Date of acceptance Grade

Instructor

Chaining with Maximal Exact Matches for Fast and Accurate Approximation of Edit Distance

Peter Porttinen

Helsinki December 21, 2020

Master's Thesis

UNIVERSITY OF HELSINKI
Department of Computer Science

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Laitos — Institution — Department
Faculty of Science	Department of Computer Science

Tekijä — Författare — Author
Peter Porttinen

Työn nimi — Arbetets titel — Title

Chaining with Maximal Exact Matches for Fast and Accurate Approximation of Edit Distance

Oppiaine — Läroämne — Subject
Computer Science

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Master's Thesis	December 21, 2020	47 pages + 2 appendix pages

Tiivistelmä — Referat — Abstract

Computing an edit distance between strings is one of the central problems in both string processing and bioinformatics. Optimal solutions to edit distance are quadratic to the lengths of the input strings. The goal of this thesis is to study a new approach to approximate edit distance. We use a chaining algorithm presented by Mäkinen and Sahlin in "Chaining with overlaps revisited" CPM 2020 implemented verbatim. Building on the chaining algorithm, our focus is on efficiently finding a good set of anchors for the chaining algorithm. We present three approaches to computing the anchors as maximal exact matches: Bi-Directional Burrows-Wheeler Transform, Minimizers, and lastly, a hybrid implementation of the two.

Using the maximal exact matches as anchors, we can efficiently compute an optimal chaining alignment for the strings. The chaining alignment further allows us to determine all such intervals where mismatches occur by looking at which sequences are not in the chain. Using these smaller intervals lets us approximate edit distance with a high degree of accuracy and a significant speed improvement. The methods described present a way to approximate edit distance in time complexity bounded by the number of maximal exact matches.

ACM Computing Classification System (CCS):
Applied computing-Life and medical sciences-Computational biology-Molecular sequence analysis

Theory of computation-Design and analysis of algorithms-Data structures design and analysis-Pattern matching

Avainsanat — Nyckelord — Keywords

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — övriga uppgifter — Additional information

Contents

1 Introduction 1

1.1 Notation 2

1.2 Basic Concepts 2

1.2.1 Levenshtein edit distance 2

1.2.2 Maximal Exact Match 2

1.2.3 Suffix Array 4

1.2.4 Rank and Select 4

1.2.5 BWT - BDBWT 5

1.2.6 K-mers and Minimizers 9

2 Finding Maximal Exact Matches 11

2.1 Using BD-BWT 11

2.1.1 Implementation 12

2.1.2 Analysis on the implementation 18

2.1.3 Optimization - Parallelization 20

2.2 Using Minimizers 20

2.2.1 Implementation 21

2.2.2 Analysis 24

2.2.3 Optimization 25

2.3 Hybrid Solution 26

2.3.1 Necessary additional data structures 26

2.3.2 Suffix- and inverse suffix arrays 27

2.3.3 Longest common prefix array 28

2.3.4 Partitions and conversion 28

2.3.5 Implementation 30

2.3.6 Optimization 33

2.3.7 Analysis 33

3 Chaining 34

3.1 Motivation 34

3.2 Definition of parts 35

3.3 Implementation 36

3.4 Edit Distance from chaining output 39

4 Results **40**

5 Discussion **44**

References **46**

Appendices

1 Results data table

2 Location of implementation

1 Introduction

Edit distance computation and pair-wise string alignment are two of the most fundamental problems in both string processing and bioinformatics. Optimal results can only be guaranteed at the cost of quadratic worst-case time complexity, which is shown to be hard to improve upon unless the strong exponential time hypothesis ($SETH$) is proven false [1].

In this thesis, we present an alternative approach built on a chaining algorithm defined in a recent paper by Mäkinen and Sahlin: "Chaining with overlaps revisited" [2]. Their algorithm used with maximal exact matches allows us to compute an efficient chaining-alignment between two strings. For then computing the edit distance, our main interest lies in the sequences absent from the chain. The absent sequences illustrated in Figure 1 contain all the mismatches between the input texts and ultimately allow us to approach the edit distance computation from a different angle.

