). In Step 13, the left diversity of these occurrences is examined, and if successful, the discovered supermaximal repeats are returned as output (Step 15). The sequential examining of POL records proceeds until all the POL records are examined or until the length of a candidate becomes smaller than the *min_len*. Since the records are sorted in descending order of lengths of candidates, no other supermaximal repeats which would satisfy the specified length constraint exist after the length comparison fails. Thus, the SMR algorithm correctly terminates without examining unnecessary nodes. This feature optimizes the search time

Algorithm 3 SMR Search Algorithm(Sequence S, index POL, requested minimum length min_len)

```
1: if $min\_len < MIL$ then
2:     load STTD64 index
3:     run Algorithm 1
4: else
5:     load $POL$ index
6:     $unit$ = first $POL$ record;
7:     while $unit.header.length > min\_len$ do
8:         $y = unit.header.numofoccurrences$
9:         for $i = 0$ to $y$ do
10:            $position[i] = unit.data[i]$
11:            retrieve left character for the $position$ from sequence $S$
12:        end for
13:        compare left characters of all occurrences
14:        if all left characters are distinct then
15:            output array $position$ and $unit.header.length$
16:        end if
17:        $unit$ = next $POL$ record;
18:    end while
19: end if
```

performance at the SMR algorithm especially when the MRL is much greater than the MIL. For example, searching for supermaximal repeats larger than 2000 and using the POL10 index, it is possible that most records in this index do not satisfy the length condition, and hence examining only the few possible candidates improves the search performance in this case.

Let us consider the running sample sequence S = AGAGAGC\$ (shown in Figure 2.2) and a *min_len* value 4. The SMR algorithm starts with reading the first POL record, which represents the candidate repeat at node 5, whose length is 4 characters and satisfies the *min_len* constraint. Then SMR reads the two subsequent data blocks (as instructed by $y = 2$) and retrieves the two positions $S[0]$ and $S[2]$, where the candidate of supermaximal repeat starts. Since the left character of the suffix starting at position $S[0]$, i.e., $S[-1]$, is different by default from any characters in the sequence, the two suffixes are left diverse and thus SMR outputs the supermaximal repeat found, which is of length 4 and its two occurrences start at positions $S[0]$ and $S[2]$. The algorithm then reads the next POL block, which is the header for the candidate repeat at node 9. Since its length is 3, which is less than *min_len*, there is no need to further examine the POL index and the search process terminates.

The main advantage of our SMR algorithm over the Gusfield's algorithm (Algorithm 1) is that it does not read the entire ST index for a sequence but rather only a considerably smaller POL index. By using POL, which replaces the ST index, the SMR algorithm avoids the first step of Gusfield's algorithm for finding suitable parent nodes from ST, which is rather costly in terms of search time. Further, in some cases (i.e., MRL >> MIL), even not the full POL index has to be processed in order to find all existing supermaximal repeats. As a result, our SMR algorithm exhibits a considerable decrease in the number of disk I/O operations, which in turn leads to faster supermaximal repeats search time, compared to Gusfield's solution, as shown in the chapter 5.

Chapter 5

Experiments and Results

In this chapter, we first study the cost of construction of POL index for different minimum lengths (10, 25, 100, 200) both in terms of time and storage requirements. We then evaluate the performance of our proposed SMR technique on real-life DNA sequences using the four POL indexes. We compare our search times with Vmatch [Kurtz, 2000], a suffix array based search tool. Also, we investigate the number of supermaximal repeats found. Finally, we conduct additional experiments with synthetic DNA sequences in order to further evaluate SMR technique.

All experiments are performed on a typical desktop computer with Intel Pentium 4@3GHz, 2GB RAM, 300GB HDD, and 2MB L2 cache, running Linux kernel 2.6.14. The construction and search times reported are real times in seconds (measured using the time command in Linux). The POL index construction and SMR search algorithms are implemented in C. The SMR search service is available online for evaluation and use from the web site of the FASST project at http://sepehr.cs.concordia.ca.

As real-life DNA data, we used the 24 homo sapiens chromosomes which include 22 autosomes, X and Y chromosomes as sequences to be searched for supermaximal repeats. The data was obtained from NCBI (National Center for Biotechnology Information). We removed all the unknown nucleotides (indicated by character N), resulting in sequences of size range 26 to 238 million bases. As for synthetic sequences, we build 24 sequences with the same size as 24 real human chromosomes, but generate letters " A, C, G, T" randomly.

5.1 POL index construction

As discussed in Chapter 4, recording all candidate nodes from ST will result in huge POL index size. Instead, in our first set of experiments, we consider 4 alternative minimum index lengths (MIL): 10, 25, 100, and 200 nucleotides, for which we construct the corresponding POL indexes POL10, POL25, POL100, and POL200. For example, POL200 records all candidate ST nodes whose repeat lengths are at least 200 nucleotides.

Figure 5.1 shows construction time for these four indexes. The construction time for each POL index is the sum of the construction time for all 24 chromosomes. The POL200 index has the fastest construction time, while POL10 is the slowest, being about twice slower than POL200.

Next, we study the sizes of these four POL indexes. To show the relationship between the index size and sequence size, we consider the ratio of POL index size and sequence size in Figure 5.2, in which we show the average sizes for all the 24 chromosomes. POL200 has the smallest storage requirement, which is on average about 6% of the sequence size. Note that the size of STTD64 index is 13 times of the sequence size on average. That is, POL200 index is more than 200 times smaller than the STTD64 index. The POL10 index is the largest among the four indexes, nearly 4 times bigger than the sequence, and about one third of the STTD64 index. POL100 is comparable to POL200, and POL25 is about half size of the input sequence.

In conclusion, the POL200 index has the fastest construction time and the small-

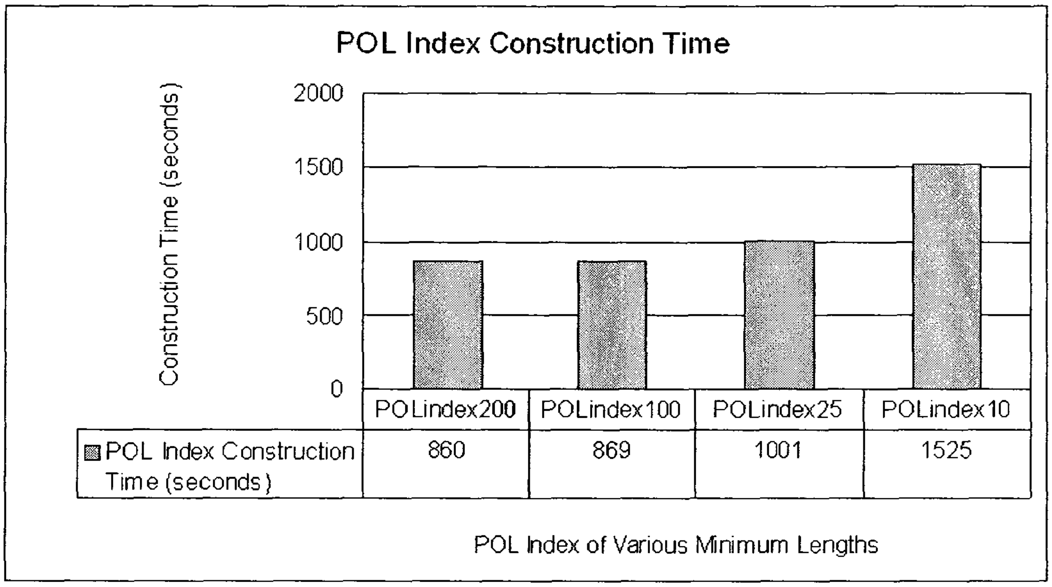


Figure 5.1: Construction time for various POL index lengths

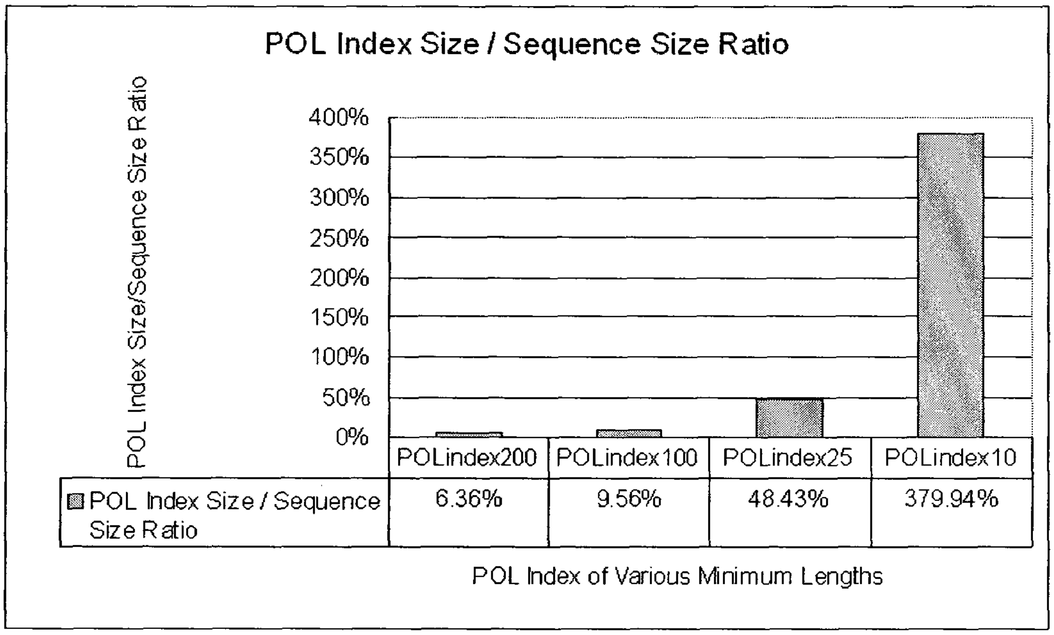


Figure 5.2: Index size/sequence size ratio for various POL index lengths

est storage requirement. This index can be used to improve the search performance of SMR only if the minimum repeat length (MRL) of supermaximal repeats is at least 200 nucleotides. POL100, POL25, and POL10 can be used for smaller MRL values, at the cost of increased construction time and storage space, since decreasing minimum repeat length leads to more candidate nodes to be identified from STTD64 and recorded in POL index. For example, POL25 requires additional 140 seconds in order to record 8 times more candidate nodes compared to POL200, but supports efficient SMR search for threshold value 25 or more nucleotides. Constructing the POL10 index requires almost twice the construction time of POL200 and results in a 60 times larger index (but still 3 times smaller than STTD64), and supports searching for supermaximal repeats of almost all practical sizes. We remark that for the 24 human chromosomes considered, searching for repeats with minimum lengths smaller than 10 nucleotides is not practical in general, as discussed later in section 5.3.

The choice of an "appropriate" minimum repeat length for the POL index construction is application dependent. Our construction algorithm allows the user to specify a value for this parameter which suits the needs of a particular application, thus providing a suitable trade-off between construction time and storage space on one hand, and search time on the other.

5.2 SMR search performance

In our second set of experiments, we evaluate the search time performance of SMR when using the 4 POL indexes and compare our results with Vmatch [Kurtz, 2000]. In these experiments, we used 14 different threshold values for the supermaximal repeats, ranging from 1 to 10,000 nucleotides. If the threshold is smaller than the MIL in a particular POL index, the SMR algorithm uses the general STTD64 index instead. Figure 5.3 reports the measured cumulative search times (for all 24 chromosomes) for the four SMR runs and Vmatch.

We make the following two important observations. First, if the MRL is greater than the MIL of two or more POL indexes, the SMR algorithm provides the similar

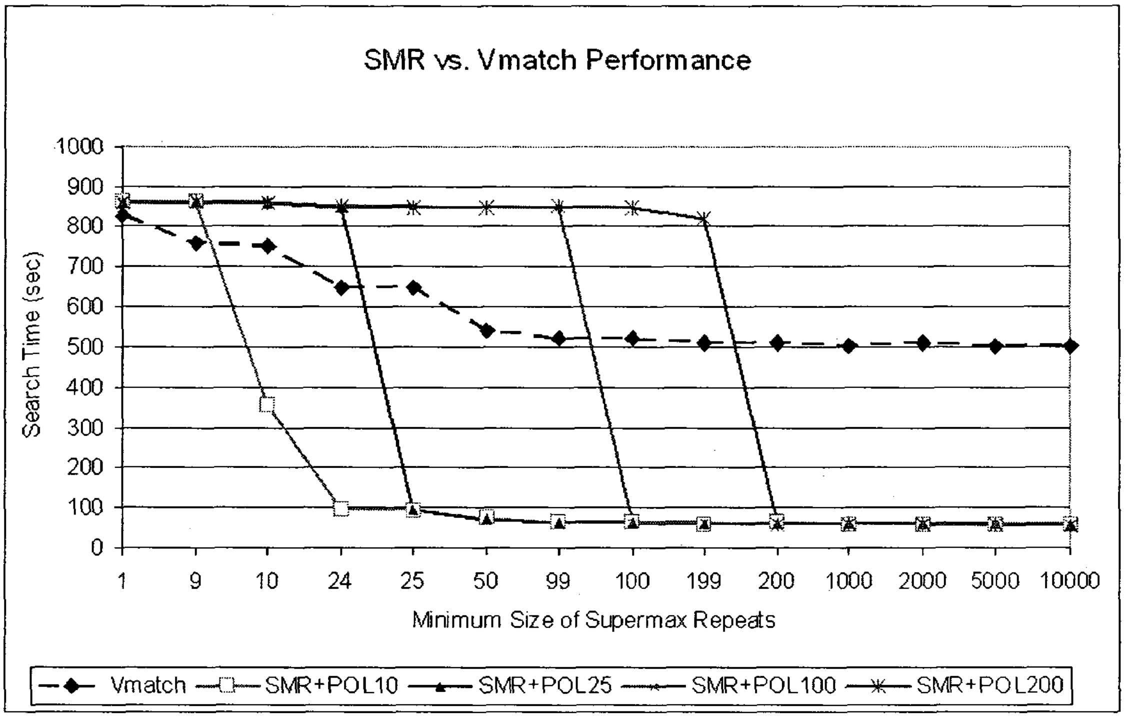


Figure 5.3: Vmatch vs. SMR with different POL index

search time performance, regardless of which POL index is used. For example, SMR exhibits very similar search times using either POL10 or POL25 for MRL larger than 24 nucleotides. Also, SMR exhibits identical performance using any of the 4 POL indexes for MRL above 199 nucleotides. This is explained by noting that regardless of which particular POL index is used, the number of candidate nodes that represent supermaximal repeats of desired lengths is the same. Since the records in the POL index about the candidate nodes are kept in descending order, the SMR algorithm processes the same number of POL records, which leads to identical search times using any of the 4 POL indexes. This observation implies that if a particular search application requires only finding supermaximal repeats of size hundreds or thousands of nucleotides, then POL200 would be a suitable index choice due to its fast construction time and small storage requirement.

