Practical algorithms for biological sequence analysis methods and applications

Retha, Ahmad

Awarding institution:
King's College London

Practical Algorithms for Biological Sequence Analysis

Methods and Applications

Ahmad Retha

Department of Informatics

King's College London

This dissertation is submitted for the degree of

Doctor of Philosophy

June 2019

Dedicated to my daughter Lara.

Acknowledgements

I would like to thank my supervisor, Dr. Solon P. Pissis, and second supervisor, Prof. Costas Iliopoulos, for their help and support throughout my PhD. I am especially grateful for the time and effort my supervisor invested in me and the help provided in times of crisis.

I would like to thank King's College London and the Department of Informatics for funding my PhD under the Graduate Teaching Assistanship scheme.

My wife and parents have been especially supportive during the difficult times of my studies and I am very grateful for that.

Abstract

Biological sequences such as DNA, RNA and proteins can be represented as a sequence or string of characters. In this thesis I present the work I have done to implement new heuristic and exact string algorithms and show how they can be applied with existing algorithms to perform sequence analysis. I present examples of applications of these algorithms to solve problems in Bioinformatics and show how they are competitive or orders of magnitude superior to existing algorithms.

At the start of this thesis we present a flexible implementation of the MaxShiftM bit-parallel algorithm (Crochemore et al., 2010) for performing Fixed-Length Approximate String Matching (FLASM) under the Hamming distance model. We also describe how we can apply Myers' bit-parallel approximate pattern matching algorithm (Myers, 1999) for solving the FLASM problem under the Edit distance model. FLASM is the problem of searching for all factors of length ℓ of a pattern p in a text t. The two algorithms mentioned were packaged into a software library called libFLASM. We then presented multiple applications for these algorithms – we used the algorithms for implementing an algorithmic text pre-processing step known as the Chang and Marr Index (Chang, W.I. and Marr, T.G., 1994), we showed how to use the algorithms for pairwise circular sequence alignment, we incorporated it into BEAR (BEst Aligned Rotations), a state-of-the-art tool for improving Multiple Circular Sequence Alignment (MCSA), and finally we incorporated it into MoTeX-II a state-of-the-art tool to identify single and structured motifs. We also compare the algorithms to two competitors and show that whilst not being extremely fast, the FLASM algorithms are very effective and robust in practice, especially under increasing error rate.

Next, the thesis deals with the pairwise circular sequence comparison problem (CSC) which we solve using a technique involving q-grams and splitting the circular sequences into blocks. The CSC problem consists in finding an optimal linear alignment of a pair of circular strings. Circular sequences are abundant in nature and can be found in Mitochondrial DNA, bacterial and viral genomes and in some protein structures. We describe two exact algorithms and one heuristic algorithm for solving the problem and demonstrate their speed and accuracy using synthetic and real data. We compare our

algorithms to each other for speed and accuracy and compare them to other accurate but comparatively slow algorithms and show one of the algorithms, saCSC, to be very fast. We then explain how we improved its accuracy by a refinement step to the point it was fit to be incorporated into BEAR for performing multiple circular sequence alignment.

Finally, the thesis deals with algorithms for searching for patterns in Elastic-Degenerate (ED) texts, a text-based format for expressing variation in a group of related sequences such as a pan-genome. We present three algorithms in total to solve the problem - the first two search a single pattern in an ED text while the third searches for multiple patterns simultaneously. We tested our algorithms for speed against each other and one other competitor using synthetic and real data to find that the bit-parallel algorithms we implemented perform very well. We applied our algorithms to search for patterns in real data taken from the 1000 Human Genomes project. We used the multiple pattern matching algorithm for validating Minimal Absent Words (MAWs) in the Human Genome and for validating the work of researchers (Silva et al, 2015) who identified three MAWs thought to exist in the Ebola virus but not in humans. We discovered these patterns do in fact exist in the greater human population and will present as a false positive for Ebola viral infection.

Table of contents

List of figures

List of tables

Chapter 1

Publications

The following relevant publications comprise the subject matter of this thesis:

1. [109] S. P. Pissis, A. Retha, "Generalised Implementation for Fixed-Length Approximate String Matching Under Hamming Distance and Applications", in Proceedings of the 2015 IEEE International Parallel and Distributed Processing Symposium Workshop, Washington, DC, USA: IEEE Computer Society, pp. 367-374.

2. [4] L. A. Ayad, S. P. Pissis and A. Retha, "libFLASM: a software library for fixed-length approximate string matching", BMC Bioinformatics, vol. 17, no. 1, 2016, pp. 454.

3. [53] R. Grossi, C. S. Iliopoulos, R. Mercas, N. Pisanti, S. P. Pissis, A. Retha, F. Vayani, "Circular Sequence Comparison with q-grams", in Algorithms in Bioinformatics: 15th International Workshop, WABI 2015, Atlanta, GA, USA, September 10-12, 2015, Proceedings, M. Pop, H. Touzet, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 203-216.

4. [54] R. Grossi, C. S. Iliopoulos, R. Mercas, N. Pisanti, S. P. Pissis, A. Retha, F. Vayani, "Circular sequence comparison: algorithms and applications", Algorithms for Molecular Biology, vol. 11, no. 1, 2016, pp. 1-14.

5. [52] R. Grossi, C. S. Iliopoulos, C. Liu, N. Pisanti, S. P. Pissis, A. Retha, G. Rosone, F. Vayani, L. Versari, "On-Line Pattern Matching on Similar Texts", in 28th Annual Symposium on Combinatorial Pattern Matching (CPM 2017), J. R. Juha Kärkkäinen, W. Rytter, Eds., Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, pp. 9:1-9:14.

6. [110] S. P. Pissis, A. Retha, "Dictionary Matching in Elastic-Degenerate Texts with Applications in Searching VCF Files On-line", in 17th International Symposium on Experimental Algorithms (SEA 2018), G. D'Angelo, Ed., Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, pp. 16:1-16:14.

Chapter 2

Introduction

In this thesis I will be introducing some of the work I have done during my PhD in Bioinformatics. I have published six papers in this time with my colleagues, dealing with three core problems. This thesis will describe these projects – the solutions that we arrived to, the various applications and the experimental results we obtained. I place extra emphasis on explaining technical approaches and methods that may have been excluded from the published papers, usually for lack of space. I make extra effort to go through some of the algorithm psuedocode and describe alternative approaches and their impact on the computational complexity or efficiency.

The motivation for this research is to improve the running times, memory consumption and accuracy of Bioinformatics tools. The subject of Bioinformatics is known by many different definitions as it is a hold-all term for the application of computer science to solve problems surrounding the storage, processing and visualisation or interpretation of biological or biomedical data. We concentrate on the molecular biological analysis aspect of Bioinformatics; that is the study of the primary sequences of DNA, RNA and Proteins. We work primarily with genomic (DNA) or mitochondrial (mtDNA) sequences, represented as strings made up of the characters `A`, `C`, `G` and `T` (in DNA).

Genetic material in the form of DNA or RNA is present in all living organisms and is the primary vehicle for biological inheritance. It is this material that allows the continuation of species from one generation to the next, passing on the traits that increase their likelihood of survival. But this genetic material has the property of mutability and imperfect duplication, repair and recombination, however, in its folly lies the ingenious mechanism by which new traits are obtained, changing environments are conquered and new species come to be. A proper understanding of the genetic sequence will help us understand our evolutionary origins, ultimately what makes us

Human, as well as discern the causes and treatments of disease, whether it be inherited or acquired through infection. Thus, whatever tools we can develop to better probe the secrets held inside the genetic code is invaluable.

One of the many challenges in Bioinformatics is to search for patterns/motifs in long sequences of DNA many million of bases in length. As DNA tends to mutate through the passage of time and even *next generation sequencing* (NGS) technologies introduce errors, the problem of finding a pattern is not always a simple case of finding an exact match. Even conserved regions of DNA are not immune to some differences. Conserved motifs between species often indicates their indispensability to biological processes as their mutation may be inconducive to life. They can be used to study the evolutionary relationship of extant species and infer their common ancestry by building a phylogeny tree – the underlying intuition being the more similar sequences are to each other, the more closely related they are and are likely to share a common ancestor. We can also infer their functionality by studying genetic knockouts strains in other species such as mice [72] and nematode worms [139, 141]. Other motifs researchers are interested in finding include transcription factor binding sites, promoter binding sites, transcription start sites, restriction enzyme binding sites, transposable elements and so on.

Another challenge in Bioinformatics concerns the circular nature of some sequences. The genomes of viruses, archaea, bacteria and organelles such as mitochondria (which are believed to have once been an independent organism that found a permanent symbiotic niche inside eukaryotic cells [145]) are circular in composition. When they are sequenced they are linearised by cutting them at an arbitrary position. This introduces some problems to their analysis with conventional Bioinformatics tools. In pairwise sequence alignment programs they would register a great difference between cloned cells when they are in fact identical. Therefore, circular sequences need to be re-aligned first before they are compared. Another problem that arrises with using motif search tools to search circular sequences is some matches would be missed because a motif may span the start and end of a sequence.

We created two functional groups of algorithms: One group for the alignment of pairs of related circular sequences, and another for searching for approximate or exact patterns in circular sequences or sets of closely related sequences (pan-genomes). The implementations of some of these algorithms were incorporated into two state-of-the-art Bioinformatics tools: BEAR (BEst Aligned Rotations) [8] and MoTeX-II [108]. Additionally, a general-purpose software library, libFLASM [4], was also

created. All applications created in this project are released as open-source software with a permissive license.

Algorithms already exist to solve many of the problems described in this thesis but they tend to run slowly or have certain limitations or restrictions. We aim to solve the same problems quicker and with increased speed and efficiency where possible. We also put special emphasis on maintaining or improving the accuracy of the new algorithms where possible. We present multiple applications of the algorithms and how they can be used to solve different problems in Bioinformatics. We then go further and show that some of these algorithms can be used outside of a biological scope and be applied to solving general purpose problems. We expect some of these algorithms are sufficiently flexible to be applied to solving problems in many different fields in the sciences and hope that their uses are identified in the near future.

Two approaches to devising algorithms, especially text-searching algorithms, can be found in the literature and are termed *off-line* and *on-line*. Off-line algorithms have a preprocessing stage where they take the input datasets and generate exhaustive intermediate indexes which can then be searched quickly. Off-line algorithms presuppose the search datasets are known beforehand and are readily available on disk and remain unchanged going forward. There is also the assumption that index-storage space is abounding and indexes, sometimes as large as (if not larger than) the original dataset, can be stored on disk alongside the original datasets. In addition to this, the time it takes to index a dataset is of secondary importance to the search time and can take many times longer to generate. Off-line solutions are a good match for static datasets, vast libraries of information that should remain accessible and be searchable quickly after indexing.

On-line search algorithms are better suited to smaller, changing and expanding datasets whose contents are not known beforehand. On-demand search requests are made frequently on different datasets and are executed on-the-fly with little or no preprocessing. Often, preprocessing is only performed on the pattern being searched or light-weight data structures are generated for the dataset and stored in memory instead of disk to enhance the search performance. On-line search algorithms are better suited for immediate and on-demand searching of variable datasets. In this thesis we mainly consider solving the problems on-line because the datasets we are searching are not known beforehand and we expect our algorithms to return results as quickly as possible without waiting for indexes to be built before performing the searches. Furthermore, advances in on-line algorithm design often translate to performance improvements in offline solutions.

We place special emphasis on reducing computational complexity of the algorithms we present in this thesis because this reduces intractable problems to ones that can be solved in a reasonable time, albeit, in some cases, at a cost to accuracy. For this reason we also find ways of improving accuracy when using the new methods. We use the asymptotic (Big-O) notation, denoted by $\mathcal{O}$, to define the upper-bound [24] computational complexity of the algorithms we describe in this thesis. Whilst this type of computational complexity analysis ignores efficiency, optimisations and constants (which can sometimes be large), it has been found in practice to be a good predictor of relative running time expectations. In [24], Cormen et al. compare the relative performance of two different algorithms: the *Merge Sort* algorithm which has a worst-case $\mathcal{O}(n \log n)$ time complexity and the *Insertion Sort* algorithm with worst-case time complexity $\mathcal{O}(n^2)$. Judging by their worst-case time complexity and the growth rate of the the data that the algorithms operate on, they deduce how they perform relative to one another. They explain that with small datasets, an algorithm's efficiency can dominate the running time such that a more asymptotically complex algorithm can finish faster than its competitor, but with increasingly larger inputs, the amount of data dominates the effect on the running time of an algorithm rather than any other constant factors or efficiency optimisations.

By defining the complexity of an algorithm based on the growth rate or changing amount of data that is processed by an algorithm, it can effectively determine the relative speed of the new algorithm to a (usually naïve) competing algorithm. Reducing an algorithm's complexity can result in running time difference or memory consumption difference by many orders of magnitude, such that some algorithm requiring quadratic time, in practice taking many hours or even days to complete a job, can be replaced by an algorithm requiring time linear in the size of the input dataset and finishes its computation in just a few minutes. This change is highly significant and can have a major impact on data analysis pipelines, perhaps removing processing bottlenecks that once restricted how much research could be performed and how quickly results could be obtained. It also means some limitations are lifted and more data can be processed in a shorter time period.

Massive datasets are being produced at an ever-faster rate, especially with evolving NGS technologies being used to sequence the genomes of more species and healthcare patients [114]. At some point we will not be capable of storing and processing the ever-increasing amounts of *Big Data* simply by dividing computation and storage across many machines in *The Cloud*, because we are producing data at a rate faster than we can handle [114]. This underscores the importance of reducing computational

complexity and finding different storage approaches. Although the algorithms we demonstrate in this thesis show results using a single processing core, they are still very much relevant and can have a significant impact on cloud computing. By producing more effective and efficient algorithms, the amount of memory, the number of processing cores, power consumption, time and ultimately money can be reduced greatly when it comes to distributing a large workload.

We now summarise the contents of chapters of this thesis:

Contribution [4, 109] : In Chapter 4, we introduce the concept of Fixed-length Approximate String Matching (FLASM) – a generalisation of approximate string matching where any factor of length ℓ of a pattern p of length $m \geq \ell$ can be found in a text t of length n. We introduce two bitwise algorithms, one using the Hamming distance model and the other the edit distance model, to solve the FLASM problem efficiently in time $\mathcal{O}(m\lceil \ell/w \rceil n)$ and space $\mathcal{O}(m\lceil \ell/w \rceil)$, where w is the size of the computer word (typically 64). We packaged these algorithms into libFLASM, a free and open-source C++ software library and provided example programs showing how to use it. We also incorporated the library into two state-of-the-art Bioinformatics tools: BEAR [8] and MoTeX-II [108]. We then show how FLASM can be applied to solving various problems:

- Approximate Circular String Matching
- Multiple Circular Sequence Alignment
- Simple/Structured Motif Extraction
- Building the Chang and Marr Index [20]

Finally, we show how the algorithms perform with increasing dataset, factor and error-limit size, and we show how well they do when put toe-to-toe against competing programs Cyclope [92], CMFN [42], ACB [59] and the naïve approach to building the Chang and Marr index. We further demonstrate the importance of being able to use longer factors in terms of increasing accuracy of results.

My direct contribution for the first [109] of the two publications of this chapter is the writing of the MaxShiftM algorithm C code, testing and optimising the code, running all experiments, and incorporating it into BEAR [8] as one of the choosable algorithms. I co-authored the paper with Solon P. Pissis. For the second publication [4], I co-authored the C++ code with Lorraine Ayad and incorporated the SeqAn library [32]

to build the libFLASM library. I also wrote Example Program 1 which can be used to perform pairwise circular sequence alignment. I wrote and ran Experiments I, II and V and incorporated libFLASM into BEAR. The user of BEAR now has the option to choose between the Hamming and Edit distance models when using the *FLASM* method. All authors contributed to writing the paper.

Contribution [53, 54] : In Chapter 5, we make effective use of q-grams to perform pairwise circular sequence alignment (CSA). We introduce the concept of *β-blockwise q-gram distance* between two circular strings x of length n and y of length m, which, by splitting a string into β blocks, gives a more powerful generalization of the q-gram distance described in [143]. We present three algorithms, where q is the q-gram length and Σ is the alphabet of x or y:

1. An exact naïve algorithm (nCSC), requiring time $\mathcal{O}(m(m+n))$ and space $\mathcal{O}(m+n+|\Sigma|^q)$.
2. A heuristic algorithm (hCSC), requiring time $\mathcal{O}(\beta(m+n)+\frac{m(m+n)}{\beta})$ and extra space $\mathcal{O}(m+n+|\Sigma|^q)$.
3. An exact suffix array-based algorithm (saCSC), requiring time and space $\mathcal{O}(\beta m+n)$.

Finally we show how the algorithms compare in terms of accuracy and speed for performing pairwise sequence alignment, and we discuss the inclusion of a refinement step to make algorithm saCSCr that runs in time $\mathcal{O}(\beta m+n+L^3)$ (where L is a user-defined parameter $0 \leq L \leq \frac{m}{3}$), but greatly improves the accuracy of the result.

My direct contribution for this chapter is writing the C++ code of algorithms nCSC and hCSC, cNW, hSW, authoring the pseudocode of all three CSC algorithms, and writing the code of the refinement step of algorithm saCSC. I wrote and ran Experiment II for Time Performance and Experiment III for testing the accuracy of the algorithms with synthetic data.

Contribution [52, 110] : In Chapter 6, we describe the Elastic-Degenerate (ED) text format used to represent a set of closely related sequences such as a pan-genome in a compact form. An ED text has a length n and total size N. We describe how we perform on-line searching for a pattern in an ED text and how we incorporated variants such as SNPs (Single Nucleotide Polymorphisms) from VCF files to search patterns in the Human pan-genome. We then introduce three ED String Matching

(EDSM) algorithms for searching a pattern of length m or sets of patterns of combined length M in an ED text on-line:

1. Naïve EDSM for single pattern matching, requiring time $\mathcal{O}(nm^2 + N)$ and space $\mathcal{O}(m)$.
2. EDSM-BV for single pattern matching, requiring time $\mathcal{O}(N \cdot \lceil \frac{m}{w} \rceil)$ and preprocessing time and space $\mathcal{O}(m \cdot \lceil \frac{m}{w} \rceil)$.
3. Multi-EDSM for multiple pattern matching, requiring time $\mathcal{O}(N \cdot \lceil \frac{M}{w} \rceil)$ and space $\mathcal{O}(M)$.

We compare the speed of the algorithms against each other using both synthetic and real datasets—Human genomic data obtained from Phase 3 of the 1000 Genomes Project [138] containing variants from 2,504 individuals. We also show how results can be verified and how we used Multi-EDSM as part of a pipeline for determining the validity of Minimal Absent Words (MAWs) in the Human genome. MAWs are patterns/motifs that are absent from a text/genome but whose factors are present in the text. They are sometimes employed in the study of genomes. We were able to validate the results of Silva et al. [128] who found 3 MAWs absent in the Human genome *reference* sequence but present in the Ebola virus genome. We discovered that all three MAWs actually exist in individuals present in the 1000 Genomes dataset. This means these MAWs are unsuitable for diagnosing Ebola infection because they naturally exist in the greater human population and could lead to a false diagnosis.

My contribution for this chapter is writing the `C++` and `Python` code for EDSM-BV and Multi-EDSM along with the tools EDSO and EDSRand, for creating ED texts from real data and generating uniformly random synthetic ED texts, respectively. I contributed optimisations and amendments to the EDSM-BV algorithm. I devised using the Multiple Shift-And algorithm for finding multiple borders simultaneously and performing multiple exact pattern matching in the Multi-EDSM algorithm. I also devised various optimisations for the alternative implementation of the Occ-Vector$_P$ data structure of the Multi-EDSM algorithm to reduce memory complexity and improve performance. I wrote and ran all the code for the experiments and I co-authored the Multi-EDSM paper [110] with Solon P. Pissis.

Chapter 3

Preliminaries

In this chapter we present some of the definitions and data structures that form the algorithmic foundations for the rest of the thesis and that are relevant to all or at least two of the following chapters.

3.1 Strings

Introduced here are general definitions and fundamentals in string algorithms presented in a summarised form following the example of *Crochemore et al.* in [26].

We define an *alphabet* Σ as a finite non-empty set of letters of size $\sigma = |\Sigma|$. In this thesis we set the condition that we will be working only with fixed alphabets of constant size ($|\Sigma| = \mathcal{O}(1)$). We define a *string* x as a linear array of letters taken from Σ where each letter has size 1. A string has length $m = |x|$ and an *empty string*, denoted by ε, has length $m = 0$. We use the term *sequence* when referring to a non-empty string in the context the biological macromolecule it represents, e.g. DNA.

By $x[i]$, we denote the position of a single letter in a non-empty string by its index i in x, for $i = 0, 1, \ldots, m-1$. We follow the convention that $i = 0$ is the first index of a letter in the string. By $x[i..j]$ we denoted a range of letters spanning from index i to index j in x, inclusive of j.

A string x is said to be *identical* to another string y, or $x = y$, if all their letters are identical at every position, $x[0..|x|-1] = y[0..|y|-1]$, and $|x| = |y|$. The operation of *concatenation* of two strings, that is combining two strings x and y together, produces a new string z of length $|z| = |x| + |y|$. We simply represent the concatenation operation like so: $z = xy$. Let $x = \texttt{CONCAT}$ and $y = \texttt{ENATION}$, then performing concatenation, $z = xy$, results in $z = \texttt{CONCATENATION}$.

A *circular string* of length m can be informally defined as a standard linear string where the first-occurring letters and last-occurring letters are wrapped-around and positioned beside each other. In essence, the starting and ending positions of a circular string are incidental and although presented as a single linear string, it can be regarded as a set of m linear strings each of length m all of which are considered equivalent. What makes the strings distinct from each other is their j-th *rotation*. Let $x = \texttt{abcd}$ be a linearised circular string of length m in its initial rotation ($j = 0$). The following rotations for $0 \leq j < m$ are equivalent: $x_0 = \texttt{abcd}$, $x_1 = \texttt{bcda}$, $x_2 = \texttt{cdab}$ and $x_3 = \texttt{dabc}$.

Considering some strings x, y, u and v. When $y = uxv$, x is said to be a *factor* or *substring* of y. If $u = \varepsilon$, then x is said to be a *prefix* of y; but if $v = \varepsilon$, then x is said to be a *suffix* of y. Additionally, if neither u and v are empty strings then x or any non-empty factor of x is called an *infix*. The term *proper* is traditionally used to describe prefixes or suffixes where one of the strings u or v are empty but not both at the same time. x is described as a proper suffix or prefix of y when $x \neq y$.

A set of *q-grams* are all the factors of a string x of length q and $q \leq |x|$. The maximum number of q-grams that can be created for a string x is $|x| - q + 1$. In practice, q is kept small so a reasonably-sized set of q-grams, a *profile* of a given string, can be collected, and the q-gram can be retrieved efficiently from an array when encoded to its numerical form.

In the context of text searching, if x is a factor of a string y we say that there is an *occurrence* of x in y that *starts* at position i and *ends* at position $i + |x| - 1$, such that $y[i \,..\, i + |x| - 1] = x$.

3.2 Dynamic Programming for String Matching

Dynamic Programming refers to a technique for finding a solution to a complex problem by combining the solutions to overlapping subproblems to give an answer in polynomial time where bruteforce naïve approaches to solving the problem prove intractable with increasing data size. Subproblems are computed only once and their resulting values stored in a table/matrix to be reused instead of being recalculated. The goal of performing calculations with dynamic programming is to find either the minimum or maximum score and, optionally, to enumerate one or more of the optimal solutions to a problem. The dynamic programming approach to solving the *global sequence alignment* problem, that is the comparison of two sequences from end to end to determine their best alignment to one-another, was first applied in bioinformatics in 1970 by Needleman and Wunsch [99]. They used it to determine homology between pairs of evolutionarily-

related protein sequences. They compared two similar sequences x and y and identified the minimum number of single-character changes — substitutions, insertions or deletions — required to change x into y (so that $y = x$). Their algorithm gives a measure of similarity or homology of a pair of sequences. Later advances generalised this approach to solving semi-global alignment, local alignment, alignments with affine gap-penalties and even approximate pattern matching problems [50, 124, 125, 146]. Please refer to the cited papers for definitions of the problems and descriptions of how they solved them.

The basic idea of the generalised algorithm for pairwise sequence alignment involves comparison of all nucleotide bases of sequence x one by one against all the nucleotide bases of sequence y and placing the score at the position of the comparison in the dynamic programming (DP) matrix. At each position in the matrix, three immediate neighbouring cells containing previously calculated scores are used to find the local optimal score. By the end of the comparison, a global optimal score can be determined and the best alignment sequence can be obtained. Although different approaches to scoring schemes have been proposed, a common default approach in many algorithms is to use the simplified *Edit/Levenshtein distance* model [28, 78], where each substitution, deletion or insertion operation increases the distance by one (cost/penalty) unit, and minimising difference (reducing *distance*) or maximising homology (increasing similarity) are equivalent [132].

Consider a sequence $x = \texttt{CGAGTC}$ of length $n = 6$ and sequence $y = \texttt{CGGTC}$ of length $m = 5$, we define the edit distance, denoted by the function $\delta_E(x, y)$, as the minimum total cost of operations to change x into y, where each change operation (substitution, insertion or deletion) has a cost of 1. We can guarantee the best possible alignment is discovered in $\mathcal{O}(mn)$ operations in $\mathcal{O}(mn)$ space. Under the edit distance model the following operations are allowed:

- *Insertion*: insert a letter in y, not present in x; $(\varepsilon, b),\ b \neq \varepsilon$
- *Deletion*: delete a letter in y, present in x; $(a, \varepsilon),\ a \neq \varepsilon$
- *Substitution*: replace a letter in y with a letter in x; $(a, b),\ a \neq b, and\ a, b \neq \varepsilon$.

The operations being modeled are insertion, deletion and substitution. In the case of the insertion operation, it models inserting a letter b from sequence y into x in a position represented by an empty string ε. We denote this with the tuple (ε, b). This increases the length of x by one. And in the case of modeling the deletion operation, we remove a letter from y, and represent this as ε in the tuple (a, ε). This decreases

the length of y by one. Finally, in the case of substitution, we replace letter b of y with a of x. The length of both strings would remain unchanged. Each operation has a unit cost.

Before performing these operations, we create the data structure to hold the result of the calculations. We create a dynamic programming matrix M of dimensions $(n+1) \times (m+1)$, with string x going across (x-axis) and string y going down (y-axis). The first cell represents a comparison of ε to itself giving a score of 0. We initialise the rest of the cells in the first column and the first row with monotonically increasing distance costs representing increasing-size gaps (insertions/deletions). Then we do the rest of the calculations, comparing the characters of each string and storing the results from position $M[1..m][1..n]$. We move cell-by-cell going left-to-right doing the calculation. This is the traditional direction of doing the calculation but it can also be done top-to-bottom. Given a cell in the DP matrix $M[i][j]$ where $0 < i \leq |y|$ and $0 < j \leq |x|$, we check three neighbouring cells and store the result of the calculation in the current cell. We check:

- $M[i-1][j]$: The cell above, representing an insertion of a letter of y.
- $M[i][j-1]$: The left cell, representing a deletion of a letter in x.
- $M[i-1][j-1]$: The diagonal (top-left) cell, containing the score of the last matched pair of bases $x[i-2]$ and $y[j-2]$.

Initialisation:

$$\mathsf{M}[0][i] = i \text{ for } i = 0..n$$
$$\mathsf{M}[j][0] = j \text{ for } j = 1..m$$

Calculation:

$$\mathsf{M}[i][j] = \min \begin{cases} \mathsf{M}[i-1,j]+1 \\ \mathsf{M}[i,j-1]+1 \\ \mathsf{M}[i-1,j-1]+(1 \,\mathsf{if}\, x[i-1] \neq y[j-1]) \end{cases}$$

$$\text{for all } i = 1..m, j = 1..n.$$

We are now in a position to draw the completed dynamic programming matrix as seen below in Table 3.1.

To obtain the optimal solution we can follow a pathway through the matrix from the score in the bottom right corner of the matrix back to score 0 in the top left of the

Table 3.1 The Dynamic Programming Matrix for $x = \texttt{CGAGTC}$ and $y = \texttt{CGGTC}$ for optimal global sequence alignment under the edit distance model, minimising errors, with an optimal solution highlighted in blue.

	ε	C	G	A	G	T	C
ε	0	1	2	3	4	5	6
C	1	0	1	2	3	4	5
G	2	1	0	1	2	3	4
G	3	2	1	1	1	2	3
T	4	3	2	2	2	1	2
C	5	4	3	3	3	2	1

matrix. We *backtrack* through the matrix following a path that moves from cell to cell by picking the one with the lowest penalty score. Ties are broken arbitrarily indicating there could be more than one optimal solution. Where there is a horizontal transition it indicates an insertion operation in y, so the optimal solution in converting y to x is $y = x = \texttt{CGAGTC}$ by inserting $x[2] = \texttt{A}$, giving an edit distance of 1.

The DP matrix can be used to perform string comparisons, global sequence alignment, under the Hamming distance model [55], denoted by $\delta_H(x, y)$. This model only takes substitutions into consideration (only the diagonal cell $M[i-1][j-1]$ is checked) and has the condition that the strings compared should be of the same length, $|x| = |y|$. Under this model, the first row and column of the DP matrix are initialised to ∞, or in practice a very large number, as any insertion or deletion operation is not permitted: $M[0][j] = \infty$ for $j = 0..n, M[i][0] = \infty$ for $i = 1..m$. This model is faster to process because it compares the value of one cell and is therefore simpler but it is not suitable for use with strings of different lengths or that contain deletions or insertions.

The DP matrix can also be used to perform approximate pattern matching, also known as approximate string matching (ASM), as first proposed by Peter Sellers in 1980 [125], and it can be done under either the edit or Hamming distance model. The ASM problem involves finding factors of a text that match a pattern approximately (in addition to matching exactly). How approximately a factor must match a pattern before it is considered a valid match is determined by a distance threshold parameter set by person doing the search. In some cases the desired goal is to find a matching factor with the least distance, i.e. the best match.

Let x be the sequence of length n being searched and y be the pattern of length $m \leq n$; in order that the pattern be able to start at any position i in x without penalty, the first row of the matrix is initialised to zero, while the first column's distance scores increase monotonically up to m; $M[0][j] = 0$ for $j = 0..n, M[i][0] = i$ for $i = 1..m$. The

smallest score in the last row indicates the ending position to trace back from to obtain the best match. This factor of the text has the least distance from y. There may also be more than one best factor if their distances are identical.

Let $x = \texttt{CGAGTC}$ and $y = \texttt{ATT}$, we produce the following DP matrix M in Table 3.2 and observe the lowest score (highlighted) in the last row, showing that the best match ends at position $x[4]$, with $\delta_E(x, y) = 1$, due to a single $\texttt{T} \rightarrow \texttt{G}$ substitution.

Table 3.2 The Dynamic Programming Matrix for $x = \texttt{CGAGTC}$ and $y = \texttt{ATT}$ for approximate string matching under the edit distance model, minimising errors, with the lowest score in the last row highlighted in blue.

	ε	C	G	A	G	T	C
ε	0	0	0	0	0	0	0
A	1	1	1	0	1	1	1
T	2	2	2	1	1	1	2
T	3	3	3	2	2	1	2

One further point on the DP matrix — if the optimal solution is not required, we do not have to trace back through the DP matrix and all we need is the edit distance value at the end of the calculation, so we need only store two rows/columns at any point in time during processing. This is because at any position in traversing the matrix we check only the neighbouring cells from the current row and the row above. This reduces the space complexity of the algorithm to $\mathcal{O}(\min(m, n))$.

3.3 Bitwise Techniques

Bitwise or *bit-parallel* techniques take advantage of technological advancements in processor design for quick, simultaneous manipulation of *bits* in a *computer word* of memory. On modern processors, the size of a computer word in bits is $w = 64$, thus offering speed-ups up to a factor of w [37]. The other advantage besides parallel processing is memory savings when data can be encoded in a more compact form.

The examples in this thesis will follow the convention of *little-endianness* – that is, when considering the numeric representation of bits in a computer word, the most significant bits lie towards the left-hand side and the least significant bits towards the right. So when presenting multiple computer words, i.e. an array of computer words, they should be read word by word from the right-most word to the first word on the left.

We now present a few definitions, explain some terminology and describe some techniques used when working with computer words.

A *computer word* or just *word* stores information as a binary sequence of 1's and 0's in w *bits* of computer memory. It can be viewed as an w-sized sequence over the alphabet $\Sigma = \{0, 1\}$. Although we treat a computer word as one unit, it in fact occupies $w/8$ bytes in memory. Modern processors can access a position in memory and perform operations on a single computer word, all $w/8$ bytes simultaneously, in $\mathcal{O}(1)$ time.

By b_i, we denote the position of an individual bit in a computer word numbered from right-to-left, $i = 0..w-1$, in increasing numerical significance. We call the bit at position i in a word a *set bit* when $b_i = 1$. We say a word has been *cleared* or it is described as *clear* when all positions $b_0..b_{w-1} = 0$.

A *bit mask* or simply *mask* is a computer word where certain bits have been set to 1. Masks are generally used for testing the presence of set bits at certain positions or to maintain or clear combinations of bits at predetermined positions.

We call an array of computer words a *word vector* and we call the series of bits of length m in the word vector a *bit vector*. A bit vector b of length $m = |b|$ bits occupies $\lceil m/w \rceil$ computer words in a word vector. Sometimes, the terms word vector and bit vector are used interchangeably because they are intrinsically related.

Various different operations can be performed in constant time on a computer word. We present some of these operations in the list below along with their arguments (words x and y and integer α) and their *operator* symbols. We use these symbols following the example of the C programming language [120]. Note that the Pop-Count function does not have an *operator* symbol to represent it, so the function name itself is used in this thesis.

- Pop-Count(x) — `pop-count()` — Returns the count of set bits in word x.
- Left-Shift(x,α) — $\ll$ — Alters x by shifting all bits α positions to the left. Bits shifted beyond the word size are discarded.
- Right-Shift(x,α) — $\gg$ — Alters x by shifting all bits α positions to the right. Bits shifted beyond the word size are discarded.
- Logical-NOT(x) — $\sim$ — Returns an inverse representation of the word, swapping 0s for 1s and vice-versa.
- Logical-AND(x,y) — & — Takes two words as arguments and returns a new word with set bits at all positions i where $x_i = y_i$.

- Logical-XOR(x,y) —^— Takes two words as arguments and returns a new word with set bits at all positions where $x_i = 1$ or $y_i = 1$ but not $x_i = y_i$.

- Logical-OR(x,y) — | — Takes two words as arguments and returns a new word combining the set bits of both x and y.

To illustrate this better, we present in Table 3.3 operations performed on computer words $x = 12345 = \texttt{00110000 00111001}$ and $y = 4619 = \texttt{00010010 00001011}$. Note that the shift functions modify x in place. Also note that we specify a word size $w = 16$.

Table 3.3 Separate bitwise operations on x and what they return or what x becomes after an operation.

Operation Name	**Operation**	**Binary Result**	**Integer Result**
Pop-Count	Pop-Count(x)	N/A	6
Left-Shift	$x \ll 1$	01100000 01110010	24690
Right-Shift	$x \gg 1$	00011000 00011100	6172
Logical-NOT	$\sim x$	11001111 11000110	53190
Logical-AND	x & y	00010000 00001001	4105
Logical-XOR	$x \hat{} y$	00100010 00110010	8754
Logical-OR	$x \mid y$	00110010 00111011	12859

For single words, a bitwise shift operation can cause a bit to be moved beyond the bounds of the word, in which case it is discarded. When performing a bitwise Left-Shift or Right-Shift operation on a word vector b of size $m > w$, the left or right most bit is carried to the next word. Bits are carried over until the final word in b and if there are any more bits to carry they are discarded beyond this point. The size of the word vector is not altered. But in the situation where $m \bmod w > 0$, a Left-Shift operation can cause a side-effect where m is effectively increased, introducing errors into an algorithm. To prevent this kind of error from being introduced, a bit mask is used to clear set bits that surpass the permitted size of the bit vector. Let $w = 8$, $m = 6$, $x = \texttt{00010100}$ and mask $s = \texttt{00111111}$. If during a hypothetical algorithm we do a Left-Shift operation, $x = \texttt{00101000}$, this moves the 1 to the last position allowed. Another Left-Shift operation would cause a side-effect, making $x = \texttt{01010000}$, effectively extending the bit vector to $m = 9$, which is not what we want. We stop this problem from occurring by applying a Logical-AND operation using the mask after the shift operation on x to ensure only the valid set of bits are maintained:

$$x = (x \ll 2)\ \&\ s$$

$$\texttt{00010000} = (\texttt{00010100} \ll 2)\ \&\ \texttt{00111111}$$

When x spans across multiple computer words we run into another potential problem — if one of the words has the last bit set and we perform a Left-Shift operation, this bit would be discarded instead of being carried to the next word. We can solve this problem using a carry-check bit mask to check the presence of a set bit at the last position of a word. Let $w = 8$, $m = 11$, $x = \texttt{00000010 10001000}$, the number of words $d = 2$, carry mask $v = \texttt{10000000}$ and c be the carry word which we add to the next word using bitwise-OR. We also use a bit mask $s = \texttt{00000111}$ to clear any bits shifted past the last position in the bit vector. We also use a temporary variable z to store the a copy of the word before the shift operation and we use it to check if we need to carry a bit to the next word. The operation Left-Shift-With-Carry is shown below (Algorithm 1).

Algorithm 1 Left-Shift-With-Carry performs the left-shift operation on multiple words.
x: a bit vector spanning multiple words in length.
d: the number of computer words used to represent x.
v: a bit mask (10000000) used to determine if a carry is required.
s: a bit mask (00000111) used to ensure no bits are shifted beyond the length of x.

```
procedure LEFT-SHIFT-WITH-CARRY(x, d, v, s)
    c ← 00000000
    for i ∈ {0..d−1} do
        z ← x[i]
        x[i] ← (x[i] ≪ 1) | c
        if (z & v) = 00000000 then
            c ← 00000000                          ▷ do not carry
        else
            c ← 00000001                          ▷ carry bit
    x[d−1] ← x[d−1] & s
    return x
```

Bit-parallel techniques have been found to be particularly effective when applied to string algorithm design, especially in their ability to simulate a non-deterministic finite automata (NFA) efficiently. NFAs can have multiple active states at any one time and these are effectively simulated using 1s at various positions in a computer word, which are shifted, set or cleared in one or more quick bitwise operations. It is outside the scope of this thesis to elaborate on NFAs and their bitwise implementations, but we refer the reader to various descriptions of bitwise algorithms in the literature [37, 98].

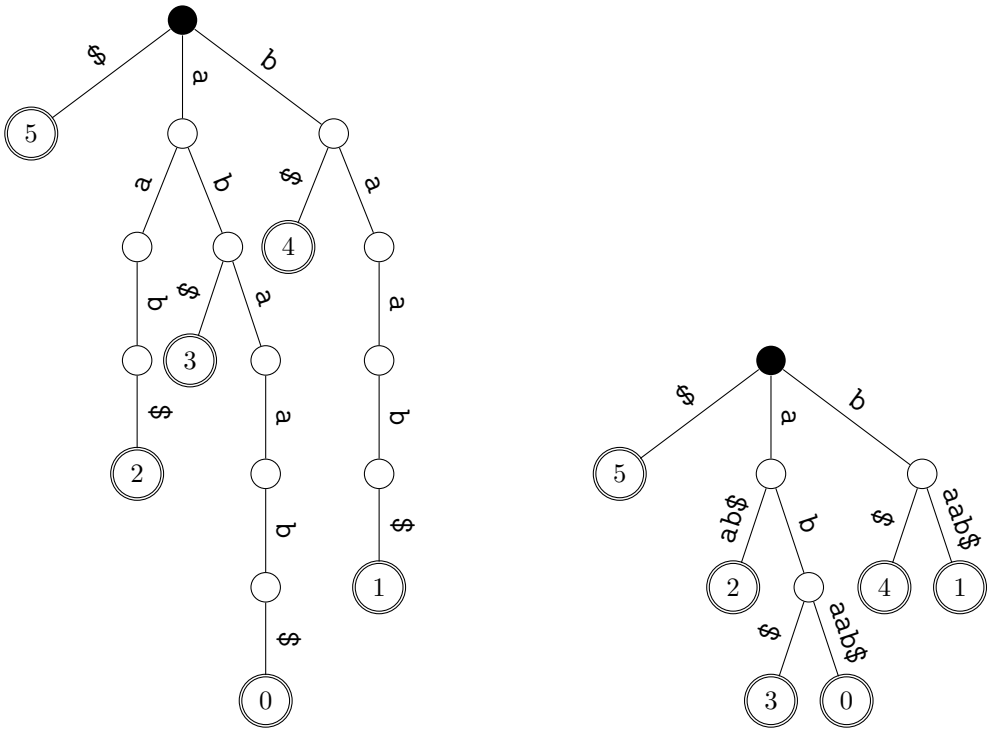
3.4 Suffix Trees and Suffix Arrays

The suffix tree [89, 148] and suffix array [86] are data structures used for indexing a given string x of length $n = |x|$ and can be used as a *deterministic automaton* (see [26]) to recognise any factor of length m of that string quickly and efficiently. Both data structures can be built efficiently requiring only $\mathcal{O}(n)$ time and space, with each search of a string y of length $m \leq n$ taking $\mathcal{O}(m)$ time.

The *suffix tree* of string x, denoted by ST_x, is a compacted *trie* representing all suffixes of x. Please refer to Figure 3.1 for a graphical depiction of the conversion from a suffix trie to a suffix tree. A trie is an *acyclical graph* structure laid out in the form of a *rooted tree* with each character of x represented as a *node* connected by a labeled *edge*. The first node in the trie is known as the *root*. If we imagine building a trie over all the suffixes of x we obtain a *suffix trie*. The suffix trie can be compacted into a *suffix tree* by deleting *internal nodes* with only one child and keeping internal nodes with *forks* (two or more children) or *leaves* (terminal nodes). The internal nodes of the suffix tree are now called *explicit* nodes, while the nodes that were deleted from the suffix trie, now represented by edges, are called *implicit* nodes. Each edge is labeled with a substring of one or more characters and no two edges starting out from a node begin with the same character. If we follow each node from the root down to a leaf, the concatenation of their edge labels spell out a unique string. We call the maximal length of the concatenated strings from the root down to any non-root node its *string depth* d_s and we call the level in the tree at which each explicit node exists its *node depth* d_n where $1 \leq d_n \leq d_s$.

Take for example the leaf of the suffix tree labeled 2 in Figure 3.1b – this has $d_s = 4$ and $d_n = 2$. Leaves mark the termination of a suffix and are numbered with the index at which they start in x – their *suffix number*. A problem arises where some suffix of x matches a prefix of another suffix of x, creating an internal node with only one child as if it were an implicit node. This problem is avoided by building the tree on $x = x\$$, that is x concatenated with a 'sentinel' character $\$ \notin \Sigma$, ensuring each suffix is unique. In this thesis we stipulate the sentinel character is lexicographically smaller than any alphabetical character.

By SA_x we denote the *suffix array* of $x = x\$$. SA_x is an integer array of size n storing the starting positions (*suffix number*) of all lexicographically sorted non-empty suffixes of x. The suffix array presents a space-saving alternative to the suffix tree requiring exactly n 4-byte integers in a linear array as opposed to $< 2n$ nodes requiring around $12 - 20$ bytes each [74]. Each position $\mathsf{SA}_x[i]$ in the suffix array represents a suffix $x[i \,..\, n-1]$, equivalent to a leaf from the suffix tree. The suffix array is often



(a) Suffix Trie of x (b) Suffix Tree of x

Fig. 3.1 The suffix trie (3.1a) and corresponding suffix tree (3.1b) for the string $x = \texttt{abaab\$}$. The root is filled in black, internal nodes are filled in white, leaves are shown with a double border containing the suffix number and edges are labeled with their respective substring.

used in conjunction with other arrays that augment its functionality. The suffix array and additional arrays are shown in Table 3.4. The *longest common prefix array* of x, denoted LCP_x, determines the longest common prefix of consecutive suffixes in the suffix array in constant time but it is primarily used to accelerate pattern matching. For every index $j = 1..n-1$, the array gives the longest common prefix between $\mathsf{SA}_x[j-1]$ and $\mathsf{SA}_x[j]$, and $\mathsf{LCP}_x[0] = 0$. The *inverse suffix array* of x, denoted iSA_x, gives the *rank* or index of a suffix in the suffix array given a suffix number. It behaves as a reverse index, so when $\mathsf{SA}_x[j] = i$, $\mathsf{iSA}[i] = j$. All three arrays of length n, over a fixed alphabet, can be computed in time and space $\mathcal{O}(n)$ [40, 101].

Table 3.4 The lexicographically ordered suffixes and Suffix Array (SA), Rank/Inverse Suffix Array (iSA) and Longest Common Prefix (LCP) array for $x = \texttt{abaab\$}$.

i	**Suffix** i	$\mathsf{SA}_x[i]$	$\mathsf{iSA}_x[\mathsf{SA}_x[i]]$	$\mathsf{LCP}_x[i]$
0	`$`	5	0	0
1	`aab$`	2	1	0
2	`ab$`	3	2	1
3	`abaab$`	0	3	2
4	`b$`	4	4	0
5	`baab$`	1	5	1

Chapter 4

Fixed Length Approximate String Matching

This chapter presents the work done in the following publications:

1. [109] S. P. Pissis, A. Retha, "Generalised Implementation for Fixed-Length Approximate String Matching Under Hamming Distance and Applications", in Proceedings of the 2015 IEEE International Parallel and Distributed Processing Symposium Workshop, Washington, DC, USA: IEEE Computer Society, pp. 367-374.

2. [4] L. A. Ayad, S. P. Pissis and A. Retha, "libFLASM: a software library for fixed-length approximate string matching", BMC Bioinformatics, vol. 17, no. 1, 2016, pp. 454.

4.1 Introduction

Approximate string matching is the problem of finding all factors of a text t of length n with a distance at most k from a pattern x of length $m \leq n$. Fixed-length approximate string matching is the problem of finding all factors of t with a distance at most k from *any factor of* x of length ℓ, where $\ell \leq m$. Fixed-length approximate string matching is therefore a generalisation of approximate string matching. We present two bit-parallel algorithms to solve the fixed-length approximate string matching problem *on-line* in time $\mathcal{O}(m\lceil \ell/w \rceil n)$ and space $\mathcal{O}(m\lceil \ell/w \rceil)$ under both the *Hamming* and *edit* distance models, where w is the size of the computer word. As the performance of these techniques are independent of the distance threshold k or the alphabet size $\sigma = |\Sigma|$,

they offer great performance and flexibility for general use and application in many fields including in computational molecular biology, as we demonstrate below.

4.1.1 Technical Background

Various algorithms exist to tackle the problem of finding an exact or similar string *pattern* in a given string *text* applied to various branches of scientific research [96]. Many of these solve the *approximate string matching* (ASM) problem. Approximate matching allows for a limited number of *errors* or *edit operations* needed for the searched pattern to have a match in the text. In the field of molecular biology, ASM algorithms are used for extracting motifs or regions of interest [108] or for the alignment of short reads to longer DNA or protein sequences [136]. This is important for solving problems such as gene classification, protein function identification, mutation discovery and genetic sequencing.

The use of ASM algorithms is required because *exact string matching* (ESM) algorithms are unsuitable given the nature of the text being searched; DNA sequence changes occur at a steady rate with the average single nucleotide substitution mutation rate in humans estimated to be 1.20×10^{-8} per nucleotide per generation [71]. Cumulative mutations passed on through heredity introduce significant sequence variation within a population and apart from single-nucleotide substitution, there also occur nucleotide insertion and deletion events (*indels*) but these occur are significantly lower rates. The more genetically diverse a population is, or the higher the rate of mutation in a species, or the greater the evolutionary distance between species, the more likely that one or more bases in a stretch of DNA will have changed, so an ESM approach will not always find the expected pattern being searched for, especially when longer patterns are considered.

The premise for comparison between two or more divergent amino acid or nucleotide sequences is that the sequences represent evolutionarily related *homologs* being present once upon a time as one sequence in a common ancestor. As a direct result, bioinformaticians present models of divergence, incorporating them into *scoring/weighted matrices*, to estimate the degree by which two sequences are related, with more homologous sequences exhibiting a smaller *distance* considered to be evolutionarily closer. Models can take into account substitution rates between different nucleotides or amino-acids, as well as inversion events and the frequency and significance of indel events [30, 67, 90, 94]. Different scoring schemes have also been created for use with more or less divergent sequences, the most popular being the *PAM* and *BLOSUM* matrices [30, 58]. Although several such biological models for ASM exist, they tend to be more computationally

expensive [50], so the edit/Levenshtein and Hamming distance models [28, 55, 78], although not taking biological models of evolution into account, are still extensively used as a reasonable accuracy-efficiency trade-off for use in biological as well as general contexts. The edit distance model uses substitutions, deletions, and insertions of letters to find the minimum number of operations to change one string into another. We specify an edit cost of 1 for all operations. The Hamming distance model considers equally-sized strings and allows only for substitutions operations and like the edit distance we set the cost of the operation to 1.

In 1980 Sellers [125] presented the first edit-distance ASM algorithm for pattern matching based on *dynamic programming* requiring $\mathcal{O}(mn)$ time, where m is the length of the pattern and n is the length of the text. Later major theoretical and practical advancements brought the complexity of this thread of research down to an $\mathcal{O}(kn)$-time algorithm (see [45, 46, 75]), where k is the maximum edit distance allowed. This specific approach to the problem is termed the *k-mismatches* problem or *approximate string matching with k-errors*. Still later, more progress was made on this by exploiting a thread of practice-oriented research employing the hardware-based word-level parallelism of bitwise operations. By using this fact cleverly, the number of operations that a bit-parallel algorithm performs can be cut down by a factor of at most w, where w is the size of the computer word. In order to relate the behaviour of bit-parallel algorithms to other work, it is normally assumed that $w = \Theta(\log n)$, as dictated by the RAM model [96]. Wu and Manber, in [151], gave an $\mathcal{O}(k\lceil m/w \rceil n)$-time algorithm for the k-errors problem by simulating a non-deterministic finite automaton [142] using word-level parallelism. Baeza-Yates and Navarro, in [6], gave an $\mathcal{O}(\lceil km/w \rceil n)$-time variation of the Wu-Manber algorithm, implying $\mathcal{O}(n)$ performance when $km = \mathcal{O}(w)$. In 1994, Wright [150] presented an $\mathcal{O}(m \log(\sigma) n/w)$-time algorithm, where $\sigma = |\Sigma|$, the size of the input alphabet. This was the first work using word-level parallelism on the dynamic programming matrix. Then in 1999, Myers [95] introduced a fast and efficient bit-parallel algorithm based on the dynamic programming matrix for approximate string matching under the edit/Levenshtein distance model in time $\mathcal{O}(\lceil m/w \rceil n)$. The mentioned advances in ASM presented a good starting point for solving the generalised version of the FLASM problem. In 2001, Iliopoulos *et al.* [65] introduced the first bit-vector algorithm based on dynamic programming to solve the problem and in 2010 Crochemore *et al.* devised MaxShiftM [27], a bit-parallel algorithm for fixed-length approximate string matching with k-errors under the Hamming distance model with time complexity $\mathcal{O}(m\lceil \ell/w \rceil n)$.

4.1.2 Motivation

Circular sequences are found in various places in the domain of biology [25, 57]. The bulk of bacterial DNA is stored in a circular chromosome but bacteria have additional genes stored in plasmids, circular loops of double-stranded DNA. Bacteria may harbour a variety of plasmids, giving them distinctive properties such as the ability to produce toxins or resist the effects of antibiotics [29]. Viral genomes also happen to be circular and the ability to identify regions of interest helps us in the discovery of druggable sites. Circular DNA is also present in specialised organelles, chloroplasts and mitochondria, of eukaryotic plant and animal cells. Mitochondrial DNA (mtDNA) is found inside mitochondria [145] and is commonly used in phylogenetic reconstruction and research into ancestry and evolution because it is consistently passed on from mother to offspring without recombination and it has a faster mutation rate than chromosomal DNA. Additionally, some proteins are noted to share homologous domains but do not align because of swaposins, domains that are relocatable and have the property of circular permutability [83, 112]. Additionally, some proteins bind on their N and C termini in order to form a circular chain [25]. The wide presence of the circular structures in biology attests the importance of analysing circular sequences and finding algorithms suitable for its study [54].

4.1.3 Applications

Application I: Multiple Circular Sequence Alignment

Circular sequences have no point of reference by which they are sequenced or aligned to one another and treating them as linear sequences leads to poor alignments. By identifying the correct rotation for a pair of circular sequences, sequence alignment can be carried out to produce more reliable results. This is evident when analysing the linearised human (NC_001807) and chimpanzee (NC_001643) mtDNA sequences which start at different biological regions. Without refining the sequences, the pairwise sequence alignment of the mtDNA using the pairwise sequence alignment tool EMBOSS Needle [119] gives a similarity score of 85.1% with 1,195 gaps. Aligning different rotations of the same sequences yields a similarity of 91% with only 77 gaps [8] – a far better result! *Multiple Circular Sequence Alignment* (MCSA) involves aligning three or more circular sequences simultaneously, which is a common task in computational molecular biology. As similar to the standard setting, this alignment can be used to find patterns within protein sequences and specifically, identify homology between new and existing groups of related sequences [127]. Just as importantly, it helps

in identifying novel regions or mutations that give a species or breed its distinctive properties or highlights the cause of disease. A few tools currently exist to tackle the MCSA problem [8, 39, 92].

Application II: Simple/Structured Motif Extraction

Motif extraction (ME), or motif discovery, involves detecting overrepresented DNA motifs (patterns) as well as conserved DNA motifs in a set of orthologous DNA sequences. Such conserved motifs may serve as potential candidates for transcription factor binding sites for regulatory proteins [56]. The pattern, which is usually fairly short, 5 to 20 base-long, can be located in different genes or several times within the same gene. ME, however, may also be relevant for extracting longer regions within DNA sequences. A study in [11] shows that there exist 481 regions longer than 200 bases that are absolutely conserved in the genomes of the human, rat, and mouse. This fact suggests the possibility of the existence of long motifs in the presence of substitutions, insertions or deletions, underscoring the necessity of ME for larger lengths. Many tools exist to tackle the ME problem for single motifs [87, 104, 111, 130].

In addition to this simple form of single motifs, structured motifs are another special type of DNA motif. A structured DNA motif is made up of two (or even more) smaller conserved sites with a variable-sized spacer (gap) located between these sites. The spacer is found in the middle of the motif due to the transcription factors binding as a dimer. This means that the transcription factor is made up of two subunits, having two separate contact points to bind with the DNA sequence. A non-conserved, usually fixed-length or slightly variable-length spacer separates these contact points. Such conserved structured motifs may serve as potential candidates for transcription factor binding sites for a composite regulatory protein [35]. A few tools exist to tackle the ME problem for structured motifs [19, 108, 152].

ME can also be used on *immunoglobulins* and *T cell receptors* found on the surface of *T lymphocytes*, important components of the immune system of humans and other vertebrates. Immunoglobulins, or *antibodies*, have the principle function of eliminating foreign objects and pathogens such as bacteria, by attaching to them and neutralizing their effects on the body [77]. Obtaining the structural information of an antigen is essential for monoclonal antibody engineering. Synthetically-engineered antibodies [116] are created by introducing synthetic DNA into specific regions of the antibody, which has proven successful for the treatment of diseases such as Rheumatoid Arthritis [18].

Application III: Chang and Marr Index

Some average-case optimal algorithms for approximate string matching, such as [42], are based on a pre-processing step implementing the *Chang and Marr index* [20]. This step involves constructing an array $D[0..\sigma^q - 1]$ of integers, such that for every q-gram string s of length q over an alphabet of size σ, s is searched for in pattern x, and $D[s]$ contains the smallest distance between s and any match found in x. Storing D requires space $\mathcal{O}(\sigma^q)$ and computing it naïvely takes time $\mathcal{O}(\sigma^q qm)$ for the edit and Hamming distance models. The Chang and Marr index is used in a number of algorithms as a preprocessing step because the cost of computing it is outweighed by the performance improvements it presents in searching a large amount of text. It is used to obtain the minimal distance between a q-gram of the text being searched and the pattern in $\mathcal{O}(1)$ time with a simple index lookup.

Application IV: Approximate Circular String Matching

Approximate circular string matching (ACSM) is the problem of finding all factors of a given text that are at a distance k from any of the m rotations of a given circular pattern of length m. ACSM has various applications such as finding permutations in proteins [147], but it has other uses outside of biology and has been applied extensively in pattern recognition (see [85], for instance). Many average-case algorithms exist to tackle the ACSM problem efficiently for low error ratios k/m [9, 10, 42, 59].

4.1.4 Our Contributions

Our contribution in [109] is to present a generalised implementation of the MaxShift algorithm as proposed by [27] — MaxShiftM — employing the Hamming distance model [55] to perform Fixed Length Approximate String Matching (FLASM). Our implementation overcomes the limitation of $\ell \leq w$ of a naïve implementation, meaning the length of a factor can be longer than the computer word size. And in [4] we make use of Myers' algorithm [95] for approximate pattern matching, employing the edit distance model [28], and tailor it to solving the FLASM problem. We have released both algorithms bundled in an open-source C++ software library — libFLASM — which we provide with example applications and documentation. In this thesis we demonstrate practical applications of the algorithms in biological as well as general purpose contexts.

4.2 Definitions

For general definitions in regards to dynamic programming and the edit and Hamming distance models, please refer to Chapter 3.2 above. We only consider edit and Hamming distance with unit-cost operations of 1:

- *Edit distance* – Given an integer $k > 0$ and strings x and y, if $\delta_E(x, y) \leq k$ we say that x and y have at most k *differences*.
- *Hamming distance* – Given an integer $k > 0$ and strings x and y, if $\delta_H(x, y) \leq k$ we say that x and y have at most k *mismatches*.

We provide some further definitions following Carvalho et al. [19] to describe the two types of motif.

We define a *single motif* (also known as a *simple motif*) as a string of letters on an alphabet Σ. We are given an integer value k denoting a distance threshold (error threshold). We then say that a motif on Σ k-*occurs* in another string s on Σ, if there is a distance (edit or Hamming) of at most k between the motif and a factor of s. A set $s_1, \ldots, s_N$ of strings on Σ, where $N \geq 2$, the quorum $1 \leq q \leq N$, the error threshold k, and the length ℓ for the motifs are taken as input for the *single motif extraction* problem. Specifically, it involves identifying all motifs of length ℓ, with each motif k-occurring in at least q input strings. In this case, such single motifs are called *valid*.

We define a *structured motif* by pairs of the form (m, d), where m_i represent simple motifs, numbered $m_1, m_2 \ldots m_\beta$ and the structured motif ends with m_β; and d is a tuple representing an interval or spacer between simple motifs. A tuple of d is represented by $d_i = (d_{\min_i}, d_{\max_i})$ where each element specifies the minimum and maximum length of the spacer and there are $i = 1 \,..\, \beta - 1$ such intervals in the structured motif seperating the simple motifs from each other.

We denote a structured motif by:

$$m_1[d_{\min_1}, d_{\max_1}]m_2, \ldots, m_{\beta-1}[d_{\min_{\beta-1}}, d_{\max_{\beta-1}}]m_\beta.$$

Simply put, a structured motif is two or more motifs separated by one or more variable regions (spacers/intervals) in the middle. The contents of the variable region intervals consist of *don't care* bases, *spacers*, while the elements $m_1, m_2, \ldots, m_\beta$ of a structured motif that matter and are pattern-matched are called *boxes*. The length of box m_i is denoted by ℓ_i while the length of a variable region is denoted by $d_{\min_j} \,..\, d_{\max_j}$.

We are given a β-tuple $(k_i)_{1 \leq i \leq \beta}$ of error thresholds — an error threshold k for each of the boxes. We then say that a structured motif (m, d) has a $(k_i)_{1 \leq i \leq \beta}$-occurrence in another string s on Σ, if there is a k_i-occurrence m_i' of m_i, for all $1 \leq i \leq \beta$, such that:

1. $m'_1, m'_2, \ldots, m'_\beta$ occur in s and
2. the distance between the ending position of m'_i and the starting position of m'_{i+1} in s is in interval $[d_{\min_i}, d_{\max_i}]$, for all intervals $d_1 \ldots d_{\beta-1}$.

A set $s_1, \ldots, s_N$ of strings on Σ, where $N \geq 2$, the quorum $1 \leq q \leq N$, β lengths $(\ell_i)_{1 \leq i \leq \beta}$, β error thresholds $(k_i)_{1 \leq i \leq \beta}$, and $\beta - 1$ intervals $(d_{\min_i}, d_{\max_i})_{1 \leq i < \beta}$ of distance are taken as input for the *structured motif extraction* problem. Specifically, it involves identifying all structured motifs that have a $(k_i)_{1 \leq i \leq \beta}$-occurrence in at least q input strings. In this case, such structured motifs are called *valid*. A problem instance is denoted by:

$$< (\ell_1, k_1)[d_{\min_1}, d_{\max_1}](\ell_2, k_2) \ldots (\ell_{\beta-1}, k_{\beta-1})[d_{\min_{\beta-1}}, d_{\max_{\beta-1}}](\ell_\beta, k_\beta), q >.$$

We are now in a position to define the FLASM problem, first under the Hamming distance model:

FixedLengthApproximateStringMatching (Hamming distance)
Input: A pattern x of length m, a text t of length $n \geq m$, an integer $\ell \leq m$, and an integer $k < \ell$.
Output: All factors u of t such that $\delta_H(u, v) \leq k$, where v is any factor of length ℓ of x.

We then extend the definition to the edit distance model:

FixedLengthApproximateStringMatching (Edit distance)
Input: A pattern x of length m, a text t of length $n \geq m$, an integer $\ell \leq m$, and an integer $k < \ell$.
Output: All factors u of t such that $\delta_E(u, v) \leq k$, where v is any factor of length ℓ of x.

Theorem 1 ([109])**.** *The FLASM problem under the Hamming distance model can be solved in time* $\mathcal{O}(m\lceil\frac{\ell}{w}\rceil n)$ *and space* $\mathcal{O}(m\lceil\frac{\ell}{w}\rceil)$*, where* w *is the size of the computer word.*

Theorem 2. *The FLASM problem under the edit distance model can be solved in time* $\mathcal{O}(m\lceil\frac{\ell}{w}\rceil n)$ *and space* $\mathcal{O}(\lceil\frac{\ell}{w}\rceil)$*, where* w *is the size of the computer word.*

Given a pattern x of length m, a text t of length $n \geq m$, and an integer $k < m$, Myers bit-vector algorithm solves the ASM problem under the edit distance model in time $\mathcal{O}(n\lceil\frac{m}{w}\rceil)$ and space $\mathcal{O}(\lceil\frac{m}{w}\rceil)$ [95]. Applying this algorithm for all $\mathcal{O}(m)$ factors of length ℓ of x separately solves the FLASM problem under the edit distance model.

		0	1	2	3	4	5	6	7	8	9
		ϵ	C	G	A	A	A	G	T	A	T
0	ϵ	0	0	0	0	0	0	0	0	0	0
1	C	1	0	1	1	1	1	1	1	1	1
2	A	2	2	1	1	1	1	2	2	1	2
3	A	3	3	3	1	1	1	2	3	2	2
4	A	3	3	3	2	1	0	1	2	2	2
5	C	3	2	3	3	2	1	1	2	3	2
6	C	3	2	2	3	3	2	2	2	3	3
7	T	3	3	2	2	3	3	3	2	3	2
8	T	3	3	3	2	3	3	3	2	2	2
9	T	3	3	3	3	3	3	3	2	2	1

Fig. 4.1 Dynamic programming matrix D' for $x = \texttt{CGAAAGTAT}$ and $y = \texttt{CAAACCTTT}$ with $\ell = 3$. Each cell contains the minimum number of mismatches between factors ending at that position.

4.3 Algorithms

4.3.1 MaxShiftM

We implemented Crochemore's algorithm [27] for performing FLASM under the Hamming distance model. The pseudocode for this is presented below in Algorithm 5. Let $\mathsf{D}'[0..m][0..n]$ be a matrix, where $\mathsf{D}'[i][j]$ contains the Hamming distance between some factor $x[\max\{0, j-\ell\}..j-1]$ of a string x and factor $y[\max\{0, i-\ell\}..i-1]$ of string y, for all $1 \leq j \leq n$, $1 \leq i \leq m$. The naïve way to obtaining this matrix is through a straightforward $\mathcal{O}(m\ell n)$-time algorithm by constructing matrices $\mathsf{D}^s[0..\ell][0..n]$, for all $0 \leq s \leq m-\ell$, where $\mathsf{D}^s[i][j]$ is the Hamming distance between some factor of $x[j-\ell..j-1]$ and the prefix of length i of $y[s..s+\ell-1]$. We obtain D' by collating D^0 and the last row of D^s, for all $0 \leq s \leq m-\ell$. Matrix D^s can be obtained using the standard dynamic programming algorithm. We say that $x[\max\{0, i-\ell\}..i-1]$ occurs in y ending at $y[j-1]$ with k mismatches *iff* $\mathsf{D}'[i][j] \leq k$, for all $1 \leq i \leq m$, $1 \leq j \leq n$.

The results of the algorithm are shown in Figure 4.1 and finds $\mathsf{D}'[5][4] = 0$, since $\delta_H(x[1..3], t[2..4]) = 0$. The algorithm also finds $\mathsf{D}'[9][9] = 1$, since $\delta_H(x[6..8], t[6..8]) = 1$.

We can improve the performance, both in terms of time and space, reducing the space and time complexity to $\mathcal{O}(m\lceil\frac{\ell}{w}\rceil n)$, by encoding D^s as a matrix of bit vectors.

		0	1	2	3	4	5	6	7	8	9
		ϵ	C	G	A	A	A	G	T	A	T
0	ϵ	000	000	000	000	000	000	000	000	000	000
1	C	001	000	001	001	001	001	001	001	001	001
2	A	011	011	001	010	010	010	011	011	010	011
3	A	111	111	111	010	100	100	101	111	110	101
4	A	111	111	111	110	100	000	001	011	110	101
5	C	111	110	111	111	101	001	001	011	111	101
6	C	111	110	101	111	111	011	011	011	111	111
7	T	111	111	101	011	111	111	111	110	111	110
8	T	111	111	111	011	111	111	111	110	101	110
9	T	111	111	111	111	111	111	111	110	101	010

Fig. 4.2 Dynamic programming matrix B for $x = \texttt{CGAAAGTAT}$ and $y = \texttt{CAAACCTTT}$ with $\ell = 3$. Each cell contains a bit vector with mismatches between factors encoded as 1s at that ending position.

Let $\mathsf{B}[i][j] = b_{\ell-1}, b_{\ell-2}, \ldots, b_0$, a bit vector matrix holding the binary encoding for obtaining the Hamming distance between a factor of the text $x[j-\ell..j-1]$ and $y[i-\ell..i-1]$ with a total of $\mathsf{D}'[i][j]$ mismatches. The maintenance of matrix B is done via operations defined as follows:

- `shift`(v): an operation that, given a bit-vector v, shifts the bits of v one position to the left, and returns the resulting bit-vector.
- `shiftc`(v): same as `shift`, but also truncates the leftmost bit of v past ℓ bits.
- `pop-count`(v): an operation that, given a bit-vector v, returns the number of 1s in v.

The operations encode matches and mismatches onto each bit vector in matrix B with the position of 1s representing the position of base-mismatches between the factors of length ℓ ending at $x[j-1]$ and $y[i-1]$. The matrix is represented in Figure 4.2. The algorithm finds the bit vector at $\mathsf{B}[9][9] = \texttt{010}$, with 1 at the position of a mismatch as realised from comparing $x[6..8] = \texttt{TAT}$ and $y[6..8] = \texttt{TTT}$. It must be noted however that if $j < \ell$ or $i < \ell$, that is $|x[j-\ell..j-1]| < \ell$ or $|y[i-\ell..i-1]| < \ell$, then the bit vector set bits and Hamming distance reported in the cells of the respective matrices are not always representative of the true mismatch positions or distance. In FLASM, this does not pose a problem because we only report the position of the best match if both $j \geq \ell$ and $i \geq \ell$, that is, we report a factor of length ℓ and no shorter.

We now detail the implementation and functional behaviour of the algorithm's components, but we make two preliminary definitions before we progress — let v be a word vector used to represent a cell in B; and let $s = \lceil \ell / w \rceil$ be the size of v in computer words.

The `mw-shift` function (Algorithm 2) is a generalised implementation of the `left-shift` ($\ll$) bitwise operation that runs on a word vector. It starts from the first word $v[0]$ and performs a bitwise left-shift on each word up to $v[s-1]$ while checking to see if it needs to carry a bit in c. To check if we need to carry a bit, we first copy $v[i]$ into temporary variable t and use bit mask a, which is assigned a word with the most numerically-significant bit set to 1. We then bitwise AND t with a; if a bit is set in the resulting word ($(t \ \& \ a) > 0$) then we need to carry a bit to the next word in the word vector, so we set $c = 1$, otherwise $c = 0$. In the procedure, c is bitwise ORed with $v[i] \ll 1$ so the bit is carried if needed and the result is assigned back to $v[i]$. The function requires time $\mathcal{O}(s)$ and space $\mathcal{O}(1)$ as only three extra words are needed.

Algorithm 2 mw-shift function performs the left-shift operation on a word vector.
v: the word vector stored in a cell of matrix B.
s: the number of computer words required to store v.
w: the size of a computer word (typically 64 bits).

```
1:  procedure MW-SHIFT(v, s, w)
2:      a ← 1 ≪ (w − 1)
3:      c ← 0
4:      for i ∈ {0..s − 1} do
5:          t ← v[i]
6:          v[i] ← (v[i] ≪ 1) | c
7:          if (t & a) > 0 then
8:              c ← 1                                  ▷ carry a bit
9:          else
10:             c ← 0                                  ▷ don't carry
11:     return v
```

An example of this function is shown below in Example 1.

Example 1: mw-shift for $\ell = 32$ and $w = 8$

v_3	v_2	v_1	v_0
10000011	10000010	10011101	01111111
00000111	00000101	00111010	11111110

The `mw-shiftc` function (Algorithm 3) is an implementation of operation `shiftc` on multiple words and is almost identical to the `mw-shift` function, except that it

makes sure that no more than ℓ bits are counted in v. The function uses a bit-mask (variable y below) which is calculated based on ℓ and w to keep only the necessary set bits in the last computer word of v so no extra errors are counted when calling mw-popcount. It requires time $\mathcal{O}(s)$ as it relies on `mw-shift` and the other actions take constant time.

Algorithm 3 The mw-shiftc function: perform mw-shift and clear bits past ℓ positions.
v: the word vector stored in a cell of matrix B.
s: the number of computer words required to store v.
w: the size of a computer word (typically 64 bits).
ℓ: the length of a factor of pattern x.

```
1: procedure MW-SHIFTC(v, s, w, ℓ)
2:     v ← mw-shift(v, s, w)
3:     j ← ℓ mod w
4:     y ← j 1s
5:     v[s − 1] ← v[s − 1] & y
6:     return v
```

An example of this function is shown below in Example 2.

Example 2: mw-shiftc for $\ell = 20$, $w = 8$, $y =$ `1111`

Operation	v_2	v_1	v_0
`mw-shift`	00001011	10011101	01111111
$v[s-1]$ `&` y	00010111	00111010	11111110
Result	00000111	00111010	11111110

The `mw-popcount` function (Algorithm 4) is an implementation of operation `pop-count` on multiple words. It returns the total count of set bits in all the elements of v. It counts the errors in an alignment since the 1s represent errors. Function `mw-popcount` requires time $\mathcal{O}(s)$ as operation `pop-count` runs in constant time for each word.

Algorithm 4 The mw-popcount function: returns total set bits in a word vector.
v: the word vector stored in a cell of matrix B.
s: the number of computer words required to store v.

```
1: procedure MW-POPCOUNT(v, s)
2:     c ← 0
3:     for i ∈ {0..s − 1} do
4:         c ← c + pop-count(v[i])
5:     return c
```

An example of this function is shown below.

Example 3: mw-popcount for $\ell = 16$ and $w = 8$		
$\mathsf{D}'[i,j]$	v_1	v_0
8	10011010	00111001

We are now ready to present the MaxShiftM algorithm (Algorithm 5) that populates matrices D' and B.

Algorithm 5 The MaxShiftM Algorithm.
x: the pattern string.
m: the length of the pattern string.
y: the text string.
n: the length of the text string.
ℓ: the length of a factor of x.
k: the maximum distance threshold.

```
1:  procedure MaxShiftM(x, m, y, n, ℓ, k)
2:      D'[0..m][0..n] ← 0
3:      B[0..m][0..n] ← 0
4:      for i ∈ {1..m} do
5:          B[i][0] ← min(i, ℓ) 1s
6:          for j ∈ {1..n} do
7:              B[i][j] ← mw-shiftc(B[i−1][j−1]) | δ_H(x[i−1], y[j−1])
8:              D'[i][j] ← mw-popcount(B[i][j])
9:              if j ≥ ℓ and i ≥ ℓ and D'[i][j] ≤ k then
10:                 Report match found <i, j, e>
```

Assuming $\ell \leq w$, the algorithm can run in time $\mathcal{O}(mn)$ because $\lceil \ell/w \rceil = 1$, otherwise it runs in time $\mathcal{O}(m\lceil \ell/w \rceil n)$. The algorithm's space complexity can be reduced to only $\mathcal{O}(m\lceil \ell/w \rceil)$ since each row of D' only depends on the immediately preceding row. We need only store the cell or ending-position of the factor in x having the minimum distance from a factor in y. Also, we only need to report matches found having a mismatch less than or equal to k, the distance threshold of our choosing (satisfying $k < \ell$). We report $<i,j,e>$, where i is the ending position of the match in pattern x, j is the ending position of the match in the text y, and e is the number of mismatches ($e \leq k$).

4.3.2 Myers' Algorithm for FLASM

To perform FLASM under the edit distance model we used the implementation of Myers' bit-vector algorithm [95] in SeqAn [32], a free open-source C++ library of algorithms for sequence analysis. Specifically, we adapted it to search for all factors of pattern x of length ℓ in text y in order to solve the FLASM problem.

Given a pattern x of length m, a text y of length $n \geq m$, and an integer $k < m$, Myers bit-vector algorithm solves the ASM problem under the edit distance model in time $\mathcal{O}(n\lceil\frac{m}{w}\rceil)$ and space $\mathcal{O}(\lceil\frac{m}{w}\rceil)$ [95]. Applying this algorithm for all $\mathcal{O}(m)$ factors of length ℓ of x separately solves the FLASM problem under the edit distance model in time $\mathcal{O}(n\lceil\frac{\ell}{w}\rceil m)$ while the space requirement remains the same. Simply put, we call Myers' function on every factor $x[i..i+\ell-1]$ for $i = 0..m-\ell$ and assuming ℓ is small and m is large, this means we call it $\mathcal{O}(m)$ times. In the psuedocode below (Algorithm 6), we show how we apply Myers' search algorithm to solve the FLASM problem, by calling the function MyersFLASM $\mathcal{O}(m)$ times and reporting only matches with an error e less than or equal to our chosen threshold, k.

Algorithm 6 Applying Myers' Search Algorithm to solve the FLASM problem.
x: the pattern string.
m: the length of the pattern string.
y: the text string.
n: the length of the text string.
ℓ: the length of a factor of x.
k: the maximum distance threshold.

```
1: procedure MyersFLASM(x, m, y, n, ℓ, k)
2:     for i ∈ {0..m − ℓ} do
3:         MyersSearch(x[i..i + ℓ − 1], ℓ, y, n, k)
4:         Report matches found <i, j, e>
```

4.3.3 libFLASM

We created libFLASM, a free open-source C++ software library for solving FLASM under both the edit and the Hamming distance models. Our library is able to handle factor lengths of any length; in particular, factor lengths greater than the computer word size. The library implementation is distributed under the *GNU General Public License* (GPL), and it is freely available for download at repository https://github.com/webmasterar/libFLASM.

libFLASM exposes two functions: one to solve FLASM under the edit distance model; and one for the Hamming distance model. Both functions require the following parameters to operate:

t The text to search

n The length of the text

x The pattern text

m The length of the pattern

ℓ The length of the factor

k The maximum distance allowed between a factor of x and a factor of t

r A flag to indicate if all or the best matches should be returned

The functions return a set of tuples in the form $<i, j, e>$, where:

i is the ending position of the match in x

j is the ending position of the match in t

e is the error/distance of the match

We then used the library to show how it can solve various problems:

a) We incorporate libFLASM into a state-of-the-art tool to improve the accuracy of MCSA in terms of the inferred likelihood-based phylogenies;

b) We incorporate libFLASM into a state-of-the-art tool for ME of patterns longer than what was previously possible;

c) We show how libFLASM can be used efficiently for implementing the Chang and Marr index;

d) We show how libFLASM can be used efficiently for ACSM with high error ratios.

4.3.4 Incorporation of libFLASM into BEAR

libFLASM was incorporated into BEAR (BEst Aligned Rotations) [8], a state-of-the-art tool for improving Multiple Circular Sequence Alignment (MCSA). BEAR uses the library to find most similar factors under a pre-specified distance model, between pairs of the input sequences, which can then determine suitable rotations for all input sequences. The similar factors identified can be used as anchors by which to rotate one string around the other so that they are repositioned or rotated to their linearised form in such a way as to reduce the distance between them. In other words, one string is rotated relative to the other so their best-matching factors line up. The output of BEAR after it has computed all pairs of rotations can then be used as input for any Multiple Sequence Alignment (MSA) program.

BEAR takes a MultiFASTA file containing a set of N sequences $s_1, s_2, \ldots, s_N$ of roughly equal length (m) as input, performs all pairwise sequence comparisons, and stores the results in a matrix M of size $N \times N$. We use FLASM to perform the comparison for every pair (s_i, s_j), with the best factor of s_j of length ℓ being discovered in $s'_i = s_i[0..m-1]s_i[0..\ell-1]$; From the best match we obtain the distance/error and calculate the rotation (e, r) of every pairwise sequence comparison (s_i, s_j) and store it in each cell $M[i][j]$. Although setting the factor length $\ell = m$ solves exactly the approximate circular string matching problem, in practice ℓ can be set to a much smaller value and still give accurate results while taking far less time. This whole process takes time $\mathcal{O}(N^2m^2\lceil\ell/w\rceil)$ which is faster in practice than the competing algorithm employed in Cyclope [92] which requires time $\mathcal{O}(N^2m^3)$.

This matrix M is then used as input for standard agglomerative hierarchical clustering [133] in order to group closely-related sequences together and apply the rotations to output a *refined* dataset of the N sequences. This refined dataset can in turn be passed to an MSA program such as MUSCLE [33] or Clustal Ω [127] to produce the final MSA. This concludes the MCSA pipeline.

BEAR provides a selection of algorithms to do the pairwise sequence comparison and one of these approaches is based on FLASM. This was previously restricted to using only factors of length $\ell \leq w$ under the Hamming distance model, but with the incorporation of libFLASM, it is now possible to use arbitrary values for ℓ under either the edit or the Hamming distance model.

4.3.5 Incorporation of libFLASM into MoTeX-II

libFLASM was incorporated into MoTeX-II [108] (the successor of MoTeX [111]), a state-of-the-art tool to identify single and structured motifs. MoTeX-II uses the library to find occurrences, under a pre-specified distance model, of each factor of length ℓ of every sequence in the input sequences.

MoTeX-II takes a MultiFASTA file containing a set of N sequences $s_1, s_2, \ldots, s_N$ as input. For single ME, it performs all pairwise sequence comparisons and stores the number of occurrences for each factor of length ℓ of every sequence. A similar approach is followed for structured ME. Hence it can determine all valid motifs of length ℓ. This was previously restricted to finding motifs of length $\ell \leq w$, but with the incorporation of libFLASM, it is now possible to find motifs of any length.

4.3.6 Using libFLASM to implement the Chang and Marr Index

The libFLASM library software repository contains an example program showing the use of libFLASM to implement the Chang and Marr index, that is represented by an integer array D of size σ^q. Given a pattern x of length m and a value q, the index can be implemented under the Hamming or edit distance model.

A *de Bruijn* sequence of order q on an alphabet Σ of size $\sigma = |\Sigma|$ is a cyclic sequence of characters containing all combinations of q-long substrings (q-grams) without repeats [1, 31]. Each position $i = 0..\sigma^q - 1$ in the sequence represents a unique q-gram. Consider an alphabet $\Sigma = \{0,1\}$ and a q-gram of size $q = 3$, we can create a linearised and extended de Bruijn sequence denoted by function $B(\sigma, q)$ of length $\sigma^q + q - 1$: $B(2,3) = \texttt{0001011100}$. The q-grams of this string are: `000`, `001`, `010`, `101`, `011`, `111`, `110` and `100`.

The minimal distance between each q-gram in the linearised version of the de Bruijn sequence and any factor of x can be computed using libFLASM by setting the factor length to $\ell := q$. We create the Chang and Marr integer array D of size $\mathcal{O}(\sigma^q)$ to store the minimum distances between any q-gram s in the de Bruijn sequence and any factor of x. By using a numerical representation of s, we can quickly and easily populate array D. With libFLASM, we can produce a list of tuples in the form $< i, e >$, where i is the ending position of a q-gram s in the de Bruijn sequence and e is the minimal distance found between s and any factor in x. This is used to update D, producing the Chang and Marr index in time $\mathcal{O}(\sigma^q \lceil q/w \rceil m)$ and space $\mathcal{O}(\sigma^q + \lceil q/w \rceil m)$.

4.3.7 Using libFLASM for performing Approximate Circular String Matching

The libFLASM library software repository comes with another example program that can be used to perform FLASM under a pre-specified distance model. There was no need to modify this program to make it suitable for ACSM. All that is needed is to double-up the pattern x of length m, by concatenating it with itself to create string $x' = x[0..m-1]x[0..m-2]$, and then set the factor length to $\ell := m$.