Second, we note that provided with a suitable POL index, SMR is significantly faster in finding supermaximal repeats compared to Vmatch. For example, for MRL value of 10 nucleotides, SMR with POL10 is 2 times faster than Vmatch; for a MRL value 25, SMR is 7 times faster using either POL10 or POL25. SMR is more than 8 times faster than Vmatch for searching MRL value of 100 nucleotides using any one of POL10, POL25, or POL100 indexes. We are about an order of magnitude faster for MRL values at least 200 nucleotides, using any of the four POL indexes. On the other hand, for threshold values less than 10 nucleotides, the construction of a POL index is not recommended for being too costly. While cases with threshold values less than 10 may not be frequent in practice, our proposed SMR algorithm in such cases can directly use the STTD64 index, resulting in only about 10% slower times compared to Vmatch. Appendix A shows detail experimental data.

The above results are based on the assumption that a POL index has already been constructed and is available to SMR. However, an important practical question is: how many requests for computing supermaximal repeats should be posed against a particular sequence so that the cost constructing the POL index by processing the available STTD64 index is justified and amortized, and SMR would be preferable to Vmatch solution? We consider this question from two points of view, as follows.

First, Figure 5.4 answers this question at a higher level. It reports SMR search performance including the POL index construction time, for various number of searches in a particular sequence. The depicted search time is the average for 12 different MRL values of the supermaximal repeats, ranging from 10 to 10,000 nucleotides for all the 24 chromosomes. We observe that the search cost reduces as the number of SMR searches increases. The SMR algorithm using POL25 outperforms Vmatch when performing two or more on a particular sequence. SMR with POL10 costs less than Vmatch over 3 searches. Also, performance of SMR with POL100 exceeds Vmatch after 5 searches. In the worst case, SMR with POL200 has comparable performance with Vmatch after 10 searches. We observe that SMR using POL25 has the best performance in our four POL indexes to compete with Vmatch when considering POL

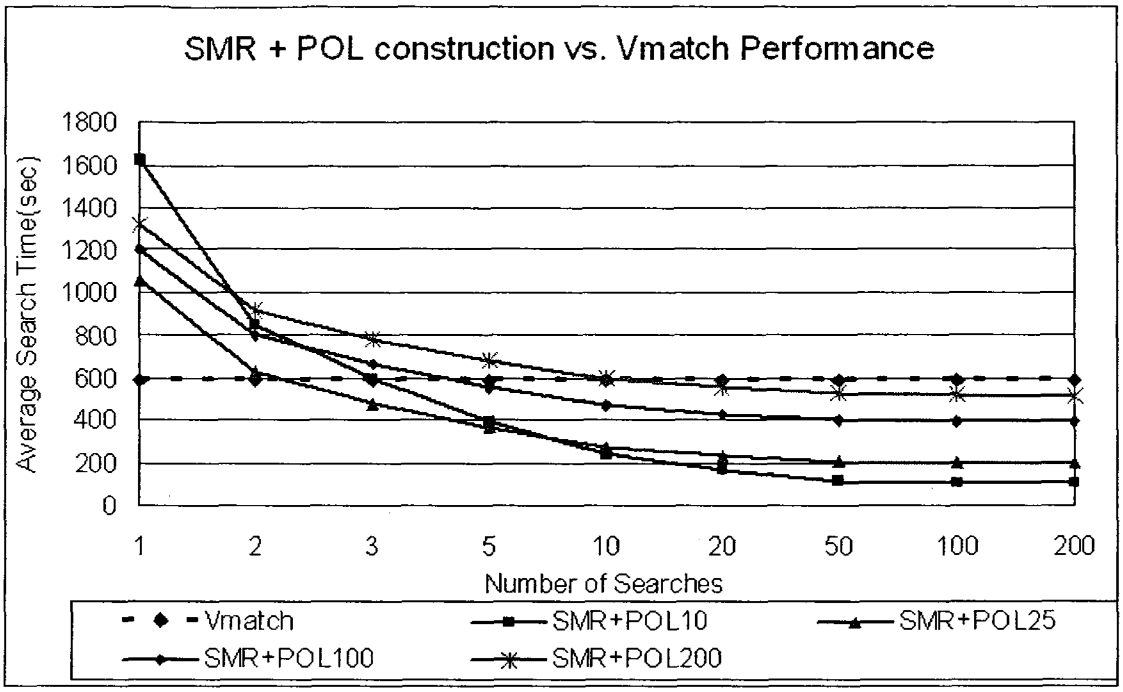


Figure 5.4: Vmatch vs. SMR considering POL index construction time

index construction cost. But is it always true whatever minimum repeat length we request?

To answer this question, we evaluate the performance at more detailed level. Figures 5.5, 5.6, 5.7, and 5.8 illustrate the performance results for various MRL value in POL10, POL25, POL100, and POL200, respectively.

In Figure 5.5, we observe that SMR with POL10 (including construction time) outperforms Vmatch when performing at least 4 searches of minimum size of 10 nucleotides supermaximal repeats. SMR+POL10 is compared to Vmatch more than 3 searches of minimum 25 nucleotides. As can be seen in Figure 5.6, SMR with POL25 has better performance than Vmatch after 2 searches for supermaximal repeats of minimum 25 nucleotides. Similarly, in Figure 5.7, SMR with POL100 performs better than Vmatch at more than 2 searches of minimum 100 nucleotides of supermaximal repeats. As shown in Figure 5.8, SMR with POL200 is better than Vmatch when searching at least twice of supermaximal repeats of minimum 200 nucleotides.

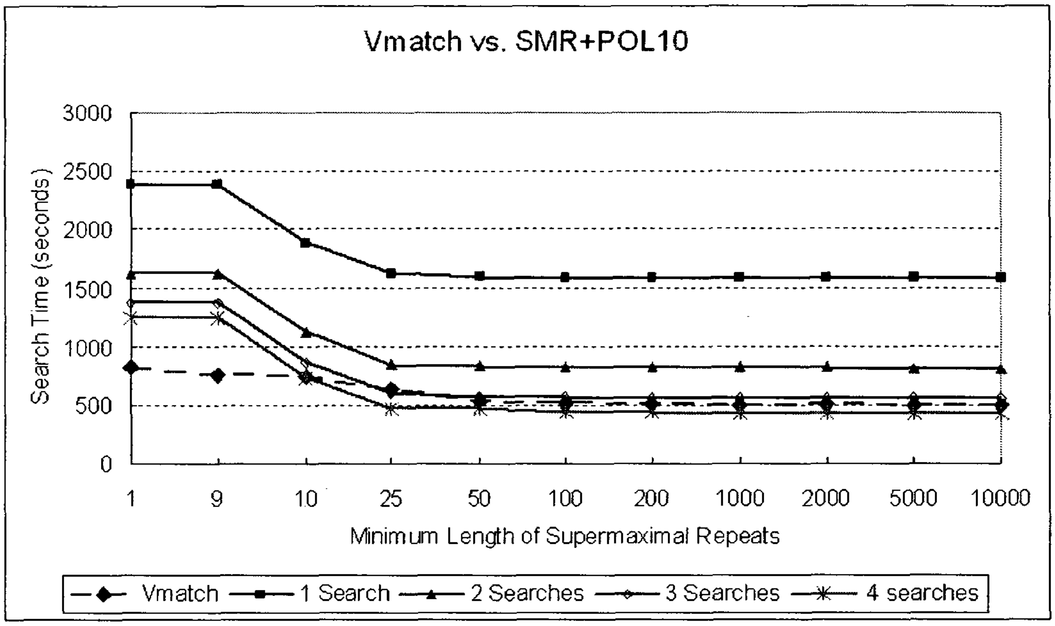


Figure 5.5: Vmatch vs. SMR + POL10 with construction time

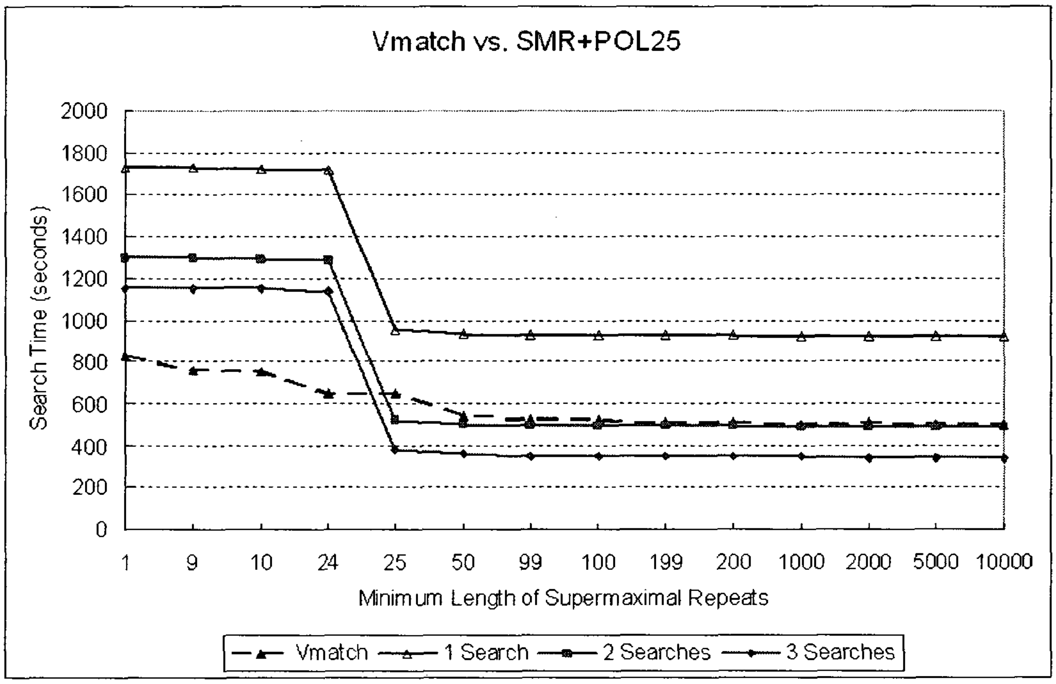


Figure 5.6: Vmatch vs. SMR + POL25 with construction time

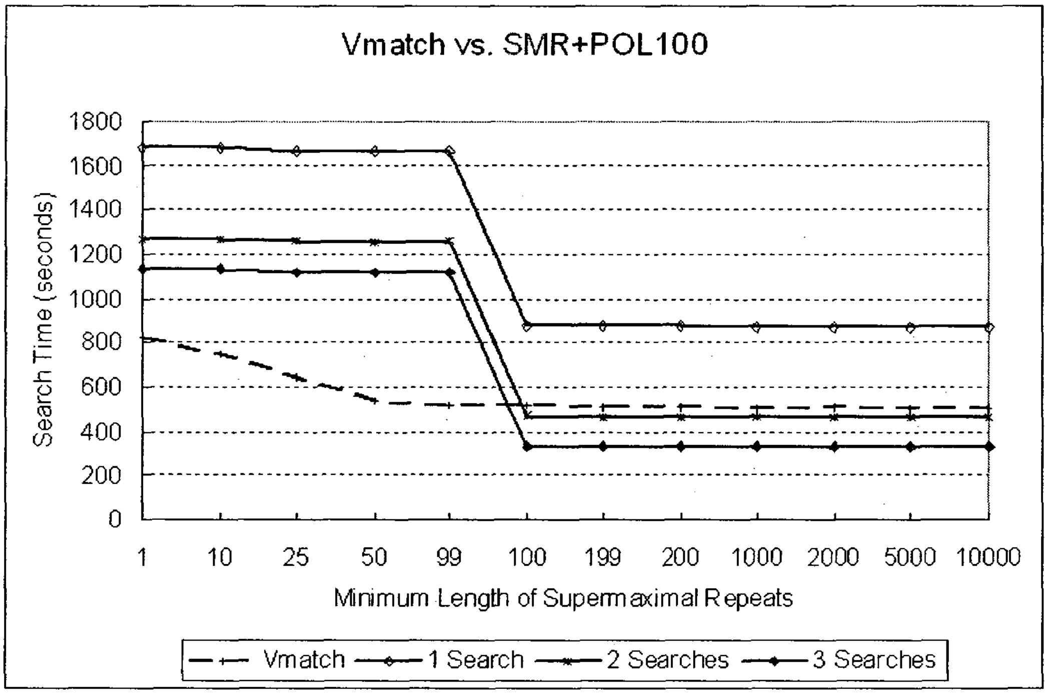


Figure 5.7: Vmatch vs. SMR + POL100 with construction time

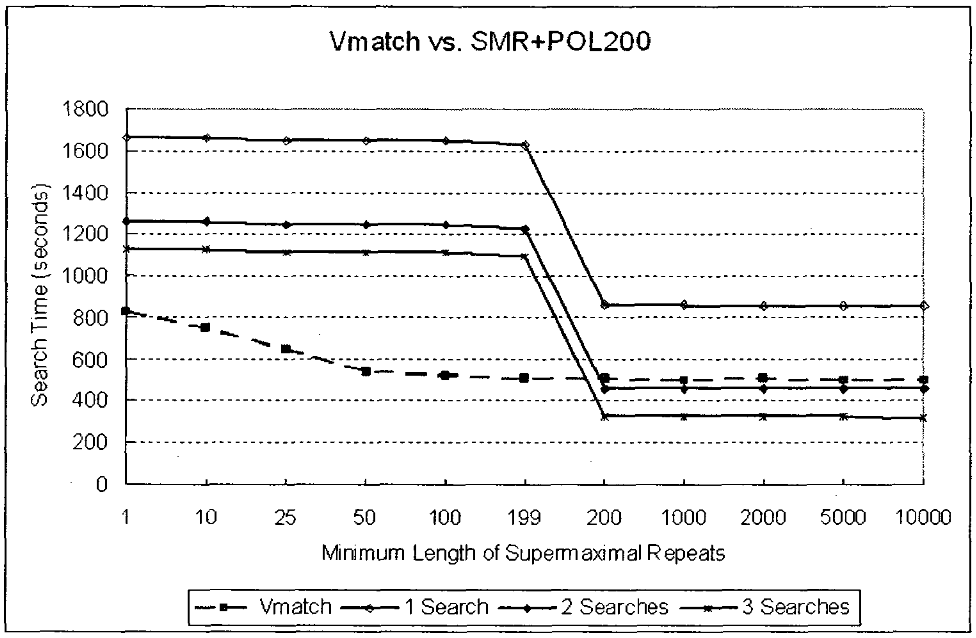


Figure 5.8: Vmatch vs. SMR + POL200 with construction time

From our results above and their analysis, we conclude that our approach of using SMR with proper POL index is better when there are 2 or more search tasks with length at least 25 nucleotides or there are 4 or more searches with length at least 10 nucleotides on the same sequence.