4.4 Experiments

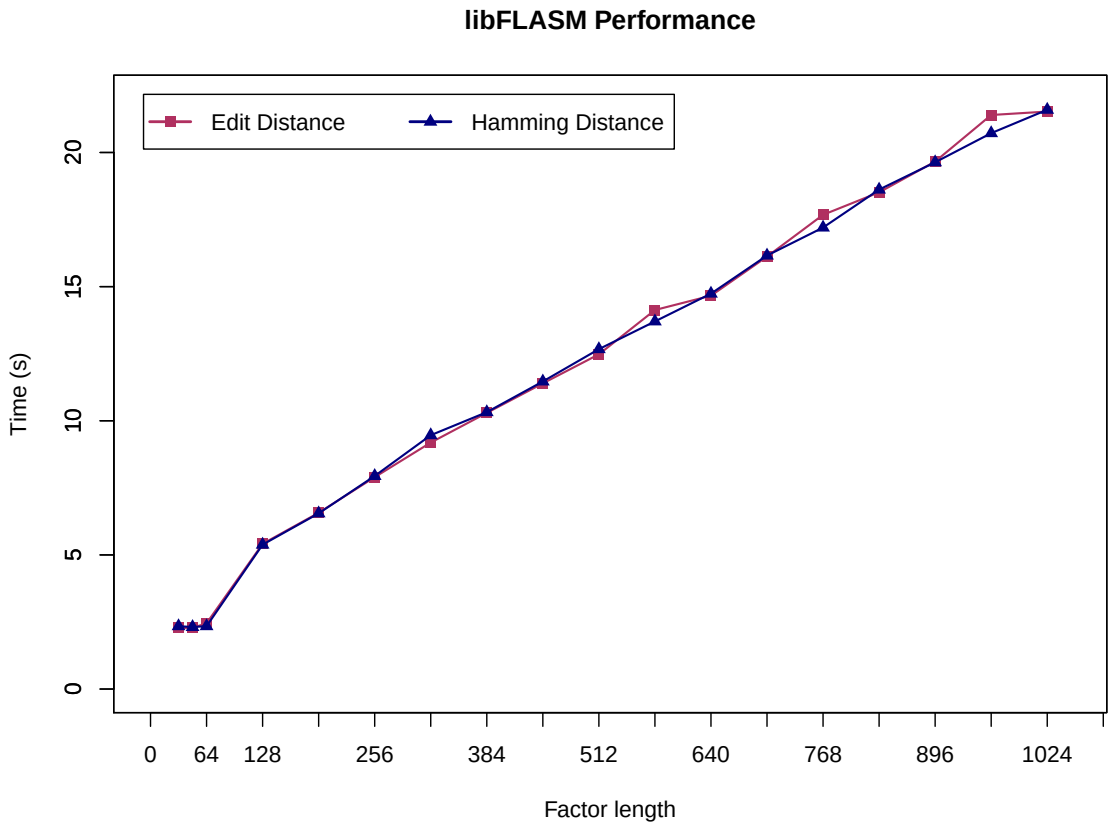
The experiments were run on a 64-bit GNU/Linux operating system on a machine with a quad-core 64-bit Intel™ 2.8GHz Core-I7 processor with 8GB of RAM. libFLASM, our example programs, Cyclope [92], CMFN [42] and ACB [59] were compiled with gcc v.4.7.3 using optimisation flags `-O3`, `-funroll-loops` and `-msse4.2`.

4.4.1 Experiment I: Performance

We first evaluated the performance of libFLASM by generating two uniformly pseudo-random $10,000$ base-long DNA sequences for the text t and the pattern x. Different factor lengths ℓ were used in the range $32, \ldots, 1024$. The distance threshold k was set to $\frac{1}{2}\ell$, while the length n of the text, the length m of the pattern, and the word size $w = 64$ remained constant. The size of the text and pattern would not play any role here because they remain constant, but we expected the time taken by the algorithm to grow relative to the factor $\lceil \frac{\ell}{w} \rceil$.

The results shown in Figure 4.3 confirm the theoretical findings; they show linear growth of the time required to complete the computation with respect to the factor length ℓ in accordance with the time complexity of $\mathcal{O}(m\lceil \frac{\ell}{w} \rceil n)$. Note also that the first three points, which use factors of length 32, 48, and 64 are represented on the graph as a straight horizontal line, indicating that there is no increase in the time required to compute them since they require the same number of computer words to be computed ($\lceil \frac{\ell}{w} \rceil = 1$ when $\ell \leq 64$). The performance shown in Figure 4.3 is trivially explained by the $\lceil \frac{\ell}{w} \rceil$ factor in the time complexity $\mathcal{O}(m\lceil \frac{\ell}{w} \rceil n)$ since m, n and w remain constant throughout the experiment.

Fig. 4.3 Elapsed time of libFLASM under edit and Hamming distance models with $n = m = 10,000$.



4.4.2 Experiment II: Multiple Circular Sequence Alignment

Nine synthetic datasets were generated using INDELible [41] in order to test the accuracy of BEAR with libFLASM for improving MCSA. INDELible is a program which, given a starting sequence, generates new DNA or protein sequences, simulating biological sequence evolution with substitutions, insertions, and deletions at rates defined by the user. It suuports many different substitution models of DNA evolution available to the user and we chose the popular model JC69 [67]. This model assumes uniform mutation rates where the rate of change from one base to any of the other three nucleotides occurs with equal likelihood. Whilst this is an oversimplification of the sequence mutation in the real world, it does a good job of modelling sequences with few mismatches between them [73] and INDELible only requires the user to specify the expected substitution, insertion and deletion rates to generate the sequences.

We started with one uniformly psuedo-random generated synthetic DNA sequence of length $2,500$. We created three files containing 12, 25, or 50 sequences each (number denoted by α) and used INDELible to simulate their molecular evolution with three unique substitution rates 5%, 20%, and 35% (denoted by θ) applied to each dataset seperately. The insertion and deletion rates were set, respectively, to 4% and 6% (denoted by κ and ω), relative to a substitution rate of 1. This resulted in 9 datasets being created in total. We call these datasets the *Original* datasets.

We then proceeded to randomly rotate each of the sequences in the datasets to create a new set of files. We call these the *Random* datasets. The goal of this experiment was to use BEAR with libFLASM under the edit distance model to refine the random rotation of each of the sequences in the *Random* datasets. The refined datasets we would obtain after rotating the sequences are called the *Restored* datasets. We ran BEAR using the FLASM method for pairwise sequence comparisons under the edit distance model. We used two combinations of factor length ℓ and distance threshold k to run the experiments: $\ell = 40$, $k = 10$; and $\ell = 100$, $k = 45$.

We then used MUSCLE [33], a fast and accurate MSA program, to produce the alignments in *PHYLIP* format for each dataset. This completed the MCSA pipeline.

Next, we had to ascertain if the *restored* MCSA alignments were accurate when compared to the *original* datasets. The PHYLIP files were then passed to RAxML [134], a program for heuristically inferring a *phylogenetic tree*, under the *Maximum Likelihood* [38] approach. This approach considers the statistical likelihood of every nucleotide substitution in an aligned set of sequences and builds a phylogenetic tree putting similar sequences closer together. RAxML was used again to compare the trees against each other via calculating the pairwise Robinson and Foulds (RF) distance [121]. The RF distance is a measure indicating how many changes to an unrooted tree's branches are required to make it match another and an RF-distance of 0 indicates the trees are identical. In particular we calculated the RF distance between the *Original* trees and the *Random* trees, as well as the distance between the *Original* trees and the *Restored* ones, to measure how well the programs had performed refining the sequences in each of the datasets.

The results in Table 4.1 show the RF distances between the *Original* datasets and the *Random* datasets in column one; the distance between the *Original* datasets and the *Restored* ones using Cyclope [92], another tool for MCSA, in column two; and the distance between the *Original* datasets and the *Restored* ones using BEAR in columns three and four. BEAR was run under the edit distance model with $\ell = 40$ and $k = 10$, shown in column three and again with $\ell = 100$ and $k = 45$, shown in column four.

Table 4.1 shows that BEAR produces very good results: an RF distance of 0 was obtained for all datasets when using $\ell = 100$. Table 4.1 shows the necessity of using a factor length larger than the word size, as it produces more accurate results than when using $\ell = 40$. This is expected as longer factors are more likely to provide information about reliable rotations than shorter factors, which could potentially have multiple occurrences with at most k differences. The Elapsed-time comparisons are shown in Table 4.2. The results show that BEAR performs significantly faster than Cyclope. Ultimately, these results demonstrate that libFLASM can be applied effectively and efficiently in BEAR for improving MCSA.

Table 4.1 The RF distance between the *Original* and *Random* datasets as well as the RF distance between the *Original* and *Restored* datasets using Cyclope and BEAR under the edit distance model.

Dataset $<\alpha,\gamma,\theta,\kappa,\omega>$	*Random*	Cyclope	BEAR $\ell = 40$	BEAR $\ell = 100$
<12,2500,0.05,0.06,0.04>	0	0	0	0
<12,2500,0.20,0.06,0.04>	0	0	0	0
<12,2500,0.35,0.06,0.04>	0	0	0	0
<25,2500,0.05,0.06,0.04>	0	0	0	0
<25,2500,0.20,0.06,0.04>	0	0	0	0
<25,2500,0.35,0.06,0.04>	0.045	0.045	0	0
<50,2500,0.05,0.06,0.04>	0.085	0	0.021	0
<50,2500,0.20,0.06,0.04>	0.043	0	0	0
<50,2500,0.35,0.06,0.04>	0.043	0	0.021	0

Table 4.2 Elapsed-time comparison in seconds of Cyclope and BEAR.

Dataset $<\alpha,\gamma,\theta,\kappa,\omega>$	Cyclope	BEAR $\ell = 40$	BEAR $\ell = 100$
<12,2500,0.05,0.06,0.04>	79.09	15.92	46.53
<12,2500,0.20,0.06,0.04>	77.47	15.06	44.52
<12,2500,0.35,0.06,0.04>	76.76	14.85	45.44
<25,2500,0.05,0.06,0.04>	332.69	69.81	203.78
<25,2500,0.20,0.06,0.04>	342.94	69.28	208.85
<25,2500,0.35,0.06,0.04>	344.50	71.14	208.82
<50,2500,0.05,0.06,0.04>	1,317.81	293.45	851.07
<50,2500,0.20,0.06,0.04>	1,303.51	300.37	837.66
<50,2500,0.35,0.06,0.04>	1,359.90	286.88	854.83

4.4.3 Experiment III: Motif Extraction

We carried out experiments on real data retrieved from the Kyoto Encyclopedia of Genes and Genomes (KEGG) Database [68]. Three sets of data were obtained made up of: RNA polymerase proteins; viruses; and hypothetical proteins. Single ME was carried out on these datasets. All datasets contain 11 sequences: the first dataset is made up of RNA Polymerase sequences, with all sequences containing 4 distinct matching motifs of varying length; the second dataset is made up of virus sequences, all containing 4 matching motifs; the third dataset is made up of hypothetical proteins, containing 2 distinct matching motifs.

Table 4.3 shows the results obtained when parameters of the form $<\ell, k>$ were used to extract the single motifs from the datasets. The quorum shows the percentage of sequences which contained the listed motif. A quorum of 100% shows that for each set of input sequences, all sequences contained the same single motif, as expected. Table 4.3 shows that MoTeX-II was able to identify all motifs of various lengths when used with libFLASM. These real datasets show the necessity of using a factor length larger than the word size for ME.

Table 4.3 Results for single motif extraction from real datasets.

Dataset	Parameters	Motif	Quorum (%)
RNA Polymerase	$<350, 110>$	RNA polymerase Rpb2, domain 6	100
	$<60, 28>$	RNA polymerase Rpb2, domain 4	100
	$<40, 12>$	RNA polymerase Rpb2, domain 5	100
	$<90, 30>$	RNA polymerase Rpb2, domain 7	100
Viruses	$<350, 150>$	Viral methyltransferase	100
	$<130, 50>$	Cucumber mosaic virus 1a protein	100
	$<70, 48>$	Cucumber mosaic virus 1a protein C terminal	100
	$<250, 130>$	Viral (Superfamily 1) RNA helicase	100
Hypothetical Proteins	$<130, 45>$	Type III restriction enzyme, res subunit	100
	$<60, 30>$	Helicase conserved C-terminal domain	100

Synthetic data was also used to extract structured motifs from sequences. We generated 50 random $1,000$ base-long DNA sequences. Structured motifs were implanted in half of the DNA sequences. Table 4.4 shows that the incorporation of libFLASM into MoTeX-II has allowed for structured motifs with lengths $\ell_1, \ldots, \ell_\beta > w$ to be extracted. The parameters are in the form $<(\ell_1, k_1)[d_{\min_1}, d_{\max_1}](\ell_2, k_2)[d_{\min_2}, d_{\max_2}](\ell_3, k_3)>$ where $[d_{\min_i}, d_{\max_i}]$ represents the range of the distance interval allowed between each motif box. In each test, 25 structured motifs were implanted into the sequences and MoTeX-II was able to identify all of these structured motifs in all cases.

Table 4.4 Results for structured motif extraction from synthetic datasets.

Parameters	No. structured motifs implanted/extracted
$<(80,15)[5,15](60,10)[5,20](230,20)>$	25/25
$<(100,15)[5,15](80,10)[5,20](250,20)>$	25/25
$<(120,15)[5,15](100,10)[5,20](270,20)>$	25/25
$<(140,15)[5,15](120,10)[5,20](290,20)>$	25/25
$<(160,15)[5,15](140,10)[5,20](310,20)>$	25/25
$<(180,15)[5,15](160,10)[5,20](330,20)>$	25/25
$<(200,15)[5,15](180,10)[5,20](350,20)>$	25/25

4.4.4 Experiment IV: Chang and Marr Index

We carried out experiments on synthetic data to test the implementation of the Chang and Marr index under the Hamming and edit distance models when using libFLASM in comparison to a naïve implementation. Table 4.5 shows the elapsed time to compute the Chang and Marr index for q-grams of lengths 5 to 10 under the Hamming and edit distance model, using a pattern of length 32. Table 4.6 shows likewise for a pattern of length 64. Both patterns were generated randomly (uniform distribution) over the DNA alphabet.

The time taken to compute the Chang and Marr index using a naïve implementation for the Hamming and edit distance can be seen under the columns titled *Naïve*. The time taken to compute it using the libFLASM library can be seen under the columns titled *libFLASM*. It is evident that using libFLASM significantly decreases the time taken to compute the Chang and Marr index. This can clearly be seen as the value of q increases, where we obtain a speedup by more than an order of magnitude. libFLASM's ability to calculate the Chang and Marr index in time $\mathcal{O}(\sigma^q\lceil q/w\rceil m)$ presents a clear speed advantage for algorithms that could make use of it.

Table 4.5 Elapsed-time comparison in seconds for implementing the Chang and Marr index using a pattern of length 32.

	Edit Distance		Hamming Distance	
q-gram length	Naïve (s)	libFLASM (s)	Naïve (s)	libFLASM (s)
5	0.01	0.00	0.01	0.00
6	0.08	0.02	0.08	0.01
7	0.67	0.90	0.55	0.05
8	6.20	0.50	4.81	0.25
9	34.00	2.74	23.99	1.43
10	145.56	11.76	96.71	6.24

Table 4.6 Elapsed-time comparison in seconds for implementing the Chang and Marr index using a pattern of length 64.

	Edit Distance		Hamming Distance	
q-gram length	Naïve (s)	libFLASM (s)	Naïve (s)	libFLASM (s)
5	0.04	0.01	0.04	0.00
6	0.23	0.03	0.22	0.02
7	1.45	0.15	1.31	0.09
8	10.76	0.82	9.27	0.46
9	95.01	5.29	76.21	2.76
10	673.17	24.51	520.12	12.51

4.4.5 Experiment V: Approximate Circular String Matching

We then evaluated the performance of libFLASM against state-of-the-art algorithms for solving the ACSM problem. Two fast average-case algorithms, both of which support the edit and Hamming distance models, were identified from the literature: CMFN [42] and ACB [59]. The corresponding implementation of the algorithms were obtained via communication with the authors.

CMFN [42] is an average-optimal algorithm designed to solve the general problem of multiple approximate string searching—that is, searching for a set of N patterns $x_1, x_2, \ldots, x_N$ simultaneously in the text. It works by sliding a fixed-sized window from left to right along the text and reading backwards enough q-grams before determining whether verification of the window is required or the window can be moved forward. If enough q-grams are read and the accumulated distance is less than the error threshold a relatively slow verification step is performed to determine if one or more matches exist. This algorithm is sensitive to the the length of the q-grams and the error threshold as well. The average complexity of the algorithm is $\mathcal{O}((k+\log_\alpha(rm))n/m)$ for $\alpha < 1/2 - \mathcal{O}(1/\sqrt{\sigma})$, where k is the error threshold, m is the length of the patterns, α is the difference ratio k/m, r is the number of patterns in the set, n is the size of the text being searched and σ is the size of the alphabet.

ACB [59] is an algorithm designed with the purpose of solving the ACSM problem and is based on the CBNDM algorithm introduced by Chen et al. [21], a simplified version of BNDM [97]. It is a bit-parallel algorithm that simulates a nondeterministic finite automaton that has been modified to recognise circular patterns and their factors backwards. As with CMFN, it also matches the pattern backwards over a window of the text and skips alignments where a match is not possible using q-grams. It does a verification step similar to the CMFN algorithm as well. It therefore acheives average case sublinear performance when the error threshold k is bounded such that

$k < m/(log_\sigma m + \mathcal{O}(1))$, acheiving a search time complexity of $\mathcal{O}((n/m)k \log_\sigma m)$ when $n \gg m$. The algorithm is sensitive to the value of the error threshold k.

To compare the speed of the three programs we adapted CMFN to search for all rotations of x in order to solve the ACSM problem. This was done by considering all rotations of x as the set of input patterns. Since ACB is designed to solve the ACSM problem we did not need to change anything for it. Unfortunately, the implementation of ACB is restricted to searching patterns of length less than or equal to the computer word size ($m \leq w$).

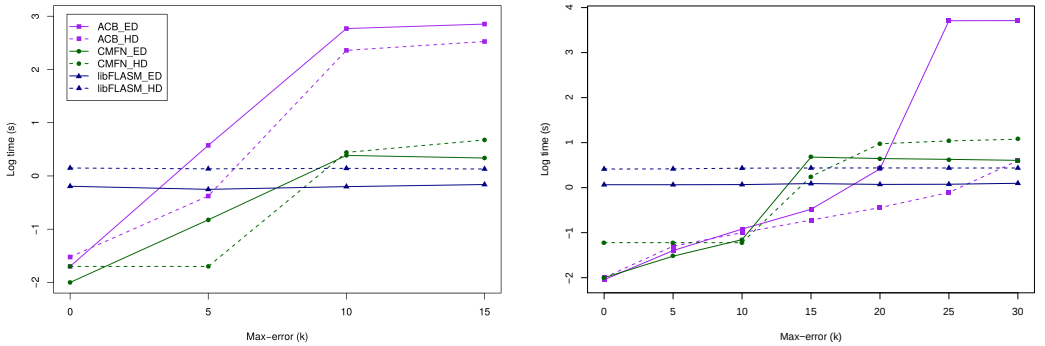
We ran the three programs with the following settings. ACB was run with the option `-k` to set the distance threshold and the list of patterns were read in on the console from a file. CMFN was run with the options: `-D -B -Sb -t 6 -k` k. This uses a q-gram length of 6, which is noted for its suitability in [42]. Hamming distance was enabled using the option `-s`. To test libFLASM, we used the example application that is packaged with the library. The distance model, the factor length ℓ, and the distance threshold k options were set accordingly.

A one million base-long DNA sequence was pseudo-randomly generated (with uniform distribution) as the text t to be searched. Patterns of length $m = 32, 64, 128, 256$ were randomly picked from t. In the case of libFLASM, each pattern x was concatenated with itself to make $x' = x[0..m-1]x[0..m-2]$ and the factor length was set to $\ell := m$. With ACB, only 32 and 64 base-long patterns were tested because of the limitations of its implementation. Both the Hamming and edit distance versions of the programs were tested. The other parameter considered was the performance of the programs with regards to the distance threshold k. A series of values for k from the range $0, \ldots, \frac{1}{2}\ell$ was used. The results are shown in Figure 4.4.

The results of this experiment show that ACB and CMFN are fast for small values of k, but once k is increased we find that libFLASM becomes competitive and goes on to perform significantly better. This is explained by the fact that the time complexity of ACB and CMFN depends on k, while libFLASM is independent of k. This is clearly captured by the graphs in Figure 4.4.

It can be observed from the graphs that the lines for libFLASM are mostly horizontal and that is because the size of the factor ($\ell = m$) does not change during the course of each experiment. Another observation is that the time taken increases slightly with increasing error-threshold under the Edit distance model and this is attributed to SeqAn's implementation of Myer's algorithm being k-dependent, although the effect is not so large.

(a) $m = 32$ (b) $m = 64$



(c) $m = 128$ (d) $m = 256$

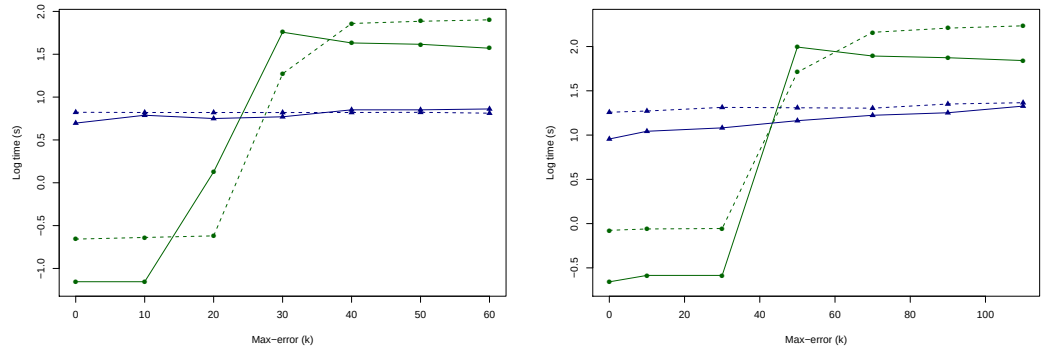


Fig. 4.4 Elapsed time in $\log_{10}$ seconds of the different programs using different pattern length m and increasing distance threshold k.

Apart from its speed and robustness, another benefit of using libFLASM, not shown here, is alphabet independence.

4.5 Conclusion

Approximate string matching is a core computational task in molecular biology and has been extensively studied over the past decades. Fixed-length approximate string matching is a generalisation of this problem that has beneficial applications in biology and beyond.

In this thesis we have presented full details for a generalised implementation of a Hamming distance-based algorithm, MaxShiftM, and have also shown how clever

application of Myers' fast approximate string matching algorithm under the edit distance model can be applied to solve the fixed-length approximate string matching problem. Having incorporated both these algorithms into a software library, libFLASM, we presented various examples of applications of FLASM. We also compared the performance of the library against competing algorithms ACB and CMFN.

Both algorithms use bit-parallel techniques applied through a dynamic programming matrix to reduce the amount of computation performed compared to a naïve approach. This has helped reduce the worst-case time and space requirement to $\mathcal{O}(n\lceil \ell/w \rceil m)$ and $\mathcal{O}(m\lceil \ell/w \rceil)$, respectively. This complexity is not dependent of the alphabet size σ, nor is it restricted by the number of errors/mismatches k that are allowed, thus making the algorithm robust for general purpose use under potentially high error conditions. Also, the ability to choose a longer factor size allows for higher accuracy in cases where factors of length $\ell \leq w$ are unsuitable or inadequate.

In this thesis we have shown how FLASM can be incorporated into existing software or applied as a tool for solving the following problems:

- libFLASM was incorporated into the state-of-the-art Bioinformatics tool BEAR to perform Multiple Circular Sequence Alignment. This was shown to be fast and increasingly accurate with higher factor lengths.
- libFLASM was incorporated into the state-of-the-art Bioinformatics tool MoTeX-II to perform simple and structured motif extraction and was shown to perform with perfect recall.
- The library was used to produce the Chang and Marr index efficiently, vastly outperforming the naïve approach with increased q-gram size.
- The library was used to perform pairwise circular sequence re-alignment quickly under the Hamming and edit distance models, outperforming its competitors when the factor length and error rate increased.

Chapter 5

Circular Sequence Comparison with q-grams

This chapter presents the work done in the following publications:

1. [53] R. Grossi, C. S. Iliopoulos, R. Mercas, N. Pisanti, S. P. Pissis, A. Retha, F. Vayani, "Circular Sequence Comparison with q-grams", in Algorithms in Bioinformatics: 15th International Workshop, WABI 2015, Atlanta, GA, USA, September 10-12, 2015, Proceedings, M. Pop, H. Touzet, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 203-216.

2. [54] R. Grossi, C. S. Iliopoulos, R. Mercas, N. Pisanti, S. P. Pissis, A. Retha, F. Vayani, "Circular sequence comparison: algorithms and applications", Algorithms for Molecular Biology, vol. 11, no. 1, 2016, pp. 1-14.

5.1 Introduction

Circular molecular structures are present, in abundance, in all domains of life: bacteria, archaea, and eukaryotes; and in viruses. We refer the reader to Section 4.1.2 in the previous chapter (Chapter 4) for more details about circular sequences in nature. Sequence comparison is a fundamental step in many important tasks in bioinformatics, from phylogenetic reconstruction to the reconstruction of genomes. Traditional algorithms for measuring approximation in sequence comparison are based on the notions of distance or similarity, and are generally computed through sequence alignment techniques. A caveat of the adaptation of alignment techniques for circular sequence comparison is that they are computationally expensive, requiring from super-quadratic

to cubic time in the length of the sequence. Thus, heuristic methods are required for solving the problem quickly while being sufficiently accurate.

The problem: We consider the pairwise circular sequence comparison problem. Under the edit distance model, it consists in finding an optimal linear alignment of two circular strings. This problem for two strings x and y of length m and $n \geq m$, respectively, can be solved under the edit distance model in time $\mathcal{O}(nm \log m)$ [84]. Several other super-quadratic [88] and quadratic-time [15, 91] algorithms exist with varying degrees of accuracy. Trivially, for molecular biology applications, the same problem can be solved exactly in time $\mathcal{O}(nm^2)$ with scoring matrices and affine gap penalty scores. A *direct* application of pairwise circular sequence comparison is for the purpose of multiple circular sequence alignment (MCSA) [8, 39, 92].

To the best of our knowledge of past literature, there is no fast (that is, with sub-quadratic time complexity) and exact (or at least very accurate) algorithm for circular sequence comparison under some realistic or flexible model (that is, allowing *indels*), such as the edit distance model. Taking into account edit distance rather than Hamming distance is computationally challenging as the search space for seeking similarity is wider. Moreover, filters that work for Hamming distance do not work in general for edit distance [107] as well. An exception to this are the q-gram filtering techniques [143] that have successfully been used for string matching under the edit distance model (e.g. [17, 105, 118]), as well as for multiple local alignments both under the Hamming [106] and edit [105] distance model.

Our Contribution: In this research, we make effective use of q-grams to perform pairwise circular sequence alignment. We present two new efficient q-gram-based algorithms for solving the problem; specifically:

1. We introduce the *β-blockwise q-gram distance* between two strings x and y, that is, a generalisation of the q-gram distance introduced as a string distance measure in [143]. Intuitively, and similarly to [17, 105, 118], this generalisation consists of partitioning x and y into β *blocks* each, as evenly as possible, computing the q-gram distance between the corresponding block pairs, and then summing up the distances computed blockwise.

2. We present an exact naïve algorithm based directly on the *β-blockwise q-gram distance* metric.

3. We present another algorithm that uses *β-blockwise q-gram distance* metric to provide a rough heuristic solution that is further refined to give the best local-minimal β-blockwise q-gram distance.

4. Then we present our third algorithm, one which is exact and based on the suffix array [86] that finds the rotation of x such that the β-blockwise q-gram distance between the rotated x and y is minimal, in time and space $\mathcal{O}(\beta m + n)$, where $m = |x|$ and $n = |y|$.

5. We present a post-alignment *refinement* process for the third algorithm based on dynamic programming techniques to improve the pairwise alignment with respect to the beginning and end of the sequence.

6. We present an experimental study, using real and synthetic data, which demonstrates orders-of-magnitude superiority of our approach, in terms of efficiency, while maintaining an accuracy very competitive to the *optimal* obtained after considering all rotations of x against y using EMBOSS Needle [119].

5.2 Definitions

We mention a few definitions that have not already been described in Chapter 3 (Section 3.1).

A *Parikh vector* [123] associated with a string $r \in \Sigma^*$, denoted by $\mathcal{P}(r)$, represents a vector of size $\sigma = |\Sigma|$, where each component denotes the number of occurrences in r of the corresponding letter from a constant-sized alphabet Σ of size $\sigma = |\Sigma|$. It is simply a data structure that holds the tally/count or frequency of how many times a character occurs in a string. Consider a string $r = \texttt{ABACAB}$ over the alphabet $\Sigma = \{\texttt{A}, \texttt{B}, \texttt{C}, \texttt{D}\}$, the Parikh vector will contain:

Table 5.1 $\mathcal{P}(r)$: the Parikh vector of $r = \texttt{ABACAB}$ over the alphabet $\Sigma = \{\texttt{A}, \texttt{B}, \texttt{C}, \texttt{D}\}$.

Letter	**Frequency**
A	3
B	2
C	1
D	0

Consider two strings r and r' where $|r| = |r'|$, their Parikh vectors are equivalent, i.e. $\mathcal{P}(r) = \mathcal{P}(r')$, if one can be turned into the other by permuting its characters. Considering the fact that a rotation of a string is a permutation of that string, it follows that its Parikh vector will be identical.

Lemma 1. *Let r be a string and r' be a permutation of r; the Parikh vectors of both strings will be identical — $\mathcal{P}(r) = \mathcal{P}(r')$.*

We extend the definition of the Parikh vector to represent a table of q-grams and the frequency they occur in r. Let $q = 2$, we obtain the q-grams: `AB`, `BA`, `AC` and `CA`. If we assign each letter in Σ a number, $\Sigma' = \{\texttt{A}:0,\texttt{B}:1,\texttt{C}:2,\texttt{D}:3\}$, we can encode the q-grams in their numerical/binary form. With the DNA alphabet we can simply represent this using 2 bits per letter of a q-gram to make the integer string $r' = \{1,4,2,8,1\} = \{0001,0100,0010,1000,0001\}$, so each q-gram becomes an integer that corresponds to an index in the Parikh vector $\mathcal{P}(r')$. This is done using a two step algorithm—we use function Encode-Q-Gram (Algorithm 7) to encode the first q-gram at position $i = 0$ and for each successive q-gram at position $i = 1..|r| - q + 1$, the previous q-gram index can be updated with the next character in constant time using the operation Update-Q-Gram (Algorithm 8). This is faster than regenerating a brand new q-gram from scratch with every character in r, which would otherwise require $\mathcal{O}(|r| \cdot q)$ time instead of $\mathcal{O}(|r|)$ with this method. For each q-gram obtained in this way, we can update the Parikh vector, incrementing the q-gram index by 1 with each character, taking overall linear time in the length of the sequence being encoded/read. The size of the Parikh vector is $\mathcal{O}(\sigma^q)$. Table 5.2 below shows $\mathcal{P}(r')$:

Table 5.2 $\mathcal{P}(r')$: The Parikh vector of $r = \texttt{ABACAB}$ counting q-grams where $q = 2$ and $\Sigma' = \{\texttt{A}:0,\texttt{B}:1,\texttt{C}:2,\texttt{D}:3\}$.

Index	**q-gram**	**Encoded q-gram**	**Frequency**
0	AA	0000	0
1	AB	0001	2
2	AC	0010	1
3	AD	0011	0
4	BA	0100	1
5	BB	0101	0
6	BC	0110	0
7	BD	0111	0
8	CA	1000	1
9	CB	1001	0
10	CC	1010	0
11	CD	1011	0
12	DA	1100	0
13	DB	1101	0
14	DC	1110	0
15	DD	1111	0

We give some further definitions following [143]. The *q-gram profile* of a string x of length m is denoted by vector $G_q(x)$, where $q > 0$ and $G_q(x)[v]$ denotes the total

Algorithm 7 Encoding an initial q-gram of string r.
r: the string being encoded.
q: the length of the q-gram.
Σ': letter to number mapping {A:0,B:1,C:2,D:3}.
σ: the size of the alphabet.

procedure ENCODE-Q-GRAM(r, q, Σ', σ)
  $qg \leftarrow 0$
  for $i \in \{0..q-1\}$ **do**
    $qg = qg + (\Sigma'[r[i]] \times \sigma^i)$
  return qg

Algorithm 8 Updating q-gram qg with the next position $i = i+1$ in constant time.
r: the string being encoded.
q: the length of the q-gram.
Σ': letter to number assignment {A:0,B:1,C:2,D:3}.
σ: the size of the alphabet.
qg: the previously encoded q-gram.
i: the next position in string r.

procedure UPDATE-Q-GRAM($r, q, \Sigma', \sigma, qg, i$)
  $qg \leftarrow qg - \Sigma'[r[i-q]]$
  $qg \leftarrow qg \div \sigma$
  $qg \leftarrow qg + (\Sigma'[r[i]] \times \sigma^{q-1})$
  return qg

number of occurrences of $v \in \sigma^q$ in x. The q-gram distance between two strings x and y is defined as:

$$D_q(x,y) = \sum_{v \in \sigma^q} |G_q(x)[v] - G_q(y)[v]|. \tag{5.1}$$

Note that D_q is a *pseudo-metric* as $D_q(x,y)$ can be 0 even if $x \neq y$. Also note that Lemma 1 does not always hold for $q > 1$ but it can happen that two strings have identical q-gram profiles without being identical themselves. For example, let $x =$ CACACA, $y =$ ACACAC and $q = 3$, we find each string has two of each q-gram CAC and ACA, making $D_q(x,y) = 0$ even when $x \neq y$. The D_q metric has the following properties [143] for all $x, y, z \in \Sigma^*$ of length at least $q = 1$:

1. Positivity: $D_q(x,y) \geq 0$
2. Symmetry: $D_q(x,y) = D_q(y,x)$
3. Triangular inequality: $D_q(x,y) \leq D_q(x,z) + D_q(z,y)$

4. $|(|x|-|y|)| \leq D_q(x,y) \leq |x|+|y|-2q-2$

5. $D_q(x_1x_2, y_1y_2) \leq D_q(x_1,y_1)+D_q(x_2,y_2)+2(q-1)$

6. $D_q(h(x),h(y)) \leq D_q(x,y)$, for a non-length-increasing homomorphism h on Σ^*

We define a generalisation of the q-gram distance in Equation (5.1) by partitioning x and y as evenly as possible into β *blocks*. β is a parameter specified by the user for the number of blocks to partition the sequences into and affects the performance and accuracy of the algorithms presented below. The rationale for partitioning the sequences is to reduce the time complexity and enforce *locality* improving blockwise similarity and reducing chances a permutation of a string presents with the same or similar q-gram profile. Using q-gram distance for pairs of blocks between x and y, it is possible to find a good rotation alignment with high locality giving the lowest overall distance between x and y, following a refinement step. For the sake of presentation in the rest of the thesis, we assume that the lengths $|x| = |y|$ and the lengths of the strings are both multiples of β, so that x and y are conceptually partitioned into β blocks each of size $\frac{m}{\beta} = \frac{n}{\beta}$. In practice, there is no requirement that the strings be of equal length, that block sizes be factors of the string length or that the same block size is used for both strings.

Definition 1. *Given strings x of length m and y of length $n \geq m$ and integers $\beta \geq 1$ and $q > 0$, the* β-blockwise q-gram distance $D_{\beta,q}(x,y)$ *is defined as:*

$$D_{\beta,q}(x,y) = \sum_{j=0}^{\beta-1} D_q\left(x\left[\frac{jm}{\beta}..\frac{(j+1)m}{\beta}-1\right], y\left[\frac{jn}{\beta}..\frac{(j+1)n}{\beta}-1\right]\right). \tag{5.2}$$

Here, we consider the following problem, where we seek for the ith rotation of x, denoted x_i, that minimises its blockwise distance from y as defined in Equation (5.2). Ties are broken arbitrarily.

Circular Sequence Comparison (CSC)
Input: strings x and y of lengths m and $n \geq m$, respectively, and integers $\beta \geq 1$ and $q \geq 1$, such that $1 \leq \beta \leq m-q+1$ and $q < m$.
Output: A position i in x such that $D_{\beta,q}(x_i, y)$ is minimal.

Obtaining the ith position of x with the minimal β-blockwise q-gram distance will give us a rotation of x that aligns well to y.

5.3 Algorithms

5.3.1 Algorithm nCSC: An Exact Naïve Algorithm

We use the following approach to first give a naïve solution to the CSC problem. We call this algorithm nCSC. Although there is more than one way to solve the problem using the naïve approach, we present one such approach that removes the need to create multiple Parikh vectors, greatly saving on pre-processing time and space.

To find any rotation of y in x, we concatenate x with itself to create xx, allowing a match with y spanning across the start and end of x. As a preprocessing step, to avoid re-encoding q-grams with every iteration, we can store two new integer strings x' and y' that hold the numerical representation of each q-gram in xx and y, respectively, in time and extra space $\mathcal{O}(m+n)$. Although this step is not necessary (because q-grams can be encoded in real-time), it improves the general efficiency of the algorithm.

Additional space $\mathcal{O}(|\Sigma|^q)$ is required for Parikh vectors $\mathcal{P}(x)$ and $\mathcal{P}(y)$ to store the frequency of all possible q-grams of each block of strings x' and y', respectively. The Parikh vectors in this naïve design are mostly sparse and in practical terms the space can be reduced by standard hashing techniques without increasing the running time significantly [143]. Nonetheless, we continue to present the naïve algorithm according to the original design. We update the frequencies of the q-grams in the Parikh vectors in constant time, with the traversal of both strings taking total time $\mathcal{O}(m+n)$.

Lemma 2 ([143])**.** *If we have space $\mathcal{O}(|\Sigma|^q)$ available, then the q-gram distance $D_q(x,y)$ can be computed in time $\mathcal{O}(m+n)$ and extra space $\mathcal{O}(m+n)$, where $m=|x|$ and $n=|y|$.*

While one naïve approach to solving the problem is to create a Parikh vector for each β block and reset it after every iteration, it would greatly impact on the performance of the naïve algorithm, requiring extra space and initial setup time $\mathcal{O}(\beta \cdot |\Sigma|^q)$. We avoid the need to create multiple Parikh vectors by first updating $\mathcal{P}(x)$ and $\mathcal{P}(y)$, then resetting the updated values only, after having calculated the β-blockwise q-gram distance of each block. They are reset by traversing the q-grams in the blocks again and setting their counts in the Parikh vectors to 0.

Lemma 3. *If we have space $\mathcal{O}(|\Sigma|^q)$ available, then the β-blockwise q-gram distance $D_{\beta,q}(x,y)$ can be computed in time $\mathcal{O}(m+n)$.*

By Lemma 3 we know that reading the q-grams requires time linear in the size of the block of both x and y. Reading all the blocks of x and y will therefore require $\mathcal{O}(m+n)$ time. Updating the Parikh vector for each q-gram is done in constant time

because the numeric representation corresponds to an index in the Parikh vector and calculating an individual distance for a q-gram can be done in constant time as well. After the β-blockwise q-gram distance calculation is performed for a block, the Parikh vectors can be reset by reading the q-grams in the block a second time and setting the values to 0, ready for the next block's calculation. All together, for all blocks in x and y, this operation requires time $\mathcal{O}(m+n)$.

To computate the β-blockwise q-gram distance for all rotations of x, we create $x'' = xx$ and calculate $\delta_i = D_{\beta,q}(x''[i..i+m-1], y)$, for all positions $i = 0..m-1$. This requires the application of Lemma 3 m times to solve the problem exactly, and we report position i such that δ_i is minimal, marking the best position to rotate x_i at so it has the least distance from y. This is how we arrive to Lemma 4. The algorithm solves the CSC problem naïvely in time $\mathcal{O}(m(m+n))$ and space $\mathcal{O}(|\Sigma|^q)$.

Lemma 4. *If we have space $\mathcal{O}(|\Sigma|^q)$ available, then algorithm nCSC solves the CSC problem in time $\mathcal{O}(m(m+n))$.*

The pseudo-code for the implementation of the naïve algorithm is shown in Algorithm 9. The nCSC algorithm requires time $\mathcal{O}(m(m+n))$ and space $\mathcal{O}(|\Sigma|^q)$. No extra space is required if we generate the q-grams on-the-fly, but in our practical implementation in the pseudo-code below, we created two new integer strings requiring extra time and space $\mathcal{O}(m+n)$ which means the q-grams are calculated only once and this improves the efficiency of the algorithm.

5.3.2 Algorithm hCSC: A Heuristic Algorithm

Here we give a simple heuristic algorithm, denoted by hCSC, to solve the CSC problem faster than nCSC, and return an approximation of the best rotation.