5.3 Number of supermaximal repeats

We also studied the number and the size of supermaximal repeats in the 24 human chromosomes and present the results in Figure 5.9.

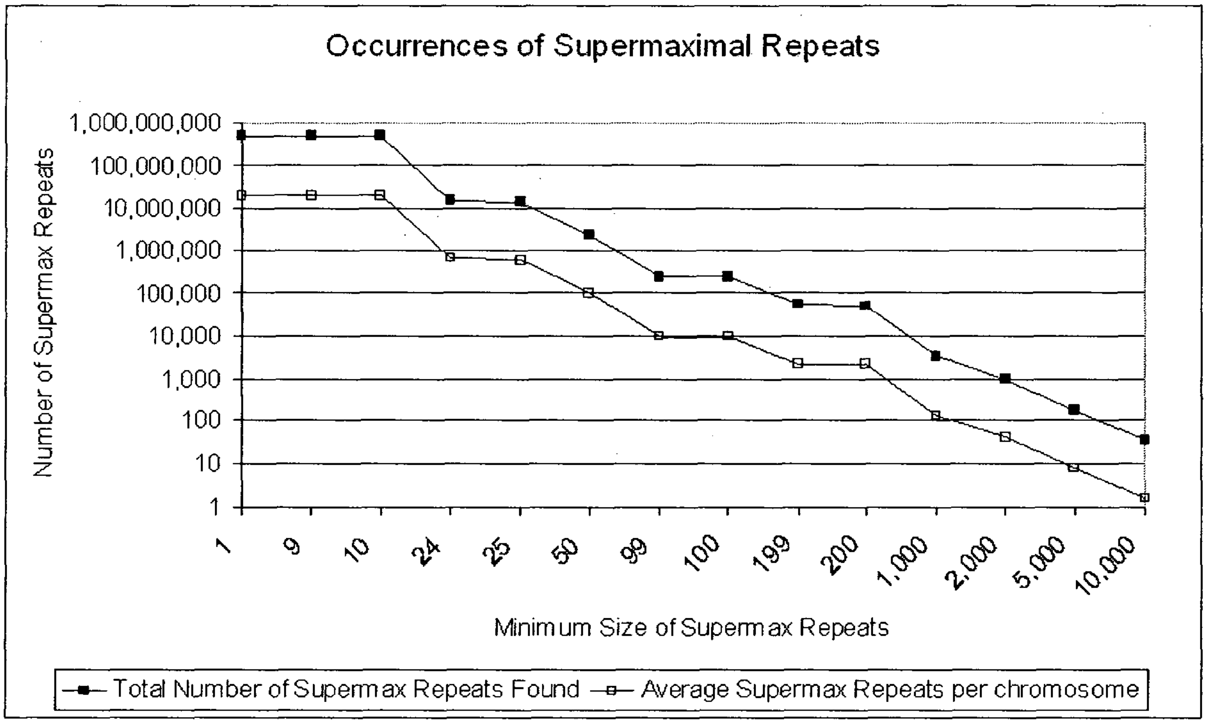


Figure 5.9: Occurrences found in 24 human chromosomes

As can be seen from the figure, increasing the minimum repeat length in the range from 1 to 10 nucleotides does not lead to a significant decrease in the number of supermaximal repeats found. For such small MRL values, we find almost half a billion repeats in the collection of 24 chromosomes with total size of around 2.8 billion bases. The size of search results contradicts the idea of supermaximal repeats search, which is intended as a high-level and concise investigation tool for initial analysis of repetitive structures in biological sequences. Further, there is a high possibility that most supermaximal repeats of size less than 10 nucleotides found in sequences containing tens and hundreds million bases occur purely by chance, and thus may not carry any structural or functional information. For these reasons, we believe searching with MRL values less than 10 nucleotides should not be viewed as a primary application of supermaximal repeats search in large DNA sequences. Thus, the slower SMR performance in such cases (up to 10% slower than Vmatch) would not pose a restriction in its use.

5.4 Synthetic DNA data

To further evaluate the performance of SMR, we study how the POL construction and SMR technique work with synthetic DNA sequences in this section. Does the POL index for synthetic data occupy reasonable space and have satisfying construction time? Does SMR have comparable performance running with synthetic data as with real-life DNA data? Does it still outperform Vmatch solution?

To answer these questions, we evaluate the POL index construction, SMR search performance, and number of supermaximal repeats and their sizes using a set of synthetic DNA sequences, which are generated randomly by computer program and have the same character set as real-life DNA, i.e. A, C, G, T. We build 24 synthetic DNA sequences with the same sizes as corresponding human chromosomes. Then we compare the performance of our POL construction algorithm and SMR search algorithm running on synthetic sequences against real-life DNA sequences. We also compare SMR to Vmatch on synthetic sequences. Finally, we study the occurrences of supermaximal repeats for synthetic data.

5.4.1 POL index construction

We construct various POL indexes for synthetic DNA sequences, which are POL10, POL15, POL20. Figure 5.10 shows the ratio of average POL index size/original sequence size for synthetic and real DNA data. The ratio of synthetic data is similar to the real DNA data in POL10, but size of POL15 of synthetic data is only half of its original sequence, while that of real data is one and half times of its original sequence. For POL20, synthetic data has much less size than real data. This result implies that the occurrences of supermaximal repeats in DNA data are much more than synthetic data. Thus, efficient techniques for repeats finding, such as SMR, are needed.

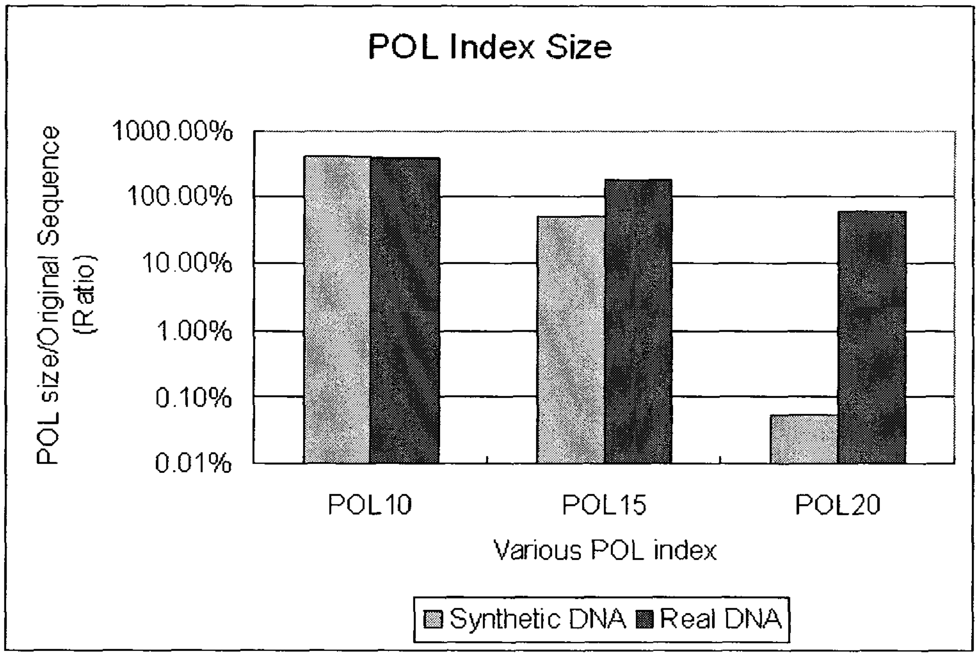


Figure 5.10: POL index size analysis : synthetic vs. real

Furthermore, we evaluate POL index construction time for synthetic data and real data. As shown in Figure 5.11, they have similar construction time for POL10,

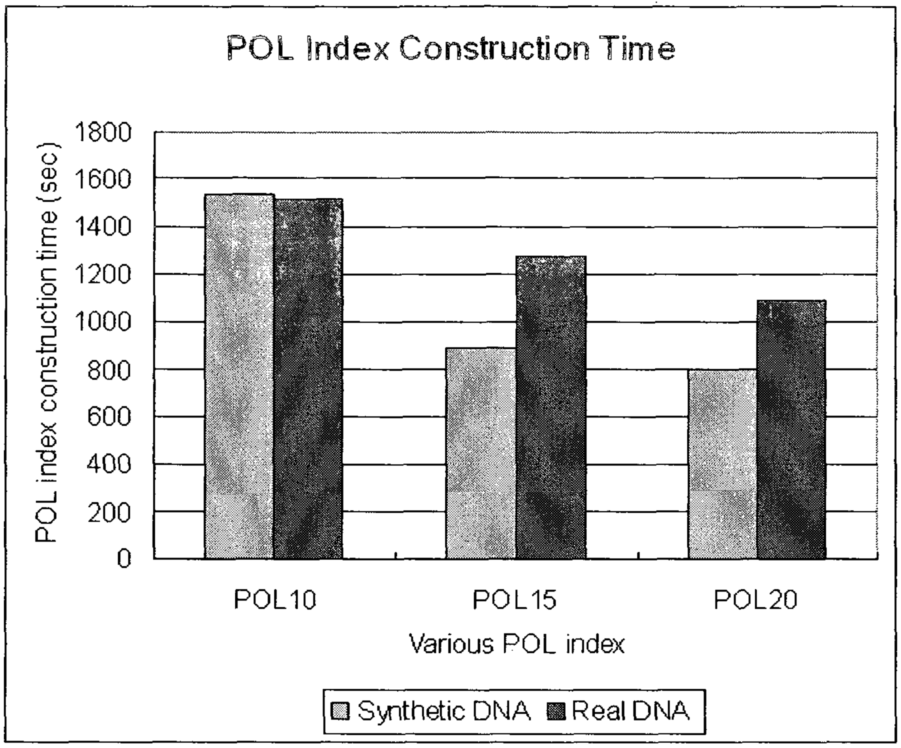


Figure 5.11: POL index construction time : synthetic vs. real

but POL15 and POL20 index constructions for synthetic data are much faster than real data due to their less occurrences of repeats.

5.4.2 SMR search performance

In this set of experiments, we evaluate the SMR search time performance using POL10, POL15, and POL20 indexes running with synthetic data, and then compare our results with Vmatch [Kurtz, 2000] and real-life data. In these experiments, we used 8 different threshold values for the supermaximal repeats, ranging from 1 to 25 nucleotides.

Figure 5.12 reports SMR search performance using POL10, POL15, and POL20, comparing with Vmatch performance. We observe that SMR is significantly faster than Vmatch when the requested minimum length is greater than or equal to the threshold of POL index. This observation is in accord with our finding for real-life DNA.

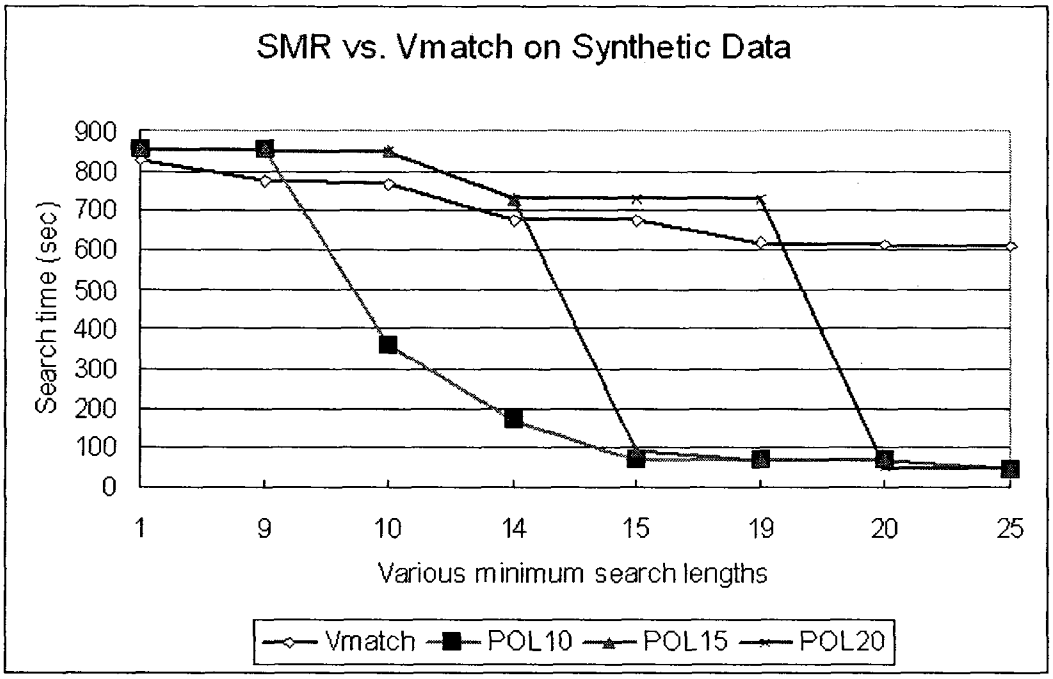


Figure 5.12: SMR vs. Vmatch performance on synthetic data

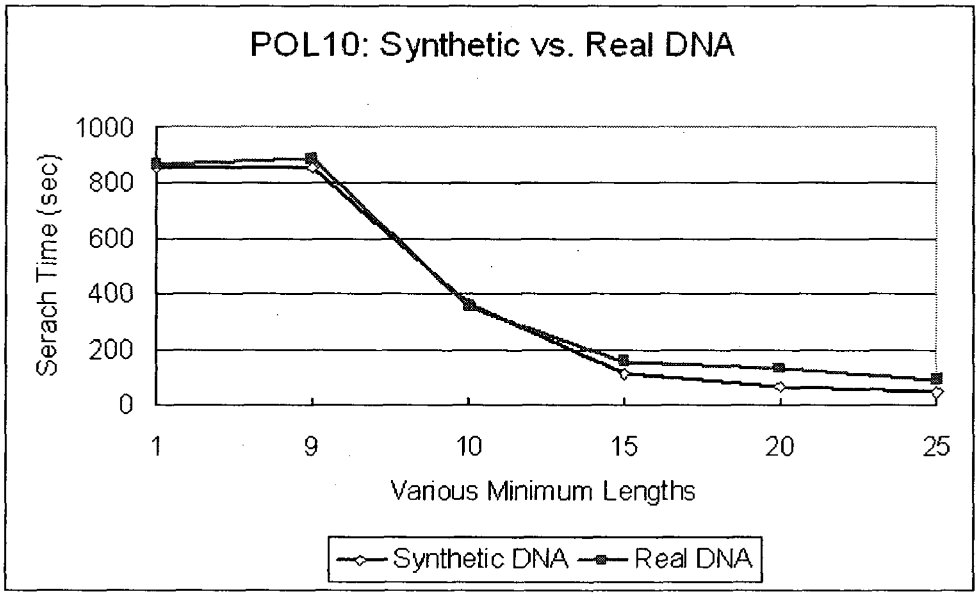


Figure 5.13: SMR search + POL10 : synthetic vs. real

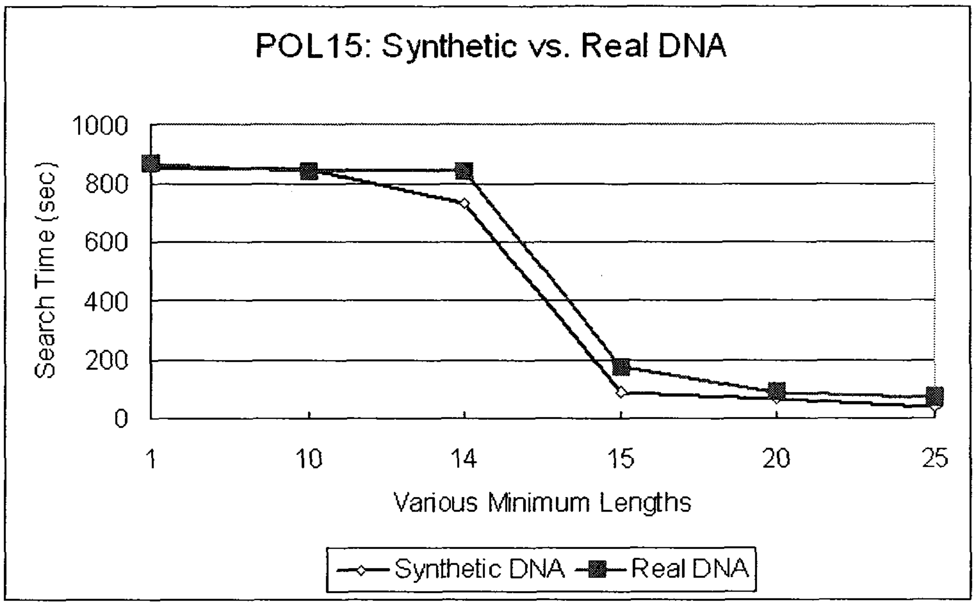


Figure 5.14: SMR search + POL15 : synthetic vs. real

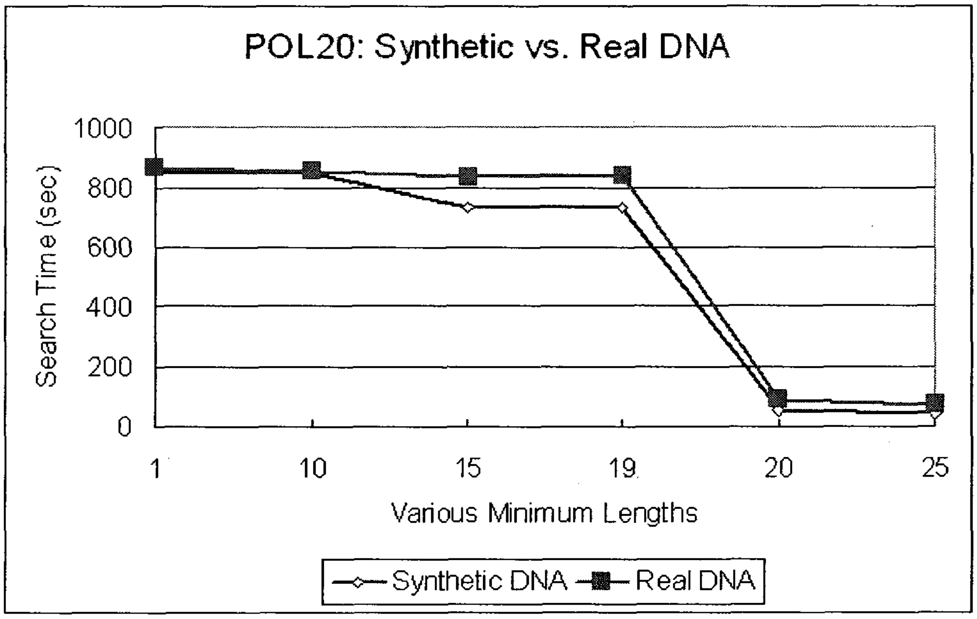


Figure 5.15: SMR search + POL20 : synthetic vs. real

Next, we evaluate SMR search performance on synthetic DNA sequences and human chromosomes. Figures 5.13, 5.14, and 5.15 exhibit SMR performance on synthetic DNA data and real-life human DNA data. As shown in the Figures, SMR with POL10 working on synthetic data performs as good as running on real data. And SMR using POL15 or POL20 on synthetic data is even faster than it runs on real data.

5.4.3 Number of supermaximal repeats

As already discussed in section 5.4.1, there are much fewer supermaximal repeats in synthetic data compared to real DNA. In this section, we study this issue in more details. Figure 5.16 shows the number of repeats found in synthetic data for different length thresholds. We observe that about 360 million supermaximal repeats have 10 to 15 characters, which represent more than 75% of all supermaximal repeats in the sample sequences. Also, in synthetic data, supermaximal repeats longer than 20 nucleotides are very rare. This observation shows the difference between synthetic random DNA sequences and real-life DNA sequences, and explains why POL15 and POL20 indexes of synthetic data are much smaller than real-life DNA data.

From above experiments and discussions, we observe that repeats in DNA occur much more often than in randomly generated strings. Therefore, repeats are biologically important, and efficient techniques for their finding, such as SMR, are needed. Moreover, SMR search algorithm using POL index exhibits outstanding performance in real DNA data, as well as in random strings.

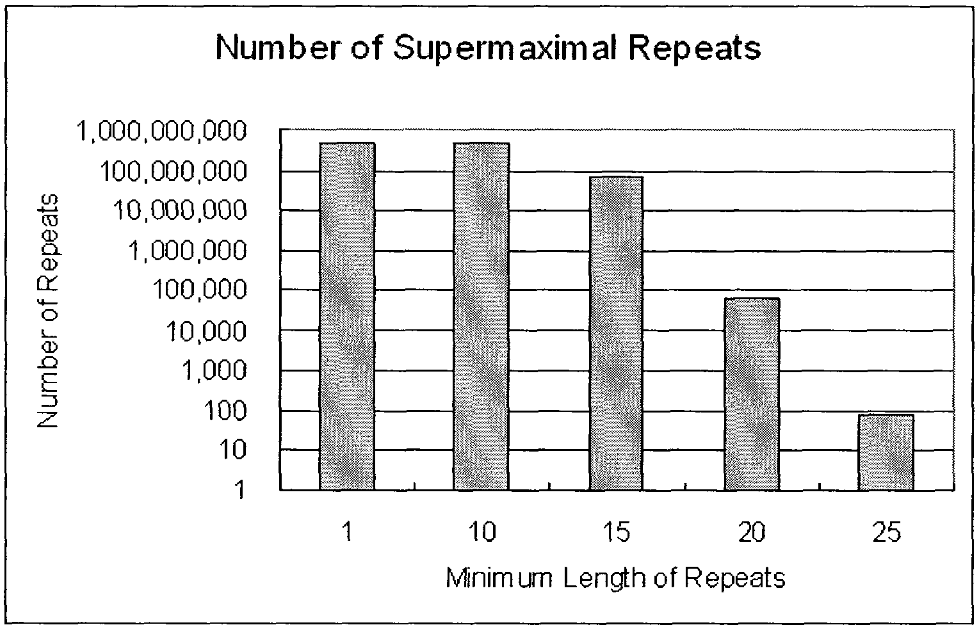


Figure 5.16: Number of supermaximal repeats

Chapter 6

Web Based Interface

In this chapter, we briefly introduce FASST project (Fast And Scalable Search Tool for biological sequence data) and its web site http://sepehr.cs.concordia.ca/ developed using HTML, PHP and Perl languages. We then demonstrate the SMR application through this web interface.

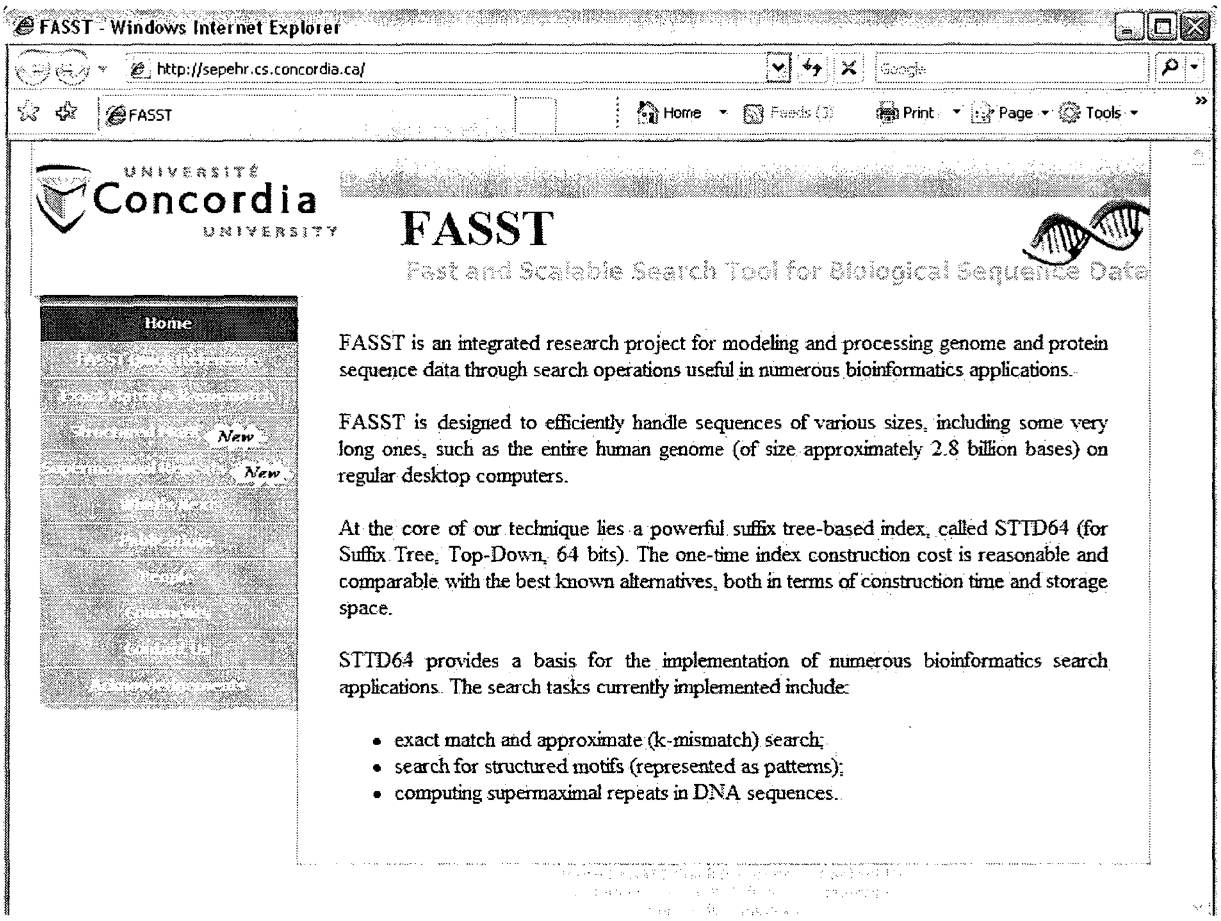
6.1 FASST project

FASST (**F**ast **A**nd **S**calable **S**earch **T**ool for biological sequence data), is an integrated research project for modeling and processing genome and protein sequence data and which provides support for various search applications. The tool uses the STTD64 index, proposed and developed by [Halachev et al., 2007] in our project.

FASST is designed to efficiently handle sequences of various sizes, including some very long ones, such as the entire human genome (of size approximately 2.8 billion bases) on a typical desktop computers.

The search tasks currently implemented and supported as part of the FASST project include:

- exact match and approximate (k-mismatch) search;
- search for structured motifs (represented as patterns);
- computing supermaximal repeats in DNA sequences.