This algorithm is described as being heuristic because it considers a limited number of positions to start with; It calculates the score at the starting position of each block $((j+1) \bmod (m/\beta) = 0))$ instead of naïvely doing it for every position as we do above in nCSC. It finds the first arrangement of blocks that gives the lowest score and then does a refinement step in order to find the local minimal score, checking every position within the flanking blocks. There may be other arrangements of the blocks that have the same minimal score or even some blocks that have a higher distance, but we break ties by choosing the first one and perform the refinement step for it. It may be that some other position gives a better alignment after refinement. In more divergent sequences with many substitutions, insertions and deletions, it is possible that the best combination of blocks determined using this heuristic approach may not even be

Algorithm 9 The naïve algorithm nCSC for solving the CSC problem.
x: string x.
m: the length of x.
y: string y.
n: the length of y.
β: the number of blocks to partition the strings into.
q: the length of a q-gram.

```
1: procedure nCSC(x, m, y, n, β, σ, q)
2:     Initialise Parikh vectors P_x and P_y of size σ^q each with 0s
3:     Make q-gram integer strings of xx (x') and y (y')
4:     bScr ← m + n
5:     bPos ← 0
6:     for i ∈ {0..2m − n − q} do
7:         scr ← 0
8:         xPos ← i
9:         yPos ← 0
10:        for j ∈ {0..n − q} do
11:            P_x[x'[i + j]] ← P_x[x'[i + j]] + 1
12:            P_y[y'[j]] ← P_y[y'[j]] + 1
13:            if j > 0 and j mod (m/β) = 0 or j = n − q then
14:                for k ∈ {xPos..i + j} do
15:                    if P_x[x'[k]] ≠ −1 then
16:                        scr ← scr + abs(P_x[x'[k]] − P_y[x'[k]])
17:                        P_x[x'[k]] ← P_y[x'[k]] ← −1
18:                for k ∈ {yPos..j} do
19:                    if P_y[y'[k]] ≠ −1 then
20:                        scr ← scr + P_y[y'[k]]
21:                        P_y[y'[k]] ← −1
22:                for k ∈ {xPos..i + j} do
23:                    P_x[x'[k]] ← P_y[x'[k]] ← 0
24:                for k ∈ {yPos..j} do
25:                    P_y[y'[k]] ← 0
26:                xPos ← i + j + 1
27:                yPos ← j + 1
28:        if scr < bScr then
29:            bScr ← scr
30:            bPos ← i
31:    Report best score bScr at position bPos
```

close to the best rotation. We expect the accuracy of this algorithm to degrade when tested with more divergent sequences but we have not explored its theoretical bounds

or practical limitations further since Algorithm saCSC described in the next section is an exact algorithm and offers superior performance and accuracy.

In algorithm hCSC, we manage a pair of Parikh vectors of size $\mathcal{O}(|\Sigma|^q)$ in the same way as the naïve algorithm above. We now describe how it operates:

Step 1: We split $x' = xx$ into 2β non-overlapping string *blocks* of length m/β. We obtain a set of strings $x_0, x_1, \ldots, x_{2\beta-1}$, such that $x_i = x'[\frac{im}{\beta} .. \frac{(i+1)m}{\beta} - 1]$, for all $0 \leq i < 2\beta$. We do the same for y and split it into β non-overlapping string blocks of length $n/\beta = m/\beta$. We obtain strings $y_0, y_1, \ldots, y_{\beta-1}$, such that $y_i = y[\frac{in}{\beta} .. \frac{(i+1)n}{\beta} - 1]$, for all $0 \leq i < \beta$.

Step 2: We compute the β-blockwise q-gram distance of each of the blocks of strings x' and y as follows:

$$\delta_j = D_{\beta,q}(x'[\frac{jm}{\beta} .. \frac{jm}{\beta} + m - 1], y) = \sum_{i=0}^{\beta-1} D_q(x_{j+i}, y_i)$$

We compute δ_j, for all $0 \leq j \leq \beta$ and we select the block with the smallest distance, where δ_j is minimal. We call this starting position j_{best}. What we have found is a *window* of text in x of length m starting at block position j_{best} and consisting of β blocks of length m/β each, that minimises its β-blockwise q-gram distance from y. This alignment is coarse and does not guarantee us the best alignment between x and y. To potentially improve the alignment, we have to perform a localised refinement process in the next step.

Step 3: To perform a refinement on the window of size m, we consider all positions k included inside the two blocks flanking position j_{best} on either side. We need to calculate the β-blockwise q-gram distance from position $k = j_{best} - m/\beta + 1 \ldots j_{best} + m/\beta - 1$. That is $2m/\beta - 2$ starting positions and in addition to this we do not need to recalculate position j_{best} as this was already calculated in Step 2. So we calculate the β-blockwise q-gram distance δ_i of each window of length m for each starting position k. We report position $i_{best} = k$ such that δ_i is minimal.

The pseudo-code for Steps 1 and 2 are given below in Algorithm 10. Step 3 for refinement is a simple application of the naïve **nCSC** algorithm's approach for calculating the β-blockwise q-gram distance, where $j_{best} = bPos$, we check xx from position $j_{best} - m/\beta + 1 \ldots j_{best} + m/\beta - 1$ against y to find the local-minimal δ_i. If $j_{best} = 0$, we set $j_{best} = m$.

Analysis. Step 1 can be done trivially in time $\mathcal{O}(m+n)$, requiring space $\mathcal{O}(|\Sigma|^q)$ for the Parikh vectors. If we have space $\mathcal{O}(|\Sigma|^q)$ available, then, by Lemma 2, the

Algorithm 10 The heuristic algorithm hCSC for solving the CSC problem - only steps 1 and 2.
x: string x.
m: the length of x.
y: string y.
n: the length of y.
β: the number of blocks to partition the strings into.
q: the length of a q-gram.

```
1: procedure hCSC(x, m, y, n, β, σ, q)
2:     Initialise Parikh vectors P_x and P_y of size σ^q each with 0s
3:     B_x ← xx split into 2β non-overlapping string blocks each of length m/β and encode into q-gram integer strings
4:     B_y ← y split into β non-overlapping string blocks each of length m/β and encode into q-gram integer strings
5:     ℓ = m/β − q
6:     bScr ← m + n
7:     bPos ← 0
8:     for i ∈ {0..β} do
9:         scr ← 0
10:        for j ∈ {0..β} do
11:            for k ∈ {0..ℓ} do
12:                P_x[B_x[i+j][k]] ← P_x[B_x[i+j][k]] + 1
13:                P_y[B_y[j][k]] ← P_y[B_y[j][k]] + 1
14:            for k ∈ {0..ℓ} do
15:                if P_x[B_x[i+j][k]] ≠ −1 then
16:                    scr ← scr + abs(P_x[B_x[i+j][k]] − P_y[B_x[i+j][k]])
17:                    P_x[B_x[i+j][k]] ← P_y[B_x[i+j][k]] ← −1
18:            for k ∈ {0..ℓ} do
19:                if P_y[B_y[j][k]] ≠ −1 then
20:                    scr ← scr + P_y[B_y[j][k]]
21:                    P_y[B_y[j][k]] ← −1
22:            for k ∈ {0..ℓ} do
23:                P_x[B_x[i+j][k]] ← P_y[B_x[i+j][k]] ← 0
24:            for k ∈ {0..ℓ} do
25:                P_y[B_y[j][k]] ← 0
26:        if scr < bScr then
27:            bScr ← scr
28:            bPos ← i × m/β
```

β-blockwise q-gram comparison of a single pair of blocks, $D_q(x_{j+i}, y_i)$, can be computed in time $\mathcal{O}(\frac{m+n}{\beta})$.

In Step 2, as per Lemma 3, we can calculate the β-blockwise q-gram distance of all blocks $0 \leq j \leq \beta$ of y against x once, giving us δ_j in time $\mathcal{O}(\beta(\frac{m+n}{\beta})) = \mathcal{O}(m+n)$. Since we calculate the β-blockwise q-gram distance β times for all blocks $0 \leq j \leq \beta$ of y against all blocks $0 \leq i \leq \beta$ of x, Step 2 is completed in time $\mathcal{O}(\beta(m+n))$.

In Step 3, the β-blockwise q-gram distance δ_i between a single window of size m and y can be computed in time $\mathcal{O}(\beta\frac{m+n}{\beta}) = \mathcal{O}(m+n)$. Since there exist $2m/\beta - 2$ such windows, Step 3 can be done in time $\mathcal{O}(\frac{m}{\beta} \cdot \beta\frac{m+n}{\beta}) = \mathcal{O}(\frac{m(m+n)}{\beta})$.

In conclusion, the algorithm requires time $\mathcal{O}(\beta(m+n) + \frac{m(m+n)}{\beta})$ and extra space $\mathcal{O}(|\Sigma|^q + m + n)$.

For practical purposes, and as we will soon show in the experimental results below, it is beneficial to set $\beta = \mathcal{O}(\sqrt{m})$ and $q = \mathcal{O}(\log_\sigma m)$ to give an algorithm with time complexity $\mathcal{O}(\sqrt{m}(m+n))$ and space complexity $\mathcal{O}(m+n)$. This is explained by the fact that $\sigma^{\log_\sigma m} = m$ and $\frac{m}{\sqrt{m}} = \sqrt{m}$.

5.3.3 Algorithm saCSC: An Exact Suffix Array-based Algorithm

The heuristics solution of hCSC does not guarantee to find the exact position i, for which $\delta_i = D_{\beta,q}(x_i, y)$ is minimal. In particular, when we identify in Step 2 j_{best}, that is, position j for which δ_j is minimal, we take into account only the starting positions of each block ($(j+1) \bmod (m/\beta) = 0$). Thus, Step 3 cannot guarantee that i_{best}, the local minimum obtained by shifting the window $2m/\beta - 2$ positions to the right and left of j_{best}, is minimal for all $0 \leq i < m$. In this section, we give a fast and exact algorithm, denoted by saCSC, to find i such that $\delta_i = D_{\beta,q}(x_i, y)$ is minimal, based on the suffix array (see Section 3.4).

We partially follow the idea from [34]. This work investigates the string matching problem in the setting of k-abelian equivalences: two strings are considered k-abelian equivalent for some positive integer k, if they have the same length and share the same factors of length at most k, including multiplicities. Note that if k is greater than or equal to the string's length, then the strings must be equal. A version of this result, called extended k-abelian equivalence, focuses only on the factors of length k. By setting $k = q$, it is quite straightforward to notice the equivalence with q-grams. Therefore, in order to avoid confusion we will refer to the former notion from now on as *q-abelian equivalence*.

In [34], the authors propose a linear-time algorithm to solve the string matching problem when looking at q-abelian equivalent strings: given a string x of length m, a

string y of length $n \geq m$, and a positive integer $q < m$, all factors of y that are q-abelian equivalent to x can be found in time and space $\mathcal{O}(m+n)$. The idea of the algorithm in [34] consists in constructing the suffix array of the string xy, and ranking sets of identical q-length prefixes of suffixes in the suffix array in the order of their appearance. Then it constructs new strings based on this ranking, and solves the problem as in the *jumbled matching* case [16], i.e. identifying all factors of y that have the same Parikh vector as x.

We note that we can have at most $g \leq n-q+1$ distinct matching q-grams between y and x and we use this knowledge to our advantage in designing the algorithm used to solve the problem. We create a pair of q-gram integer strings, $x' = xx$ and $y' = y$, of size $2m-q+1$ and $n-q+1$, respectively, to hold the numeric representation of each string's q-grams. We traverse the Suffix Array of the $z = xxy$ and update positions in x' and y' with a *rank*, a number $r \in \{0\,..\,g-1\}$ representing each distinct q-grams that is found in both strings. We create two specialised q-gram indices, a_x and a_y; the index a_x signifies any q-gram that exists in x but not in y, and the index a_y signifies a q-gram that exists in y but does not exist in x.

For calculating the β-blockwise q-gram distance we create a *difference* Parikh vector $\mathcal{P}(\mathsf{diff})$ of size $g+2 \leq n-q+3$ containing the indices of all the matching q-grams of x' and y' including the two special indices a_x and a_y. For position $i = 0$ in x' we update $\mathcal{P}(\mathsf{diff})$ to hold the β-blockwise q-gram distance of x' and y' for the first set of blocks of total window length $n-q+1$. For subsequent positions $i = 1\,..\,i+n-q$, moving the window along one position in x' at a time, we update the difference vector $\mathcal{P}(\mathsf{diff})$ by incrementing the count for the newly added q-gram on the right-hand side and decrementing the count for the old q-gram on the left-hand side, as well as updating δ_i accordingly. We report position i_{best} such that δ_i is minimal. As a recommended optional step, we perform some refinement to this result using a technique based on dynamic programming to ensure the best circular alignment is obtained.

We first describe our algorithm for a single block ($\beta = 1$) and then address the general case ($\beta \geq 1$).

Example of the basic algorithm for $\beta = 1$. Let $x = \texttt{GAGTCTA}$, $y = \texttt{TCTAGCG}$, $m = n = 7$, and $q = 3$.

Step 1: We construct the suffix array, iSA and LCP array of the string $z = xxy$ without a sentinel character (because it is not required) and we create two new q-gram integer strings x' of length $2m-q+1$ and y' of length $n-q+1$, respectively. We initialise x' to contain a_x and likewise we initialise y' to contain a_y at each position.

We do this because we will be assigning a rank number ($r \notin \{a_x, a_y\}$) for the q-grams shared by both strings at the positions where the q-grams are found in xx and y, respectively. We create a ranking variable r and set it to $r = -1$ before traversing $\mathsf{ST}(z)$. The rank will be incremented during the algorithm such that the first rank inserted into x' and y' will be $r = 0$.

Step 2: We then traverse the suffix array from index $i = 1..2m+n-1$. At each position i we determine if the prefix of the suffix at i is new by looking at its LCP value — if $LCP[i] \geq q$ then it must share the same prefix as the previous suffix, $SA[i-1]$. We use the suffix numbers SA[i] to determine if they belong in both xx and y and increment the rank, $r = r+1$, for each new q-gram, otherwise the rank remains unchanged. If the new q-gram is first discovered in xx, we assign the new ranks: $x'[SA[i-1]] = r$ and $y'[SA[i]-2m] = r$; And if the new q-gram is first discovered in y, we assign the new ranks – $x'[SA[i]] = r$ and $y'[SA[i-1]-2m] = r$. In this way we assign a rank number to every shared q-gram position in both xx and y. As for q-grams that are not shared, they were already set to a_x or a_y when x' and y' were first created. The observe the resulting arrays in Table 5.8.

Step 3: By the end of Step 2, we realise that r will hold the last index of the unique q-grams that are shared in both x' and y'. Let $g = r+1$, be the number of shared q-grams. We create a temporary Parikh vector $\mathcal{P}(y')$ of size $g+1 \leq n-q+2$ with indices from $0..g$. Index g is made to represent a_y. We run through the q-grams of y' to update the counts in $\mathcal{P}(y')$. We then create another Parikh vector $\mathcal{P}(\mathsf{diff})$ of size $g+2$, containing all the indices of $\mathcal{P}(y')$ as well as a_x at index $g+1$, and we initialise the counts of this vector: $\mathcal{P}(\mathsf{diff})[i] = \mathcal{P}(y')[i]$ for $0 \leq i \leq g$. To accompany the difference vector, we create an integer array δ of size $m-1$ to keep the score of the β-blockwise q-gram distance and initialise the value in δ_0:

$$\delta_0 = \sum_{i=0}^{g} \mathcal{P}(y')[i]$$

This resulting Parikh vectors are shown in Table 5.3.

Table 5.3 The initialised values of $\mathcal{P}(y')$ and $\mathcal{P}(\mathsf{diff})$.

	0	1	2	a_y	a_x
$\mathcal{P}(y')$	1	1	1	2	
$\mathcal{P}(\mathsf{diff})$	1	1	1	2	0

Step 4: Now we come to calculating the β-blockwise q-gram distance. Each position $i = 0..m-1$ in δ will hold the q-gram distance between y' and the window

of size $i..i+n-1$ in x', which is the ith rotation x_i of x. At position x_0, we update $\mathcal{P}(\mathsf{diff})$ for every character we read from $x'[0..m-1]$, decreasing by 1 the value of the newly read letter, and updating δ_0, by either increasing the current value of the distance when we read too many of the current letters, or decreasing it, when more of these letters still occur in y':

$$\mathcal{P}(\mathsf{diff})[x'[i]] = \mathcal{P}(\mathsf{diff})[x'[i]] - 1 \qquad \text{and} \qquad \delta_0 = \begin{cases} \delta_0 - 1, & \text{if } \mathcal{P}(\mathsf{diff})[x'[i]] \geq 0 \\ \delta_0 + 1, & \text{if } \mathcal{P}(\mathsf{diff})[x'[i]] < 0 \end{cases}$$

The resulting difference vector is shown in Table 5.4.

Table 5.4 The contents of $\mathcal{P}(\mathsf{diff})$ at position $i = 0$. δ_0 is 6.

	0	1	2	a_y	a_x
$\mathcal{P}(\mathsf{diff})$	0	1	0	2	3

For each subsequent position $i = 1..m-1$ we perform a constant number of operations – we first set $\delta_i = \delta_{i-1}$, then we subtract 1 from the difference vector for the old q-gram from the left-hand side of the window at position x_{i-1} and add 1 for the new q-gram at the right-hand side of the window at position x_{i+m-1}, updating δ_i with each operation. The updated value in the difference vector and the distance score is calculated sequentially applying the following two rules for an q-gram that is shared between x and y:

$$\delta_i = \begin{cases} \delta_i - 1, & \text{if } \mathcal{P}(\mathsf{diff})[x'[i]] \leq 0 \\ \delta_i + 1, & \text{if } \mathcal{P}(\mathsf{diff})[x'[i]] > 0 \end{cases}$$

$$\delta_i = \begin{cases} \delta_{i+1} - 1, & \text{if } \mathcal{P}(\mathsf{diff})[x'[i+m]] \geq 0 \\ \delta_{i+1} + 1, & \text{if } \mathcal{P}(\mathsf{diff})[x'[i+m]] < 0 \end{cases}$$

But when moving the window, if we find character a_x, that is a character not in y' — we update $\mathcal{P}(\mathsf{diff})$ for every new a_x character we read from $x'[i..i+m-1]$, increasing by 1 the value of $\mathcal{P}(\mathsf{diff})[a_x]$, and updating δ_i, or decreasing $\mathcal{P}(\mathsf{diff})[a_x]$ by 1, and updating δ_i, for every a_x character we remove from the left-hand side:

$$\delta_i = \begin{cases} \delta_i + 1, & \text{if } \mathcal{P}(\mathsf{diff})[x'[i]] \leq 0 \\ \delta_i - 1, & \text{if } \mathcal{P}(\mathsf{diff})[x'[i]] > 0 \end{cases}$$

$$\delta_i = \begin{cases} \delta_{i+1} + 1, & \text{if } \mathcal{P}(\mathsf{diff})[x'[i+m]] \geq 0 \\ \delta_{i+1} - 1, & \text{if } \mathcal{P}(\mathsf{diff})[x'[i+m]] < 0 \end{cases}$$

Note that the details of how we deal with a_x is not shown in the pseudo-code of Algorithm saCSC (Algorithm 11) for the purpose of clarity and due to lack of space.

Table 5.5 The contents of $\mathcal{P}(\mathsf{diff})$ at position $i = 1$. δ_1 is 4.

	0	1	2	a_y	a_x
$\mathcal{P}(\mathsf{diff})$	0	0	0	2	2

At position $i = \{1,2,3\}$ we find the lowest distances, $\delta_{1..3} = 4$, and the difference vector (Table 5.6) contains:

Table 5.6 The contents of $\mathcal{P}(\mathsf{diff})$ at positions $i = 1..3$.

	0	1	2	a_y	a_x
$\mathcal{P}(\mathsf{diff})$	0	0	0	2	2

At position $i = 4$ the distance increases so $\delta_4 = 6$, and the difference vector (Table 5.7) contains:

Table 5.7 The contents of $\mathcal{P}(\mathsf{diff})$ at position $i = 4$. δ_4 is 6.

	0	1	2	a_y	a_x
$\mathcal{P}(\mathsf{diff})$	0	0	1	2	3

After we have run through all positions $0 \leq i < m$, we report position i_{best} where δ_i is minimal, that is, the window $x'[i..i+m-1]$ in x with the least distance from y.

We also present the pseudo-code for saCSC for $\beta = 1$ in Algorithm 11. Let the array s created on line 3 be of size $2m+n$. For each index i in SA that can represent a valid q-gram, we give a number 0 or 1 if the suffix number is found in xx or y of z, respectively. We assign $s[i] = 0$ **if** $\mathsf{SA}[i] \in \{0..2m-q\}$ and we assign $s[i] = 1$ **if** $2m \leq \mathsf{SA}[i] \leq 2m+n-q$, otherwise $s[i] = -1$. This helps in properly assigning the ranks as seen in the pseudo-code from lines 6–16. In practice we do not need to store array s because we can check the position of a suffix in constant time, but we include it here to help explain the algorithm.

From lines 22–32 we create the difference Parikh vector $\mathcal{P}(\mathsf{diff})$ and initialise it with all the values of the $\mathcal{P}(y')$. Then from position $i = 0$, we scan through xx updating the difference vector to get the score δ_0 for the first window of size $n-q$. For the rest of the positions $i = 1..m-q$, we update the difference vector by subtracting the trailing character, adding the new character and updating the score δ_i, as shown in lines 34–45. In performing this process we can identify the position i_{best} that gives the lowest score where δ_i is minimal and report it accordingly.

Algorithm 11 The exact algorithm saCSC for solving the CSC problem, $\beta = 1$.
x: string x.
m: the length of x.
y: string y.
n: the length of y.
β: the number of blocks to partition the strings into.
q: the length of a q-gram.

```
1:  procedure saCSC(x, m, y, n, β, q)
2:      Build the suffix array SA and LCP array LCP on string z = xxy
3:      Create array s of size 2m + n for determining positions of q-grams in SA
4:      Create integer string x' of length 2m − q + 1 initialised with a_x
5:      Create integer string y' of length n − q + 1 initialised with a_y
6:      r ← −1
7:      for i ∈ {1..2m + n − 1} do
8:          if LCP[i] ≥ q then
9:              if s[i − 1] = 0 and s[i] = 1 then
10:                 if x'[SA[i − 1]] ≠ r then
11:                     r ← r + 1
12:                 x'[SA[i − 1]] ← y'[SA[i] − 2m] ← r
13:             else if s[i − 1] = 1 and s[i] = 0 then
14:                 if x'[SA[i − 1] − 2m] ≠ r then
15:                     r ← r + 1
16:                 x'[SA[i]] ← y'[SA[i − 1] − 2m] ← r
17:     Create Parikh vector P(y') of size r + 2 initialised with 0s
18:     scr ← 0
19:     for i ∈ {0..n − q} do
20:         P(y')[y'[i]] ← P(y')[y'[i]] + 1
21:         scr ← scr + 1
22:     Create Parikh vector P(diff) of size r + 3 initialised to P(y')s
23:     for i ∈ {0..n − q} do
24:         if x'[i] = a_x then
25:             P(diff)[a_x] ← P(diff)[a_x] + 1
26:             scr ← scr + 1
27:         else
28:             P(diff)[x'[i]] ← P(diff)[x'[i]] − 1
29:             if P(diff)[x'[i]] ≥ 0 then
30:                 scr ← scr − 1
31:             else
32:                 scr ← scr + 1
33:     Create scores array δ of size m − q + 1 and initialise δ[0] ← scr
```

Table 5.8 Let $x = \texttt{GAGTCTA}$, $y = \texttt{TCTAGCG}$ and $q = 3$. The table shows the Suffix Array and LCP array for $z = xxy = \texttt{GAGTCTAGAGTCTATCTAGCG}$, as well as the q-gram integer strings for xx and y. $x'[3] = y'[0] = 2$ denotes that $x[3..5] = y[0..2] = \texttt{TCT}$. $x'[0] = a_x$ denotes that $x[0..2] = \texttt{GAG}$ does not occur in y. Array s indicates with 0 or 1 whether a valid q-gram falls in xx or y, respectively.

i	**Suffix**	SA$[i]$	LCP$[i]$	$x'[i]$	$y'[i]$	$s[i]$
0	AGAGTCTATCTAGCG	6	0	a_x	2	0
1	AGCG	17	2	a_x	0	1
2	AGTCTAGAGTCTATCTAGCG	1	2	a_x	1	0
3	AGTCTATCTAGCG	8	6	2	a_y	0
4	ATCTAGCG	13	1	0	a_y	−1
5	CG	19	0	1		−1
6	CTAGAGTCTATCTAGCG	4	1	a_x		0
7	CTAGCG	15	4	a_x		1
8	CTATCTAGCG	11	3	a_x		0
9	G	20	0	a_x		−1
10	GAGTCTAGAGTCTATCTAGCG	0	1	2		0
11	GAGTCTATCTAGCG	7	7	0		0
12	GCG	18	1			1
13	GTCTAGAGTCTATCTAGCG	2	1			0
14	GTCTATCTAGCG	9	5			0
15	TAGAGTCTATCTAGCG	5	0			0
16	TAGCG	16	3			1
17	TATCTAGCG	12	2			−1
18	TCTAGAGTCTATCTAGCG	3	1			0
19	TCTAGCG	14	5			1
20	TCTATCTAGCG	10	4			0

Algorithm 11 Algorithm saCSC continued...

for $i \in \{1..m-q\}$ **do**
  $\delta[i] \leftarrow \delta[i-1]$
  $\mathcal{P}(\mathsf{diff})[x'[i-1]] \leftarrow \mathcal{P}(\mathsf{diff})[x'[i-1]]+1$ ▷ Removing previous character
  if $\mathcal{P}(\mathsf{diff})[x'[i-1]] \leq 0$ **then**
    $\delta[i] \leftarrow \delta[i]-1$
  else
    $\delta[i] \leftarrow \delta[i]+1$
  $\mathcal{P}(\mathsf{diff})[x'[i+n-q]] \leftarrow \mathcal{P}(\mathsf{diff})[x'[i+n-q]]-1$ ▷ Adding new character
  if $\mathcal{P}(\mathsf{diff})[x'[i+n-q]] \geq 0$ **then**
    $\delta[i] \leftarrow \delta[i]-1$
  else
    $\delta[i] \leftarrow \delta[i]+1$
Report index i of lowest score in δ

General algorithm for $\beta \geq 1$. We can now generalise this algorithm to solve the CSC problem for $\beta \geq 1$, which gives algorithm saCSC. We maintain a Parikh vector for each block, and apply the above basic algorithm for each pair of blocks, computing their q-gram distance. If we denote by $\mathcal{P}_j(y')$ and $\mathcal{P}_j(\mathsf{diff})$, for all $0 \leq j < \beta$, the β Parikh vectors of y' and of the q-gram distances, respectively, as well as by $\delta_{i,j}$ the q-gram distance between the jth block of y and x^i, then the updates will be given by the formulae below. Hence, at each position $i < m$, we can update all of the β Parikh vectors corresponding to the blocks, as previously described, in time $\mathcal{O}(\beta)$. As an example, see here the modification of the previous Step 3, with the other two steps being easily adapted in a similar fashion.

When shifting the window one position to the right from position i, update the information for every $\mathcal{P}_j(\mathsf{diff})$, where $0 \leq j < \beta$, as follows

$$\mathcal{P}_j(\mathsf{diff})[x'[i+\frac{jm}{\beta}]] = \mathcal{P}_j(\mathsf{diff})[x'[i+\frac{jm}{\beta}]]+1$$
$$\mathcal{P}_j(\mathsf{diff})[x'[i+\frac{(j+1)m}{\beta}]] = \mathcal{P}_j(\mathsf{diff})[x'[i+\frac{(j+1)m}{\beta}]]-1,$$

and calculate $\delta_{i+1,j}$, based on this information, sequentially applying the two following rules

$$\delta_{i+1,j} = \begin{cases} \delta_{i,j}-1, & \text{if } \mathcal{P}_j(\mathsf{diff})[x'[i+\frac{jm}{\beta}]] \leq 0 \\ \delta_{i,j}+1, & \text{if } \mathcal{P}_j(\mathsf{diff})[x'[i+\frac{jm}{\beta}]] > 0 \end{cases}$$

$$\delta_{i+1,j} = \begin{cases} \delta_{i+1,j} - 1, & \text{if } \mathcal{P}_j(\mathsf{diff})[x'[i + \frac{(j+1)m}{\beta}]] \geq 0 \\ \delta_{i+1,j} + 1, & \text{if } \mathcal{P}_j(\mathsf{diff})[x'[i + \frac{(j+1)m}{\beta}]] < 0. \end{cases}$$

Analysis. In Step 1, constructing SA, iSA, and LCP of xxy can be done in time and extra space $\mathcal{O}(m+n)$ (see Preliminaries section 3.4). Furthermore in Step 2, the construction of x' and y' takes linear time and space. Populating the two integer strings by reading through the suffix array and LCP array takes time $\mathcal{O}(m+n)$. In Step 3, where we create $\mathcal{P}(y')$ and $\mathcal{P}(\mathsf{diff})$ and populate them requiring extra space and time $\mathcal{O}(m+n)$. The array δ is created requiring time $\mathcal{O}(m)$. In Step 4, reading through the first window $x[0..m-1]$ and updating $\mathcal{P}(\mathsf{diff})$ and δ_0 takes constant time for each letter, thus taking $\mathcal{O}(m)$ time in total. Then for each subsequent window from position $1..m-1$, updating $\mathcal{P}(\mathsf{diff})$ and δ_i requires two steps – removing the old q-gram and adding a new q-gram. These operations take constant time for each new position, so in total they require time $\mathcal{O}(m)$. Hence, we can solve the CSC problem for $\beta = 1$ in time and space $\mathcal{O}(m+n)$.

Theorem 1. *Algorithm saCSC solves the* CSC *problem in $\mathcal{O}(\beta m+n)$ time and space.*

For the general algorithm where $\beta > 1$, x' and y' are subdivided into β blocks, with Parikh vectors $\mathcal{P}_j(y')$ and $\mathcal{P}_j(\mathsf{diff})$ being created and maintained for all blocks $0 \leq j < \beta$. Updating each vector takes time $\mathcal{O}(m)$ and there are β vectors, so in total, along with Steps 1-3, algorithm saCSC takes time and space $\mathcal{O}(\beta m+n)$ to solve the CSC problem.

saCSC Refinement. The application of the β-blockwise q-gram distance via algorithm saCSC suggests that an optimal or a close-to-optimal rotation of x can be found when compared to the rotation discovered by less efficient approaches like the dynamic programming-based solution, Circular Needleman-Wunsch (cNW). Due to the locality property offered by the newly introduced distance notion, it is reasonable to assume that the close-to-optimal rotation returned by saCSC may be refined via some quick heuristics that take into consideration the blocks at both ends. Let x_i be the close-to-optimal rotation of x returned by saCSC. We introduce a new input parameter p, a positive integer $0 \leq p \leq \frac{\beta}{3}$, which defines the number of blocks to use and the length $L = \lfloor p \cdot \frac{m}{\beta} \rfloor$ of the prefixes and suffixes of x_i and y to be considered in the refinement.

We take p block(s) of the prefix of x_i, concatenate it with a string of equal length L comprised only of letter \$, where $\$ \notin \Sigma$, and concatenate that with p block(s) of the suffix of x_i to form a new string x'' of length $3L$. We do the same with y to form a new string y''. See the Example 5.3.3 below.

The refinement algorithm works by taking all rotations of x'' and comparing their similarity to y''. Each rotation of x'' is compared to y'' excluding when a $ letter is found at index 0 of the rotation of x''. We measure the similarity between the strings for which equality between letters are positively valued; inequalities, insertions, and deletions are negatively valued; and comparisons involving $ are neither positively nor negatively valued. The goal of rotating x'' serves to find the rotation that maximizes the similarity to y'' and, to this end, we make use of the Needleman-Wunsch dynamic programming algorithm. The rotation of x'' which results in the maximum score is chosen as the best rotation, and hence, the final rotation x_i of x is computed based on this rotation of x''. Ties are broken arbitrarily. We denote this new algorithm, consisting of saCSC and the refinement stage, by saCSCr.

An example of saCSCr. Consider the following pair of strings obtained from saCSC,

$$x_i = \texttt{GACACCCCCCACAGTTTATGTAGCTT...ACCCCGAACCAACCAAACCCCAAA}$$
$$y = \texttt{GTTTATGTAGCTTACCTCCCCAAAGC...CAAACCCCAAAGACACCCCACACA}$$

and the following pair of strings formed for $L = 25$.

$$x'' = \texttt{GACACCCCCCACAGTTTATGTAGCTT\$ACCCCGAACCAACCAAACCCCAAA}$$
$$y'' = \texttt{GTTTATGTAGCTTACCTCCCCAAAGC\$CAAACCCCAAAGACACCCCACACA}$$

The Needleman-Wunsch algorithm for all rotations of x'' and string y'' gives the following optimal alignment,

```
GTTTATGTAGCTT$$$$$$$$$$$$$$$$$$$$$$$$$ACCCCGAACCAACCAAACCCCAAAGACACCCCCCACA
GTTTATGTAGCTTACCTCCCCAAAGC$$$$$$$$$$$$$$$$$$$$$$$$$CAAACCCCAAAGACACCCCACACA
```

which tells us that a refined rotation is in fact $x_{i'}$, where $i' = i - 13$.

$$x_{i'} = \texttt{GTTTATGTAGCTT...CAAACCCCAAAGACACCCCCCACA}$$
$$y = \texttt{GTTTATGTAGCTT...CAAACCCCAAAGACACCCCACACA}$$

Theorem 2. *Algorithm saCSCr solves the* CSC *problem in* $\mathcal{O}(\beta m + n + L^3)$ *time and space* $\mathcal{O}(\beta m + n + L)$.

The application of the Needleman-Wunsch algorithm on a pair of strings of length $3L$ has a time and space complexity of $\mathcal{O}(L^2)$ [99]. The time complexity can be

reduced further to $\mathcal{O}(L)$ by storing and updating two rows of length L as mentioned in Chapter 3. Considering all L rotations of x'' results in a final time complexity of $\mathcal{O}(L^3)$ for the refinement step. So, algorithm saCSCr, the result of executing the saCSC algorithm followed by a refinement step, requires total time $\mathcal{O}(\beta m + n + L^3)$ and space $\mathcal{O}(\beta m + n + L)$.

5.4 Experiments

We initially implemented algorithms nCSC, hCSC, and saCSC as the program CSC. After seeing the exceptional speed and accuracy performance advantages of the algorithms presented in this paper, they were incorporated into the state-of-the-art multiple circular sequence alignment program BEAR [8].

Given one of the three methods, nCSC, hCSC, and saCSC, two sequences x and y in (Multi)FASTA format, the number β of blocks, the length q of the q-grams, and an optional refinement parameter p, CSC finds the rotation of x (or an approximation of it) that minimises its β-blockwise q-gram distance from y. The implementation is distributed under the *GNU General Public License* (GPL), and it is available at http://github.com/solonas13/csc. For comparison purposes, we implemented a naïve dynamic programming-based algorithm Circular Needleman-Wunsch (cNW) that compares all rotations of x against y with substitution matrices and affine gap penalty scores [50]. We also implemented the Smith-Waterman local alignment algorithm [146] to search for the best local alignment of x and y and then used the central match from this local alignment to anchor the global alignment; we denote this implementation hSW.

The following experiments were conducted on a desktop computer using one core of Intel® Core™ i7-2600S CPU at 2.8GHz and 8GB of RAM under 64-bit GNU/Linux. All programs were compiled with gcc version 4.7.3. We used both synthetic data (Sections 5.4.1–5.4.3) and real data (Section 5.4.4). All input datasets referred to in this section are publicly maintained at the same website.

5.4.1 Experiment I: Accuracy

We used the same three synthetic datasets containing 12 sequences at as described in Section 4.4.2 of Chapter 4. Each of these datasets contains 12 sequences mutated at substitution rates 5%, 20%, and 35% as mentioned before. We used these datasets to compare the accuracy of the algorithms in realigning randomly-rotated sequences with

varying degrees of disimilarity. We remind the reader that the original datasets after being produced with INDELible [41] are named *Original* and the datasets after being randomly rotated where termed *Random*.

For each *Random* dataset, an all-against-all sequence comparison was performed. That is, all possible pairs, 66 in total, of sequences in each dataset were input to both hCSC and saCSC. β was set to $\lceil\sqrt{m}\rceil = 50$ and q was set to $\lceil\log_\sigma m\rceil = 6$. The resultant re-rotated sequences were aligned using EMBOSS Needle [119] (with default parameters) and the similarity scores were compared to those of the *Original* and *Random* datasets, which were input directly to EMBOSS Needle. EMBOSS Needle gives a similarity score as a percentage representing the homology of the pair of sequences by performing optimal global sequence alignment using the Needleman-Wusch algorithm [99]. As we are testing three sets of sequences at different mutation rates, the homology percentage score will decrease with increasing mutation rate.

The results can be found in Figure 5.1. Note that nCSC and saCSC always return the same rotation because they are exact algorithms. hCSC also does a good job of finding the same rotation in most cases in this experiment but is not guaranteed to do so for every dataset. You can see that the points for each pairwise alignment overlap indicating the algorithms give the same rotation or a very similar rotation. You can observe that the score of the pairwise alignments of the *Random* dataset (before rotation) are worse in almost all cases except a few which, by coincidence, were randomly rotated close to the optimal alignment.

The results show that: (a) hCSC and saCSC yield improved similarity scores compared to those obtained from inputting *Random* datasets directly to EMBOSS Needle; and (b) hCSC and saCSC yield similarity scores that are identical or almost identical—notice that the black (Original), green (hCSC), and blue (nCSC/saCSC) points *coincide*—to those obtained from inputting *Original* datasets directly to EMBOSS Needle. This implies that algorithms hCSC, nCSC, and saCSC return the rotation maximizing the similarity score for all pairwise comparisons.

Hence, what we establish here is that the introduced distance measure coupled with the respective algorithms consistently yield a very high accuracy, compared to the standard measure [50, 99, 119], for both *low* and *high* substitution rates.

5.4.2 Experiment II: Time Performance

We then compared the time performance of the algorithms. Each algorithm was given a pair of randomly generated sequences starting from $m = n = 50$bp and doubling 8 times to a length of $m = n = 12800$bp. It was expected that the slowest algorithm would be

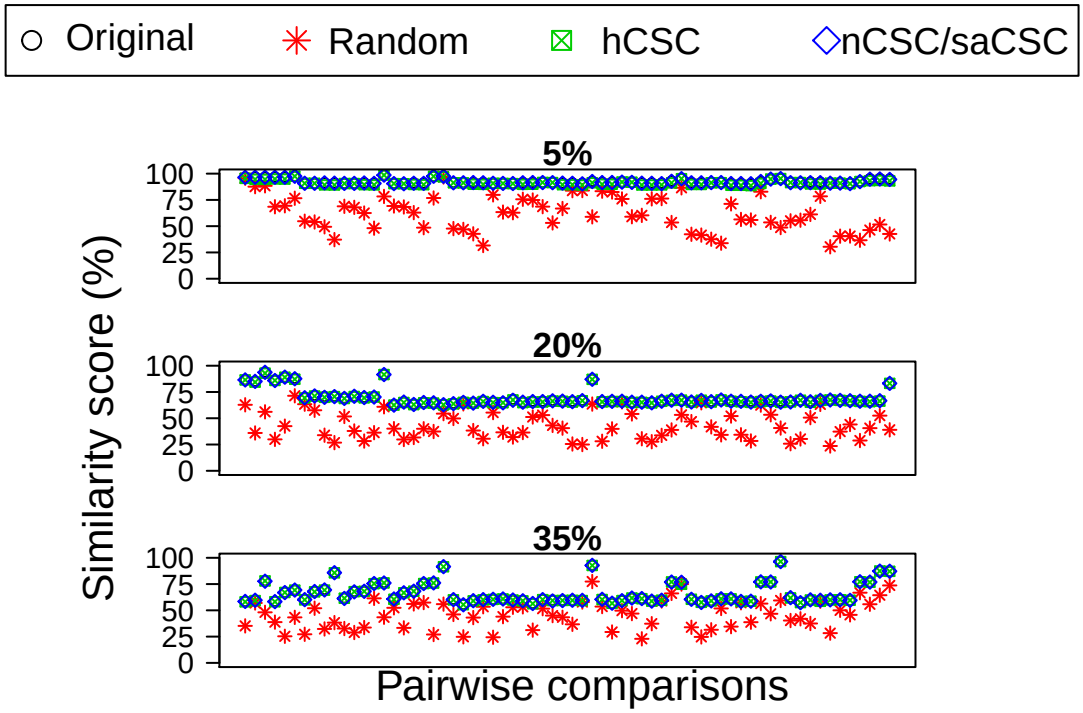


Fig. 5.1 Accuracy comparison charts for substitution rates 5%, 20%, and 35%, as titled; Each point on the chart represents a pairwise sequence comparison of one sequence and another in each of their respective datasets, resulting in 66 possible pairs being compared per substitution rate. The black (Original), green (hCSC), and blue (nCSC/saCSC) points coincide implying that algorithms hCSC, nCSC, and saCSC return the rotation maximizing the similarity score for all pairwise comparisons.

cNW which runs in time $\mathcal{O}(nm^2)$. Then it would be algorithm nCSC which runs in time $\mathcal{O}(m(m+n))$, then algorithm hCSC, which runs in time $\mathcal{O}(\beta(m+n)+\frac{m(m+n)}{\beta})$, and lastly algorithm saCSC, which runs in time $\mathcal{O}(\beta m+n)$.

Initially, β was set to $\lceil\sqrt{m}\rceil$ and q was set to $\lceil\log_\sigma m\rceil$. The reasons we decided on these parameter values was that they give some space and time complexity advantages to our algorithms. Recall the heuristic algorithm has a time complexity of $\mathcal{O}(\beta(m+n)+\frac{m(m+n)}{\beta})$ and requires extra space $\mathcal{O}(|\Sigma|^q+m+n)$. But if we set the parameters as described, we acheive a time complexity of $\mathcal{O}(\sqrt{m}(m+n))$ and space complexity $\mathcal{O}(m+n)$ as described in the analysis of the hCSC in Section 5.3.2.

And since the time complexities of hCSC and saCSC depend on β, we repeated the same experiment with these two algorithms setting β to $\lceil m/25\rceil$, giving us least 2 blocks each for $m=n=50$, and setting q to $\lceil\log_\sigma m\rceil$. Recall that q does not affect the

time efficiency of the algorithms but the number of blocks does and using $\beta = \lceil m/25 \rceil$ means more blocks will be created with respect to m compared to setting β to $\lceil \sqrt{m} \rceil$.

The results in Figure 5.2 show that for β-dependent algorithms hCSC and nCSC, the number of blocks that are processed increase the time required to finish the computations, but not by very much. When m is small and consequently the number of blocks is small, the algorithms run quickly, however, when m is longer, the difference in the number of blocks should begin to show its impact. We start off from $m = 50$ ($2 = \beta = \lceil m/25 \rceil$ and $8 = \beta = \lceil \sqrt{m} \rceil$), and double the sequence length all the way up to $m = 12800$ ($512 = \beta = \lceil m/25 \rceil$ and $114 = \beta = \lceil \sqrt{m} \rceil$). Figure 5.2 shows fluctuations in the running time depending on the number of blocks, especially for saCSC towards the right hand side of the chart. It is likely that these small differences will become more significant with much longer sequences. Establishing the right parameters, the balance between the number of blocks and block size with respect to performance and accuracy, needs further study.

Also shown in Figure 5.2 is the speed comparison of our algorithms to others. It demonstrates the orders-of-magnitude superiority of saCSC compared to cNW (circular Needleman-Wunsch) and nCSC, confirming our theoretical findings. Algorithm hCSC was found to be the second fastest though even when β was set to $\lceil \sqrt{m} \rceil$, saCSC clearly outperformed hCSC because it uses a very optimized implementation of the suffix-array construction [48], low memory overheads and a minimal number of simple linear-time operations, thus highlighting the importance of suitably implemented data structures such as suffix arrays.

More algorithms could have been included in the comparison but their (at least) quadratic time complexity [15, 88] prevents them to compete with saCSC.

5.4.3 Experiment III: Application to Synthetic Data

For evaluating the proposed methods for circular sequence comparison in some relevant application, we also implemented the following pipeline for distance-based phylogenetic reconstruction of a dataset starting with N randomly-rotated circular sequences. These are the same datasets as described in Section 4.4.2 of Chapter 4.

1. For each pair (x, y) of the N sequences, we use one method for circular sequence comparison to compute the best rotation x_i.

2. A similarity score for (x_i, y) is then computed using EMBOSS Needle (default parameters) and stored in cell $[x, y]$ of an $N \times N$ similarity score matrix.

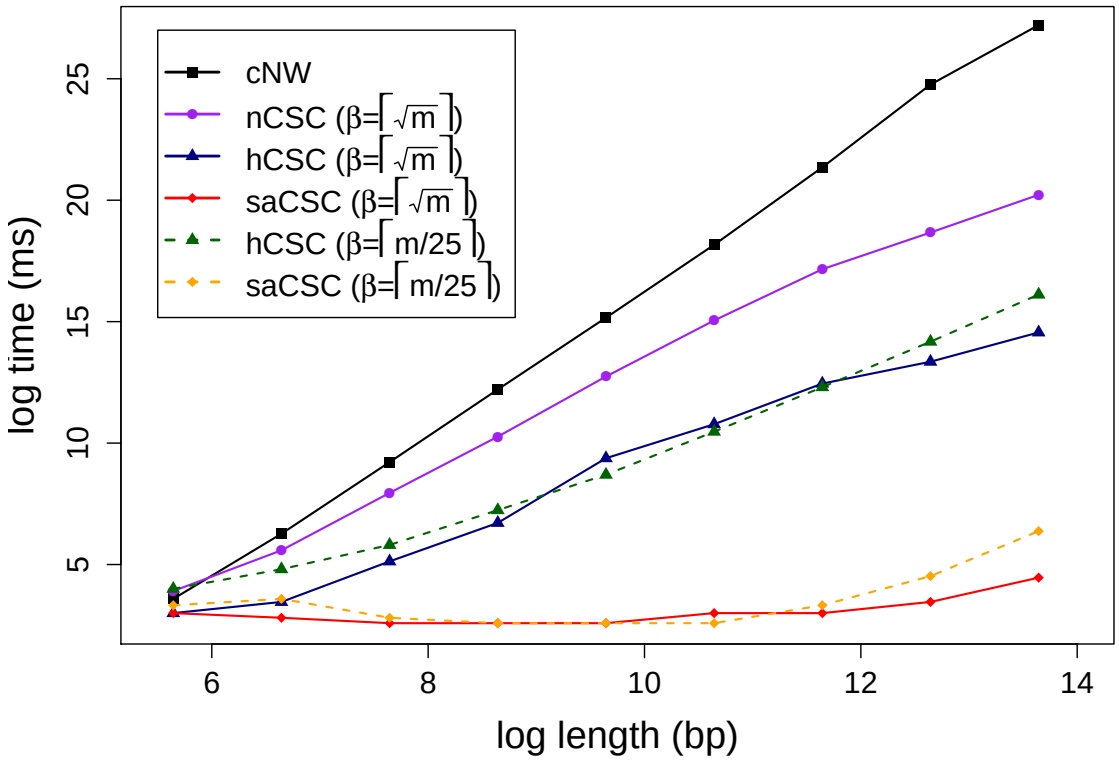


Fig. 5.2 CSC algorithms elapsed-time comparison.

3. The similarity score matrix is transformed into a distance matrix by converting each score into a distance relative to the maximum score in the similarity score matrix.

4. Neighbour joining clustering is performed on the distance matrix, using NINJA [149], to produce a phylogenetic tree.

Phylogenetic trees were constructed by NINJA [149], for the aforementioned Random datasets, using output from the following algorithms: cNW (using EMBOSS default parameters), hSW (Smith-Waterman) (using EMBOSS default parameters), and saCSCr ($\beta = 50$, $q = 5$, and $p = 1$). Notice that, output from cNW should be the same as from EMBOSS Needle (default parameters) with the *Original* datasets as input. In terms of accuracy, the *Robinson-Foulds* (RF) distance metric [121, 137] was used to compare the three resultant phylogenetic trees with the tree resulting from EMBOSS Needle (default parameters) on the *Original* and *Random* datasets, denoted by NW(o) and NW(r), respectively. RF distance is the number of elementary steps (topological

rearrangements by moving branches one by one) required to transform one tree into another, following the method defined by Robinson and Foulds [121]. If two isomorphic trees share the same topological labeling then they have an RF distance of 0. The results displayed in Table 5.9 clearly show that saCSCr and cNW produce the most accurate results with these nine datasets. As also shown in [92], hSW followed by EMBOSS Needle can sometimes result in sub-optimal global alignments.

Table 5.9 RF distances between the tree obtained from running NW(r) and those obtained after restoring the rotations with cNW, hSW, and saCSCr.

Dataset $<\alpha,\gamma,\theta,\kappa,\omega>$	NW(r)	cNW	hSW	saCSCr
$<12, 2500, 0.05, 0.06, 0.04>$	16	0	0	0
$<12, 2500, 0.20, 0.06, 0.04>$	12	0	0	0
$<12, 2500, 0.35, 0.06, 0.04>$	4	0	0	0
$<25, 2500, 0.05, 0.06, 0.04>$	44	0	0	0
$<25, 2500, 0.20, 0.06, 0.04>$	24	0	0	0
$<25, 2500, 0.35, 0.06, 0.04>$	16	0	0	0
$<50, 2500, 0.05, 0.06, 0.04>$	86	0	6	0
$<50, 2500, 0.20, 0.06, 0.04>$	84	0	0	0
$<50, 2500, 0.35, 0.06, 0.04>$	56	0	0	0

In terms of time performance, the elapsed time required for each method to process each dataset was recorded and the results are displayed in Table 5.10. It is clear, from the results presented heretofore, that saCSCr outperforms all other algorithms by at least one order of magnitude.

Table 5.10 Elapsed time comparison for algorithms cNW, hSW, and saCSCr.

Dataset $<\alpha,\gamma,\theta,\kappa,\omega>$	cNW	hSW	saCSCr
$<12, 2500, 0.05, 0.06, 0.04>$	10139.36	72.43	6.90
$<12, 2500, 0.20, 0.06, 0.04>$	9888.84	80.91	6.57
$<12, 2500, 0.35, 0.06, 0.04>$	10052.33	80.16	6.28
$<25, 2500, 0.05, 0.06, 0.04>$	46311.85	369.02	27.61
$<25, 2500, 0.20, 0.06, 0.04>$	46230.07	375.41	28.92
$<25, 2500, 0.35, 0.06, 0.04>$	46289.99	400.30	30.44
$<50, 2500, 0.05, 0.06, 0.04>$	122165.95	1563.96	125.63
$<50, 2500, 0.20, 0.06, 0.04>$	121810.69	1617.89	123.12
$<50, 2500, 0.35, 0.06, 0.04>$	120679.32	1662.82	123.77

5.4.4 Experiment IV: Application to Real Data

We have concluded thus far that using $\beta = \lceil\sqrt{m}\rceil$ and $q = \lceil\log_\sigma m\rceil$ results in a reasonable trade-off between running time and accuracy, nothwithstanding the fact that the parameters require further study. In the following section, where necessary, we adopt these values and multiply or divide them by a constant factor (factor of two), depending on the length of the input sequences. From the following results we hope to prove the algorithms we have presented here work just as well on real data as they do on the synthetic datasets described above.

DNA sequences.

Pairwise sequence comparison. As the input dataset, we used two real sequences from GenBank [12]: human (NC_001807) and chimpanzee (NC_001643) mtDNA sequences. The mtDNA genome size for human is 16,571 bp and for chimpanzee is 16,554 bp. Their pairwise sequence alignment using EMBOSS Needle with the default parameters (Gap opening penalty 10.0 and Gap extension penalty 0.5) gives a similarity of 85.1%. We used cNW (EMBOSS default parameters) to obtain the rotation of NC_001807 that maximizes its similarity score with NC_001643. This experiment took approximately 28 hours and the resultant rotation of 578 of NC_001807 improved the similarity score to 91%. This result was then compared to those obtained from saCSC (equivalent to saCSCr with $p = 0$) and saCSCr with varying parameters, displayed in Table 5.11.

Table 5.11 Rotations of Genbank mtDNA sequence NC_001807 (human) obtained when compared to NC_001643 (chimpanzee) with varying refinement of parameter p of saCSCr. The table shows that the optimal rotation 578 is found when refinement is performed using $p = 1$ blocks of length m/β as specified in the table below. The q-gram size is kept constant in this experiment.

q	β	p	**Rotation**
5	50	0	566
5	50	1	578
5	$\sqrt{m}$	0	567
5	$\sqrt{m}$	1	578
5	$2\sqrt{m}$	0	583
5	$2\sqrt{m}$	1	578
5	$\sqrt{m}/2$	0	566
5	$\sqrt{m}/2$	1	578

The convergence of the results after the additional refinement step of saCSCr demonstrates the accuracy obtained over simply using saCSC.

For clarity of presentation hereafter, instead of using β, we denote by ℓ the length of the block chosen in algorithm saCSCr.

We repeated this experiment with the human and gorilla (NC_011120) mtDNA sequences. The mtDNA genome size for gorilla is 16412 bp. Their pairwise sequence alignment using EMBOSS Needle with the default parameters gives a similarity of 83.5%. After using saCSCr to rotate sequence NC_001807, EMBOSS Needle gave a significantly improved similarity of 88.4%.

Finally, note that the experiments which used saCSC and saCSCr each took a fraction of a second to run.

Distance-based phylogenetic reconstruction. Three datasets of 16 primate, 12 mammalian and 19 mixed mammalian and primate mtDNA sequences, of average length 16,500 bp, were obtained from GenBank. We followed the same pipeline as described in Section 5.4.3. The RF distance between the trees produced by cNW (EMBOSS default parameters), and the trees produced by saCSCr ($L = \lceil \sqrt{m} \rceil = 129$, $q = 5$, and $p = 1$) followed by EMBOSS Needle (default parameters), was 0.

RNA sequences.

Eighteen viroid sequences were obtained from RefSeq, a database of curated molecular biological sequences [113]. Their lengths and target hosts vary, ranging from 348 to 371 bp and infecting peppers and citrus fruits, respectively. We followed the same pipeline as described in Section 5.4.3. The RF distance between the tree produced by cNW (EMBOSS default parameters), and the tree produced by saCSCr ($L = \lceil \sqrt{m} \rceil = 19$, $q = \lceil \log_\sigma m \rceil = 5$, and $p = 1$) followed by EMBOSS Needle (default parameters), was 0.

Protein sequences.

Linear, circularly-permuted protein sequences. Eight sequences of proteins, of average length 950 amino acids, belonging to β-glucosidase family [122] were obtained from the UniProt protein database [23]. We followed the same pipeline as described in Section 5.4.3. The RF distance between the tree produced by cNW (EMBOSS default parameters), and the tree produced by saCSCr ($L = \lceil \sqrt{m} \rceil = 31$, $q = \lceil \log_\sigma m \rceil = 5$, and

$p = 1$) followed by EMBOSS Needle (default parameters), was 0.

Naturally-occurring circular proteins. Ten bacteriocin protein sequences, of average length 20 amino acids, were obtained from Cybase [144], a database of cyclical protein sequences. We followed the same pipeline as described in Section 5.4.3. The RF distance between the tree produced by cNW (EMBOSS default parameters), and the tree produced by saCSCr ($L = 2\lceil\sqrt{m}\rceil = 10$, $q = 2\lceil\log_\sigma m\rceil = 6$, and $p = 1$) followed by EMBOSS Needle (default parameters), was 0.

5.5 Conclusion

In this chapter we introduced an effective technique for pairwise circular sequence alignment by way of using q-grams and splitting up the sequences into β-blocks to enforce locality and improve accuracy. We presented three algorithms, two exact and one heuristic algorithms, and demonstrated how they can be used on synthetic and real world examples to give accurate results. We showed the superior performance of algorithm saCSC which has a space and time complexity of $\mathcal{O}(\beta m + n)$ and demonstrated how it could give more accurate results using a *refinement* step in saCSCr. Our thorough experimentation and satisfying results led us to incorporate the algorithm into the state-of-the-art tool BEAR [8] as the default algorithm for quickly and accurately performing multiple circular sequence alignment.

Chapter 6

On-line Pattern Matching in Elastic-Degenerate Texts

This chapter presents the work done in the following publications:

1. [52] R. Grossi, C. S. Iliopoulos, C. Liu, N. Pisanti, S. P. Pissis, A. Retha, G. Rosone, F. Vayani, L. Versari, "On-Line Pattern Matching on Similar Texts", in 28th Annual Symposium on Combinatorial Pattern Matching (CPM 2017), J. R. Juha Kärkkäinen, W. Rytter, Eds., Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, pp. 9:1-9:14.

2. [110] S. P. Pissis, A. Retha, "Dictionary Matching in Elastic-Degenerate Texts with Applications in Searching VCF Files On-line", in 17th International Symposium on Experimental Algorithms (SEA 2018), G. D'Angelo, Ed., Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, pp. 16:1-16:14.

6.1 Introduction

It is possible to represent closely-related sequences that have been aligned using a multiple sequence alignment (MSA) algorithm into one compacted form that is able to represent the non-polymorphic sites (columns) of the MSA as well as the polymorphic ones [62]. This representation compresses maximal sequences of non-polymorphic sites, while the polymorphic ones, containing substitutions, insertions, and deletions of letters, are represented as a set containing all possible variants observed at that location. Consider, for instance, the following MSA of three closely-related sequences:

ATGCAACGGGTA--TTTTA

ATGCAACGGGTATATTTTA

ATGCACCTGG----TTTTA

These sequences can be compacted into a single string $\tilde{T}$ containing some *deterministic* and some *non-deterministic segments*. Note that a non-deterministic segment is a finite set of deterministic strings and may contain an empty string ε corresponding to a deletion.

$$\tilde{T} = \{\texttt{ATGCA}\} \cdot \{\texttt{A}, \texttt{C}\} \cdot \{\texttt{C}\} \cdot \{\texttt{G}, \texttt{T}\} \cdot \{\texttt{GG}\} \cdot \{\texttt{TA}, \texttt{TATA}, \varepsilon\} \cdot \{\texttt{TTTTA}\}$$

This representation has been defined in [66] as an *elastic-degenerate* (ED) text. The first five columns of the MSA all match, so when this is compacted, it creates a deterministic *segment* with a single string `ATGCA`. The next letter in the MSA is `A` for the first and second sequence but `C` for the third, making it the site of a variant. The second variant in the example is similarly a single-base substitution variant. But the third variant site consists of insertions and deletions.

The total size of an ED text, denoted by N, is simply the sum of the combined lengths of all strings in the ED text. In this case, ε counts as an individual character even though it represents a deletion. The length of an ED text, denoted by n, is calculated by summing up the lengths of deterministic segments and adding degenerate positions as having a length of one. This means each character in a deterministic segment counts as an individual position while a degenerate position counts as one position regardless of how many strings it contains and how long they are. In our example, $N = 24 = ||\tilde{T}||$ and $n = 16 = |\tilde{T}|$.

An elastic-degenerate text can represent, for example, a set of closely-related DNA sequences. For instance, a *pan-genome* [69, 100, 126, 140] is a reference sequence which is not just a single genome, but the result of an MSA of several of them that share large consensus regions and also exhibit differences at some positions consisting of letter substitutions, insertions and deletions.

The natural problem that arises is finding all matches of a deterministic pattern p of length m in text $\tilde{T}$. We call this the Elastic-Degenerate String Matching (*EDSM*) problem. The simplest version of this problem assumes that a degenerate

segment can contain only single letters [61] but this has limited practical use. Some previous work has been done on finding patterns in similar texts, such as in [102, 103] applying the *Boyer-Moore* algorithm [14] for an exact match or using the *KMP* algorithm [70] to solve the problem with a Hamming distance of 1, and in [115] by searching through a *Journaled string tree* where variants are represented by compressed tree structures. Furthermore, prior to our publications, an on-line solution we call Algorithm IKP in this thesis was presented in [66] for searching in elastic degenerate texts to provide the starting and ending positions of all matches (at a cost to performance), instead of just reporting the ending position as we do in the algorithms we present here. In either case, whether both the start and end of a match are reported or just the end alone, a caveat of searching compacted pan-genomes is the potential to find patterns that do not actually exist in the original genome sequences. This issue is resolved through a verification step which we discuss later on.

Elastic-Degenerate String Matching (EDSM)
Input: An Elastic Degenerate Text $\tilde{T}$ of total size N and a deterministic string pattern p of length $m \leq N$.
Output: A list of all the ending positions in $\tilde{T}[i]$ where p was found.

Multiple Elastic-Degenerate String Matching (MEDSM)
Input: An Elastic Degenerate Text $\tilde{T}$ of total size N and a set (dictionary) of deterministic string patterns P each of length $m \leq N$.
Output: A list of tuples $<i, j>$ with ending positions in $\tilde{T}[i]$ where $P[j]$ was found.

Various data structures to store pan-genomes have been suggested — some researchers choose to represent the variants of the combined sequence in the form of *De Bruijn* graphs [140] or specialized implementations of the data structure – *Variation Graphs* [131]. Other researchers use *Trie*-based data structures such as the *Bloom Filter Trie* in [60] or compressed *Suffix Tree* data structures [7]. All of these function as indexes to help solve the off-line version of the problem. Much effort has gone into solving this problem, due to the cataloguing of human genetic variation in the 1000 Genomes Project [138]. The 1000 Genomes Project store variants in the *Variant Call Format* (VCF) file format which has become the standard way of storing variants for pan-genomes and in next-generation sequencing. These specially-formatted, often compressed text files, in combination with a reference genome, are able to document all insertions, deletions and substitutions that occur in a population. While it is possible – for the purpose of searching for patterns – to recreate the genome of all

individuals (samples) in the pan-genome as deterministic strings, it is very impractical and requires a lot of processing power and disk space. It also defeats the purpose of storing the information in the VCF format in the first place. We were motivated to make it possible to do on-line searching of one or more patterns in a pan-genome without extracting sample sequences. Our solutions take the position of variants in the VCF file and encodes them as degenerate segments of an ED text. In this way, we are able to search a pan-genome on-line given the reference sequence and associated VCF files. We created a tool EDSO (available at https://github.com/webmasterar/edso) for creating ED files which are faster to search.

To-date, we find few attempts in the literature to solve the *on-line* version of this problem [13, 66, 102, 103]. The motivation for solving the on-line version of the problem is to remove the burden of building disk-based indexes or rebuilding them with every update in the sequences. Indexes are often cumbersome, take a lot of time and space to build, and require lots of disk space to be stored. Their usage carries the assumption that the data is static or changes very infrequently. Solutions to the on-line version can be beneficial for a number of reasons:

a) efficient on-line solutions can be used in combination with partial indexes as practical trade-offs;

b) efficient on-line solutions for exact pattern matching can be applied for fast average-case approximate pattern matching similar to standard strings;

c) on-line solutions can be useful when one wants to search for a few patterns in many degenerate texts similar to standard strings.

Our Contributions:

- In [52], we present two solutions to the EDSM problem for searching a single pattern. The first algorithm requires time $\mathcal{O}(nm^2 + N)$ after a preprocessing stage with time and space $\mathcal{O}(m)$; the second one requires time $\mathcal{O}(N \cdot \lceil \frac{m}{w} \rceil)$ after a preprocessing stage with time and space $\mathcal{O}(m \cdot \lceil \frac{m}{w} \rceil)$, where w is the size of a computer word. Thus the second algorithm requires time linear in the size of the texts representation using a pattern of length $m \leq w$.
- In [110], we present an algorithm, MultiEDSM, for searching multiple patterns of combined length M in an ED Text simultaneously, requiring time $\mathcal{O}(N \cdot \lceil \frac{M}{w} \rceil)$ with pre-processing time and space $\mathcal{O}(M)$. Any pattern length can be searched now.

- We present experimental results with both real and synthetic data confirming our theoretical findings.
- We present a real application of MultiEDSM's use as part of identifying and verifying *minimal absent words* (MAWs) in the *Homo sapiens* pan-genome. Specifically, we show that a significant number of previously discovered MAWs are in fact false-positives when a population of genomes is considered.

6.2 Definitions and Notation

In addition to the general string definitions presented in the Section 3.1, we introduce the following concepts:

6.2.1 Minimal Absent Words

We say that the string x is an *absent word* of a string y if x does not occur in y — x is absent from y. We consider absent words of length at least 2 only. An absent word x of length m, $m \geq 2$, of y is *minimal* if and only if all its proper factors occur in y but x itself does not exist in y. This is equivalent to saying that a minimal absent word (MAW) of y is of the form aub, $a, b \in \Sigma, u \in \Sigma^*$, such that au and ub are factors of y but aub is not.

Example 1. *Let* $y =$ `ABAACA`*. Its factors of lengths* 1 *and* 2 *are* `A`*,* `B`*,* `C`*,* `AA`*,* `AB`*,* `AC`*,* `BA`*, and* `CA`*. The set of MAWs of* y *(of maximum length* 3*) is obtained by combining the aforementioned factors:* {`BB`, `BC`, `CB`, `CC`, `AAA`, `AAB`, `BAB`, `BAC`, `CAA`, `CAB`, `CAC`}*.*

6.2.2 Elastic Degenerate Strings

An *elastic degenerate string* (ED string) $\tilde{X} = \tilde{X}[0]\tilde{X}[1]\cdots\tilde{X}[n-1]$, of *length* n, on an alphabet Σ, is a finite sequence of n *degenerate letters*. Every *degenerate letter* $\tilde{X}[i]$, for all $0 \leq i < n$, is a non-empty set of strings $\tilde{X}[i][j]$ with $0 \leq j < |\tilde{X}[i]|$, where each $\tilde{X}[i][j]$ is a deterministic string on Σ. The total *size* of $\tilde{X}$ is defined as

$$N = \sum_{i=0}^{n-1} \sum_{j=0}^{|\tilde{X}[i]|-1} |\tilde{X}[i][j]|.$$

By *segment*, we refer to a single degenerate position $\tilde{X}[i]$ in the ED string or a series of one or more deterministic characters that are in consensus in the pan-genome

and are (possibly) flanked by degenerate positions. The presence of ε in a degenerate segment represents a deletion of the segment, meaning patterns can be matched by skipping over or ignoring the segment. Only for the purpose of computing N does $|\varepsilon| = 1$, otherwise it simply functions as an indicator of a deletion. In practice, the use of ε can be avoided, just as variants are encoded in VCF files.

We remark that, for an ED string $\tilde{X}$, the size and the length are two distinct concepts. By the size N, we include the combined length of all strings in $\tilde{X}$ including ε. By length, we count the number of degenerate positions as having length equal to 1 and deterministic segments consisting of a single string being counted as the length of the string contained. When reporting positions of matches, we report the position ending $\tilde{X}[i]$ relative to the length of the ED string where $0 \leq i \leq n-1$.

We say that a deterministic string y of length m *matches* an ED string $\tilde{X} = \tilde{X}[0] \dots \tilde{X}[m'-1]$ of length $1 < m' \leq m$, denoted by $y \approx \tilde{X}$, if and only if string y can be decomposed into $y_0, \dots, y_{m'-1}$, $y_j \in \Sigma^*$, such that:

1. there exists a string $s \in \tilde{X}[0]$ such that a suffix of s is $y_0 \neq \varepsilon$;
2. if $m' > 2$, there exists $s \in \tilde{X}[i]$, for all $1 \leq j \leq m'-2$, such that $s = y_i$;
3. there exists a string $s \in \tilde{X}[m'-1]$ such that a prefix of s is $y_{m'-1} \neq \varepsilon$.

Note that in the above definition we require that both y_0 and $y_{m'-1}$ are non-empty to avoid degenerate cases at the beginning or at the end of an occurrence. A deterministic string y, of length m, is said have an *occurrence* ending at position i in an ED string $\tilde{T}$ if there exist $j < i$ such that $\tilde{T}[j] \dots \tilde{T}[i] \approx y$, or, if there exists $s \in \tilde{T}[i]$ such that y occurs in s.

6.2.3 Borders and Overlaps

Finding Proper Prefixes in Suffixes

A *border* of a non-empty string x of length m is a proper factor of x that is both a prefix and a suffix of x. This presents as one or more overlaps of the string on itself if we imagine a copy of the string being placed over itself and moved along from left to right, position by position, $i = 1 \dots m-1$, and all the characters at one or more positions $i \geq 1$ match to the end of the string. Consider the string $x = \texttt{ABACABA}$, the borders of the string, `ABA` and `A` are found at positions 4 and 6, respectively.

We introduce the function $\mathsf{border}(x)$ defined for every non-empty string x as the longest border of x. We define the *border table* of x, $\mathcal{B}_x$: $\{0, 1, \dots, m-1\} \to \{0, 1, \dots, m-1\}$ by $\mathcal{B}_x[k] = |\mathsf{border}(x[0 \dots k])|$, for $k = 0, 1, \dots, m-1$.

Fact 1 ([26])**.** *Given a string x of length m, the border table of x can be computed on-line in time $\mathcal{O}(m)$. All borders of x are recorded in the border table and can be specified within the same time complexity.*

We remark that the border table $\mathcal{B}_x[k]$ and the notion of border refer to a proper prefix and a proper suffix of the same string. We extend the definition of the border table function to handle a string x and one or more strings y, denoted in the manner $\mathcal{B}_{x,y}$. This refers to a string which is a prefix of a string x and a suffix of another string y, or $y = \{y_0 .. y_k - 1\}$ if y is a set of strings of size $k > 1$. For example, let $x = \{\texttt{ABACABA}\}$ and $y = \{\texttt{ABAABAABAC}, \texttt{AC}\}$. Note that $|y_0| > |x|$, so it is not possible to find a border of x in the prefix of y because it is longer than x. If $|y_i| \geq m$ then we need only consider the suffix of y_i of length $m-1$, otherwise all of it. To calculate the border table, we concatenate with distinct separator characters $\$_{0..j-1} \notin \Sigma$, x and each string in y to create $X = x\ \$_0\ y_0[\max(0, |y| - m + 1) .. |y| - 1]\ \$_1\ y_1$, and compute the overlaps in time linear to the size of X which is bounded by $\mathcal{O}(N)$, as explained in Lemma 5.

Lemma 5. *Given a pattern p of length m and $\tilde{T}$ of length n and size N, the sets $\mathcal{B}_{p,s}$ with $s \in \tilde{T}[i]$, for all $i \in [0, n-1]$, can be computed in time $\mathcal{O}(N)$.*

Proof. For each position i, we generate a string $X_i = P\$_0 s_0 \$_1 s_1 \$_2 s_2 \ldots \$_k s_k$ where $s_j \in \tilde{T}[i]$, $0 \leq j < k$, and $\$_j$'s are distinct letters not in Σ. We build the border table of string X. By traversing X from left to right we can compute $\mathcal{B}_{p,s}$. Specifically, for any string s_j, all borders that are suffixes of s_j and prefixes of p can be computed in time $\mathcal{O}(|s_j|)$ since there exists at most $|s_j|$ such borders. By Fact 1, we can build all border tables, and hence compute all $\mathcal{B}_{p,s_j}$ for all $s_j \in \tilde{T}[i]$ in time $\mathcal{O}(|p| + \sum_{j=0}^{k-1} |s_j|)$. Since the length, number of positions or segments in $\tilde{T}$ is n, sets $\mathcal{B}_{p,s}$ can be computed in time $\mathcal{O}(nm + N)$. By noting that the border table for P can be computed only once per segment and that the border table computation can be done on-line (Fact 1), the whole computation is bounded by $\mathcal{O}(N)$. □

Finding Infixes

Given a proper factor p of x of length $|p| \leq m - 2$, we can use a Suffix Tree to find the positions of all the occurrences of p in x, i.e. the position of all infixes, in time $\mathcal{O}(m + Occ_p)$.

Fact 2 ([36])**.** *Given a string x of length n, ST_x can be constructed in time and space $\mathcal{O}(n)$. Finding all occurrences Occ_p of a string p of length m in x can be performed in time $\mathcal{O}(m + Occ_p)$ using ST_x.*

The suffix tree can be used to determine the existance of p as a factor of x in time $\mathcal{O}(m)$ but finding all occurences of these positions in x requires additional time $\mathcal{O}(Occ_p)$. Let $\mathcal{L}(v)$ denote the *path-label* of a node v, i.e. the concatenation of the edge labels along the path from the root to v. We say that v is path-labeled $\mathcal{L}(v)$. Additionally, $\mathcal{D}(v) = |\mathcal{L}(v)|$ is used to denote the *string-depth* of node v. Node v is a *terminal* node if its path-label is a suffix of x, that is, $\mathcal{L}(v) = x[i..n-1]$ for some $0 \leq i < n$; here v is also labeled with index i. It should be clear that each factor of x is uniquely represented by either an explicit or an implicit node of ST_x. In standard suffix tree implementations, we assume that each node of the suffix tree is able to access its parent. Once ST_x is constructed, it can be traversed in a depth-first manner to compute $\mathcal{D}(v)$ for each node v.

Lemma 6. *Given a string y of length m and the suffix tree ST_x of a string x of length n, $\mathcal{B}_{y,x}$ can be computed in time $\mathcal{O}(m)$.*

Proof. We can traverse ST_x in linear time to find the terminal node v corresponding to the longest prefix of y which is path-labeled $\mathcal{L}(v)$. While traversing ST_x with y, we add index $n-1-i$ to $\mathcal{B}_{y,x}$ if we encounter a terminal node (leaf node) u, such that $\mathcal{L}(u) = y[i..n-1]$. The longest such prefix of y is of length at most m. No longer prefix of y can be a suffix of x as it does not occur in x. □

6.2.4 The *Shift-And* Algorithm

The *Shift-And* algorithm is an exact pattern matching algorithm that takes advantage of the parallelism of bitwise operations performed on a computer word [98]. It works by simulating a *Nondeterministic Finite Automaton* (NFA) and uses bit-level operations to simultaneously update the states of the NFA in a single CPU cycle. This offers speed-ups bounded by the number of bits in a computer word w, where, typically, on modern computer architectures, we have that $w = 64$. For short patterns where $m \leq w$, searching a text of length n runs in $\mathcal{O}(n)$ time but for longer patterns, the search takes $\mathcal{O}(n \cdot \lceil \frac{m}{w} \rceil)$ time. The pre-processing time of the algorithm is $\mathcal{O}(\sigma \cdot \lceil \frac{m}{w} \rceil + m)$ making it suitable for small constant-sized alphabets, and this complexity is dominated by m giving an $\mathcal{O}(m)$ practical time complexity. The *Shift-And* algorithm can be easily generalized for a set of patterns; it is then known as the *Multiple Shift-And* algorithm [98].

Fact 3 ([98])**.** *Given a single pattern p of length M or set of patterns P of combined length M, a string x of length N, and the computer word of size w, finding all*

occurrences of the patterns in P in x takes time $\mathcal{O}(N \cdot \lceil \frac{M}{w} \rceil)$ after pre-processing time $\mathcal{O}(M)$.

In addition to using the *Shift-And* algorithm for pattern searching, we take advantage of its ability to compute suffix/prefix overlaps between string $s \in \tilde{T}[i]$, where $\tilde{T}[i]$ is the ith segment of an ED text arriving on-line, and the set P of patterns we are searching for. Given s, we can find all the prefixes of a pattern $p \in P$ of length m by searching $s[|s|-m+1..|s|-1]$ if $|s| \geq m$ or $s[0..|s|-1]$, otherwise. This updates the NFA to mark any prefixes of the patterns present in a suffix of s. We store the states in a bit vector which we bitwise-OR to itself for each string $s \in \tilde{T}[i]$. The resulting state bit vector memorizes all the prefixes ending at position i in $\tilde{T}$, and we then use *Shift-And* in the searching stage of the algorithm to search the $(i+1)$th segment to find a suffix, that either completes the match for some pattern in P, or further extend some prefix of a pattern, in which case the algorithm updates the search state. We summarize the above description in the following fact.

Fact 4. *Given a set of strings S of total length $N = \sum_{s \in S} |s|$ and a set P of strings of total length $M = \sum_{p \in P} |p|$, computing the suffix/prefix overlaps of S and P can be done in time $\mathcal{O}(N \cdot \lceil \frac{M}{w} \rceil)$.*

6.3 Algorithms

6.3.1 Naïve EDSM for Single Pattern Matching

Main idea. Our algorithm has a preprocessing phase where we build the suffix tree of the pattern p (line 2 in the pseudocode of Algorithm 12 below). Then, in an on-line manner, we scan $\tilde{T}$ from left to right and, for each $\tilde{T}[i]$, we:

i. Memorise the prefixes of the pattern that occur at the end of $\tilde{T}[i]$ (lines 5 and 15 in pseudocode);

ii. Check whether at $\tilde{T}[i]$ it is possible to extend an occurrence of the pattern which has started earlier in the ED text (lines 17—22 in pseudocode);

iii. In both previous cases we finally check whether such an occurrence of p actually also ends in $\tilde{T}[i]$ (lines $6-8$ and $17-22$ in pseudocode).

iv. For strings $s \in \tilde{T}[i]$ of length $|s| \geq m$, we search for p in s using the KMP algorithm [70] (lines 26—28 in pseudocode).

We perform these steps by computing and storing, for each $0 \leq i < n$, the list $\mathcal{L}_i$ of the rightmost positions of prefixes of p that occur at the end of $\tilde{T}[i]$. The KMP algorithm [70] is a fast linear time exact pattern matching algorithm which we utilise for searching in strings where $|s| \geq m$, requiring $\mathcal{O}(|s|)$ time.

Below, we formally present Algorithm EDSM (Algorithm 12) that solves the *EDSM* problem in an on-line manner. Note that by $\textsc{Insert}(\mathcal{L}, \mathcal{A})$, we denote the operation that inserts the elements of a set A into a linked-list $\mathcal{L}$. Also recall that, by $\mathcal{B}_{u,v}$ we denote a border table representing the set containing all indices i, such that the prefix $u[0..i]$ of string u is also a suffix of string v.

6.3.2 Naïve EDSM Running Example

Suppose we have a deterministic pattern $p = \texttt{ACACA}$ of length $m = 5$, and an ED string $\tilde{T}$, of length $n = 6$ and size $N = 18$; the first occurrence of p starts at position 1 and ends at position 2 of $\tilde{T}$ and the second one starts at position 2 and ends at position 4.

$$\tilde{T} = \left\{ \texttt{C} \right\} \cdot \left\{ \begin{matrix} \texttt{A} \\ \texttt{C} \end{matrix} \right\} \cdot \left\{ \begin{matrix} \texttt{AC} \\ \texttt{ACC} \\ \texttt{CACA} \end{matrix} \right\} \cdot \left\{ \begin{matrix} \texttt{C} \\ \varepsilon \end{matrix} \right\} \cdot \left\{ \begin{matrix} \texttt{A} \\ \texttt{AC} \end{matrix} \right\} \cdot \left\{ \texttt{C} \right\}$$

Assume we have already computed $\mathcal{L}_0$ and $\mathcal{L}_1$, and we move to position $i = 2$, where at $\tilde{T}[i]$ we have three strings $\{s_0, s_1, s_2\}$, where $s_0 = \texttt{AC}$, $s_1 = \texttt{ACC}$ and $s_2 = \texttt{CACA}$. We generate the string $X_i = X_2 = p\$_0 s_0 \$_1 s_1 \$_2 s_2 = \texttt{ACACA}\$_0\texttt{AC}\$_1\texttt{ACC}\$_2\texttt{CACA}$ and build its border table (see line 15 in pseudocode) as shown in Table 6.1.

Table 6.1 The border table $\mathcal{B}_{p,s}$. We concatenate $p = \texttt{ACACA}$ and the strings in $\tilde{T}[2]$ to create string $X = p\$_0 s_0 \$_1 s_1 \$_2 s_2 = \texttt{ACACA}\$_0\texttt{AC}\$_1\texttt{ACC}\$_2\texttt{CACA}$.

k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$X[k]$	A	C	A	C	A	$\$_0$	A	C	$\$_1$	A	C	C	$\$_2$	C	A	C	A
B$[k]$	0	0	1	2	3	0	1	2	0	1	2	0	0	0	1	2	3

In order to Compute $\mathcal{B}_{p,s}$ (lines 5 and 15):

- we read $\mathsf{B}[7] = 2$, which gives the length of the longest string which is a prefix of p and a suffix of s_0.
- We check if there exist borders of length shorter than 2 — we read $\mathsf{B}[2-1] = 0$, telling us that no shorter border exists; so we have $\mathcal{B}_{p,s_0} = \{1\}$.

Algorithm 12 Algorithm EDSM: A naïve algorithm to solve the EDSM problem.
p: the pattern string.
m: the length of the pattern string.
$\tilde{T}$: an elastic degenerate text.
n: the length of $\tilde{T}$.

```
 1: procedure EDSM(p, m, T̃, n)
 2:     Construct ST_p
 3:     L_0 ← ∅                                                  ▷ Empty list
 4:     for s ∈ T̃[0] and s ≠ ε do
 5:         Compute B_{p,s} using the border table
 6:         INSERT(L_0, B_{p,s})
 7:         if |s| ≥ m then
 8:             Search p in s using KMP and...
 9:             report position 0 if p occurs in s and CHECKDUPLICATE(0)
10:     for i ∈ {1..n−1} do
11:         L_i ← ∅
12:         if ε ∈ T̃[i] then
13:             L_i ← L_i ∪ L_{i−1}
14:         for s ∈ T̃[i] and s ≠ ε do
15:             Compute B_{p,s} using the border table
16:             INSERT(L_i, B_{p,s})
17:             if |s| < m then
18:                 Search s in p using ST_p and...
19:                 insert starting positions into A
20:                 for (h ∈ L_{i−1}, j ∈ A) do
21:                     if h + 1 = j then
22:                         INSERT(L_i, {h + |s|})
23:             Compute B_{s,p} using ST_p
24:             if there exists (h ∈ L_{i−1}, j ∈ B_{s,p}) such that h + j + 2 = m then
25:                 Report i and CHECKDUPLICATE(i)
26:             if |s| ≥ m then
27:                 Search p in s using KMP and...
28:                 report i if p occurs in s and CHECKDUPLICATE(i)
```

- We then read $\mathsf{B}[11] = 0$, telling us that no prefix of p is a suffix of s_1, and hence $\mathcal{B}_{p,s_1} = \emptyset$.
- We read $\mathsf{B}[16] = 3$, which gives the length of the longest string which is a prefix of p and a suffix of s_2.
- We check if there exist shorter borders than 3 — we read $\mathsf{B}[3-1] = 1$, telling us that a shorter border of length 1 exists.
- Since $\mathsf{B}[1-1] = 0$, no shorter border exists. So we have $\mathcal{B}_{p,s_2} = \{0,2\}$.
- We return $\{0,1,2\}$.

Calling the function INSERT($\mathcal{L}_i, \mathcal{B}_{p,s}$) on line 16 of the pseudocode, where $i = 2$, makes $\mathcal{L}_i = \{0,1,2\}$.