FASST - Windows Internet Explorer
http://sepehr.cs.concordia.ca/
FASST
Home
Print
Page
Tools
UNIVERSITÉ Concordia UNIVERSITY
Home
New
New

FASST

Fast and Scalable Search Tool for Biological Sequence Data

FASST is an integrated research project for modeling and processing genome and protein sequence data through search operations useful in numerous bioinformatics applications.

FASST is designed to efficiently handle sequences of various sizes, including some very long ones, such as the entire human genome (of size approximately 2.8 billion bases) on regular desktop computers.

At the core of our technique lies a powerful suffix tree-based index, called STTD64 (for Suffix Tree, Top-Down, 64 bits). The one-time index construction cost is reasonable and comparable with the best known alternatives, both in terms of construction time and storage space.

STTD64 provides a basis for the implementation of numerous bioinformatics search applications. The search tasks currently implemented include:

- exact match and approximate (k-mismatch) search;
- search for structured motifs (represented as patterns);
- computing supermaximal repeats in DNA sequences.

Figure 6.1: The home page of FASST web interface

The FASST tool has interactive interfaces for our exact match and k-mismatch search, motif search, and supermaximal repeat search applications. We also provide quick references for these bioinformatics search applications and other relevant information on the web site. Figure 6.1 is a screenshot of the FASST home page.

As this thesis focusing on development of SMR for finding supermaximal repeats, we next illustrate the FASST interface for this search task.

6.2 Supermaximal repeats search demonstration

In this section, we demonstrate a supermaximal repeats search through FASST web interface. From the home page of FASST (Figure 6.1), we select *supermaximal repeats* option on the left menu. This opens the interface related to the supermaximal repeats search, shown in Figure 6.2. There are three options to select a sequence and proceed with a search:

Option 1: Search in the following sample sequences that are already uploaded to our server and their indexes are created. All header lines, comments, symbols that do not represent a nucleotide (e.g., blanks, new line characters, etc.), and Ns (unknown nucleotides in DNA data) are being removed prior to index construction. The sample sequences currently available are as follows:

- chr_Y (25 MB) - Human chromosome Y;
- chr_15 (81 MB) - Human chromosome 15;
- chr_8 (143 MB) - Human chromosome 8;
- chr_1 (225 MB) - Human chromosome 1;

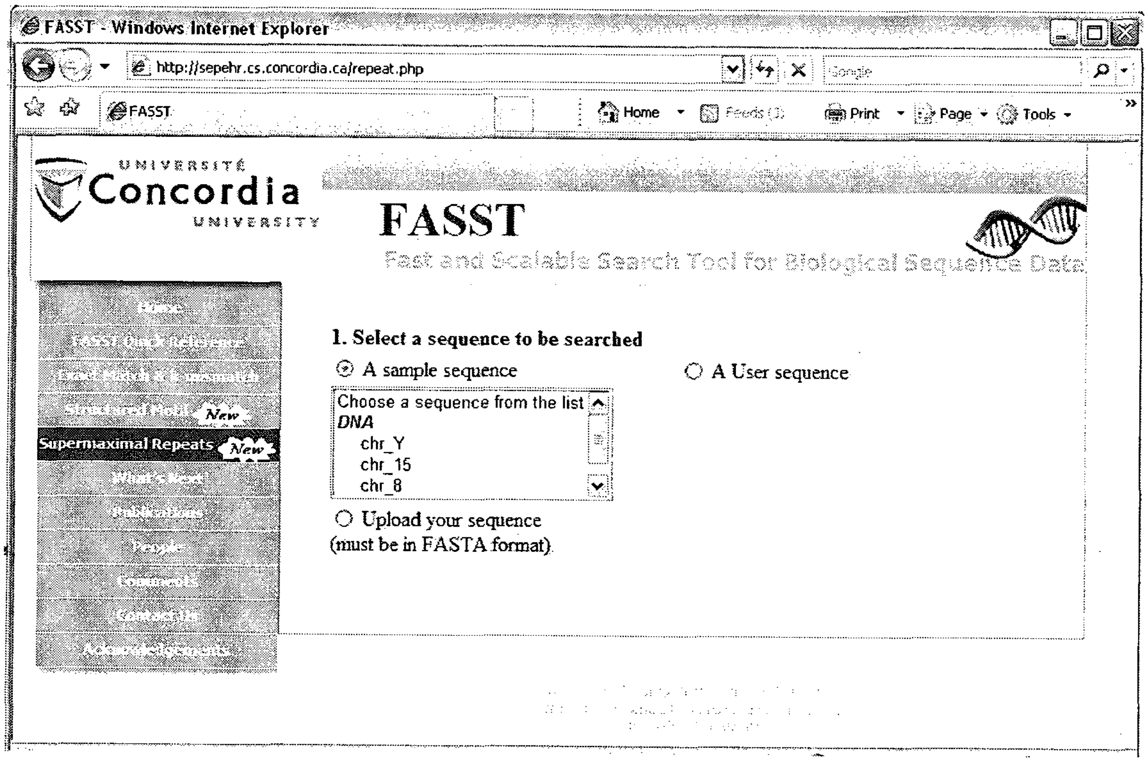
Option 2: Search in a new sequence. For this, the following steps are to be taken:

1. Choose a sequence (in FASTA format) from local computer, mark its type (i.e., DNA), and upload it to our server.
2. From the uploaded sequence the following will be removed:
   - All header and comment lines;
   - Symbols that do not represent a nucleotide (e.g., blanks, new line characters, etc.) and all Ns (i.e., unknown nucleotides);

3. Construct the POL index for the sequence.

Although our technique can handle sequences of up to 4GB, due to storage constraints, the limit on the size of a user uploaded sequence is 10MB. For the same reason, the uploaded sequence and its index will be kept on the server for no longer than 72 hours.

Option 3: Search in existing sequences uploaded by any web users in the last 72 hours. These sequences have been preprocessed as mentioned in Option 2, and the size of user sequence is at most 10MB. We set up this option to provide convenience for users who would like to reuse their uploaded sequence within 72 hours of the original loading process.


FASST - Windows Internet Explorer
http://sepehr.cs.concordia.ca/repeat.php
FASST
Home
Print
Page
Tools
Université Concordia University
FASST
Fast and Scalable Search Tool for Biological Sequence Data
Supermaximal Repeats New

1. Select a sequence to be searched

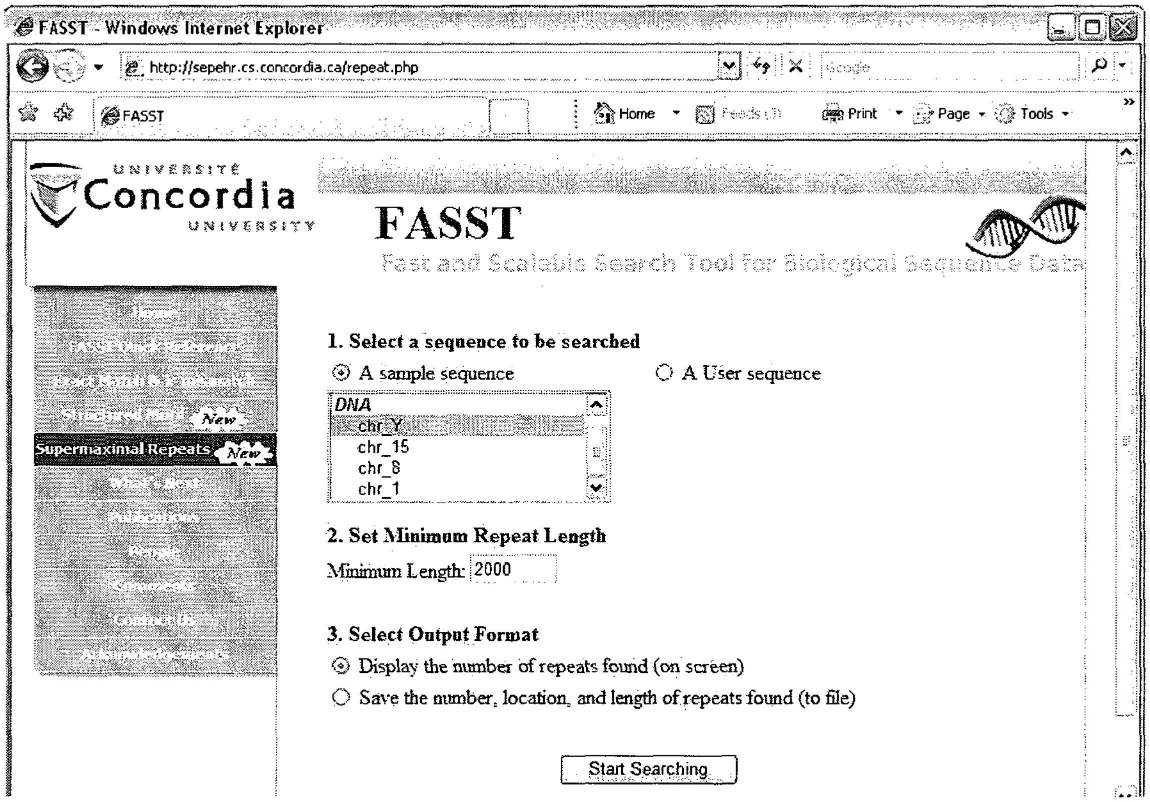
[✓ A sample sequence] A User sequence

Choose a sequence from the list
DNA
chr_Y
chr_15
chr_8

Upload your sequence (must be in FASTA format)

Figure 6.2: Supermaximal repeats search interface – select options

After a sequence is selected in the first step, the user is prompted to input a minimum length of repeats to search for, as well as the output options, shown in Figure 6.3.



1. Select a sequence to be searched

[✓ A sample sequence] A User sequence

DNA
- chr_Y (selected)
- chr_15
- chr_8
- chr_1

2. Set Minimum Repeat Length

Minimum Length: 2000

3. Select Output Format

[✓ Display the number of repeats found (on screen)]

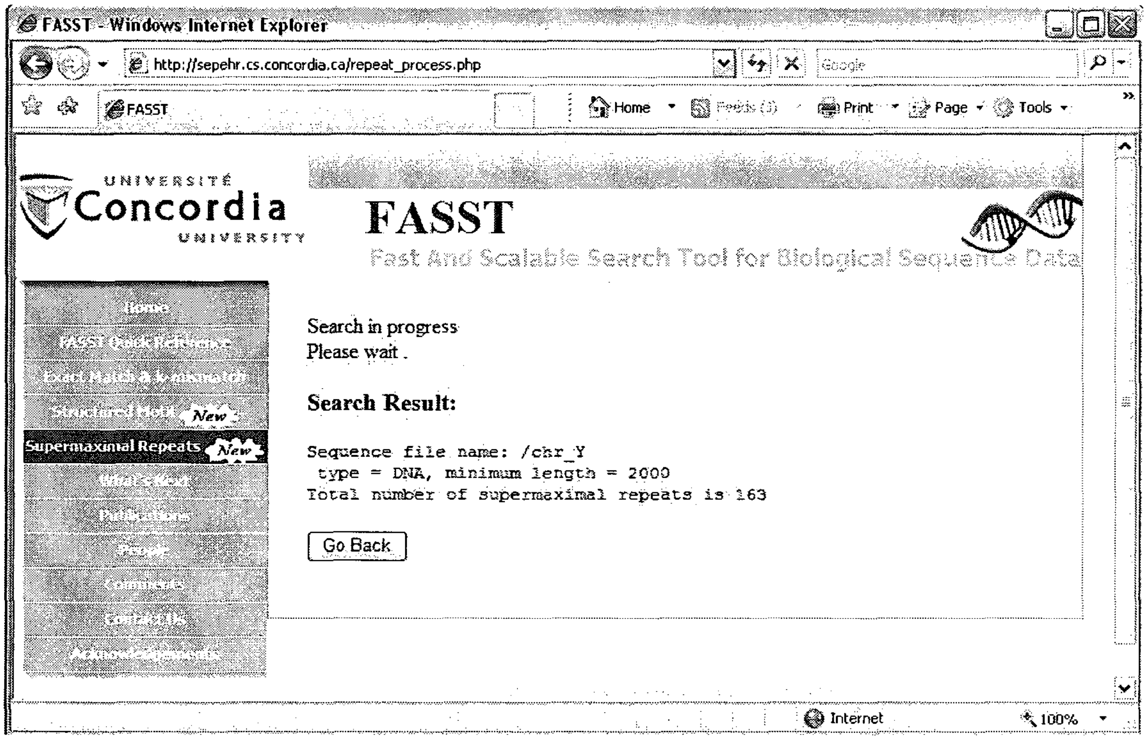
Save the number, location, and length of repeats found (to file)

Figure 6.3: Supermaximal repeats search interface – select parameters

Since it is not uncommon for a search to return millions of occurrences, displaying and/or saving the results could take considerable amounts of time. For convenience, we provide 2 output options:

- Option 1: Display on the screen, only the number of repeats found;
- Option 2: Save to a file, the number, locations, and lengths of the repeats found.

Figure 6.4 is the screen-shot of the page displaying the result of searching chr_Y with minimum length 2000 of the repeats and output option 1. Output option 2 stores detailed information such as locations and lengths of repeats. If the output size is reasonable, the user can view it directly on our server. Otherwise, he/she has to download the result file on his/her local machine. The screenshots of output page for detailed results in text format are shown in Figures 6.5 and 6.6. We also provide graphic annotations for the result, which will be described in the next section.

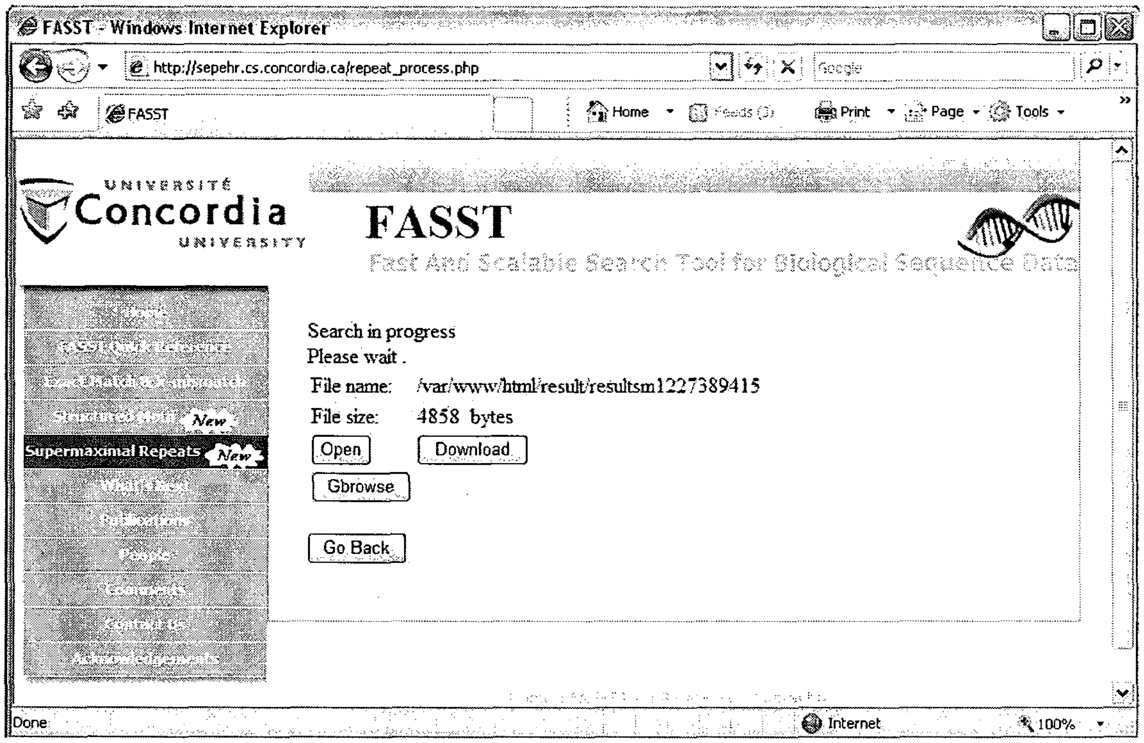


Search in progress
Please wait.

Search Result:

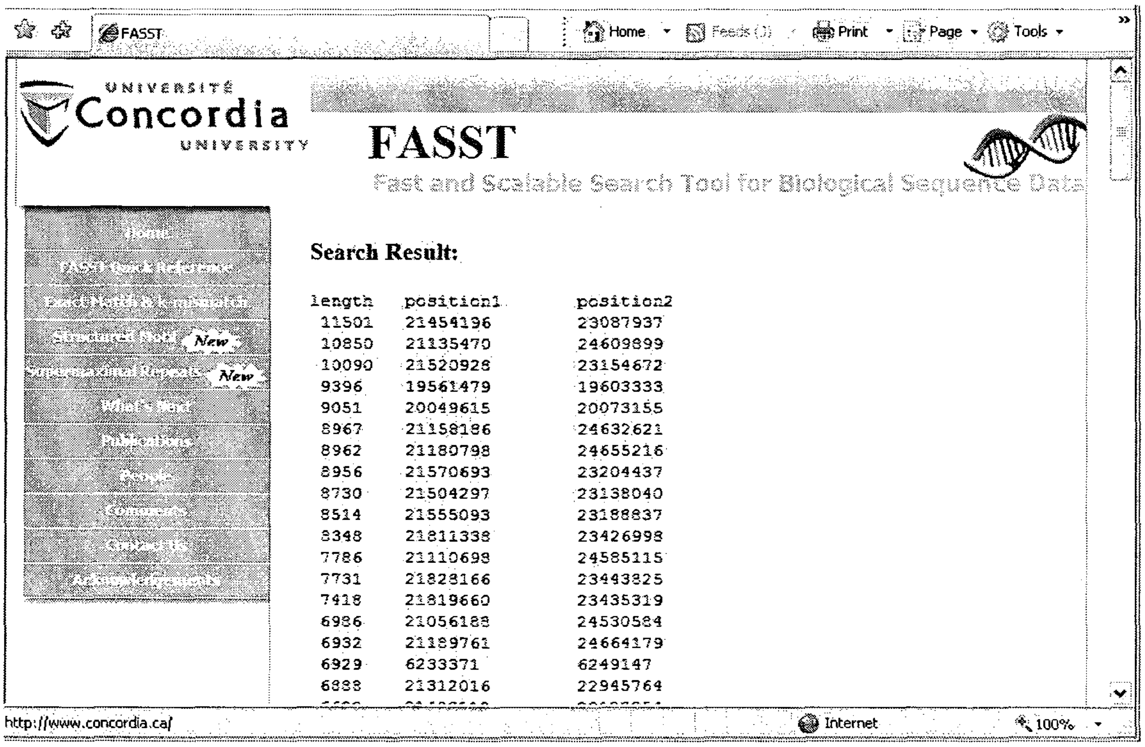
```
Sequence file name: /chr_Y
 type = DNA, minimum length = 2000
Total number of supermaximal repeats is 163
```

Figure 6.4: Supermaximal repeats search interface – display brief results



Search in progress
Please wait .
File name: /var/www/html/result/resultsm1227389415
File size: 4858 bytes

Figure 6.5: Supermaximal repeats search interface – display detailed results



Search Result:

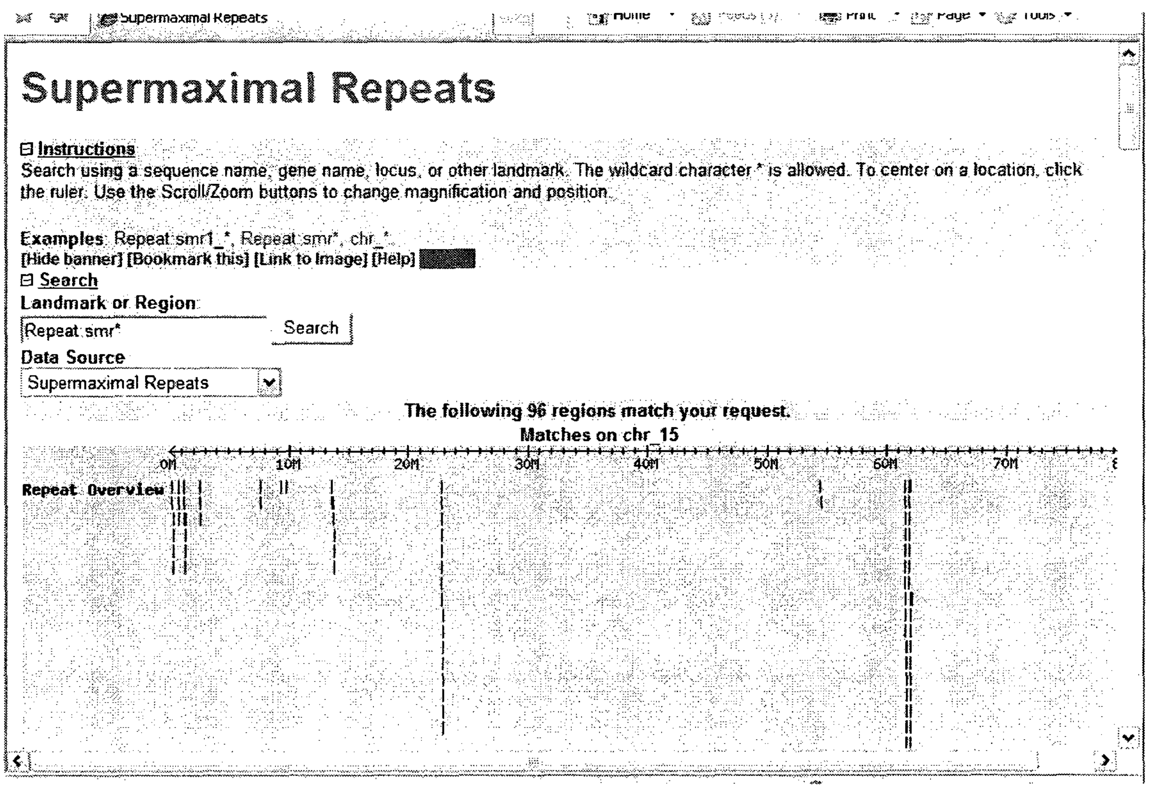
```
length  position1      position2
 11501  21454196       23087937
 10850  21135470       24609899
 10090  21520928       23154672
 9396   19561479       19603333
 9051   20049615       20073155
 8967   21158186       24632621
 8962   21180798       24655216
 8956   21570693       23204437
 8730   21504297       23138040
 8514   21555093       23188837
 8348   21811338       23426998
 7786   21110698       24585115
 7731   21828166       23443825
 7418   21819660       23435319
 6986   21056188       24530584
 6932   21189761       24664179
 6929   6233371        6249147
 6888   21312016       22945764
```

Figure 6.6: Supermaximal repeats search interface – display detailed text results

6.3 Graphic annotations

In order to view the results conveniently, we provide an alternative graphic output option for supermaximal repeats by embedding a third party viewer named GBrowse [Stein et al., 2002] into our user interface. The version of GBrowse we use in FASST is 1.68.

GBrowse is a specified browser which combines database and interactive web page to manipulate and display annotations on genomes. It is a popular viewer in GMOD(Generic Model Organism Database project) which is a collection of open source software tools for creating and managing genome-scale biological databases.


Supermaximal Repeats

⊟ Instructions
Search using a sequence name, gene name, locus, or other landmark. The wildcard character * is allowed. To center on a location, click the ruler. Use the Scroll/Zoom buttons to change magnification and position.

Examples: Repeat:smr1_*, Repeat:smr*, chr_*.
[Hide banner] [Bookmark this] [Link to Image] [Help]
⊟ Search
Landmark or Region:
Repeat:smr*
Data Source
Supermaximal Repeats

The following 96 regions match your request.
Matches on chr_15

Figure 6.7: Graphic output page part 1 – overview

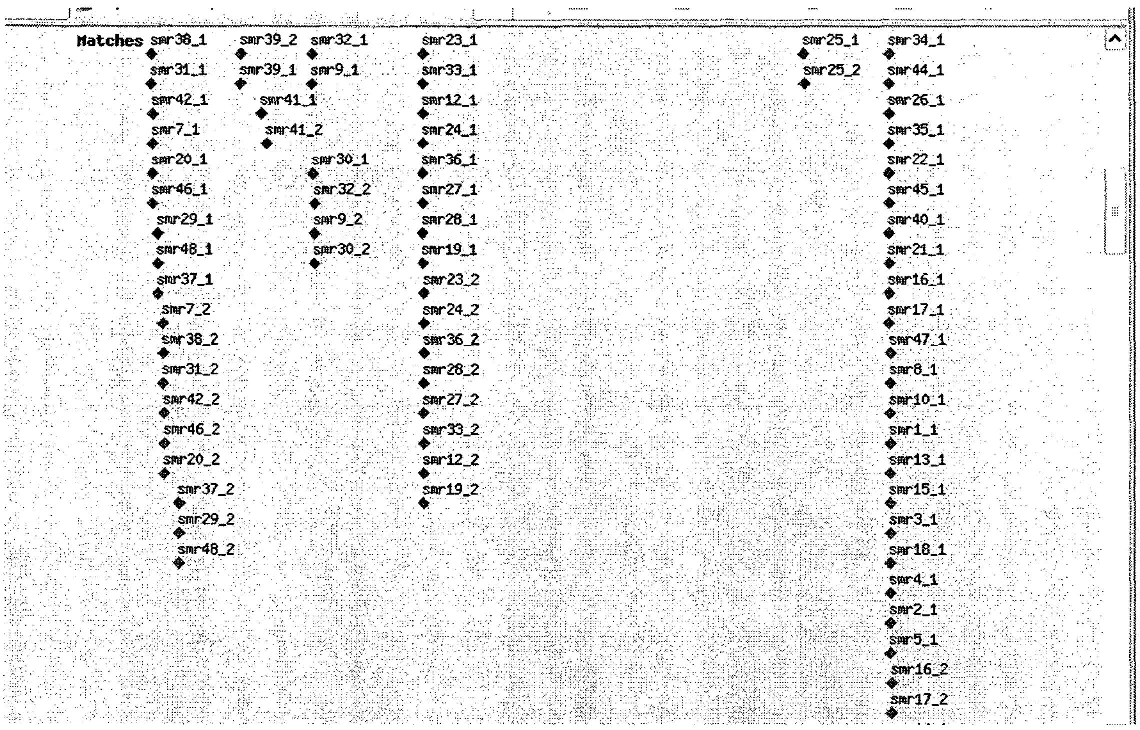
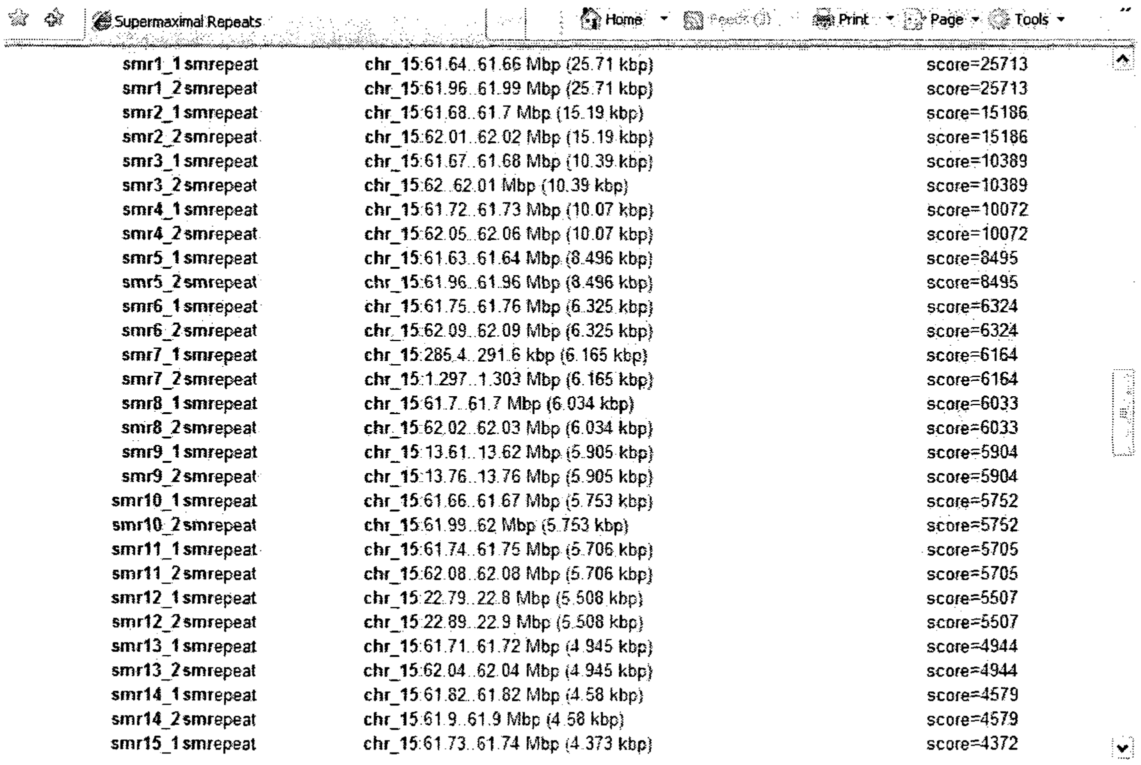