Lemma 7. *Given p, of length m, and $\tilde{T}$, of length n and size N, the sets $\mathcal{B}_{p,s}$ with $s \in \tilde{T}[i]$, for all $i \in \{0..n-1\}$, can be computed in time $\mathcal{O}(N)$.*

Proof. For each position i, we generate a string $X_i = p\$_0 s_0 \$_1 s_1 \$_2 s_2 \ldots \$_{k1} s_{k1}$, where $s_j \in \tilde{T}[i], 0 \leq j < k$, and $\$_j$s are distinct letters not in Σ. We build the border table $\mathcal{B}$ of string X_i. By traversing $\mathcal{B}$ from left to right we can compute sets $\mathcal{B}_{p,s_j}$. Specifically, for any string s_j , all borders that are suffixes of s_j and prefixes of p can be computed in time $\mathcal{O}(|s_j|)$, since there exist at most $|s_j|$ such borders. By Fact 1, we can build all border tables, and hence compute all B_{p,s_j}, for all $s_j \in \tilde{T}[i]$, in time $\mathcal{O}(|p| + \sum_{j=0}^{k-1} |s_j|)$. Since the length and the total size of $\tilde{T}$ are n and N, respectively, sets $\mathcal{B}_{p,s_j}$ can be computed in time $\mathcal{O}(nm+N)$. By noting that the border table for p can be computed only once and that the border table computation can be done on-line (Fact 1), the whole computation is bounded by $\mathcal{O}(N)$. □

Lemma 8. *Given p, ST_p, and $\tilde{T}$ of length n and size N, the sets $\mathcal{B}_{s,p}, s \in \tilde{T}[i]$, for all $i \in \{1..n-1\}$, can be computed in time $\mathcal{O}(N)$.*

Proof. By Lemma 6, for any $s \in \tilde{T}[i]$, if $|s| \leq |p|$, $\mathcal{B}_{s,p}$ can be computed in time $\mathcal{O}(|s|)$ using ST_p. Since the total size of $\tilde{T}$ is N, sets $\mathcal{B}_{s,p}$ can be computed in time $\mathcal{O}(N)$. □

Lemma 9. *Lists $\mathcal{L}_i$, for all $i \in \{0..n-1\}$, in Algorithm EDSM can be computed in time $\mathcal{O}(nm^2+N)$.*

Proof. List $\mathcal{L}_0$ consists of the elements of $\mathcal{B}_{p,s}$ for position 0, which by Lemma 7 can be done within time $\mathcal{O}(N)$. For pattern p of length m, there exist at most

$m(m+1)/2 = \mathcal{O}(m^2)$ factors. Therefore, for the strings $s_j \in \tilde{T}[i], |s_j| \leq m, 0 \leq j < k$, we can find at most $\mathcal{O}(m^2)$ occurrences in pattern p. By Fact 2, finding all occurrences can be done in time $\sum_{j=0}^{k-1}(|s_j| + Occ_{s_j})$, and this is bounded by $\mathcal{O}(nm^2 + N)$ for all positions i. This is because, by definition, no $s_j, s'_j \in \tilde{T}[i]$ exist such that $s_j = s'_j$. Each occurrence can cause only one extension from $\mathcal{L}_{i-1}$ to $\mathcal{L}_i$. To avoid duplicates in $\mathcal{L}_i$, we need to check if there exist more than one prefix extensions ending at the same position. Each check can be done in constant time using a bit vector of size m (occupying space $\mathcal{O}(\lceil \frac{m}{w} \rceil)$), which we set on only once per position i. Therefore, we can extend the prefixes in time $O(m^2)$ for each position i, and in time $O(nm^2)$ for the whole text $\tilde{T}$ of length n. By Lemma 7, sets $\mathcal{B}_{p,s}$ corresponding to new prefixes of pattern p which are suffixes of $\{s_0, s_1, \ldots, s_{k-1}\}$ at position $\tilde{T}[i]$ can be found in time $\mathcal{O}(N)$. Merging new prefixes with the prefixes extended from $\mathcal{L}_{i-1}$ can be done in time $\mathcal{O}(m)$, since both are at most m. Therefore, lists $\mathcal{L}_i$, for all $i \in \{0..n-1\}$, in algorithm EDSM can be computed in time $\mathcal{O}(nm^2 + N)$. □

In our example we have to compute position $i = 4$ and we have from the previous segments that $\mathcal{L}_3 = \{1, 3\}$. The condition for reporting a match here is ($h \in \mathcal{L}_{i-1}$, $j \in \mathcal{B}_{s,p}$) where $h + j + 2 = m$ (line 24). For $s_0 = \mathtt{A}$, doing $\mathcal{B}_{A,ACACA}$ gives us $\{0\}$. As we have 3 in $\mathcal{L}_3$, we find that $3 + 0 + 2 = 5 = m$ so we report a match for p ending at position i. Moreover, for $s_1 = \mathtt{AC}$, we have $\mathcal{B}_{AC,ACACA}$ gives us $\{0, 1\}$, so we find again that the $3 + 0 + 2 = 5 = m$ is met again showing another (duplicate) occurrence of p is found ending at the same position. Since Algorithm EDSM reports all positions i in $\tilde{T}$ where at least one occurrence of p ends, and since more than one occurrence may end at the same position, we need to avoid duplications. To this end, we can use a simple operation to check whether the current position i has already been reported, CheckDuplicate(i).

We find no futher matches for p after checking all positions in $\tilde{T}$.

Theorem 3. *Algorithm EDSM solves the* Elastic-Degenerate String Matching *problem in an on-line manner in time* $\mathcal{O}(nm^2 + N)$. *Algorithm EDSM requires preprocessing time and space* $\mathcal{O}(m)$.

Proof. The correctness of the algorithm follows from the correctness of the KMP algorithm [70] if $|s| \geq m, s \in \tilde{T}[i]$; and also from the combination of Lemmas 8 and 9, if $|s| < m$ — by definition, we cannot have any other type of (ending) occurrence.

By Fact 2, the suffix tree ST_p can be computed in time and space $\mathcal{O}(m)$. By Lemma 9, lists $\mathcal{L}_i$, for all $i \in \{0..n-1\}$, can be computed in time $\mathcal{O}(nm^2 + N)$. By Lemma 8, sets $B_{s,p}$ can be computed in time $\mathcal{O}(N)$. In case $|s| < m$, we use $\mathcal{L}_{i-1}$ and

set $\mathcal{B}_{s,p}$ to find and report occurrence i in time $\mathcal{O}(m)$ using a bit vector of size m, which we initialise only once per position i. Finally, searching p in $s \in \tilde{T}[i]$, in case $|s| \geq m$, can be done in time $\mathcal{O}(|s|)$ using the KMP algorithm [70], which is bounded by $\mathcal{O}(N)$ for $\tilde{T}$ of total size N. The algorithm reads a position i and reports whether i is an ending position of some occurrence of p, before going on to read position $i+1$. Therefore, Algorithm EDSM solves the EDSM problem in an on-line manner in time $\mathcal{O}(nm^2+N)$, with preprocessing time and space $\mathcal{O}(m)$. □

6.3.3 EDSM-BV for Single Pattern Matching

We introduce algorithm EDSM-BV, a bitwise version of algorithm EDSM. Note that in the following description, when we refer to a bit vector of size m, we mean m bits, occupying space $\mathcal{O}(\lceil \frac{m}{w} \rceil)$. Furthermore, we encode the position of letters to bit vectors as they appear in the text (from left to right) instead of how they are traditionally stored in bit vectors. E.g. For EDSM-BV, if we were encoding all As in $x = \texttt{ABACBBA}$, we would create the bit vector $\mathcal{I}_A = \texttt{1010001}$.

Main idea. The main idea of this algorithm is to simulate the previous algorithm using bit-level operations to maintain linked-lists $\mathcal{L}$ and do the matching. By encoding list $\mathcal{L}_{i-1}$ in a bit vector, we can use *Shift-And* operations to find the matches at position i. We also add a further preprocessing step to augment the suffix tree of the pattern with bit vectors. This augmented suffix tree allows us to retrieve a bit vector representation of all occurrences of any $s \in \tilde{T}[i]$ in p in time linear in $|s|$. With this structure, we can use Shift-And operations to compute $\mathcal{L}_i$ from $\mathcal{L}_{i-1}$.

We preprocess p in time $\mathcal{O}(m)$ to build a border table so that we can search for the pattern using the KMP algorithm [70]. We preprocess p another way where for each letter $c \in \Sigma$, we construct a bit vector $\mathcal{I}_c$ of size m such that for $k \in \{0..m-1\}$, $\mathcal{I}_c[m-k] = 1$ if and only if $p[k] = c$. In the search stage we maintain a bit vector B of size m such that for each position $0 \leq k < m$, $\mathsf{B}[k] = 1$ if and only if $p[0..k]$ occurs at the current position in $\tilde{T}$. We also make use of temporary bit vectors B_1, B_2 and B_3 during processing. Also during preprocessing, we construct the suffix tree of p, denoted by ST_p, and augment it with m bit vectors of length m for each explicit node as follows: for each explicit node u we create bit vector M_u and we have $\mathsf{M}_u[m-k] = 1$ if and only if the factor s_u represented by node u occurs at position k in p. The occurrences of s_u can be found at the leaves in the subtree rooted at node u. We denote this augmented suffix tree of p by $\mathsf{Occ\text{-}Vector}_p$. We wish to answer the following type of on-line queries:

Given a string α, if α is a factor of p, then $\mathsf{Occ\text{-}Vector}_p(\alpha)$ finds the node w in ST_p which represents α, and returns a pointer to the bit vector M_u, where u is the

first explicit node in the subtree rooted at w. Otherwise (if α is not a factor of p), $\mathsf{Occ\text{-}Vector}_p(\alpha)$ returns a pointer to an empty bit vector. This operation can be trivially realised in time $\mathcal{O}(|\alpha|)$.

Lemma 10. *The border table for searching with KMP can be constructed in time $\mathcal{O}(m)$. Bit vectors $\mathcal{I}_c$, for each letter $c \in \Sigma$, can be constructed in time $\mathcal{O}(m+\sigma\cdot\lceil\frac{m}{w}\rceil)$ and space $\mathcal{O}(\sigma\cdot\lceil\frac{m}{w}\rceil)$. And the augmented suffix tree data structure Occ-Vector$_p$ can be constructed in time and space $\mathcal{O}(m\cdot\lceil\frac{m}{w}\rceil)$.*

Proof. Firstly, the border table is proven from the correctness of the KMP algorithm and Fact 1. Then to prepare bit vectors $\mathcal{I}_c$, we first read the alphabet and construct σ bit vectors of size m, one for each letter in the alphabet, initialised with 0s. The space for each bit vector of size m is $\mathcal{O}(\lceil\frac{m}{w}\rceil)$, so in total $\mathcal{O}(\sigma\cdot\lceil\frac{m}{w}\rceil)$ space is required. Then we only need to read the pattern once, and for each position k in the pattern such that $p[k]=c$, we set $\mathcal{I}_c[m-k]=1$. Reading the pattern once and setting $\mathcal{I}$ costs time $\mathcal{O}(m)$, so in total we need time $\mathcal{O}(m+\sigma\cdot\lceil\frac{m}{w}\rceil)$ for the preparing bit vectors $\mathcal{I}_c$. Note that we have set the bits in the bit vector one position further left as this is an optimisation that is used during the search phase described later.

By Fact 2, ST_p can be constructed in time and space $\mathcal{O}(m)$. We traverse ST_p bottom-up and allocate a bit vector $\mathcal{M}_u$ of size m for every node u we visit. If u is a leaf node representing suffix $p[k\,..\,m-1]$, we set $\mathcal{M}_u[m-k]=1$. But if u is not a leaf node, we set $\mathcal{M}_u[m-k]=1$ for all leaf nodes representing suffixes $p[k\,..\,m-1]$ in the subtree rooted at u. This can be realised by using an bitwise OR operation between the bit vectors of the children of node u, and writing the combined positions back to the bit vector $\mathcal{M}_u$. This process is done efficiently as the tree is traversed from the bottom-up from the leaves back up to (but not including) the root in post-order tree traversal. By applying this for all explicit nodes of ST_p, we build the augmented suffix tree data structure $\mathsf{Occ\text{-}Vector}_p$. We have m leaves and no more than m non-leaf explicit nodes in ST_p, so the bit vectors for ST_p can be constructed in time $\mathcal{O}(m\cdot\lceil\frac{m}{w}\rceil)$. The space required for $\mathsf{Occ\text{-}Vector}_p$ is $\mathcal{O}(m\cdot\lceil\frac{m}{w}\rceil)$ since we have $\mathcal{O}(m)$ bit vectors and each bit vector requires space $\mathcal{O}(\lceil\frac{m}{w}\rceil)$. □

Please note again that in $\mathcal{M}_u$ bits are shifted further left by one position with respect to the pattern position they refer to; this is just an optimisation that will save us doing an extra shift operation in the algorithm. Below, we formally present Algorithm EDSM-BV (Algorithm 13) that solves the Elastic-Degenerate String Matching problem in an on-line manner.

Algorithm 13 EDSM-BV: A bitwise algorithm for solving the EDSM problem.
p: the pattern string.
m: the length of the pattern string.
$\tilde{T}$: an elastic degenerate text.
n: the length of $\tilde{T}$.
Σ: the alphabet.

```
1: procedure EDSM-BV(p, m, T̃, n, Σ)
2:     Construct I_c for all c ∈ Σ
3:     for j ∈ {0..m−1} do
4:         I_{p[j]}[m−j] ← 1
5:     Construct ST_p and Occ-Vector_p
6:     B[0..m−1] ← 0
7:     for s ∈ T̃[0] do
8:         Compute B_{p,s} using the border table
9:         for b ∈ B_{p,s} do
10:            B[b] = 1
11:        if |s| ≥ m then
12:            Search p in s using KMP and...
13:            report 0 if found and CheckDuplicate(0)
14:    for i ∈ {1..n−1} do
15:        B_1[0..m−1] ← 0
16:        if ε ∈ T̃[i] then
17:            B_1 ← B_1 | B
18:        for s ∈ T̃[i] and s ≠ ε do
19:            Compute B_{p,s} using the border table
20:            for b ∈ B_{p,s} do
21:                B_1[b] ← 1
22:            if |s| < m−1 then
23:                B_2 ← B & Occ-Vector_p(s)
24:                B_1 ← B_1 | (B_2 ≪ |s|)
25:            if |s| ≥ m then
26:                Search p in s using KMP and...
27:                report i if found and CheckDuplicate(i)
28:            B_3 ← B
29:            for j ∈ {0..min(|s|, m)−1} do
30:                B_3 ← B_3 & I_{s[j]}
31:                B_3 ← B_3 ≫ 1
32:                if B_3[0] = 1 then
33:                    Report i found and CheckDuplicate(i)
34:        B ← B_1
```

Theorem 4. *Algorithm EDSM-BV solves problem* Elastic-Degenerate String Matching *in an on-line manner in time* $\mathcal{O}(N \cdot \lceil \frac{m}{w} \rceil)$. *Algorithm EDSM-BV requires preprocessing time and space* $\mathcal{O}(m \cdot \lceil \frac{m}{w} \rceil)$.

Proof. When $|s| \geq m, s \in \tilde{T}[i]$, the correctness of the algorithm follows from the correctness of the KMP algorithm [70]. By the definition of bit vectors $\mathcal{I}$, we read each $s \in \tilde{T}[i]$ letter by letter and try to extend the prefixes of p position by position using Shift-And operations. When we reach the end of the bit vector B_3 and $\mathsf{B}_3[0] = 1$ (line 32) we may find an occurrence. No other occurences can be found since we extend position by position, which means if we cannot reach the end of B_3, we must have had at least one mismatch which prevents the extension.

By Lemma 10, the time and space for the preprocessing of Algorithm EDSM-BV is bounded by $\mathcal{O}(m \cdot \lceil \frac{m}{w} \rceil)$. And for each $s \in \tilde{T}[i]$, $|s| \geq m$, searching p in s using KMP can be done in time $\mathcal{O}(|s|)$, which is bounded by $\mathcal{O}(N)$ for $\tilde{T}$ of total size N. The Shift-And operation can be done in time $\mathcal{O}(\lceil \frac{m}{w} \rceil)$ [98], and it is repeated $\min(|s|, m) - 1$ times for each s to find an occurrence. Since we choose the minimum of $|s|$ and m, this time is bounded by $\mathcal{O}(|s| \cdot \lceil \frac{m}{w} \rceil)$, which is bounded by $\mathcal{O}(N \cdot \lceil \frac{m}{w} \rceil)$ for $\tilde{T}$.

By Lemma 6, sets $\mathcal{B}_{s,p}$ can be computed in time $\mathcal{O}(N)$. Updating B for position $i = 0$ and updating B_1 for each position $i > 0$ using sets $\mathcal{B}_{p,s}$ can be done in time $\mathcal{O}(N)$ for $\tilde{T}$. For each $s \in \tilde{T}[i]$, $|s| < m - 1$, $\mathsf{Occ\text{-}Vector}_p(s)$ requires time $\mathcal{O}(|s|)$ to return the corresponding bit vector, and updating B_1 requires time $\mathcal{O}(\lceil \frac{m}{w} \rceil)$ using the Shift-And operation. Note that B_1 needs only to be updated if $\mathsf{B} \neq 0$. So for all positions in $\tilde{T}$, the time of this step is $\mathcal{O}(N + N' \cdot \lceil \frac{m}{w} \rceil)$, where N' is the number of strings s that are shorter than p. Since $N' \leq N$, the worst-case complexity resolves to $\mathcal{O}(N \cdot \lceil \frac{m}{w} \rceil)$.

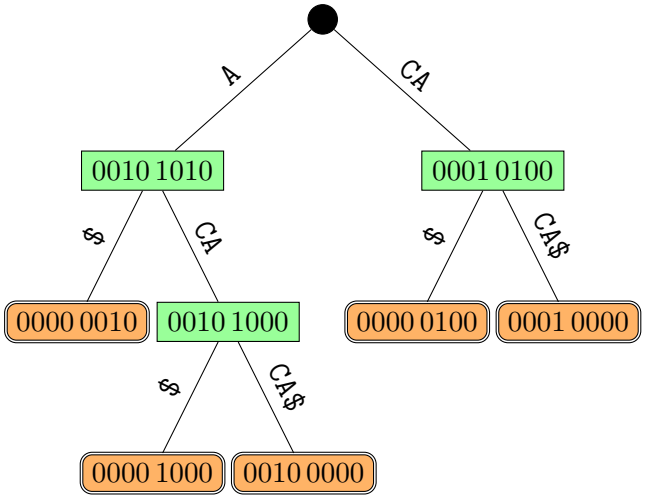
The algorithm reads a position i and reports whether i is an ending position of some occurrence of p before reading position $i + 1$. Therefore, Algorithm EDSM-BV solves problem Elastic-Degenerate String Matching in an on-line manner in time $\mathcal{O}(N \cdot \lceil \frac{m}{w} \rceil)$, with preprocessing time and space $\mathcal{O}(m \cdot \lceil \frac{m}{w} \rceil)$. □

6.3.4 EDSM-BV Running Example

We use the same example as before for the naïve EDSM running example. Suppose we have a deterministic pattern $p = \texttt{ACACA}$ of length $m = 5$, and an ED string $\tilde{T}$, of length $n = 6$ and size $N = 18$; the first occurrence of p starts at position 1 and ends at position 2 of $\tilde{T}$ and the second one starts at position 2 and ends at position 4.

$$\tilde{T} = \left\{ \texttt{C} \right\} \cdot \left\{ \begin{matrix} \texttt{A} \\ \texttt{C} \end{matrix} \right\} \cdot \left\{ \begin{matrix} \texttt{AC} \\ \texttt{ACC} \\ \texttt{CACA} \end{matrix} \right\} \cdot \left\{ \begin{matrix} \texttt{C} \\ \varepsilon \end{matrix} \right\} \cdot \left\{ \begin{matrix} \texttt{A} \\ \texttt{AC} \end{matrix} \right\} \cdot \left\{ \texttt{C} \right\}$$

In preprocessing p we have created table $\mathcal{I}$ and $\mathsf{Occ\text{-}Vector}_p$ with the suffix tree built on $p' = \texttt{ACACA\$}$ (the node for \$ coming from the root has been omitted in the diagram), shown below:



$\mathcal{I}_\texttt{A}$	0010 1010
$\mathcal{I}_\texttt{C}$	0001 0100
$\mathcal{I}_\texttt{G}$	0000 0000
$\mathcal{I}_\texttt{T}$	0000 0000

Assume we have already computed segments 0 and 1, and we move to position $i = 2$, where at $\tilde{T}[i]$ we have three strings $\{s_0, s_1, s_2\}$, where $s_0 = \texttt{AC}$, $s_1 = \texttt{ACC}$ and $s_2 = \texttt{CACA}$. We have from the previous position $i-1$, that $B = \texttt{0001 0000}$, having found a border for the prefix A.

On line 15 in the pseudo-code we calculate $B_1 = \mathcal{B}_{p,s} = \texttt{0001 1100}$ which by Lemma 7 can be done within time $\mathcal{O}(N)$.

On line 22 we try to extend previously matched prefixes. Executing $\mathsf{Occ\text{-}Vector}_p(\texttt{AC})$ returns the bit vector 0010 1000 which we bitwise-AND with B to get 0000 0000 because it does not constitute an infix at this position, leaving B_1 unaffected. By Lemma 8, this operation for a whole segment is bounded by time $\mathcal{O}(N)$. For the next string s_1, executing $\mathsf{Occ\text{-}Vector}_p(\texttt{ACC})$ is not found in the tree so it returns an empty bit vector and B_1 is again left unchanged. For $s_2 = \texttt{CACA}$, we do not run this code block because $|s_2| = m - 1$.

On line 25 we check the string is as long as p or longer. Since none of the strings in $\tilde{T}[i]$ are as long as p this code block is not executed. Searching for p in a segment where each string $|s_j| \geq m$ using the KMP algorithm takes time linear in the size of the text being searched, so $\mathcal{O}(N)$ for $\tilde{T}$ in the worst case.

From line 28 onwards we try to complete spelling the pattern continuing on from position $i-1$, using simple Shift-And bitwise operations. After assigning $B_3 = B$, we try to match character positions with those marked in $\mathcal{I}$, and we shift right one position per character to try to spell out the rest of p. If the 1 in B_3 reaches position $B_3[0]$ then we report a match for p at position i. By Fact 3 the simple updating of the search state stored in B_3 once takes time $\mathcal{O}(\lceil m/w \rceil)$ and if we do this for each character in a segment it takes time $\mathcal{O}(N \cdot \lceil m/w \rceil)$. Let us skip the explanation for extending s_0 and s_1 and concentrate on s_2 which completes the matching of p. For the first character, $j = 0$ in s_2, `C`, we bitwise-AND $\mathcal{I}_{\mathtt{C}} = \mathtt{0001\,0100}$ with $B_3 = 0001\,0000$ to get a result which we bitwise right-shift to get $B_3 = \mathtt{0000\,1000}$. We do this process again for $j = 1$, the next character `A`. $\mathcal{I}_{\mathtt{A}} = \mathtt{0010\,1010}$ is ANDed with B_3 and right-shifted so now $B_3 = \mathtt{0000\,0100}$. For $j = 2$ (`C`), $\mathcal{I}_{\mathtt{C}} = \mathtt{0001\,0100}$ is ANDed with B_3 and right-shifted so now $B_3 = \mathtt{0000\,0010}$. For the final character `A`, $j = 3$, $\mathcal{I}_{\mathtt{A}} = \mathtt{0010\,1010}$ is ANDed with B_3 and right-shifted resulting in $B_3 = \mathtt{0000\,0001}$. Note there is a 1 now at position 0 in the bit vector, so when $B_3[0] = 1$ we report a match for p at position $i = 2$.

The final line of the algorithm (line 34) copies bit vector B_1 to B, saving the current state of the search, ready for the next segment to be searched on-line. We have $B = \mathtt{0001\,1100}$ and go to position $i = 3$.

Calculating the border table for the segment makes $B_1 = \mathtt{0000\,0000}$ because `C` does not have a border with p. However, we find ε in this segment, indicating it may be treated as a deletion, so on line 17 we copy the state from the last segment into this one, $B_1 = B = \mathtt{0001\,1100}$.

Executing $\mathsf{Occ\text{-}Vector}_p(\mathtt{C})$ returns the bit vector `0001 0100` which we bitwise-AND with B to get `0001 0100`, then we right-shift by $|s_0| = 1$ position so $B_1 = \mathtt{0000\,1010}$. What we have done is extend `ACA` from s_3 in the previous segment to become `ACAC`, as well as `A` to `AC`. We bitwise-OR B_1 back to itself to update the state, so now $B_1 = \mathtt{0001\,1110}$.

Trying to find a suffix from state B does not find a match. So we finish by assigning $B = B_1$. We now go on to position $i = 4$ in $\tilde{T}$.

Calculating the border table for the segment makes $B_1 = \mathtt{0001\,1000}$. Then calling $\mathsf{Occ\text{-}Vector}_p(\mathtt{A})$ returns the bit vector `0010 1010` which when ANDed with B and right-shifted $|s_0| = 1$ positions produces `0000 0101`. Calling $\mathsf{Occ\text{-}Vector}_p(\mathtt{AC})$ returns `0010 1000`, which after the operations produces `0000 0010`. Thus, by the end of this segment, after bitwise ORing B_1 with the results, $B_1 = \mathtt{0001\,1111}$.

When we try to complete the suffix from the previous position $i-1$ with B, we are able to match $\mathcal{I}_{\mathtt{A}} = \mathtt{0010\,1010}$ for both s_0 and the prefix of s_1. Performing bitwise

operations $\mathcal{I}_{\mathtt{A}}$ & B gives $\mathtt{0000\,1010}$ which when shifted right makes $B_3 = \mathtt{0000\,0101}$ and $B_3[0] = 1$ so we report a match for p at position i. Although this is discovered twice for the segment we report it only once by CheckDuplicate(i)

There are no further matches for p in $\tilde{T}$ so we will stop here. By the end of the search we have reported matches ending at positions 2 and 4.

6.3.5 Multi-EDSM for Multiple Pattern Matching

Note that in this algorithm we encode letters in the reverse order that they appear, as is the most common convention used for bit vector algorithms, with the string index of a given string x corresponding with the bit vector index of a given bit vector $\mathcal{I}_a$ for letter a, then $\mathcal{I}_a[i] = 1$ for all positions $0 \leq i < |x|$ where $x[i] = a$.

We first define a data structure, the Occurrences Vector Data Structure, based on the suffix tree, which when queried with a simple string α, solves the following general problem:

Occurrences Vector Data Structure (*OccVec*)
Input: A string x of length n.
Output: Given a string α on-line, return a pointer to a bit vector B, with $\mathsf{B}[i] = 1$ marking all the positions α occurs in x for every position i, otherwise $\mathsf{B}[i] = 0$.

In what follows, we show how we arrive to the following lemma, Lemma 11, by way of Lemmas 12 and 13.

Lemma 11. *Given an integer parameter τ, where $1 \leq \tau \leq \lceil n/w \rceil$, a data structure of size $\mathcal{O}(\lceil n/\tau \rceil \cdot \lceil n/w \rceil)$ can be constructed in time and space $\mathcal{O}(\lceil n/\tau \rceil \cdot \lceil n/w \rceil)$ answering OccVec queries in time $\mathcal{O}(|\alpha| + \tau)$.*

It is possible to construct a binary tree from an k-ary tree, such as a suffix tree, as mentioned in the following Lemma:

Lemma 12 ([3, 47])**.** *Given a k-ary tree, where k represents edge cardinality and $k \geq 3$, its equivalent binary tree can be generated in time and space $\mathcal{O}(n)$.*

Then the tree can be decomposed/partitioned into n/τ micro-macro trees with the following properties:

Lemma 13 ([3, 44])**.** *A linear time algorithm exists to partition a tree $\mathcal{T}$ with cardinality $k \leq 3$ into n/τ disjoint connected subgraphs called micro trees. Each micro tree contains up to τ nodes and upper and lower boundary nodes connecting them with other micro trees.*

Thus we arrive to the data structure of Lemma 11 over string x, which we refer to by Occ-Vector$_x$. Observe that, if we set $\tau = 1$, we essentially have the $\mathcal{O}(n \cdot \lceil n/w \rceil)$-sized data structure proposed by Grossi et al. in [52], which is constructible in time and space $\mathcal{O}(n \cdot \lceil n/w \rceil)$. And if we set $\tau = \lceil n/w \rceil$, we create a data structure of size $\mathcal{O}(\frac{n}{\tau} \cdot \tau) = \mathcal{O}(n)$ requiring time $\mathcal{O}(|\alpha| + \tau)$ to answer queries.

Pre-processing I: Construction of the *OccVec* Data Structure.

We now describe how we construct the *OccVec* data structure, which is one of the preprocessing steps of the Multi-EDSM algorithm.

We start by constructing the suffix tree ST_x of x. By Fact 2, this can be done in time and space $\mathcal{O}(n)$.

We next create a binary tree conversion of ST_x, which we refer to as $\mathcal{T}_x$, using a standard linear time procedure (see [44, 47], for instance). We process each explicit node of ST_x with out-degree $k > 2$ as follows. Let v be a node with children $u_1, \ldots, u_k$, and $k \geq 3$. We replace v with $k-1$ new nodes $v_1, \ldots, v_{k-1}$; make u_1 and u_2 be the left and right children of v_1, respectively; and for each $\ell = 2, \ldots, k-1$, we make $v_{\ell-1}$ and $u_{\ell+1}$ be the left and right children of v_ℓ, respectively. If v is not the root of ST_x then we set the parent of v_{k-1} to be the parent of v; otherwise, v_{k-1} is the root. This procedure can create a tree at most double the size of ST_x, so $\mathcal{T}_x$ is still is in $\mathcal{O}(n)$ as mentioned in Lemma 12. (Note that we can keep a copy of the original ST_x and create a pointer for each node from the original to its binary tree node version).

We next rely on the classic notion of *micro-macro* tree decomposition as mentioned in Lemma 13. By decomposition we mean partitioning branches of the tree such that each branch has a similar number of nodes or is restricted by a maximum number, τ. We apply the process of decomposition on $\mathcal{T}_x$. Let τ be some integer input parameter, $1 \leq \tau \leq \lceil n/w \rceil$. We decompose $\mathcal{T}_x$ in $\mathcal{O}(n/\tau)$ disjoint subgraphs called *micro trees*. Each micro tree is of size *at most* τ and contains *up to two* boundary nodes that are adjacent and connect it to other micro trees. The topmost of these boundary nodes is the *root* of the whole micro tree, and the other one is called the *bottom* boundary node. Such a decomposition is always possible and can be found in time $\mathcal{O}(n)$ (see [3] for more details).

For each boundary node v of a micro tree, we store a bit vector b_v, where $b_v[i] = 1$, if the terminal node representing the ith suffix of x is a descendant of v, and otherwise $b_v[i] = 0$. For the bottom boundary node v of micro tree t, b_v can be computed by merging the bit vectors from the roots of the micro trees that are adjacent to v and

then add manually the terminal nodes within micro tree t for the root boundary node of t. By the above description and the fact that we have $\mathcal{O}(n/\tau)$ micro trees, the total size of Occ-Vector$_x$, and therefore the time to construct it, are bounded by $\mathcal{O}(n+\lceil n/\tau \rceil \cdot \lceil n/w \rceil) = \mathcal{O}(\lceil n/\tau \rceil \cdot \lceil n/w \rceil)$, for $1 \leq \tau \leq \lceil n/w \rceil$. Occ-Vector$_x$ also includes a linked-list $\mathcal{L}$ of integers $0, 1, \ldots, n-1$ used to maintain the bit vectors when a new query arrives. This completes the construction.

Querying the *OccVec* Data Structure.

The *OccVec* data structure, after having been constructed in a preprocessing step, is then queried many times in the search stage of the Multi-EDSM algorithm. While the memory of how the data structure was constructed is fresh in our minds, we take the opportunity in this section to explain how we query it.

Given a pattern α, we spell the pattern from the root of ST_x until we reach the last explicit node v. By Fact 2 this takes time $\mathcal{O}(|\alpha|)$ for constant-sized alphabets. If we inspect Figure 6.1 we see there are then two cases to consider:

1. If v in the corresponding binary tree $\mathcal{T}_x$ is a boundary node (either top or bottom) of some micro tree, we simply return a pointer to b_v; this takes constant time.

2. If v is not a boundary node – we will refer to it as v' from now on – we can find the bottom boundary node u (if it exists) of the micro tree v' in constant time. We create a new empty bit vector b'_v and if node u exists, we set $b'_v = b_u$. Then we traverse the micro tree rooted at v', obtaining the suffix number of each leaf node and setting the corresponding bits in b'_v. We then return a pointer to the updated b'_v; this takes extra time $\mathcal{O}(\tau)$.

So in the first case (1), it's simply a matter of spelling out α and returning a pointer to the bit vector b_v at the boundary node. This process takes time $\mathcal{O}(|\alpha|)$.

The second case (2) is more complicated. In following the route through the tree whilst spelling out α, we can store a pointer to the last top boundary node v we visited. In this way it is easy to get back to it to find the adjacent bottom boundary node u in constant time. We create a new bit vector and set it $b'_v = b_u$ if it exists, taking time $\mathcal{O}(\lceil n/w \rceil)$. The intuition of using b_u instead of b_v is that the upper boundary node v contains all the positions of b_u combined with the positions of the leaves in the micro tree rooted at v, however this includes more positions than are warranted in spelling out α, i.e. it would include extra leaves of prefixes of α. To avoid this problem, we combine the bits of b_u and the bits obtained from traversing the leaves of the micro

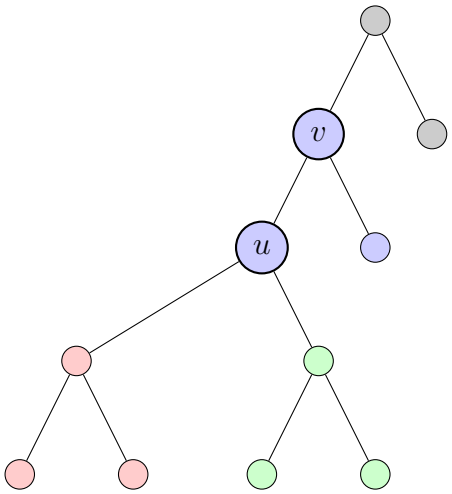


Fig. 6.1 Three micro trees: the topmost in light blue and the bottommost ones in light red and light green. If the query reaches node v or u, then the bit vector at that node is returned. If the query finds itself inside a micro tree at a node v' other than v or u, then the node itself and those below that are traversed and the positions of the leaves rooted at v' are combined with the bit vector of the bottom boundary node u and returned.

tree rooted at v', the node which we reached in spelling α. As mentioned before, by virtue of the process of micro-macro tree decomposition, there can be no more than τ leaves rooted at v', so traversing these takes no more than $\mathcal{O}(\tau)$ extra time.

Hence, any *OccVec* query requires at most time $\mathcal{O}(|\alpha| + \tau)$. By setting $\tau = \lceil \frac{n}{w} \rceil$ in the above description, we ultimately arrive at Lemma 11. We are now ready to explain the pre-processing steps required for the Multi-EDSM algorithm.

Pre-processing II: Pattern Set Preparation for *Multiple Shift-And* Search and Other Indexes.

This pre-processing stage of our algorithm consists in pre-processing the set P of $k \geq 1$ patterns. We view the set P as the concatenation of its elements to form a new string y of length M. The first step is the pre-processing of the pattern set P of combined length M in the *Mutiple Shift-And* algorithm [98]. We create σ bit vectors of size $\lceil \frac{M}{w} \rceil$ and for each letter $c \in \Sigma$ we set $\mathcal{I}_c[i] = 1$ if $y[i] = c$, otherwise $\mathcal{I}_c[i] = 0$. Therefore this first step requires time and extra space $\mathcal{O}(M + \sigma \cdot \lceil \frac{M}{w} \rceil) = \mathcal{O}(M)$, which is bounded by M for constant-sized alphabets.

The second step is a simple application of Lemma 11 over string y', that is the concatenation of all k patterns in P separated by a character $\$ \notin \Sigma$, resulting in a new

string y' of length $M+k-1=\mathcal{O}(M)$. We also keep an array of the same size that allows us to map the position of a character in y' to the corresponding position in P in constant time. We build $\mathsf{ST}_{y'}$, hereafter refered to simply as ST_P, and its corresponding Occ-Vector$_P$ data structure by setting $\tau=\lceil\frac{M}{w}\rceil$. We take the additional steps of filtering out non-infix positions and subtracting 1 from the index positions. This makes it possible to maintain infix extensions during the search stage and also includes an optimisation for the search stage. By setting $\tau=\lceil\frac{M}{w}\rceil$ during its construction, the data structure will have size and take time $\mathcal{O}(M)$ to build, and will consequently result in an $\mathcal{O}(|\alpha|+\lceil\frac{M}{w}\rceil)$ query time for the search stage.

The total time and space for the pre-processing stage are thus in $\mathcal{O}(M)$.

Theorem 5. *Algorithm Multi-EDSM solves the* Multiple Elastic-Degenerate String Matching *problem in an on-line manner in time* $\mathcal{O}(N\cdot\lceil\frac{M}{w}\rceil)$. *Algorithm Multi-EDSM requires pre-processing time and space* $\mathcal{O}(M)$.

Notably, our algorithm improves on the pre-processing time and space of [52] by a factor of $\mathcal{O}(\lceil\frac{m}{w}\rceil)$:

Corollary 1. *Algorithm Multi-EDSM solves the* Elastic-Degenerate String Matching *problem in an on-line manner in time* $\mathcal{O}(N\cdot\lceil\frac{m}{w}\rceil)$. *Algorithm Multi-EDSM requires pre-processing time and space* $\mathcal{O}(m)$.

Searching with Multi-EDSM.

After describing the pre-processing steps above, below, we formally present Algorithm Multi-EDSM (Algorithm 14) and explain how the search stage of the algorithm works.

The algorithm maintains the state of the search in bit vector B in between searches of individual segments and uses temporary bit vectors B_1, B_2 and B_3 to update the state during the search. By $m_{\min}$ and $m_{\max}$ we denote the length of the shortest and longest patterns in pattern set P, respectively. We report unique matches in the form of a tuple $<i,j>$, representing the ith position in $\tilde{T}$ for $0\leq i\leq n-1$ and the jth pattern in P for $0\leq j\leq k-1$.

In the first segment of the ED string, for every string $s\in\tilde{T}[0]$ of length $|s|\geq m_{\min}$, we call the Multi-Shift-And-Search function (line 8) with a fresh state to find any patterns that occur in s. In any case, the Overlaps function is called for computing the suffix/prefix *overlaps/borders* between every string $s\in\tilde{T}[0]$ and the set P of patterns

Algorithm 14 Multi-EDSM: A bitwise algorithm to solve the MEDSM problem.
P: a set of pattern strings.
$\tilde{T}$: an elastic degenerate text.
n: the length of $\tilde{T}$.
Σ: the alphabet.

```
1: procedure MULTI-EDSM(P, T̃, n, Σ)
2:     Construct I_c for all c ∈ Σ and MULTI-SHIFT-AND-PREPROCESS(P)
3:     Construct ST_P and OCC-VECTOR_P
4:     Construct and initialise bit vectors B, B_1, B_2 and B_3 to 0
5:     for s ∈ T̃[0] do
6:         if |s| ≥ m_min and s ≠ ε then
7:             B_3 ← 0
8:             MULTI-SHIFT-AND-SEARCH(s, B_3) and...
9:             report <0, j> if found and CHECKDUPLICATE(0)
10:    B ← OVERLAPS(T̃[0])
11:    for i ∈ {1..n−1} do
12:        B_1 ← OVERLAPS(T̃[i])
13:        if ε ∈ T̃[i] then
14:            B_1 ← B_1 | B
15:        for s ∈ T̃[i] and s ≠ ε do
16:            ▷ Pattern suffix completion / full pattern searching
17:            B_2 ← B
18:            MULTI-SHIFT-AND-SEARCH(s, B_2) and...
19:            report <i, j> if found and CHECKDUPLICATE(i)
20:            ▷ Maintain valid infix positions
21:            if |s| ≤ m_max − 2 then
22:                B_3 ← B & OCC-VECTOR_P(s)
23:                B_1 ← B_1 | LEFT-SHIFT(B_3, |s|)
24:        B ← B_1
```

we are searching for. The function essentially memorises the prefixes starting in the current segment using B. By Facts 3 and 4, lines 5–10 require $\mathcal{O}(N \cdot \lceil \frac{M}{w} \rceil)$ time. For subsequent degenerate positions, we use the state of B from the previously searched letter to continue the search with Multi-Shift-And-Search (line 18). This time the function is called regardless of the length of s because it is used to find whole patterns as well as prefixes of patterns that began in the previously searched letters and whose suffixes end in the current position. By Facts 3 and 4, this requires $\mathcal{O}(N \cdot \lceil \frac{M}{w} \rceil)$ time across all positions in the ED string.

Then we consider how to handle infixes (see line 21). We only need to process strings short enough to be considered as infixes of a pattern and we query them with the Occ-Vector$_P$ data structure to mark the positions where each infix starts. Querying Occ-Vector$_P$ for a string s requires $\mathcal{O}(|s| + \tau)$ time by Lemma 11. We bitwise-AND the bit vector it returns with B to maintain only the states started or continued from the previous letter. On the next line, we use the Left-Shift function to update the position of the bits to reflect the state of the search while ensuring that the bits are not shifted past the end of a pattern. What bits remain as 1s are bitwise-ORed with B_1 to update the state to maintain partial search states. This takes time $\mathcal{O}(\lceil \frac{M}{w} \rceil)$ per character or $\mathcal{O}(N \cdot \lceil \frac{M}{w} \rceil)$ for $\tilde{T}[i]$.

The final line of the algorithm saves the final search state of the segment to bit vector B, ready for the next on-line letter to be sent to the Multi-EDSM function. A full illustrative example of the search stage is provided below.

6.3.6 Multi-EDSM Running Example

Given an ED string $\tilde{T}$ as shown below, we wish to search for the patterns in set $P = \{\texttt{ATAT}, \texttt{TAGA}\}$ of combined length $M = 8$. A collection $\mathcal{I}$ of σ bit vectors are created during the Shift-And pre-processing stage marking the positions of the letters of the concatenated patterns. We also build Occ-Vector$_P$. Just to remind the reader, the bit vectors should be read from right to left and recall that Occ-Vector$_P$ subtracts 1 from the index positions.

$$\tilde{T} = \left\{ \begin{matrix} \texttt{AT} \\ \texttt{A} \end{matrix} \right\} \cdot \left\{ \begin{matrix} \texttt{AT} \\ \texttt{TA} \end{matrix} \right\} \cdot \left\{ \begin{matrix} \texttt{TTTA} \\ \texttt{AGA} \end{matrix} \right\}$$

$\mathcal{I}_{\texttt{A}}$	1010 0101
$\mathcal{I}_{\texttt{C}}$	0000 0000
$\mathcal{I}_{\texttt{G}}$	0100 0000
$\mathcal{I}_{\texttt{T}}$	0001 1010

The algorithm starts with $\tilde{T}[0]$, skips the Shift-And search of the strings in the segment because they are too short, and computes bit vector $\mathsf{B} = 0001\,0011 =$

OVERLAPS($\tilde{T}[0]$) on line 10. The OVERLAPS function memorises the prefixes starting in the current segment into B.

Then, the next segment is considered, $i = 1$; on line 12 we compute the bit vector $\mathsf{B}_1 = \texttt{0011 0011} = \text{OVERLAPS}(\tilde{T}[1])$. The next step is to check each string $s \in \tilde{T}[1]$ and after doing MULTI-SHIFT-AND-SEARCH(AT, B_2) with state B (from $\tilde{T}[0]$) it discovers a match for $P[0]$ and reports it. This time the Shift-And search function is called regardless of the length of s because it is used to find whole patterns as well as suffixes of patterns that began in the previous segments. The function completes the suffix by matching AT at positions $\mathcal{I}_{\mathtt{A}}[2]$ and $\mathcal{I}_{\mathtt{T}}[3]$ to spell out $P[0]$.

Then we consider how to handle infixes on line 21. We only need to process strings short enough to be considered as infixes of a pattern and we query them with the OCC-VECTOR$_P$ data structure to mark the positions where each infix starts. Querying OCC-VECTOR$_P$ requires time $\mathcal{O}(|s| + \tau)$ per query by Lemma 11. Calling OCC-VECTOR$_P$(AT) finds infix position 2 and returns 0000 0010. So $\mathsf{B}_3 = \texttt{0000 0010} = \mathsf{B}$ & 0000 0010, thus maintaining the active search state. The LEFT-SHIFT function does bitwise left-shift of B_3 by $|s|$ positions whilst ensuring no 1s end up at or beyond the ending position of each pattern in the set. This function requires $\mathcal{O}(|s| \cdot \lceil \frac{M}{w} \rceil)$ time for any given s. What bits remain as 1s are bitwise-ORed with B_1 to update the state to maintain partial search states, but in this case, the 1 is shifted too far and B_1 remains unchanged.

In the next iteration, we do MULTI-SHIFT-AND-SEARCH(TA, B_2) yielding no match. Then we call OCC-VECTOR$_P$(TA) which finds infix position 1 and returns 0000 0001 which we bitwise-AND with B to take $\mathsf{B}_3 = \texttt{0000 0001}$. Then LEFT-SHIFT is performed on B_3 and bitwise-ORing its result with B_1 makes $\mathsf{B}_1 = \texttt{0011 0111}$ because no boundaries are crossed. Having searched all the strings in the segment, we save the state of the search to B on line 24 and observe that we have thus far matched ATA of $P[0]$ spanning across $\tilde{T}[0]$ and $\tilde{T}[1]$.

Now we go ahead and search the final segment $\tilde{T}[2]$ of our example. We compute $\mathsf{B}_1 = \texttt{0010 0001} = \text{OVERLAPS}(\tilde{T}[2])$ first and then by calling MULTI-SHIFT-AND-SEARCH(TTTA, B_2) with the state $\mathsf{B}_2 = \mathsf{B}$ from the previous segment, we complete the partial match and report finding $P[0]$ in this segment. Calling this function again for the next string in the segment, MULTI-SHIFT-AND-SEARCH(AGA, B_2) also completes the suffix for $P[1]$, and we report it.

On the very last line of the algorithm we save the final search state of the segment to bit vector B, ready for the next on-line letter to be sent to the SEARCH function.

In conclusion, both patterns in P where discovered in time $\mathcal{O}(N \cdot \lceil \frac{M}{w} \rceil)$ and space $\mathcal{O}(M)$.

6.3.7 Match Verification

Multi-EDSM reports matches in the form $<i, j>$, where i is the ending position in the ED text corresponding to the position in the reference genome and j is the jth pattern in P. However, by the nature of ED strings, a match can be reported that does not really exist in any individual sequence of the pan-genome. It is therefore necessary to verify matches and confirm that a pattern $P[j]$ occurs in at least one of the genome sequences.

VCF files record the bases of each sample (individual) for every variant position relative to the reference genome sequence. Except for chromosome Y, the VCF files are *phased*, containing information for both alleles from either parent of each sample. For example, if we look at the VCF record for the Single Nucleotide Polymorphism (SNP) in chromosome 21 at position 9,411,410, we see that the reference base (REF) is `C`, identified by index 0, and alternative (ALT) non-reference alleles identified by index 1 and so on for every variant, separated by a comma. In this record there is only one alternative variant, `T`. Sample HG00096 is heterozygous — `0|1` indicates that one allele has `C` and the other allele `T`. Sample HG00097 is homozygous for `T` and sample HG00143 is homozygous for `C`.

Table 6.2 The VCF record for variant $rs78200054$ in chromosome 21 at position 9,411,410 with some columns ommitted.

#CHROM	POS	ID	REF	ALT	HG00096	HG00097	HG00143
21	9411410	rs78200054	C	T	0|1	1|1	0|0

When the variants are applied for these samples, they produce the strings shown in Table 6.3.

Table 6.3 For reference pattern $P[j] =$ `TTTGTCTATC` of length $m = 10$ and ending position $i = 9,411,410$, the table shows the variants of position 21 : 9411410 applied to each allele of samples HG00096, HG00097 and HG00143.

	allele 0	allele 1
HG00096	TTTGTCTATC	TTTGTCTATT
HG00097	TTTGTCTATT	TTTGTCTATT
HG00143	TTTGTCTATC	TTTGTCTATC

We use the tool PyVCF (built on PySAM and SAMtools [80]) to query the VCF files with the range $i-m+1 \ldots i+1$, where $m = |P[j]|$, which returns a list of zero or more variants, $\mathcal{L}_v$, present in the VCF file in this range. We also read the reference genome sequence to get the starting state of the sequence of length at least m ending at position i. We refer to this from hereon as the *original* sequence.

Each individual is assigned a pair of original sequences to represent their alleles as a starting point before applying the variants. We check each individual and recreate the differences in their alleles by applying the variants in $\mathcal{L}_v$, using the position of each variant to calculate the relative location at which to substitute the original base(s). Of course, we only need to make a substitution if the allele index isn't the reference (REF) base (index `0`). We use a variable k (initialised to 0) for each allele to correct for the relative position we insert variants at because the position changes if the reference and variant are of different lengths. For example, if a variant is shorter than the reference then we need to delete the difference in the number of bases from the allele's original sequence, so we decrement k accordingly and use it to modify the calculation of the next variant's relative location. In the case where a variant is longer than the reference, we increment k by the difference.

Because it is possible for allele sequences to be longer than the pattern after applying all the variants, we need to check the sequences using simple linear time exact string matching. If $P[j]$ is found in any allele for any sample, we say the match has been *verified*.

6.4 Experiments

6.4.1 EDSM and EDSM-BV Results

We have implemented Algorithms EDSM and EDSM-BV in the `C++` programming language. The implementation of the algorithm presented in [66], IKP, was taken from https://github.com/Ritu-Kundu/ElDeS. Algorithm IKP solves the Elastic-Degenerate String Matching problem in time $\mathcal{O}(\alpha\gamma mn + N)$ and space $\mathcal{O}(N)$; where α and γ are parameters, respectively representing the maximum number of strings in any degenerate position of the text and the maximum number of degenerate positions spanned by any occurrence of the pattern in the text. Recall that Algorithm IKP outputs both the starting and ending positions of pattern occurrences, while the output of Algorithms EDSM and EDSM-BV is only the ending positions. All three programs were compiled with `g++` version 4.7.3 at optimisation level 3 (-O3). The

following experiments were conducted on a desktop computer using one core of Intel® Core™ i7-2600S CPU at 2.8GHz and 8GB of RAM under 64-bit GNU/Linux. We compared the performance of EDSM-BV, EDSM and IKP using synthetic data, as well as the performance of EDSM-BV – shown to be the fastest – using real data. The implementation of EDSM-BV is available at https://github.com/webmasterar/edsm under the terms of the GNU General Public License. The synthetic datasets referred to in this section are maintained at the same web-site but were created with a tool obtainable from https://github.com/webmasterar/EDSRand.

Note that for the synthetically-generated sequences used in our experiments, DNA characters and positions were chosen pseudo-randomly from a uniform distribution.

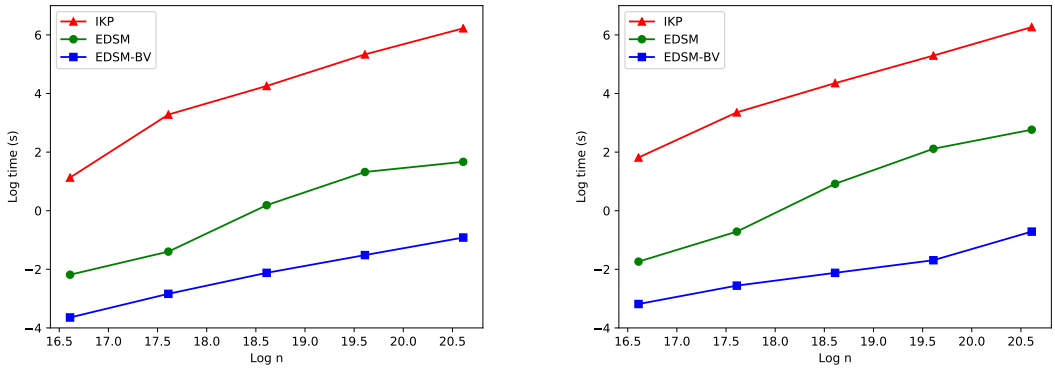
Experiment I: Synthetic Data

Synthetic ED strings on the DNA alphabet were created with n ranging from $100,000$ to $1,600,000$ positions and the percentage of indeterminate segments set to 10%. For each indeterminate segment within the synthetic ED strings, the number of strings was chosen randomly, with an upper-bound set to 10. The length of each string of a degenerate position was chosen randomly, with an upper bound again set to 10. Every non-degenerate position within the synthetic ED texts contained a single letter. Four different pseudo-randomly generated patterns of length $m = 8$, 16, 32, or 64 were given as input to all three programs, along with the aforementioned synthetic ED texts, resulting in four sets of output.

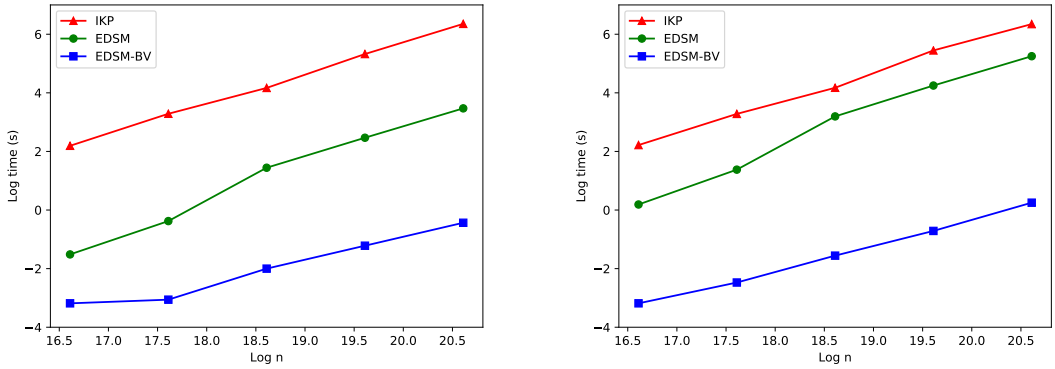
Our theoretical findings showing that Algorithms EDSM-BV and EDSM are asymptotically faster than Algorithm IKP are validated in practice by the results illustrated in Figure 6.2. Note that the axes are in $\log_2$ scale. In particular, the results confirm that algorithm EDSM-BV, which is asymptotically the fastest for short patterns, is also the fastest in practice. As for algorithm EDSM, not surprisingly, we observe that, as m grows, the m^2 factor in its time complexity becomes more and more significant overall. This is clearly observed in Figure 6.2 by the green line with the circle point marker going up the y-axis as the length of the pattern increases from 8 to 64 meaning it takes longer to finish. Note that searching for much longer patterns exactly is not relevant in applications of interest, where errors (substitutions, insertions, and deletions) must be accommodated as m grows.

Experiment II: Real Data

EDSM-BV was tested further using real-world datasets. Human genomic data was obtained from the 1000 Genomes Project [138]. Specifically, data was obtained from



(a) $m = 8$ (b) $m = 16$

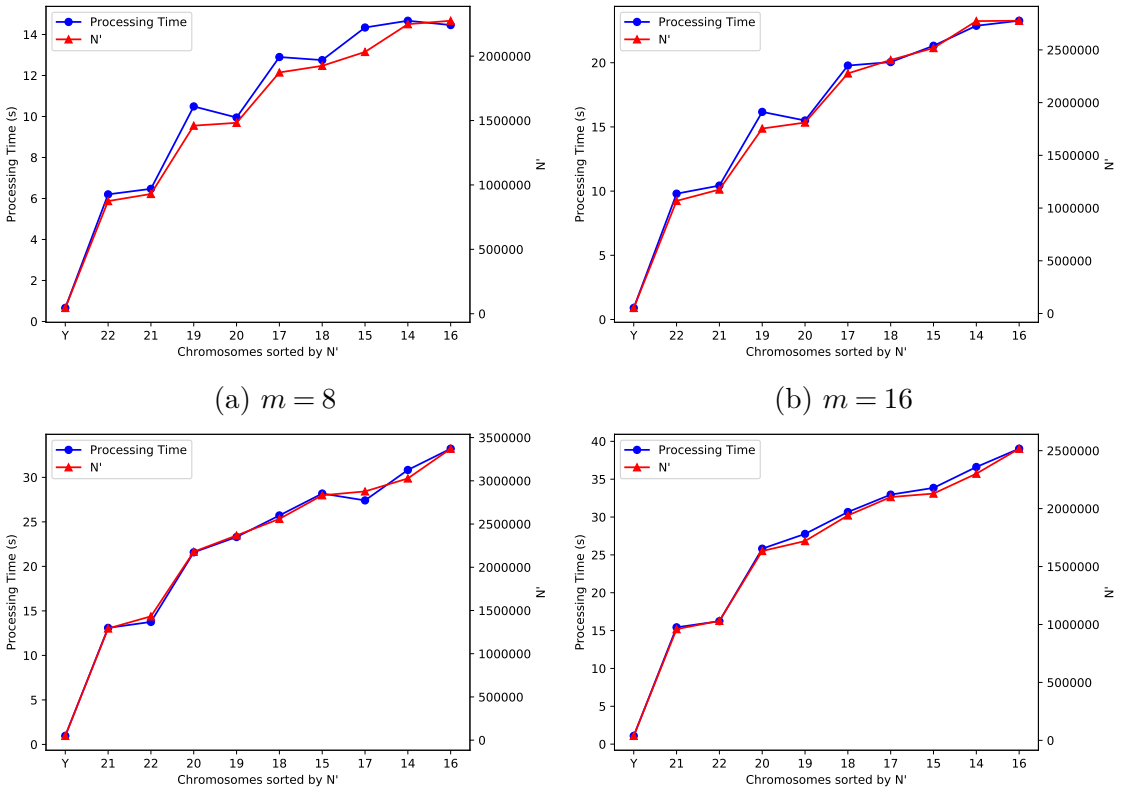


(c) $m = 32$ (d) $m = 64$

Fig. 6.2 Elapsed time in seconds ($\log_2 s$) comparison of EDSM-BV, EDSM and IKP for synthetic ED texts of length n ($\log_2 n$), and patterns of length m, as input.

Phase 3 of the project, in which the genomes of 2,504 individuals from 26 different populations were sequenced and aligned, producing a dataset which summarises the variation in the sample population. Files in Variant Call Format (VCF) include information about variations at each position in the genome, which makes the format ideal for our purposes. EDSM-BV was given a reference sequence (in FASTA format) and variation data (in VCF) for each of the ten smallest chromosomes as input, as well as synthetic patterns of length $m = 8$, 16, 32 or 64. The average percentage of degenerate positions across these chromosomes was approximately 3%; the average number of strings at degenerate positions was 2; and the average length of strings at degenerate positions was 1. The processing time of EDSM-BV was recorded; with processing we refer only to the actual CPU time used in executing the process–excluding the time to read the data in memory on-line. Chromosome 21, which is the smallest in

length, has an uncompressed VCF file of size 11.2GB. The results of this experiment are displayed in Figure 6.3.



(a) $m = 8$ (b) $m = 16$

(c) $m = 32$ (d) $m = 64$

Fig. 6.3 Elapsed time of EDSM-BV for real ED texts (Human chromosomes including variants) of size N and patterns of length m, as input. Note that the processing time is on the left y-axis and N' (denoting the number of strings in all segments having the property $|s| < |p|$ and $B \neq 0$, is on the right y-axis. The charts are ordered by N', so this sometimes changes the order of the chromosomes listed along the x-axis

The graphs in Figure 6.3 show a very clear linear relationship between the time taken for EDSM-BV to run and N', the total number of strings $s \in \tilde{T}[i]$ such that $|s| < |p|$ and $\mathcal{B} \neq 0$, per chromosome. Recall that the total time required by EDSM-BV for updating bit vector B_1 from B is $\mathcal{O}(N + N' \cdot \lceil \frac{m}{w} \rceil)$. This is the most time-consuming operation in practice as it searches for s in Occ-Vector$_P$ and then updates B_1 using bit-level operations. Note that, the total time to process strings $s \in \tilde{T}[i]$, with $|s| \geq |p|$, using the fast KMP algorithm is $\mathcal{O}(N)$, whose impact becomes insignificant overall in practice when compared to the time consumed calling Occ-Vector$_P$.

6.4.2 Multi-EDSM Results

Experiments for Multi-EDSM were run on the same machine mentioned above but the code for Multi-EDSM and EDSM-BV was compiled with a newer version of g++ version 5.4.0 at optimization level 3 (-O3). Scripts for verification and experiment automation were written in Python 2.7. The code, datasets and instructions are all freely available from https://github.com/webmasterar/multi-edsm.

Alternate Implementation

The results we show for Multi-EDSM are for a simpler implementation of the Occ-Vector$_P$ data structure that shares some of the characteristics of the theoretical implementation described above. It does *not* achieve $\mathcal{O}(M)$ memory complexity but comes close in that it avoids a naïve approach to building the Occ-Vector$_P$ data structure and employs some practical optimisations. We observe that we do not need to create a bit vector for each of the $< 2M$ nodes in ST_P:

1. Leaves represent a single position and so do not require storing permanently in a bit vector, saving space $\mathcal{O}(M)$. If we step through α and it arrives to a leaf node then we can obtain the same result with a series of constant time operations—obtaining the suffix number of the leaf and checking the corresponding bit in B can be done in constant time. We store a single bit vector of size $\mathcal{O}(\lceil \frac{M}{w} \rceil)$ which we can set the position in and return a reference to it. We also memorise the position we set so that next time we can clear the bit vector in constant time before applying the next bit position(s) of the next query.

2. Bit vectors need not be stored for deep nodes because an infix cannot be longer than $m_{max} - 2$, so $|\alpha| \leq m_{max} - 2$, and we never descend further down the tree, and in practice, the lengths of patterns are uniformly short. This is because we are doing exact string matching and the likelihood of longer patterns matching exactly diminishes greatly the longer the pattern. This optimisation does not solve the memory consumption problem by itself because for an especially high number of variable patterns, the number of nodes for which we would have to store a bit vector is high and requires extra memory $\mathcal{O}(\lceil \frac{M}{w} \rceil \cdot \sigma^{m_{max}-2})$.

3. We define a user-supplied parameter ω analogous to the τ parameter described in the algorithm above, where $1 \leq \omega \leq \lceil \frac{M}{w} \rceil$. We can obtain the number of the leaves from any node v via obtaining the suffix number of the left-most and right-most leaves and calculating the range, the number of leaves c, in constant time [93].

If $c > \omega$, then we resolve to store it as a bit vector in the OCC-VECTOR$_P$ data structure, otherwise during the query, we mark the positions in an empty bit vector in time $\mathcal{O}(c)$. This means queries take no more than $\mathcal{O}(|\alpha| + \omega)$ time. A good balance between performance and memory consumption can be obtained by setting $\omega = \lceil \frac{M}{w} \rceil$ in practice.

4. We define a second user-supplied parameter μ where the user can set a memory limit to be used by the program. If the memory runs out we store at each explicit node the suffix numbers of the left-most and right-most leaves $[l_v, r_v]$, as a pair of integers requiring at most $\mathcal{O}(M)$ space in the tree. Although storing the range is technically not necessary, it makes the program run faster in practice at little extra cost to memory.

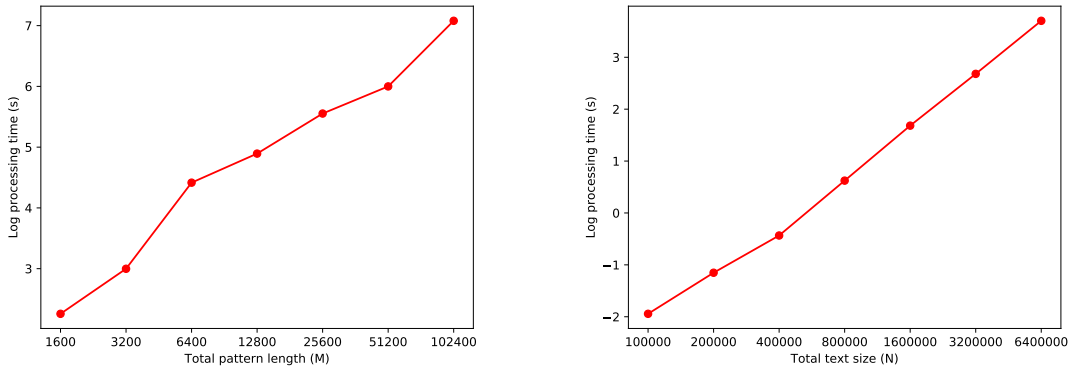
5. Finally, we use a level-order tree-traversal strategy for building the OCC-VECTOR$_P$ data structure in combination with parameters ω and μ. First we traverse down the tree from the root, checking $c > \omega$ and creating an empty bit vector on explicit nodes if memory consumption is less than μ, otherwise we store the range $[l_v, r_v]$. We do not traverse further than depth $m_{max} - 2$. Then we traverse the tree in reverse level order back up towards the root. At each level from $m_{max} - 2 \ldots 1$ as we traverse through the nodes, if node u is a leaf or a range, and its parent node v is a bit vector, we copy the positions from u to v. If u is a bit vector then its parent must also be a bit vector so we copy the positions by bitwise-ORing it to its parent. This takes time no more than $\mathcal{O}(\lceil \frac{M}{w} \rceil)$ per node, giving a worst-case time and space $\mathcal{O}(M \cdot \lceil \frac{M}{w} \rceil)$, alleviated by parameters ω and μ, resulting in a space complexity closer to $\mathcal{O}(M)$.

Experiment III: Time Performance

To test the time performance of Multi-EDSM, we devised two experiments. In both experiments we set the memory limit of Multi-EDSM to $\mu = 4$GB and $\omega = 1$ for the program to use as much memory as necessary. In Figure 6.4a we measured the processing time when searching a randomly-generated fixed ED text of length $n = 1,600,000$ ($N = 5,700,610$) over the DNA alphabet with randomly-generated pattern sets doubling in length from length $M = 1,600$ to $M = 102,400$. The text searched contains 10% degenerate segments and within each degenerate segment there are 2 to 10 random strings of length 1 to 10 each. Similarly, in Figure 6.4b we measured the processing time when searching randomly generated ED texts doubling in size from $N = 100,000$ to $N = 6,400,000$ with a fixed set of randomly-generated patterns

of length $M = 3,000$. The text had 10% degenerate positions representing single-base substitutions. (Uniform distribution has been used in all randomizations.)

As can be seen from the charts (Figure 6.4), the change in performance, whether it be an increase in the patterns total length M or the total text size N, causes a linear increase in processing time, which conforms to our theoretical findings (Theorem 5).



(a) Processing time with increasing patterns total length on a fixed ED text of length $n = 1,600,000$ ($N = 5,700,610$).

(b) Processing time with increasing ED total text size on a fixed set of patterns of total length $M = 3,000$.

Fig. 6.4 Time performance of Multi-EDSM.

Experiment IV: Comparison to EDSM-BV

To test the performance of Multi-EDSM compared against EDSM-BV we searched the same randomly-generated ED text of length $n = 1,600,000$ ($N = 5,700,610$) mentioned above against multiple sets of randomly-generated patterns of length 40 each. First we tested with a single pattern of length 40 and then the number of patterns in each set was incremented in steps of 10 from 10 to 100 patterns of length 40 each. EDSM-BV is only able to search one pattern at a time so we searched each pattern in a set individually and summed-up the total time spent. We see from the chart in Figure 6.5 that for a single pattern the EDSM-BV algorithm is fast, but it becomes immediately clear that for dictionary searching of even a handful of patterns, Multi-EDSM becomes orders of magnitude faster.

Experiment V: Real Application

We designed a three stage pipeline for determining the validity of MAWs discovered in the human genome. We obtained the GRCh37 chromosome sequences from *Ensembl* [63]

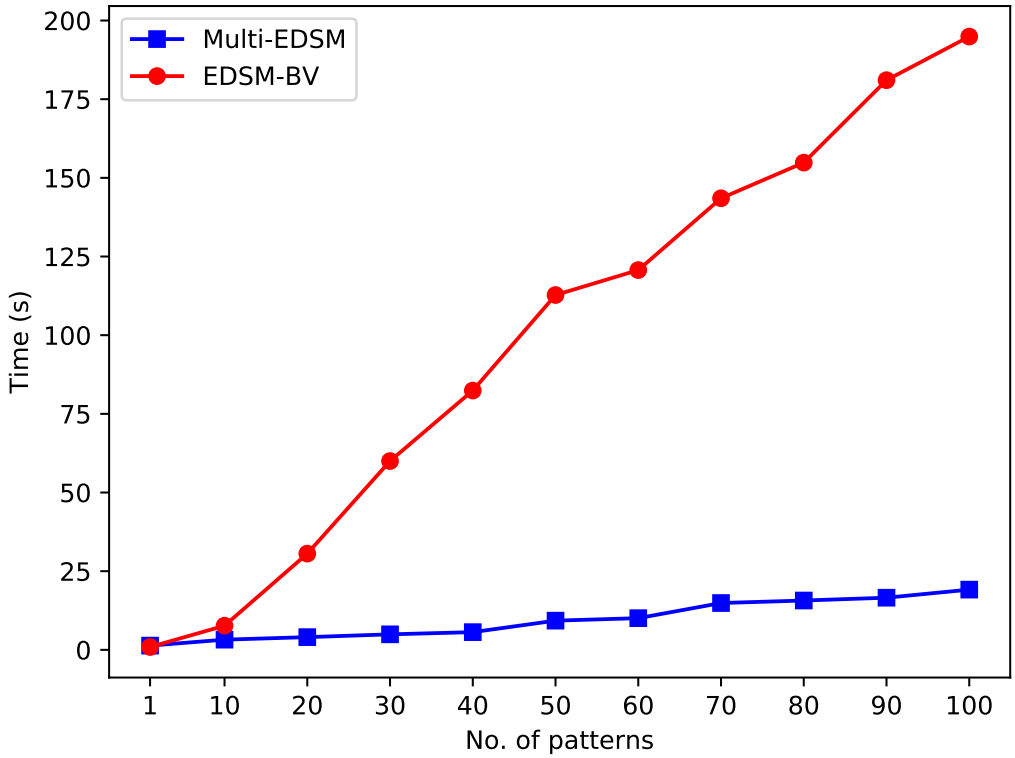


Fig. 6.5 Elapsed-time comparison of Multi-EDSM and EDSM-BV with an ED text of total size $N = 5,700,610$ and sets of randomly-generated patterns of length 40 each.

and the associated Phase 3 VCF files were obtained from the *1000 Genomes project* [138] on-line repositories. Phase 3 of the 1000 Genomes project contains 2,504 individuals from 26 populations whose variants are encoded in VCF files. The first step in the pipeline was to use *emMAW* [64] to extract MAWs of length between 3 and 12 from a concatenated file of the 22 autosomes and two sex chromosomes. The filtered list of MAWs contained 161,565 patterns with combined length $M = 1,937,789$. The second stage was to use the tool *EDSO* (obtainable from https://github.com/webmasterar/edso) to combine the reference chromosome sequences and the variants in the VCF files into searchable ED string (EDS) format files. Then Multi-EDSM was used to search each of the EDS files against the MAW patterns to produce a list of tuples marking the position of the match and pattern id. The final stage was validation. A script was written to validate each match, verifying a MAW genuinely exists for an individual at the identified position in the chromosome. We have found that for each chromosome more than half the MAWs discovered using Multi-EDSM exist in one or

more individuals and are subsequently disqualified, i.e. they are not really MAWs. Our compiled summary of the results show that 73% of MAWs were disqualified, leaving only 43,722 of 161,565 potential MAWs remaining.

Table 6.4 Ebola sequences *absent* from human genome *reference* sequence but *present* in the human pan-genome.

id	**sequence**	**position**	**variant id**	**sample id**	**ethnicity**
RAW1	`TTTCGCCCGACT`	6:93819539	rs569027564	NA18606	Han Chinese
RAW2	`TACGCCCTATCG`	1:74075482	rs578167440	HG02146	Peruvian
RAW3	`CCTACGCGCAAA`	15:71003880	rs564150197	HG03598	Bengali

We applied the results of this pipeline to validate the work of Silva et al. in [128] to identify MAWs in Ebola virus genomes that are absent from the human genome. They identify three MAWs of length 12, called RAW1, RAW2 and RAW3, that are not present in the *reference* human genome sequence. These MAWs could be used to verify an Ebola infection in a patient. However, we discovered from our results that each of the three MAWS do in fact occur in one or more individuals in the 1000 Genomes dataset, although indeed they are not that common. This means that they cannot be used as perfect identifiers for Ebola virus infection and perhaps longer unique MAWs should be used instead. In Table 6.4 we list the position of the discovery of each MAW as well as information about the variant and the id of one individual they occur in.

6.5 Conclusion

In this chapter we introduced the notion of searching for a deterministic pattern through a compacted representation of multiple closely-related sequences or a pan-genome in the form of an Elastic Degenerate text. ED strings contain degenerate positions characterising the variation in the samples contained in the original datasets but represents them in a simple text-based format that is readily searchable. We explained that this format is suited for on-line incorporation of variant data as encoded in VCF files so it can be searched immediately. The alternative off-line approach requires extensive disk space and memory for complex data structures to index a set of sequences and requires the rebuilding of that index for each new sequence added. Instead of this overhead, the solution may be to use an on-line solution which does not require re-indexing and uses very little extra disk space and memory in comparison. We envision the on-line approach being used for smaller datasets or for when a temporary

solution or an instant result is required while a more permanent off-line index is being (re-)built.

We presented three algorithms that are inspired by one another to search through ED texts. Two of the algorithms, EDSM and EDSM-BV, solve the Elastic-Degenerate String Matching problem for a single pattern in time $\mathcal{O}(nm^2+N)$ using space $\mathcal{O}(M)$ and time $\mathcal{O}(N \cdot \lceil \frac{m}{w} \rceil)$ using space $\mathcal{O}(m \cdot \lceil \frac{m}{w} \rceil)$, respectively. The third algorithm, Multi-EDSM, solves the Multiple Elastic-Degenerate String Matching problem in time $\mathcal{O}(N \cdot \lceil \frac{M}{w} \rceil)$ using space $\mathcal{O}(M)$. We have shown through our experiments the performance of these algorithms with respect to each other and one other competitor, IKP [66], that can search ED text format input. We have shown how real-world data from the 1000 Human Genomes project can be searched on-line and used it to show that many MAWs are in fact false-positives and we also checked the work of Silva et al. in [128] to prove that the three MAWs they identified in the Ebola virus genomes not present in the reference human genome actually exists naturally in some individuals in the greater human population.

Chapter 7

Concluding Remarks On...

In this chapter we make suggestions of directions for further research.

7.1 Chapter 4

The numerous examples of different applications we presented in this chapter strongly suggest that many other applications could benefit from using libFLASM or the FLASM approach in general. In fact, we propose most algorithms that make use of the Chang and Marr Index [20] will benefit from using our algorithm to calculate the index as we show in Table 4.6, thus reducing their pre-processing time. Since we provide an open-source C++ software library with a permissive license available from https://github.com/webmasterar/libFLASM making it easy to incorporate into other software, I expect more specialised applications of the algorithms in science and engineering may be identified and rapidly built with libFLASM.

I also foresee the development of variants of the algorithms presented here—algorithms modified to have reduced memory requirements and contain technical low-level optimisations that improve their running time. One technology-determined direction for further improvement on these algorithms may be to create versions for dedicated coprocessors or graphics cards with a high number of compute cores.

One direction for optimisation can be done on the Hamming distance algorithm MaxShiftM. This algorithm can be improved by making it dependent on k errors by applying Ukkonen's cut-off/block-model algorithm [46, 75, 142]. The equivalent k errors-dependant version of Myers' algorithm [95] has already been implemented in the SeqAn [32] library using Ukkonen's cut-off/block-model algorithm [95, 142].

It has been suggested that making a Hamming distance version of Myers' algorithm is a worthwhile endeavour, and it seems like a worthwhile idea for solving the FLASM

problem under the Hamming distance model, but Grabowski and Fredriksson [51] proclaim Myers' algorithm cannot be modified for the Hamming distance. Instead they propose two different Hamming-distance approximate string matching algorithms with k-errors (mismatches) and achieve a worst-case time complexity of $\mathcal{O}(n \cdot \lceil \frac{m}{w} \rceil)$ or $\mathcal{O}(n \lceil m \log\log(k)/w \rceil)$, so conceptually this could be adapted to perform FLASM in worst-case time $\mathcal{O}(n \lceil \ell/w \rceil m)$ or $\mathcal{O}(n \lceil \ell \log\log(k)/w \rceil m)$. Since this is asymptotically similar to the time complexity of the MaxShiftM algorithm, I do not imagine this would improve the overall speed significantly.

Another direction to take this research, is to implement more specialised domain-specific distance models instead of relying on the simple edit and Hamming distance models. This may be more useful for working with more divergent sequences containing many large insertions and deletions. The synthetic data and real data we used in our experiments was not particularly divergent with a maximum substitution rate of 35% (see Section 4.4), so we found the edit and Hamming distance models to be more than satisfactory when applied for motif extraction and circular sequence alignment of closely-related sequences.

7.2 Chapter 5

The first two algorithms (nCSC and hCSC) described in this chapter use a Parikh vector of size $\mathcal{O}(\sigma^q)$ to instantly access and update the frequency of q-grams. With longer q-grams and alphabet sizes, this uses up increasingly more memory and preprocessing time. As mentioned before, this can be partially alleviated with the use of a hash-table data structure which reduces the practical space requirements of the algorithms at marginal cost to execution speed. However, this is a moot point when the algorithms are compared to saCSC which solves the problem in time and space $\mathcal{O}(\beta m + n)$ without requiring $\mathcal{O}(\sigma^q)$ space and is orders of magnitude faster than its competitors. I do not see any immediate way on improving on this except to utilise machine-level multi-threading to divide the work over z processor cores giving a theoretical speedup by a factor of z.

Perhaps one approach to improving this algorithm, specifically saCSCr, is to find an alternative way to do the refinement step. This currently takes time $\mathcal{O}(L^3)$ with scoring matrices and affine gap penalty scores. But if we choose to solve this problem using the simpler edit distance model with unit scores (and accommodating the special case for \$), the complexity could be reduced to $\mathcal{O}(L^2 \log L)$ by using Maes' cyclic string-to-string correction algorithm [84] or even $\mathcal{O}(L^2 \cdot \lceil L/w \rceil)$ by using libFLASM

to solve the approximate circular string matching problem. This may not give the best solution for more divergent sequences with many insertions and deletions, but should prove satisfactory in most cases and reduce the cumulative time of executing saCSCr multiple times in the case of solving the Multiple Circular Sequence Alignment problem as implemented in BEAR.

7.3 Chapter 6

The work we presented in this chapter solves the exact string matching problem in an elastic-degenerate (ED) text, however, exact matching may not be suitable for some purposes. Some work has already been done by Bernardini et al. in [13] that describes how to solve the single-pattern approximate string matching problem on an ED text. They tackle the problem starting-off from the EDSM solution that had time complexity $\mathcal{O}(nm^2+N)$ and space complexity $\mathcal{O}(m)$. They consider the problem under the edit distance model, presenting an $\mathcal{O}(k^2mG+kN)$ time and $\mathcal{O}(m)$ space algorithm, where G is the total number of strings in an ED text and $n \leq G \leq N$ and $0 \leq k < m$ is the maximum edit distance threshold. They also present an $\mathcal{O}(kmG+kN)$ time and $\mathcal{O}(m)$ space algorithm for the Hamming distance where k is the maximum number of mismatches allowed. As of yet, no implementation of these two algorithms is available. And no paper has been published on how to solve the multiple approximate string matching in elastic-degenerate text problem as this was only just recently published. Both are potentially fruitful routes for further research.

I propose here another way to solve the single-pattern approximate string matching problem in ED texts which should work well for longer patterns with low k. We make use of the *pigeon-hole* principle—by dividing the pattern p of length m evenly into $j = k+1$ parts, we know at least one of these $p_0, p_1, \ldots, p_{j-1}$ parts must not have any errors [5]. We can then use the Multi-EDSM algorithm to find all matches for the parts of p as well as any exact matches for the complete pattern p. This would work like a filter — we would report exact matches of the whole pattern p when they are discovered and relegate potential matches for parts of p to a verification stage. The verification process would use the algorithms presented in [13], to verify the potential match. The verification step for an off-line ED text could be done with a similar approach to the *seed-and-extend* paradigm, where the exact match is considered a *seed* and extensions happen either side of the seed. Extensions continue while errors remain below error/mismatch threshold k till a match is reported. However, this particular approach cannot be used to solve the on-line version of the problem because the seed

cannot be extended forwards in advance before the next positions have come on-line. I now briefly describe an approach to solving the on-line version of the problem.

First, we have to create a list $\mathcal{L}_s$ that operates like a bounded FILO (First In Last Out) queue holding up to m positions. We push a copy of each new position we search of the ED text onto $\mathcal{L}_s$ and it maintains at most m positions at any one time and the oldest position is removed automatically. The queue allows us to search backwards (p in reverse) but requires extra memory to hold the positions/segments. We also need to keep another bounded binary list $\mathcal{L}_c$ of length m with all positions initialised to 0 and using a pointer h initially set to position 0, we would move this pointer forward one position $h = (h+1) \bmod m$ with every new position or degenerate segment we search. If while searching a position i of the ED text a match for p_j is identified, we would mark forward in $\mathcal{L}_c$ the calculated position x by setting $\mathcal{L}_c[x] = 1$. At any time during our search we move the pointer h along and discover $\mathcal{L}_c[h] = 1$ we would know to perform a verification step from that ending position of p in reverse. After this verification step, we would set $\mathcal{L}_c[h] = 0$ to reset it. In this way we can use the fast exact string matching algorithm Multi-EDSM in conjunction with Bernardini et al.'s algorithms to speed up approximate pattern matching in an ED text.

A caveat of using this on-line searching approach is that ε must be absent from the ED text, as this would require storing up to n positions in memory, which would potentially require a lot of memory and cause verification to take a long time for longer sequences with many ε-containing degenerate positions. In practice, this limitation is perfectly acceptable – the Human pan-genome generated from the 1000 Genomes Project data does not contain ε and degenerate positions are mostly SNPs and are therefore minimal in size.

One possible application for Multi-EDSM that we considered is short read mapping for genome assembly, an operation usually performed on the human genome *reference* sequence which is used as a scaffold on which to align the short reads obtained from next generation sequencing technologies. Traditionally, exact mapping is performed for short reads against the reference sequence, and for the remaining reads that do not match exactly, approximate string matching techniques are used to align them. After this process there still remains some reads that do not match up with sufficient confidence. We thought it may be more effective to perform read mapping on a pan-genome instead of a single reference genome because it better represents the diversity of a species and ensures valid reads align at positions representing allelic variants. Unfortunately, we determined that Multi-EDSM was much too slow in practice to do this as it currently stands. When there are many millions of reads and each read is of uniform length

usually between 100bp and 500bp, this produces a gigantic M factor that makes using *vanilla* Multi-EDSM to solve the problem intractable.