Figure 6.8: Graphic output page part 2 – detailed distribution

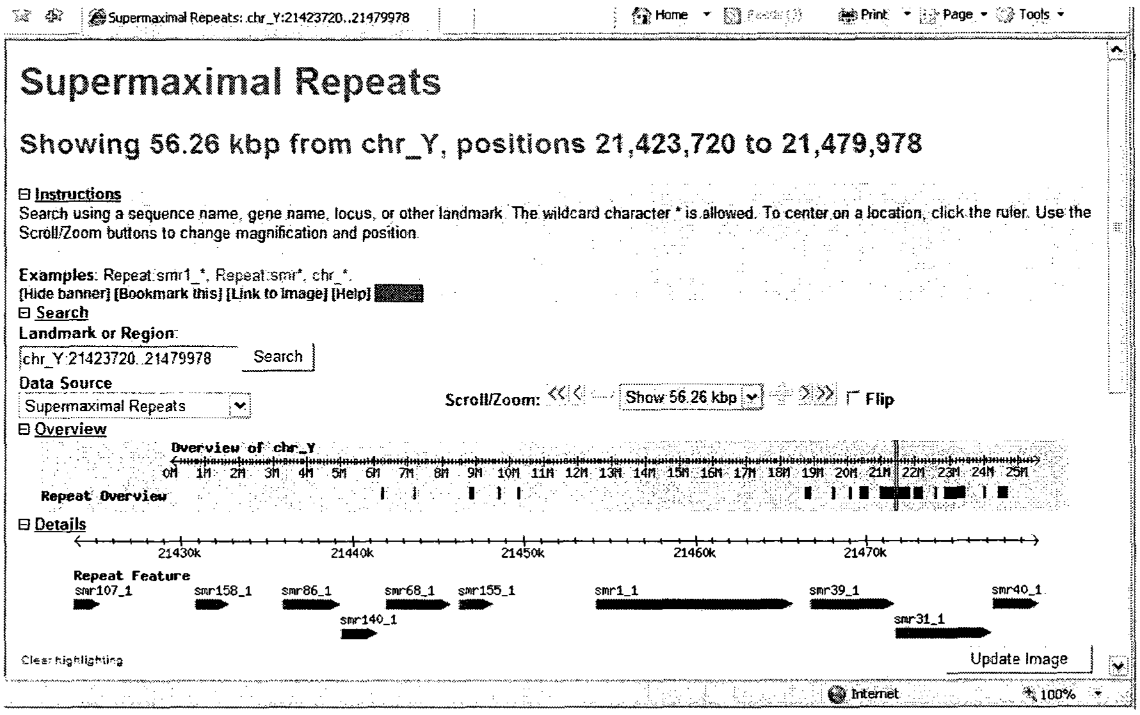


```
 smr1_1 smrepeat     chr_15:61.64..61.66 Mbp (25.71 kbp)        score=25713
 smr1_2 smrepeat     chr_15:61.96..61.99 Mbp (25.71 kbp)        score=25713
 smr2_1 smrepeat     chr_15:61.68..61.7 Mbp (15.19 kbp)         score=15186
 smr2_2 smrepeat     chr_15:62.01..62.02 Mbp (15.19 kbp)        score=15186
 smr3_1 smrepeat     chr_15:61.67..61.68 Mbp (10.39 kbp)        score=10389
 smr3_2 smrepeat     chr_15:62..62.01 Mbp (10.39 kbp)           score=10389
 smr4_1 smrepeat     chr_15:61.72..61.73 Mbp (10.07 kbp)        score=10072
 smr4_2 smrepeat     chr_15:62.05..62.06 Mbp (10.07 kbp)        score=10072
 smr5_1 smrepeat     chr_15:61.63..61.64 Mbp (8.496 kbp)        score=8495
 smr5_2 smrepeat     chr_15:61.96..61.96 Mbp (8.496 kbp)        score=8495
 smr6_1 smrepeat     chr_15:61.75..61.76 Mbp (6.325 kbp)        score=6324
 smr6_2 smrepeat     chr_15:62.09..62.09 Mbp (6.325 kbp)        score=6324
 smr7_1 smrepeat     chr_15:285.4..291.6 kbp (6.165 kbp)        score=6164
 smr7_2 smrepeat     chr_15:1.297..1.303 Mbp (6.165 kbp)        score=6164
 smr8_1 smrepeat     chr_15:61.7..61.7 Mbp (6.034 kbp)          score=6033
 smr8_2 smrepeat     chr_15:62.02..62.03 Mbp (6.034 kbp)        score=6033
 smr9_1 smrepeat     chr_15:13.61..13.62 Mbp (5.905 kbp)        score=5904
 smr9_2 smrepeat     chr_15:13.76..13.76 Mbp (5.905 kbp)        score=5904
smr10_1 smrepeat     chr_15:61.66..61.67 Mbp (5.753 kbp)        score=5752
smr10_2 smrepeat     chr_15:61.99..62 Mbp (5.753 kbp)           score=5752
smr11_1 smrepeat     chr_15:61.74..61.75 Mbp (5.706 kbp)        score=5705
smr11_2 smrepeat     chr_15:62.08..62.08 Mbp (5.706 kbp)        score=5705
smr12_1 smrepeat     chr_15:22.79..22.8 Mbp (5.508 kbp)         score=5507
smr12_2 smrepeat     chr_15:22.89..22.9 Mbp (5.508 kbp)         score=5507
smr13_1 smrepeat     chr_15:61.71..61.72 Mbp (4.945 kbp)        score=4944
smr13_2 smrepeat     chr_15:62.04..62.04 Mbp (4.945 kbp)        score=4944
smr14_1 smrepeat     chr_15:61.82..61.82 Mbp (4.58 kbp)         score=4579
smr14_2 smrepeat     chr_15:61.9..61.9 Mbp (4.58 kbp)           score=4579
smr15_1 smrepeat     chr_15:61.73..61.74 Mbp (4.373 kbp)        score=4372
```

Figure 6.9: Graphic output page part 3 – lists

Figures 6.7 to 6.9 illustrate in GBrowse the result of supermaximal repeats for sequence chr_15 with minimum length 2000. Figure 6.7 shows the overview of supermaximal repeats, where we can see the distribution of supermaximal repeats intuitively. Each "|" marks a location of supermaximal repeat. Following the overview, GBrowse lists the name of the repeats corresponding to its location in the overview part, which is shown in Figure 6.8. The last part of the GBrowse output is a detailed information list of supermaximal repeats which includes name, starting position, ending position, and its length, shown in Figures 6.9.

The above graphic annotations are displayed when we invoke GBrowse in Figure 6.5. In this GBrowse viewer, we can perform some specific search operations on the result obtained. For example, if we are interested in the supermaximal repeats located between 1Mbp to 2Mbp of the input sequence S, we can write $S : 1,000,000..2,000,000$ in *landmark or region* field located in top of the graphic web page, and then start the searching. Furthermore, wildcard character "$*$" is also allowed in this browser. For example, searching "Repeat:smr1_*" returns the locations of the first pair of supermaximal repeats in the result file (note that this is also the longest pair of repeats found since our output is arranged in descending order). Finally, we could click the ruler in overview section to identify interesting positions so that we can see detailed information about these positions. Screenshots of some examples are shown in Figures 6.10 and 6.11.



Supermaximal Repeats: chr_Y:21423720..21479978

Home Print Page Tools

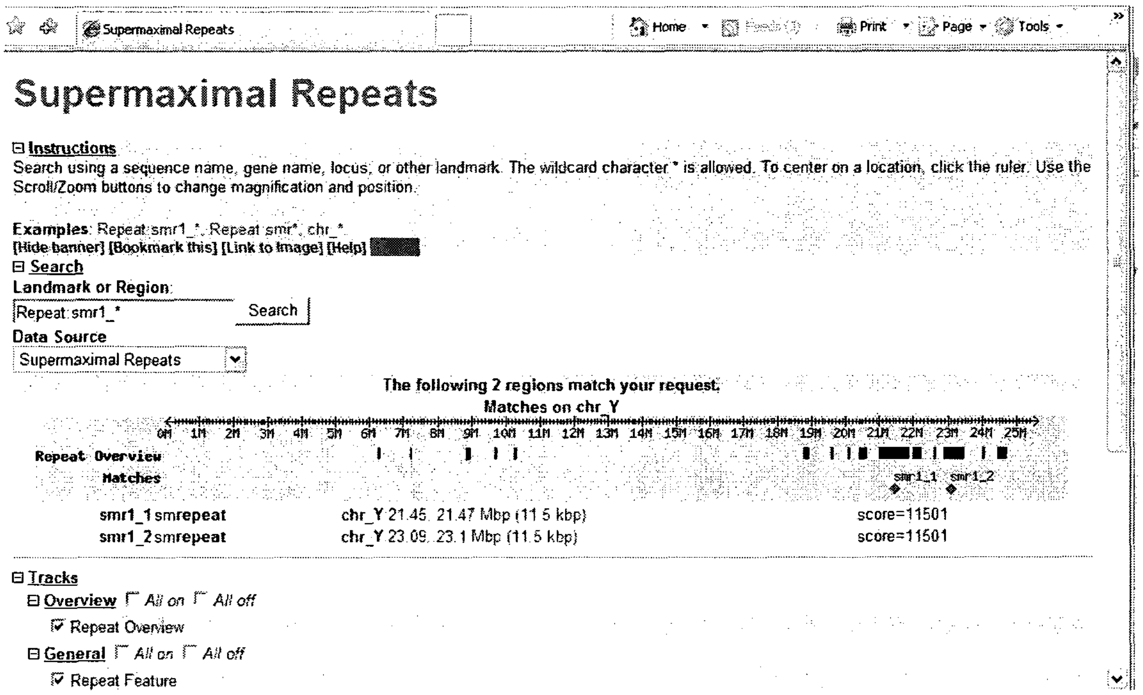
Supermaximal Repeats

Showing 56.26 kbp from chr_Y, positions 21,423,720 to 21,479,978

⊟ Instructions
Search using a sequence name, gene name, locus, or other landmark. The wildcard character * is allowed. To center on a location, click the ruler. Use the Scroll/Zoom buttons to change magnification and position.

Examples: Repeat:smr1_*, Repeat:smr*, chr_*.
[Hide banner] [Bookmark this] [Link to Image] [Help]
⊟ Search
Landmark or Region:
chr_Y:21423720..21479978 Search
Data Source
Supermaximal Repeats
Scroll/Zoom: Show 56.26 kbp Flip
⊟ Overview
⊟ Details

Figure 6.10: Graphic output of searching within a range



Supermaximal Repeats

Home Print Page Tools

Supermaximal Repeats

⊟ Instructions
Search using a sequence name, gene name, locus, or other landmark. The wildcard character * is allowed. To center on a location, click the ruler. Use the Scroll/Zoom buttons to change magnification and position.

Examples: Repeat:smr1_*, Repeat:smr*, chr_*.
[Hide banner] [Bookmark this] [Link to Image] [Help]
⊟ Search
Landmark or Region:
Repeat:smr1_* Search
Data Source
Supermaximal Repeats

The following 2 regions match your request.
Matches on chr_Y

0M 1M 2M 3M 4M 5M 6M 7M 8M 9M 10M 11M 12M 13M 14M 15M 16M 17M 18M 19M 20M 21M 22M 23M 24M 25M
Repeat Overview
Matches
smr1_1
smr1_2

smr1_1 smrepeat	chr_Y 21.45..21.47 Mbp (11.5 kbp)	score=11501
smr1_2 smrepeat	chr_Y 23.09..23.1 Mbp (11.5 kbp)	score=11501

⊟ Tracks
⊟ Overview ☐ *All on* ☐ *All off*
☑ Repeat Overview
⊟ General ☐ *All on* ☐ *All off*
☑ Repeat Feature

Figure 6.11: Graphic output of searching using wildcard (∗)

Chapter 7

Conclusions and Future Work

7.1 Conclusions

We studied the problem of finding supermaximal repeats in large DNA sequences, which is a fundamental task in bioinformatics. We proposed a new index structure, POL (parent-of-leaf) and an efficient SMR (SuperMaximal Repeats) search algorithm, which uses the POL index. We provide the user with the ability to generate POL index with his/her preferred minimum index length. We also developed a web-based interface to SMR within the FASST project, available at http://sepehr.cs.concordia.ca/, and explore the supermaximal repeat search results conveniently by using GBrowse [Stein et al., 2002].

The POL index is derived from and replaces a more powerful, but considerably larger suffix tree index. Our experiments revealed that a practical POL index for large DNA sequences, such as the 24 human chromosomes can be constructed in reasonable time and space by processing the STTD64 index of the sequence. Further, our results show that the proposed SMR algorithm which is based on POL index outperforms the enhanced suffix array based solution, provided as part of the Vmatch search package [Kurtz, 2000]. The search time improvement achieved by SMR over Vmatch ranges from 2 to 9 times faster, when searching for supermaximal repeats of size at least 10 and at least 200 nucleotides, respectively.