Recently, Cisłak et al. published a paper [22] where they managed to find an optimisation for the Elastic-Degenerate String Matching problem solution, cleverly avoiding the need to use the Occ-Vector$_P$ data structure altogether (which was the speed-limiting factor) and using simple bitwise operations (the Shift-Or algorithm) to extend prefixes. This is an observation we failed to realise in incrementally designing the EDSM-based algorithms. With their implementation, SOPanG, they achieved a speed up of more than an order of magnitude faster than EDSM-BV when searching for a pattern in the 1000 Human Genomes datasets. This optimisation could be extended to solve the Multiple Elastic-Degenerate String Matching problem and it is expected the speed-up would be highly significant, making multiple read-mapping to a pan-genome a possibility. Future research could also leverage multi-processor environments and/or co-processors to divide the work and improve its speed such that it becomes possible to do read mapping at a competitive speed compared to existing genome assembly software that use the human genome reference sequence as a scaffold, such as Bowtie [76], BWA [79], SOAP2 [81] and REAL [43], amongst others.

7.4 This Thesis and Future Research Directions

We have described the motivations for the creation of the algorithms described in this thesis and we have presented multiple applications of the algorithms in the field of Bioinformatics and beyond. The performance and accuracy of some of the algorithms was seen to qualify them for inclusion in state-of-the-art applications like BEAR and MoTeX-II, for Multiple Circular Sequence Alignment and simple and structured motif extraction, respectively. It is likely that there are many more applications for these algorithms than the ones described in this thesis and only time will tell how useful they will be. It is often the case that algorithms can be generalised into their simplest forms or specialised for solving problems they were not originally intended for. An example of this is how we applied Myers' [95] general purpose and highly efficient bit-parallel approximate pattern matching algorithm to solve the challenging problem of Fixed-Length Approximate String Matching, as described in Chapter 4.

It is somewhat difficult to predict the future emerging trends in the field of Bioinformatics and how certain algorithms or approaches become indespensible for day-to-day research pipelines. Such an example is the Suffix Array data structure [86] which we described in Section 3.4, which has become the basis of many different algorithms in

Bioinformatics [10, 49, 129] and without which it would be especially time-consuming to search for patterns in genomes today. The data structure also finds itself being used in many other algorithms including the algorithm we describe in this thesis for pairwise circular sequence alignment, saCSC. Although it has been many years since it was first presented by Udi Manber and Gene Myers in [86] in 1993 as a substitute for the bulky Suffix Tree data structure, this highly effective data structure has been improved upon, polished and used in so many different fields of research that it is likely that Manber and Myers never imagined just how far their contribution would go.

I imagine at least one of the algorithms described in this thesis will find itself being used outside of our examples, though perhaps, not be as influential as the Suffix Array data structure. I will try to mention some fields of current research that may benefit from the algorithms and ideas we describe in this thesis and it will be up to the researcher reading this thesis to find practical applications and make improvements to their end.

Very recently there has been much interest in the field of Metagenomics. Many environments such as soils, oceans, rivers, forests, hospital wards and even the guts of animals host complex and diverse ecosystems of microbial and viral genomes. Metagenomics is the study of and characterisation of biodiversity of these microbes through genotyping and sequencing all of the microbes together. Researchers doing analyses have to contend with bacteria distributing plasmids via horizontal gene transfer as well as the great variety of different species and subspecies originating through vertical transmission and evolution. The genomes of strains of these organisms are generally clustered based on compositional homology or binned by the number of domains they share [2, 82]. The process of comparing strains can be improved with multiple circular sequence alignment done with saCSC as incorporated into BEAR. The more accurate alignments may help identify the differences and similarities between the strains and help in building more accurate phylogeny trees. Furthermore, if robust circular motif searching (where a high distance between the pattern and sequence is expected) is one of the requirements, then circular pattern matching can be performed using libFLASM.

Now onto a different application the work this thesis might play a role in in the future; As mentioned in the section above, mapping of uniformly short reads in Next Generation Sequencing is done against a reference sequence. This might be the officially-recognised reference genome sequence for a particular species (GRCh37 or GRCh38 in humans) or it might be a specially-crafted version that is composed of the most frequently-occuring alleles as found in the 1000 Human Genomes Project (or equivalent

for other species). The latter reference sequence would of course be better than the former in improving read-mapping but it is not a foolproof method and it still doesn't capture the diversity in the population that was sequenced. If instead read-mapping was performed against a pan-genome it would solve the problem to the highest level possible. Mapping against a pan-genome would reduce the problem of a read not mapping properly because of the lack of variation in alleles in a singular reference sequence and in turn this would reduce the number of approximately-mapping and unmapped reads, leading to more complete and accurate genome sequences. Another benefit to this form of genome assembly is the discovery of novel SNPs as identified in graph-based pan-genome assemblies [117].

The current hot-topic technology of late is the CRISPR-Cas9 gene-editing technology and one of the most troubling aspects of this new technology is the risk of *off-target* double strand breaks or the insertion of a new sequence in the wrong place. CRISPR uses a guide-RNA sequence of length about 20bp to find the location in the genome at which to cut, but two problems abound – off-target cuts occur at positions that match exactly the guide-RNA sequence but are not the intended target which can have unexpected and potentially harmful consequences. The other problem is that the CRISPR-Cas9 system sometimes makes cuts at positions with one or two bases difference from the target sequence. The current approach to finding potential off-target positions is by using an off-target prediction tool like CCTop [135] with a read-mapper like Bowtie [76] to align potential guide-RNA sequences against a reference genome and then recommending the ones with the fewest off-target hits inside of exons. But these results are based on the reference genome and so may not be representative of a species population and the guide-RNA such programs recommend may *not* in fact be the safest ones to use! I expect the scientific community to eventually come to recognise this problem and the above-mentioned read-mapping problem and switch over to using only pan-genome based techniques and algorithms similar to Multi-EDSM.

I look forward to seeing what advancements come about in the next few years.

References

[1] van Aardenne-Ehrenfest, T., de Bruijn, N.: Circuits and Trees in Oriented Linear Graphs, pp. 149–163. Birkhäuser Boston, Boston, MA (1987), https://doi.org/10.1007/978-0-8176-4842-8_12

[2] Alneberg, J., Bjarnason, B.S., De Bruijn, I., Schirmer, M., Quick, J., Ijaz, U.Z., Lahti, L., Loman, N.J., Andersson, A.F., Quince, C.: Binning metagenomic contigs by coverage and composition. Nature methods 11(11), 1144 (2014)

[3] Alstrup, S., Secher, J.P., Spork, M.: Optimal on-line decremental connectivity in trees. Inf. Process. Lett. 64(4), 161–164 (1997), https://doi.org/10.1016/S0020-0190(97)00170-1

[4] Ayad, L.A., Pissis, S.P., Retha, A.: libFLASM: a software library for fixed-length approximate string matching. BMC Bioinformatics 17(1), 454 (2016), http://dx.doi.org/10.1186/s12859-016-1320-2

[5] Baeza-Yates, R., Navarro, G.: New and faster filters for multiple approximate string matching. Random Structures & Algorithms 20(1), 23–49 (2002)

[6] Baeza-Yates, R.A., Navarro, G.: A faster algorithm for approximate string matching. In: Hirschberg, D.S., Myers, E.W. (eds.) Proceedings of the seventh annual Symposium on Combinatorial Pattern Matching (CPM 1996). Lecture Notes in Computer Science, vol. 1075, pp. 1–23. Springer, UK (1996)

[7] Baier, U., Beller, T., Ohlebusch, E.: Graphical pan-genome analysis with compressed suffix trees and the Burrows-Wheeler transform. Bioinformatics 32(4), 497–504 (2016)

[8] Barton, C., Iliopoulos, C.S., Kundu, R., Pissis, S.P., Retha, A., Vayani, F.: Experimental Algorithms: 14th International Symposium, SEA 2015, Paris, France, June 29 – July 1, 2015, Proceedings, chap. Accurate and Efficient Methods to Improve Multiple Circular Sequence Alignment, pp. 247–258. Springer International Publishing, Cham (2015)

[9] Barton, C., Iliopoulos, C.S., Pissis, S.P.: Fast algorithms for approximate circular string matching. Algorithms for Molecular Biology 9, 9 (2014)

[10] Barton, C., Iliopoulos, C.S., Pissis, S.P.: Average-case optimal approximate circular string matching. In: Dediu, A.H., Formenti, E., Martín-Vide, C., Truthe, B. (eds.) Language and Automata Theory and Applications - 9th International

Conference, LATA 2015. Lecture Notes in Computer Science, vol. 8977, pp. 85–96. Springer, Nice, France (2015)

[11] Bejerano, G., Pheasant, M., Makunin, I., Stephen, S., Kent, W.J., Mattick, J.S., Haussler, D.: Ultraconserved elements in the human genome. Science 304(5675), 1321–1325 (2004), http://science.sciencemag.org/content/304/5675/1321

[12] Benson, D.A., Karsch-Mizrachi, I., Lipman, D.J., Ostell, J., Wheeler, D.L.: Genbank. Nucleic Acids Research 33(suppl_1), D34–D38 (2005), http://dx.doi.org/10.1093/nar/gki063

[13] Bernardini, G., Pisanti, N., Pissis, S.P., Rosone, G.: Pattern matching on elastic-degenerate text with errors. In: SPIRE. LNCS, vol. 10508, pp. 74–90. Springer International Publishing (2017)

[14] Boyer, R.S., Moore, J.S.: A fast string searching algorithm. Commun. ACM 20(10), 762–772 (Oct 1977), http://doi.acm.org/10.1145/359842.359859

[15] Bunke, H., Buhler, U.: Applications of approximate string matching to 2D shape recognition. Pattern Recognit 26(12), 1797–1812 (1993)

[16] Burcsi, P., Cicalese, F., Fici, G., Lipták, Z.: Algorithms for jumbled pattern matching in strings. Int J Found Comput Sci 23(2), 357–374 (2012)

[17] Burkhardt, S., Crauser, A., Ferragina, P., Lenhof, H.P., Rivals, E., Vingron, M.: Q-gram based database searching using a suffix array (QUASAR). In: Proceedings of the Third Annual International Conference on Computational Molecular Biology. pp. 77–83. RECOMB '99, ACM, New York, NY, USA (1999), http://doi.acm.org/10.1145/299432.299460

[18] Campbell, J., Lowe, D., Sleeman, M.: Developing the next generation of monoclonal antibodies for the treatment of rheumatoid arthritis. British Journal of Pharmacology 162, 1470–1484 (2011)

[19] Carvalho, A., Marsan, L., Pisanti, N., Sagot, M.F.: RISOTTO: fast extraction of motifs with mismatches. In: Proceedings of the 7th Latin American Symposium on Theoretical Informatics (LATIN'06). pp. 757–768. Lecture Notes in Computer Science, Springer, Valdivia, Chile (2006)

[20] Chang, W.I., Marr, T.G.: Approximate string matching and local similarity. In: Proceedings of the 5th Annual Symposium on Combinatorial Pattern Matching. pp. 259–273. CPM '94, Springer-Verlag, London, UK, UK (1994)

[21] Chen, K.H., Huang, G.S., Lee, R.C.T.: Exact circular pattern matching using the bit-parallelism and q-gram technique. In: Proc. of the 29th Workshop on Combinatorial Mathematics and Computation Theory. pp. 18–27. Citeseer (2012)

[22] Cisłak, A., Grabowski, S., Holub, J.: SOPanG: online text searching over a pan-genome. Bioinformatics p. bty506 (2018), http://dx.doi.org/10.1093/bioinformatics/bty506

[23] Consortium, T.U.: UniProt: a hub for protein information. Nucleic Acids Research 43(D1), D204–D212 (2015), http://dx.doi.org/10.1093/nar/gku989

[24] Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. MIT press (2009)

[25] Craik, D., Allewell, N.: Thematic minireview series on circular proteins. The Journal of Biological Chemistry 287, 26999–27000 (2012)

[26] Crochemore, M., Hancart, C., Lecroq, T.: Algorithms on Strings. Cambridge University Press, New York, NY, USA (2007)

[27] Crochemore, M., Iliopoulos, C.S., Pissis, S.P.: A parallel algorithm for fixed-length approximate string-matching with k-mismatches. In: Elomaa, T., Mannila, H., Orponen, P. (eds.) Algorithms and Applications, Lecture Notes in Computer Science, vol. 6060, pp. 92–101. Springer Berlin Heidelberg (2010)

[28] Damerau, F.: A technique for computer detection and correction of spelling errors. Communications of the ACM 7, 171–176 (1964)

[29] Davison, J.: Genetic exchange between bacteria in the environment. Plasmid 42(2), 73 – 91 (1999)

[30] Dayhoff, M., Schwartz, R., Orcutt, B.: A model of evolutionary change in proteins. In: Atlas of protein sequence and structure, vol. 5, pp. 345–352. National Biomedical Research Foundation Silver Spring, MD (1978)

[31] De Bruijn, N.G.: A combinatorial problem. Koninklijke Nederlandse Akademie v. Wetenschappen 49(49), 758–764 (1946)

[32] Doring, A., Weese1, D., Rausch, T., Reinert, K.: SeqAn an efficient, generic C++ library for sequence analysis. BMC Bioinformatics pp. 1–9 (2008)

[33] Edgar, R.C.: MUSCLE: multiple sequence alignment with high accuracy and high throughput. Nucleic Acids Research 32(5), 1792–1797 (2004), http://nar.oxfordjournals.org/content/32/5/1792.abstract

[34] Ehlers, T., Manea, F., Mercaş, R., Nowotka, D.: k-abelian pattern matching. In: 18th DLT. LNCS, vol. 8633, pp. 178–190 (2014)

[35] Eskin, E., Pevzner, P.A.: Finding composite regulatory patterns in DNA sequences. In: Proceedings of the Tenth International Conference on Intelligent Systems for Molecular Biology, August 3-7, 2002. pp. 354–363. Edmonton, Alberta, Canada (2002)

[36] Farach, M.: Optimal suffix tree construction with large alphabets. In: 38th Annual Symposium on Foundations of Computer Science, FOCS. pp. 137–143 (1997)

[37] Faro, S., Lecroq, T.: Twenty years of bit-parallelism in string matching. Festschrift for Borivoj Melichar pp. 72–101 (2012)

[38] Felsenstein, J.: Evolutionary trees from DNA sequences: A maximum likelihood approach. Journal of Molecular Evolution 17(6), 368–376 (Nov 1981), https://doi.org/10.1007/BF01734359

[39] Fernandes, F., Pereira, L., Freitas, A.T.: CSA: An efficient algorithm to improve circular DNA multiple alignment. BMC Bioinformatics 10(1), 230 (Jul 2009), https://doi.org/10.1186/1471-2105-10-230

[40] Fischer, J.: Inducing the LCP-array. In: Dehne, F., Iacono, J., Sack, J. (eds.) Algorithms and Data Structures - 12th International Symposium, WADS 2011. Proceedings. Lecture Notes in Computer Science, vol. 6844, pp. 374–385. Springer (2011), https://doi.org/10.1007/978-3-642-22300-6_32

[41] Fletcher, W., Yang, Z.: INDELible: A flexible simulator of biological sequence evolution. Mol Biol Evol 26(8), 1879–1888 (2009)

[42] Fredriksson, K., Navarro, G.: Average-optimal single and multiple approximate string matching. J. Exp. Algorithmics 9 (dec 2004)

[43] Frousios, K., Iliopoulos, C.S., Mouchard, L., Pissis, S.P., Tischler, G.: REAL: An efficient read aligner for next generation sequencing reads. In: Proceedings of the First ACM International Conference on Bioinformatics and Computational Biology. pp. 154–159. BCB '10, ACM, New York, NY, USA (2010), http://doi.acm.org/10.1145/1854776.1854801

[44] Gagie, T., Hermelin, D., Landau, G.M., Weimann, O.: Binary jumbled pattern matching on trees and tree-like structures. Algorithmica 73(3), 571–588 (2015), https://doi.org/10.1007/s00453-014-9957-6

[45] Galil, Z., Giancarlo, R.: Data structures and algorithms for approximate string matching. Journal of Complexity 4(1), 33–72 (1988)

[46] Galil, Z., Park, K.: An improved algorithm for approximate string matching. SIAM Journal on Computing 19(6), 989–999 (1990), https://doi.org/10.1137/0219067

[47] Ghosh, S.K., Ghosh, J., Pal, R.K.: A new algorithm to represent a given k-ary tree into its equivalent binary tree structure (2008)

[48] Gog, S., Beller, T., Moffat, A., Petri, M.: From theory to practice: Plug and play with succinct data structures. In: Gudmundsson, J., Katajainen, J. (eds.) Experimental Algorithms. pp. 326–337. Springer International Publishing, Cham (2014)

[49] Gonnella, G., Kurtz, S.: Readjoiner: a fast and memory efficient string graph-based sequence assembler. BMC bioinformatics 13(1), 82 (2012)

[50] Gotoh, O.: An improved algorithm for matching biological sequences. Journal of Molecular Biology 162(3), 705–708 (dec 1982)

[51] Grabowski, S., Fredriksson, K.: Bit-parallel string matching under hamming distance in o(n[m/w]) worst case time. Inf. Process. Lett. 105(5), 182–187 (Feb 2008), http://dx.doi.org/10.1016/j.ipl.2007.08.021

[52] Grossi, R., Iliopoulos, C.S., Liu, C., Pisanti, N., Pissis, S.P., Retha, A., Rosone, G., Vayani, F., Versari, L.: On-Line Pattern Matching on Similar Texts. In: Juha Kärkkäinen, J.R., Rytter, W. (eds.) 28th Annual Symposium on Combinatorial Pattern Matching (CPM 2017). Leibniz International Proceedings in Informatics (LIPIcs), vol. 78, pp. 9:1–9:14. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2017), http://drops.dagstuhl.de/opus/volltexte/2017/7337

[53] Grossi, R., Iliopoulos, C.S., Mercas, R., Pisanti, N., Pissis, S.P., Retha, A., Vayani, F.: Circular sequence comparison with q-grams. In: Pop, M., Touzet, H. (eds.) Algorithms in Bioinformatics: 15th International Workshop, WABI 2015, Atlanta, GA, USA, September 10-12, 2015, Proceedings. pp. 203–216. Springer Berlin Heidelberg, Berlin, Heidelberg (2015), http://dx.doi.org/10.1007/978-3-662-48221-6_15

[54] Grossi, R., Iliopoulos, C.S., Mercas, R., Pisanti, N., Pissis, S.P., Retha, A., Vayani, F.: Circular sequence comparison: algorithms and applications. Algorithms for Molecular Biology 11, 12 (2016)

[55] Hamming, R.: Error detecting and error correcting codes. The Bell System Technical Journal 29, 147–160 (1950)

[56] van Helden, J., André, B., Collado-Vides, J.: Extracting regulatory sites from the upstream region of yeast genes by computational analysis of oligonucleotide frequencies. Journal of Molecular Biology 281(5), 827–842 (1998)

[57] Helinski, D.R., Clewell, D.B.: Circular DNA. Annu Rev Biochem 40(1), 899–942 (1971)

[58] Henikoff, S., Henikoff, J.G.: Amino acid substitution matrices from protein blocks. Proceedings of the National Academy of Sciences 89(22), 10915–10919 (1992), http://www.pnas.org/content/89/22/10915

[59] Hirvola, T., Tarhio, J.: Experimental Algorithms: 13th International Symposium, SEA 2014, Copenhagen, Denmark, June 29 – July 1, 2014. Proceedings, chap. Approximate Online Matching of Circular Strings, pp. 315–325. Springer International Publishing, Cham (2014)

[60] Holley, G., Wittler, R., Stoye, J.: Bloom Filter Trie: an alignment-free and reference-free data structure for pan-genome storage. Algorithms for Molecular Biology 11, 3 (2016)

[61] Holub, J., Smyth, W.F., Wang, S.: Fast pattern-matching on indeterminate strings. Journal of Discrete Algorithms 6(1), 37–50 (2008)

[62] Huang, L., Popic, V., Batzoglou, S.: Short read alignment with populations of genomes. Bioinformatics 29(13), 361–370 (2013)

[63] Hubbard, T., Barker, D., Birney, E., Cameron, G., Chen, Y., Clark, L., Cox, T., Cuff, J., Curwen, V., Down, T., Durbin, R., Eyras, E., Gilbert, J., Hammond, M., Huminiecki, L., Kasprzyk, A., Lehvaslaiho, H., Lijnzaad, P., Melsopp, C., Mongin, E., Pettett, R., Pocock, M., Potter, S., Rust, A., Schmidt, E., Searle, S., Slater, G., Smith, J., Spooner, W., Stabenau, A., Stalker, J., Stupka, E., Ureta-Vidal, A., Vastrik, I., Clamp, M.: The Ensembl genome database project. Nucleic Acids Research 30(1), 38–41 (2002), http://dx.doi.org/10.1093/nar/30.1.38

[64] Héliou, A., Pissis, S.P., Puglisi, S.J.: emMAW: computing minimal absent words in external memory. Bioinformatics 33(17), 2746–2749 (2017), http://dx.doi.org/10.1093/bioinformatics/btx209

[65] Iliopoulos, C., Mouchard, L., Pinzon, Y.: The Max-Shift algorithm for approximate string matching. In: Brodal, G., Frigioni, D., Marchetti-Spaccamela, A. (eds.) Proceedings of the fifth International Workshop on Algorithm Engineering (WAE 2001). Lecture Notes in Computer Science, vol. 2141, pp. 13–25. Springer, Denmark (2001)

[66] Iliopoulos, C.S., Kundu, R., Pissis, S.P.: Efficient pattern matching in elastic-degenerate texts. In: 11th International Conference on Language and Automata Theory and Applications (LATA), Lecture Notes in Computer Science, vol. 10168. Springer International Publishing (2017)

[67] Jukes, T.H., Cantor, C.R.: CHAPTER 24 - Evolution of Protein Molecules. In: Munro, H. (ed.) Mammalian Protein Metabolism, pp. 21–132. Academic Press, New York (1969)

[68] Kanehisa, M., Goto, S., Sato, K., Fujibuchi, W., Bono, H.: KEGG: Kyoto encyclopedia of genes and genomes. Nucleic Acieds Research 27, 29–34 (1999)

[69] Kersey, P.J., Allen, J.E., Armean, I., Boddu, S., Bolt, B.J., Carvalho-Silva, D., Christensen, M., Davis, P., Falin, L.J., Grabmueller, C., Humphrey, J.C., Kerhornou, A., Khobova, J., Aranganathan, N.K., Langridge, N., Lowy, E., McDowall, M.D., Maheswari, U., Nuhn, M., Ong, C.K., Overduin, B., Paulini, M., Pedro, H., Perry, E., Spudich, G., Tapanari, E., Walts, B., Williams, G., Tello-Ruiz, M.K., Stein, J.C., Wei, S., Ware, D., Bolser, D.M., Howe, K.L., Kulesha, E., Lawson, D., Maslen, G., Staines, D.M.: Ensembl genomes 2016: more genomes, more complexity. Nucleic Acids Research 44(Database-Issue), 574–580 (2016)

[70] Knuth, D.E., Jr., J.H.M., Pratt, V.R.: Fast pattern matching in strings. SIAM J. Comput. 6(2), 323–350 (1977)

[71] Kong, A., Frigge, M.L., Masson, G., Besenbacher, S., Sulem, P., Magnusson, G., Gudjonsson, S.A., Sigurdsson, A., Jonasdottir, A., Jonasdottir, A., Wong, W.S., Sigurdsson, G., Walters, G.B., Steinberg, S., Helgason, H., Thorleifsson, G., Gudbjartsson, D.F., Helgason, A., Magnusson, O.T.T., Thorsteinsdottir, U., Stefansson, K.: Rate of de novo mutations and the importance of father's age to disease risk. Nature 488(7412), 471–475 (Aug 2012)

[72] Koscielny, G., Yaikhom, G., Iyer, V., Meehan, T.F., Morgan, H., Atienza-Herrero, J., Blake, A., Chen, C.K., Easty, R., Di Fenza, A., Fiegel, T., Grifiths, M., Horne, A., Karp, N.A., Kurbatova, N., Mason, J.C., Matthews, P., Oakley, D.J., Qazi, A., Regnart, J., Retha, A., Santos, L.A., Sneddon, D.J., Warren, J., Westerberg, H., Wilson, R.J., Melvin, D.G., Smedley, D., Brown, S.D.M., Flicek, P., Skarnes, W.C., Mallon, A.M., Parkinson, H.: The international mouse phenotyping consortium web portal, a unified point of access for knockout mice and related phenotyping data. Nucleic Acids Research 42(D1), D802–D809 (2014), http://nar.oxfordjournals.org/content/42/D1/D802.abstract

[73] Krane, D.E., Raymer L, M.: Fundamental concepts of bioinformatics. Pearson Education Inc. (2003)

[74] Kurtz, S.: Reducing the space requirement of suffix trees. Software-Practice and Experience 29(13), 1149–71 (1999)

[75] Landau, G.M., Vishkin, U.: Fast parallel and serial approximate string matching. Journal of Algorithms 10(2), 157 – 169 (1989), http://www.sciencedirect.com/science/article/pii/0196677489900102

[76] Langmead, B.: Aligning short sequencing reads with Bowtie. Current Protocols in Bioinformatics 32(1), 11.7.1–11.7.14, https://currentprotocols.onlinelibrary.wiley.com/doi/abs/10.1002/0471250953.bi1107s32

[77] Lefranc, M., Pommié, C., Ruiz, M., Giudicelli, V., Foulquier, E., Truong, L., Thouvenin-Contet, V., Lefranc, G.: IMGT unique numbering for immunoglobulin and T cell receptor variable domains and Ig superfamily V-like domains. Developmental and Comparative Immunology 27, 55–77 (2002)

[78] Levenshtein, V.I.: Binary codes capable of correcting deletions, insertions, and reversals. Tech. Rep. 8, Soviet Physics Doklady (1966)

[79] Li, H., Durbin, R.: Fast and accurate short read alignment with burrows–wheeler transform. Bioinformatics 25(14), 1754–1760 (2009), http://dx.doi.org/10.1093/bioinformatics/btp324

[80] Li, H., Handsaker, B., Wysoker, A., Fennell, T., Ruan, J., Homer, N., Marth, G., Abecasis, G., Durbin, R., Subgroup, .G.P.D.P.: The sequence alignment/map format and SAMtools. Bioinformatics 25(16), 2078–2079 (2009), http://dx.doi.org/10.1093/bioinformatics/btp352

[81] Li, R., Yu, C., Li, Y., Lam, T.W., Yiu, S.M., Kristiansen, K., Wang, J.: SOAP2: an improved ultrafast tool for short read alignment. Bioinformatics 25(15), 1966–1967 (2009), http://dx.doi.org/10.1093/bioinformatics/btp336

[82] Lin, H.H., Liao, Y.C.: Accurate binning of metagenomic contigs via automated clustering sequences using information of genomic signatures and marker genes. Scientific reports 6, 24175 (2016)

[83] Lo, W., Lee, C., Lee, C., Lyu, P.: CPDB: a database of circular permutation in proteins. Nucleic Acids Research 37, 328–332 (2009)

[84] Maes, M.: On a cyclic string-to-string correction problem. Information Processing Letters 35(2), 73 – 78 (1990), http://www.sciencedirect.com/science/article/pii/002001909090109B

[85] Maes, M.: Polygonal shape recognition using string-matching techniques. Pattern Recognition 24(5), 433 – 440 (1991)

[86] Manber, U., Myers, G.: Suffix arrays: A new method for on-line string searches. SIAM Journal on Computing 22(5), 935–948 (1993), https://doi.org/10.1137/0222058

[87] Marschall, T., Rahmann, S.: Efficient exact motif discovery. Bioinformatics 25(12) (2009)

[88] Marzal, A., Barrachina, S.: Speeding up the computation of the edit distance for cyclic strings. In: Proceedings 15th International Conference on Pattern Recognition. ICPR-2000. vol. 2, pp. 891–894 vol.2 (2000)

[89] McCreight, E.M.: A space-economical suffix tree construction algorithm. J. ACM 23(2), 262–272 (Apr 1976), http://doi.acm.org/10.1145/321941.321946

[90] Miyata, T., Yasunaga, T.: Molecular evolution of mRNA: a method for estimating evolutionary rates of synonymous and amino acid substitutions from homologous nucleotide sequences and its application. Journal of Molecular Evolution 16(1), 23–36 (1980)

[91] Mollineda, R.A., Vidal, E., Casacuberta, F.: Cyclic sequence alignments: approximate versus optimal techniques. International Journal of Pattern Recognition and Artificial Intelligence 16(03), 291–299 (2002), https://doi.org/10.1142/S0218001402001678

[92] Mosig, A., Hofacker, I., Stadler, P.: Comparative analysis of cyclic sequences: Viroids and other small circular RNAs. In: Giegerich, R., Stoye, J. (eds.) Lecture Notes in Informatics. pp. 93–102. Proceedings GCB (2006)

[93] Munro, J.I., Raman, V., Rao, S.S.: Space efficient suffix trees. Journal of Algorithms 39(2), 205–222 (2001)

[94] Myers, E.W.: An overview of sequence comparison algorithms in molecular biology. University of Arizona. Department of Computer Science (1991)

[95] Myers, G.: A fast bit-vector algorithm for approximate string matching based on dynamic programming. Journal of the ACM (JACM) 46, 395–415 (1999)

[96] Navarro, G.: A guided tour to approximate string matching. ACM Computing Surveys 33(1), 31–88 (2001)

[97] Navarro, G., Raffinot, M.: Fast and flexible string matching by combining bit-parallelism and suffix automata. Journal of Experimental Algorithmics (JEA) 5, 4 (2000)

[98] Navarro, G., Raffinot, M.: Flexible Pattern Matching in Strings: Practical Online Search Algorithms for Texts and Biological Sequences. Cambridge University Press (2002)

[99] Needleman, S.B., Wunsch, C.D.: A general method applicable to the search for similarities in the amino acid sequence of two proteins. Journal of Molecular Biology 48(3), 443–453 (1970)

[100] Nguyen, N., Hickey, G., Zerbino, D.R., Raney, B.J., Earl, D., Armstrong, J., Kent, W.J., andBenedict Paten, D.H.: Building a pan-genome reference for a population. Journal of Computational Biology 22(5), 387–401 (2015)

[101] Nong, G., Zhang, S., Chan, W.H.: Linear suffix array construction by almost pure induced-sorting. In: 2009 Data Compression Conference (DCC 2009). pp. 193–202 (2009), https://doi.org/10.1109/DCC.2009.42

[102] Nsira, N.B., Elloumi, M., Lecroq, T.: On-line string matching in highly similar DNA sequences. Mathematics in Computer Science 11(2), 113–126 (2017), https://doi.org/10.1007/s11786-016-0280-2

[103] Nsira, N.B., Lecroq, T., Elloumi, M.: A fast Boyer-Moore type pattern matching algorithm for highly similar sequences. IJDMB 13(3), 266–288 (2015), https://doi.org/10.1504/IJDMB.2015.072101

[104] Pavesi, G., Mereghetti, P., Mauri, G., Pesole, G.: Weeder web: discovery of transcription factor binding sites in a set of sequences from co-regulated genes. Nucleic Acids Research 32(Web-Server-Issue), 199–203 (2004)

[105] Peterlongo, P., Sacomoto, G.T., do Lago, A.P., Pisanti, N., Sagot, M.F.: Lossless filter for multiple repeats with bounded edit distance. Algorithm Mol Biol 4(3) (2009)

[106] Peterlongo, P., Pisanti, N., Boyer, F., do Lago, A.P., Sagot, M.F.: Lossless filter for multiple repetitions with Hamming distance. JDA 6(3), 497–509 (2008)

[107] Pisanti, N., Giraud, M., Peterlongo, P.: Filters and seeds approaches for fast homology searches in large datasets. In: Elloumi, M., Zomaya, A.Y. (eds.) Algorithms in computational molecular biology, chap. 15, pp. 299–320. John Wiley & sons (2010)

[108] Pissis, S.P.: MoTeX-II: structured motif extraction from large-scale datasets. BMC Bioinformatics 15(1), 1–12 (2014)

[109] Pissis, S.P., Retha, A.: Generalised implementation for fixed-length approximate string matching under Hamming distance and applications. In: Proceedings of the 2015 IEEE International Parallel and Distributed Processing Symposium Workshop. pp. 367–374. IPDPSW '15, IEEE Computer Society, Washington, DC, USA (2015), http://dx.doi.org/10.1109/IPDPSW.2015.106

[110] Pissis, S.P., Retha, A.: Dictionary Matching in Elastic-Degenerate Texts with Applications in Searching VCF Files On-line. In: D'Angelo, G. (ed.) 17th International Symposium on Experimental Algorithms (SEA 2018). Leibniz International Proceedings in Informatics (LIPIcs), vol. 103, pp. 16:1–16:14. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2018), http://drops.dagstuhl.de/opus/volltexte/2018/8951

[111] Pissis, S.P., Stamatakis, A., Pavlidis, P.: MoTeX: A word-based HPC tool for motif extraction. In: Gao, J. (ed.) ACM Conference on Bioinformatics, Computational Biology and Biomedical Informatics. ACM-BCB 2013. p. 13. ACM, Washington, DC, USA (2013)

[112] Ponting, C.P., Russell, R.B.: Swaposins: circular permutations within genes encoding saposin homologues. Trends in Biochemical Sciences 20(5), 179 – 180 (1995)

[113] Pruitt, K.D., Tatusova, T., Maglott, D.R.: NCBI reference sequences (RefSeq): a curated non-redundant sequence database of genomes, transcripts and proteins. Nucleic Acids Research 35(suppl_1), D61–D65 (2007), http://dx.doi.org/10.1093/nar/gkl842

[114] Raghupathi, W., Raghupathi, V.: Big data analytics in healthcare: promise and potential. Health information science and systems 2(1), 3 (2014)

[115] Rahn, R., Weese, D., Reinert, K.: Journaled string tree—a scalable data structure for analyzing thousands of similar genomes on your laptop. Bioinformatics 30(24), 3499–3505 (2014), http://dx.doi.org/10.1093/bioinformatics/btu438

[116] Rajan, S., Sidhu, S.S.: Chapter 1 - simplified synthetic antibody libraries. In: Wittrup, K.D., Verdine, G.L. (eds.) Protein Engineering for Therapeutics, Part A, Methods in Enzymology, vol. 502, pp. 3 – 23. Academic Press (2012), http://www.sciencedirect.com/science/article/pii/B978012416039200001X

[117] Rakocevic, G., Semenyuk, V., Spencer, J., Browning, J., Johnson, I., Arsenijevic, V., Nadj, J., Ghose, K., Suciu, M.C., Ji, S.G., Demir, G., Li, L., Toptas, B.C., Dolgoborodov, A., Pollex, B., Spulber, I., Glotova, I., Komar, P., Stachyra, A., Li, Y., Popovic, M., Lee, W.P., Kallberg, M., Jain, A., Kural, D.: Fast and accurate genomic analyses using genome graphs. bioRxiv (2018), https://www.biorxiv.org/content/early/2018/03/20/194530

[118] Rasmussen, K., Stoye, J., Myers, E.: Efficient q-gram Filters for finding all epsilon-matches over a given length. J Comput Biol 13(2), 296–308 (2006)

[119] Rice, P., Longden, I., Bleasby, A.: EMBOSS: the European molecular biology open software suite. Trends in genetics : TIG 16(6), 276—277 (June 2000), http://dx.doi.org/10.1016/S0168-9525(00)02024-2

[120] Ritchie, D.M., Kernighan, B.W., Lesk, M.E.: The C programming language. Prentice Hall Englewood Cliffs (1988)

[121] Robinson, D., Foulds, L.: Comparison of phylogenetic trees. Mathematical Biosciences 53(1-2), 131–147 (1981)

[122] Rojas, A., Romeu, A.: A sequence analysis of the betaglucosidase subfamily B. FEBS Letters 378(1), 93–97, https://febs.onlinelibrary.wiley.com/doi/abs/10.1016/0014-5793%2895%2901412-8

[123] Salomaa, A.: Counting (Scattered) Subwords, pp. 495–510. WORLD SCIENTIFIC (2012), https://www.worldscientific.com/doi/abs/10.1142/9789812562494_0061

[124] Sellers, P.H.: On the theory and computation of evolutionary distances. SIAM Journal on Applied Mathematics 26(4), 787–793 (1974)

[125] Sellers, P.H.: The theory and computation of evolutionary distances: Pattern recognition. Journal of Algorithms 1(4), 359–373 (1980)

[126] Sheikhizadeh, S., Schranz, M.E., Akdel, M., de Ridder, D., Smit, S.: Pantools: representation, storage and exploration of pan-genomic data. Bioinformatics 32(17), 487–493 (2016)

[127] Sievers, F., Wilm, A., Dineen, D., Gibson, T.J., Karplus, K., Li, W., Lopez, R., McWilliam, H., Remmert, M., Söding, J., Thompson, J.D., Higgins, D.G.: Fast, scalable generation of high-quality protein multiple sequence alignments using Clustal Omega. Molecular Systems Biology 7(1) (2011)

[128] Silva, R.M., Pratas, D., Castro, L., Pinho, A.J., Ferreira, P.J.S.G.: Three minimal sequences found in Ebola virus genomes and absent from human DNA. Bioinformatics 31(15), 2421–2425 (2015), http://dx.doi.org/10.1093/bioinformatics/btv189

[129] Simpson, J.T., Durbin, R.: Efficient de novo assembly of large genomes using compressed data structures. Genome research 22(3), 549–556 (2012)

[130] Sinha, S., Tompa, M.: YMF: A program for discovery of novel transcription factor binding sites by statistical overrepresentation. Nucleic Acids Res 31(13), 3586–3588 (2003)

[131] Sirén, J.: Indexing variation graphs. In: Fekete, S.P., Ramachandran, V. (eds.) Proceedings of the Ninteenth Workshop on Algorithm Engineering and Experiments, ALENEX 2017, Barcelona, Spain, Hotel Porta Fira, January 17-18, 2017. pp. 13–27. SIAM (2017)

[132] Smith, T.F., Waterman, M.S.: Comparison of biosequences. Advances in Applied Mathematics 2(4), 482 – 489 (1981), http://www.sciencedirect.com/science/article/pii/0196885881900464

[133] Sokal, R.: A statistical method for evaluating systematic relationships. Univ Kans Sci Bull 38, 1409–1438 (1958), http://ci.nii.ac.jp/naid/10011579647/en/

[134] Stamatakis, A.: RAxML version 8: A tool for phylogenetic analysis and post-analysis of large phylogenies. Bioinformatics (2014)

[135] Stemmer, M., Thumberger, T., del Sol Keyer, M., Wittbrodt, J., Mateo, J.L.: CCTop: An intuitive, flexible and reliable CRISPR/Cas9 target prediction tool. PLOS ONE 10(4), 1–11 (04 2015), https://doi.org/10.1371/journal.pone.0124633

[136] Stratton, M.: Genome resequencing and genetic variation. Nat Biotech 26(1), 65–66 (2008)

[137] Sukumaran, J., Holder, M.T.: DendroPy: a python library for phylogenetic computing. Bioinformatics 26(12), 1569–1571 (2010), http://dx.doi.org/10.1093/bioinformatics/btq228

[138] The 1000 Genomes Project Consortium: A global reference for human genetic variation. Nature 526(7571), 68–74 (2015)

[139] The C. elegans Deletion Mutant Consortium: Large-scale screening for targeted knockouts in the caenorhabditis elegans genome. G3: Genes, Genomes, Genetics 2(11), 1415–1425 (2012), http://www.g3journal.org/content/2/11/1415

[140] The Computational Pan-Genomics Consortium: Computational pan-genomics: status, promises and challenges. Briefings in Bioinformatics pp. 1–18 (2016)

[141] Thompson, O., Edgley, M., Strasbourger, P., Flibotte, S., Ewing, B., Adair, R., Au, V., Chaudry, I., Fernando, L., Hutter, H., et al.: The million mutation project: a new approach to genetics in caenorhabditis elegans. Genome research pp. gr–157651 (2013)

[142] Ukkonen, E.: Finding approximate patterns in strings. Journal Algorithms 6(1), 132–137 (1985)

[143] Ukkonen, E.: Approximate string-matching with q-grams and maximal matches. Theoretical Computer Science 92(1), 191 – 211 (1992), http://www.sciencedirect.com/science/article/pii/0304397592901434

[144] Wang, C.K., Kaas, Q., Chiche, L., Craik, D.J.: Cybase: a database of cyclic protein sequences and structures, with applications in protein discovery and engineering. Nucleic acids research 36(suppl_1), D206–D210 (2007)

[145] Wang, Z., Wu, M.: Phylogenomic reconstruction indicates mitochondrial ancestor was an energy parasite. PLoS ONE 9(10), 1–11 (10 2014)

[146] Waterman, M.S., Smith, T.F.: Identification of common molecular subsequences. Journal of Molecular Biology 147(1), 195–197 (1981)

[147] Weiner, J., Bornberg-Bauer, E.: Evolution of circular permutations in multidomain proteins. Mol Biol Evol 23 (2006)

[148] Weiner, P.: Linear pattern matching algorithms. In: 14th Annual Symposium on Switching and Automata Theory (swat 1973). pp. 1–11 (1973)

[149] Wheeler, T.J.: Large-scale neighbor-joining with NINJA. In: Salzberg, S.L., Warnow, T. (eds.) Algorithms in Bioinformatics. pp. 375–389. Springer Berlin Heidelberg, Berlin, Heidelberg (2009)

[150] Wright, A.H.: Approximate string matching using within-word parallelism. Software - Practice and Experience 24(4), 337–362 (1994)

[151] Wu, S., Manber, U.: Fast text searching: allowing errors. Communications of the ACM 35(10), 83–91 (1992)

[152] Zhang, Y., Zaki, M.: EXMOTIF: efficient structured motif extraction. Algorithms for Molecular Biology 1(1), 1–18 (2006)