Other advantages of our technique are its flexibility and applicability. The POL index can be tailored towards the needs of a specific supermaximal repeats search application. Depending on a desired minimum length of the supermaximal repeats for which a sequence is to be searched, the user has control over the amount of information stored in the POL index in the process of POL construction, thus providing a trade-off between index construction time and storage space on one hand, and the search time performance on the other. Further, a POL index created for a specific MIL is not used only for searching repeats with that particular length. Rather, SMR uses this POL index for search of any repeats of at least the given length. This feature could be extremely useful in the process of iterative supermaximal repeats search, until the user finds a desirable balance between the number of repeats found and their lengths.

7.2 Future plan

Providing the user the application with more control and flexibility is our first effort in the future. For example, we can provide an option which allows users to search supermaximal repeats between a minimum length and a maximum length. We also can improve our SMR search for searching the repeats containing a paticular string.

Furthermore, we can extend our POL index and develop search algorithms to support other types of repetitive structures search, such as maximal repeats, tandem repeats, approximate repeat search, etc.

Another direction of our future work is extending our technique to handle protein sequence. Since suffix trees of protein are partitioned into 23 index files and our current program can only handle suffix trees stored in a single file, extending our POL construction program to deal with partitioned suffix trees may be considered in the future.

Bibliography

[Abouelhoda et al., 2004] M.I. Abouelhoda, S. Kurtz, E. Ohlebusch. Replacing suffix tree with enhances suffix arrays. Journal of Discrete Algorithm 2, pp. 53-86, 2004.

[Alkes et al., 2004] L. P. Alkes, E. Eleazar, A.P. Pavel. Whole-genome analysis of Alu repeat elements reveals complex evolutionary history. Genome Res. 14, pp. 2245-2252, 2004.

[Andersson, 1995] A. Andersson, S. Nilsson. Efficient implementation of suffix trees.In In Software-Practice and Experience 25(2), pp. 129-141, 1995.

[Astbury, 1961] W.T. Astbury. Molecular biology or ultrastructural biology? Nature 190, pp. 1124–1125 ,1961.

[Baxevanis et al., 2005] A.D. Baxevanis, B.F.F Ouellette. (Eds.), Bioinformatics: a practical guide to the analysis of genes and proteins, third edition, Wiley, 2005.

[Bedell et al., 2000] J.A. Bedell, I. Korf, W. Gish. MaskerAid: a performance enhancement to Repeat-Masker. Bioinformatics 16, pp. 1040-1041, 2000.

[Burrows et al., 1994] M. Burrows, D. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.

[Cairns, 1998] J. Cairns. Mutation and Cancer: The Antecedents to Our Studies of Adaptive Mutation. Genetics, Vol. 148, pp. 1433-1440, 1998

[Charlesworth et al., 1994] B. Charlesworth, P. Sniegowski, W. Stephan. The evolutionary dynamics of repetitive DNA in eukaryotes. Nature 371, pp. 215-220, 1994.

[Delcher et al., 1999] A.L. Delcher, S. Kasif, R.D. Fleischmann, J. Peterson, O. White, S.L. Salzberg. Alignment of whole genomes. Nucleic Acids Res. 27, pp. 2369-2376, 1999.

[Ferragina et al., 2000] P. Ferragina, G. Manzini. Opportunistic data structures with applications. In Proc. of the 41st IEEE Symposium on Foundations of Computer Science, pp. 390-398, 2000.

[Halachev et al., 2007] M. Halachev, N. Shiri, A. Thamildurai. Efficient and scalable indexing techniques for biological sequence data. In Proc. of BIRD'07, LNBI 4414, Hochreiter S. and Wagner, R (Eds.), Springer Verlag, Germany, pp. 464-479, 2007.

[Hon et al., 2004] W.K Hon, T.W. Lam, W.K. Sung, W.L. Tse, C.K. Wong, S.M. Yiu. Practical aspects of compressed suffix arrays and FM-index in searching DNA sequences. In the 6th Workshop on Algorithm Engineering and Experiments (ALENEX), 2004.

[Irving et al., 2003] R.W. Irving, L. Love. The suffix binary search tree and suffix AVL tree. Journal of Discrete Algorithms. 1, pp. 387-408, 2003.

[Giegerich et al., 1997] R. Giegerich, S. Kurtz. From Ukkonen to McCreight and Weiner: a unifying view of linear-time suffix tree construction. Algorithmica 19(3), pp. 331-353, 1997.

[Giegerich et al., 2003] R. Giegerich, S. Kurtz, J. Stoye. Efficient implementation of lazy suffix trees. In Software-Practice and Experience 33, pp. 1035-1049, 2003.

[Gotoh, 1982] O. Gotoh. An improved algorithm for matching biological sequences. Journal of Molecu-lar Biology, 162, pp. 705-708 ,1982.

[Gregory, 2008] T.R. Gregory. quotes of interest: SINEs and LINEs. http://genomicron.blogspot.com/2008/02/quotes-of-interest-sines-and-lines.html

[Gremme et al., 2005] G. Gremme, V. Brendel, M.E. Sparks, S. Kurtz. Engineering a software tool for gene prediction in higher organisms. Information and Software Technology, 47(15), pp. 965-978, 2005.

[Grossi et al., 2005] R. Grossi, J.S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. SIAM Journal on Computing, 2005.

[Gusfield, 1997] D. Gusfield. Algorithms on strings, trees, and sequences: computer sciences and computational biology. Cambridge University Press, 1997.

[Jurka et al., 2005] J. Jurka, V.V. Kapitonov, A. Pavlicek, P. Klonowski, O. Kohany, J. Walichiewicz. Repbase Update, a database of eukaryotic repetitive elements. Cytogenetic and Genome Research 110, pp. 462-467, 2005.

[Kurtz, 1999] S. Kurtz. Reducing the space requirement of suffix trees. SoftwarePractice and Experience, vol. 29, pp. 1149-1171, 1999.

[Kurtz et al., 1999] S. Kurtz, C. Schleiermacher. REPuter: fast computation of maximal repeats in complete genomes. In Bioinformatics, pp. 426-427, 1999.

[Kurtz, 2000] S. Kurtz. Vmatch: large scale sequence analysis software. http://www.vmatch.de/

[Kurtz et al., 2000] S. Kurtz, E. Ohlebusch, C. Schleiermacher, J. Stoye, R. Giegerich. Computation and visualization of degenerate repeats in complete genomes. In

Proceedings of the International Conference on Intelligent Systems for Molecular Biology. AAAI Press, Menlo Park, CA, pp. 228-238, 2000.

[Lian et al., 2008] C.N. Lian, M. Halachev, N. Shiri. Searching for supermaximal repeats in large DNA sequences. In Proc. of BIRD'08, CCIS 13, Elloumi, M. et al.(eds.), Springer Verlag, Germany, pp. 87-101, 2008.

[Lewis et al., 2002] S.E. Lewis, S.M.J. Searle, N. Harris, M. Gibson, V. Iyer, J. Ricter, C. Wiel, L. Bayraktaroglu, E. Birney, M.A. Crosby, J.S. Kaminker, B. Matthews, S.E. Prochnik, C.D. Smith, J.L. Tupy, G.M. Rubin, S. Misra, C.J. Mungall, N.E. Clamp. Apollo: a sequence annotation editor. Genome Biology 3(12), research0082, 2002.

[Manber et al., 1993] U. Manber, E. Myers. Suffix arrays: A new method for on-line string searches. SICOMP Vol. 22 Issue 5, pp. 935-948, 1993.

[McConkey, 1993] E. McConkey. Human Genetics: The Molecular Revolution. Jones and Bartlett, Boston, MA, 1993.

[McCreight, 1976] E.M. McCreight. A Space-Economical Suffix Tree Construction Algorithm. Journal of the ACM 23 (2), pp. 262-272, 1976.

[Miki et al., 1980] B.L. Miki, J.M. Neelin. DNA repeat lengths of erythrocyte chromatins differing in content of histones H1 and H5. Nucleic Acids Res. 8(3), pp. 529-542, 1980.

[Morange et al., 1998] M. Morange. A History of Molecular Biology. Cambridge, MA, Harvard University Press, 1998.

[Morrison, 1968] D.R. Morrison. PATRICIA - Practical Algorithm to Retrieve Information Coded in Alphanumeric. Journal of the ACM, 15(4) pp. 514-534, 1968.

[Nair, 2007] A. S. Nair. Computational Biology & Bioinformatics: A Gentle Overview. Communications of the Computer Society of India, January 2007.

[Niko et al., 2007] V. Niko, G. Wolfgang, D. Kashyap, M. Veli. Compressed suffix tree-a basis for genome-scale sequence analysis. Bioinformatics 23(5), pp. 629-630, 2007.

[Sadakane, 2007] K. Sadakane. Compressed Suffix Trees with Full Functionality. Theory of Computing Systems. 41(4), pp. 589-607, 2007.

[Slezak et al., 2003] T. Slezak, T. Kuczmarski, L. Ott, C. Torres, D. Medeiros, J. Smith, B. Truitt, N. Mulakken, M. Lam, E. Vitalis, A. Zemla, C.E. Zhou, and S. Gardner. Comparative Genomics Tools Applied to Bioterrorism Defense. Briefings in Bioinformatics, 4(2), pp. 133-149, 2003.

[Smit et al., 2008] A.F.A. Smit, R. Hubley, P. Green. RepeatMasker: http://repeatmasker.org

[Stein et al., 2002] L.D. Stein, C. Mungall, S. Shu, M. Caudy, M. Mangone, A. Day, E. Nickerson, J.E. Stajich, T.W. Harris, A. Arva, S. Lewis. The generic genome browser: a building block for a model organism system database. Genome Res. 12(10), pp. 1599-1610, 2002.

[Ukkonen, 1995] E. Ukkonen. On-line construction of suffix trees. Algorithmica 14 (3), pp. 249-260, 1995.

[Weber et al., 1997] J. Weber, W. Myers. Human Whole Genome Shotgun Sequencing. Genome Res. 7, No. 5, pp. 401-409, 1997.

[Wenior, 1973] P. Wenior. Linear pattern matching algorithm. Proc. of the 14th IEEE Symposium on Switching and Automata Theory, pp. 1-11, 1973.

[Zamir et al., 1998] O. Zamir, O. Etzioni. Web document clustering: a feasibility demonstration. SIGIR '98, Proceedings of the 21st annual international ACM SIGIR conference on Research and development in information retrieval, pp. 46-54, ACM. 1998.

[Zhang, 2008] Y. Zhang. Progress and challenges in protein structure prediction. Curr Opin Struct Biol 18 (3), pp. 342-348, 2008

[Zheng et al., 2003] C. Zheng, J. Hayes. Structures and interactions of the core histone tail domains. Biopolymers 68 (4), pp. 539C546, 2003

Appendix A

Experimental Data

A.1 SMR Vs Vmatch search performance

Below we provide the details of raw data obtained in our experiments of SMR search and Vmatch performance, which were represented as graphs and charts in Chapter 5.

Minimum repeat length	SMR Algorithm (sec)	Vmatch (sec)	SMR performance vs. Vmatch
1	860.89	826.14	4.2% slower
2	860.80	814.01	5.6% slower
3	860.78	804.42	7.0% slower
4	860.82	795.78	8.1% slower
5	859.89	788.84	9.0% slower
6	859.78	780.11	10.1% slower
7	859.03	775.42	10.8% slower
8	859.92	768.78	11.8% slower
9	859.92	760.49	13.1% slower

Table A.1: SMR algorithm Vs Vmatch at the minimum repeat length from 1 to 9

Minimum repeat length	SMR Algorithm (sec)	Vmatch (sec)	SMR performance vs. Vmatch
10	358.23	752.04	2 times faster
25	92.68	645.78	7 times faster
50	71.07	541.71	7.6 times faster
100	61.92	519.18	8.4 times faster
200	59.84	509.96	8.6 times faster
1000	57.33	501.47	8.8 times faster
2000	56.78	507.83	9 times faster
5000	55.68	502.55	9.1 times faster
10000	55.97	503.15	9 times faster

Table A.2: SMR + POL10 Vs Vmatch with $MRL \geq 10$

Minimum repeat length	SMR Algorithm (sec)	Vmatch (sec)	SMR performance vs. Vmatch
10	859.31	752.04	14% slower
24	846.86	647.96	31% slower
25	92.04	645.78	7 times faster
50	70.51	541.71	7.6 times faster
100	62.11	519.18	8.4 times faster
200	59.27	509.96	8.6 times faster
1000	57.21	501.47	8.8 times faster
2000	56.75	507.83	9 times faster
5000	55.71	502.55	9.1 times faster
10000	55.95	503.15	9 times faster

Table A.3: SMR + POL25 Vs Vmatch with $MRL \geq 10$

Minimum repeat length	SMR Algorithm (sec)	Vmatch (sec)	SMR performance vs. Vmatch
10	859.31	752.04	14% slower
25	848.83	645.78	31% slower
50	847.52	541.71	56% slower
99	848.25	519.18	63% slower
100	62.11	519.18	8.4 times faster
200	59.27	509.96	8.6 times faster
1000	57.13	501.47	8.8 times faster
2000	56.06	507.83	9 times faster
5000	55.67	502.55	9.1 times faster
10000	55.89	503.15	9 times faster

Table A.4: SMR + POL100 Vs Vmatch with $MRL \geq 10$

Minimum repeat length	SMR Algorithm (sec)	Vmatch (sec)	SMR performance vs. Vmatch
10	859.31	752.04	14% slower
25	848.83	645.78	31% slower
50	847.52	541.71	56% slower
100	845.10	519.18	62% slower
199	820.84	508.17	66% slower
200	58.91	509.96	8.6 times faster
1000	56.54	501.47	8.8 times faster
2000	56.14	507.83	9 times faster
5000	55.79	502.55	9.1 times faster
10000	55.25	503.15	9 times faster

Table A.5: SMR + POL200 Vs Vmatch with $MRL \geq 10$