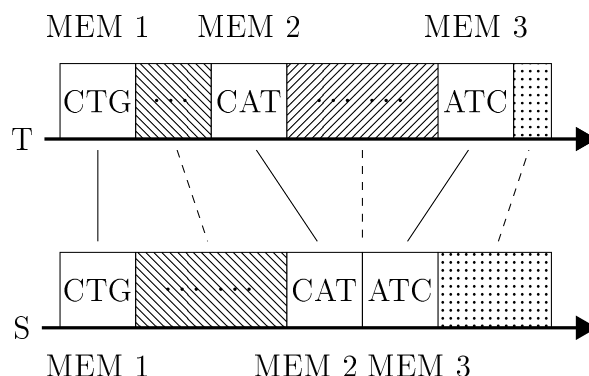

Figure 1: Extracted substrings where the texts differ in between maximal exact matches (MEMs). Note that the second substring between MEM 2 and 3 is empty on the string S. Our implementation only requires edit distance computation for the dashed and dotted sequences above.

The first result that our implementation gives is that the chaining algorithm can be used to get a good approximation of the edit distance with a little extra work. Our implementation is not able to guarantee optimality of the result, but the results are still often optimal regarding edit distance and computable in a fraction of the time taken to guarantee it.

We begin this thesis with the basic notation and brief introductions to the concepts used. Second, we discuss the three different implementations used to acquire the maximal exact matches. Following a logical order, we proceed to give a concrete example of how we use the chaining algorithm to produce our results. And in the last chapters, we will examine our experimental results and discuss possible future optimizations and implications regarding the work and results.

1.1 Notation

We assume that all indexes are zero-based, $A = A[0..n-1]$, where A is an array containing n elements and $A[i],\ 0 \leq i \leq n-1$, accesses the $i-$th element. We define intervals with notation $([a..b])$ where the endpoints $0 \leq a \leq b \leq n-1$ are inclusive. To enforce an interval being within a data structure we denote $A[a..b]$ as interval $([a..b])$ within A.

We consider input strings to be denoted by $T = T[0..n-1]$ and $S[0..m-1]$, their reverse strings as T^R and S^R respectively. T' is a substring of T, $T' = T[i..j] \subseteq T[0..n-1]$ where the endpoints $0 \leq i \leq j \leq n-1$ are bounded by the parent string T.

A prefix T_p refers to a substring starting at the beginning of the parent string, $T_p[0..i] \subseteq T[0..n-1], 0 \leq i \leq n-1$. On the other side, suffix T_s is a substring ending at the last symbol of the parent string, $T_s[j..n-1],\ 0 \leq j \leq n-1$.

We use a dot-notation $"aa"."bb" = "aabb"$ to represent concatenation between two strings. With $T[0..n-1]$, and $S[0..m-1]$, we define

$$C[0..(n-1+m-1)] = T.S \text{ where } C[0..n-1] = T \text{ and } C[n..(n-1+m-1)] = S.$$

We assume that all texts and strings are constructed of an alphabet Σ and a special character $\$ \notin \Sigma$, a symbol that does not exist in the alphabet and is lexicographically smaller than any character in it. Furthermore, for all input texts T and their reverse T^R we define that $T[n-1] = \$$ and $T^R[n-1] = \$$.

1.2 Basic Concepts

1.2.1 Levenshtein edit distance

The smallest number of insertions, deletions, and substitutions required to transform one string to another often refers to a string metric called the Levenshtein distance [3]. Though often synonymous, it is important to note that edit distance can also be implemented with, for example, the Hamming distance- or the longest common prefix metrics.

The results of this thesis require, and use the Levenshtein variation, which in the remainder will be strictly referred to as just the "edit distance" formally defined in Definition 1.

Definition 1 (Edit Distance) *Edit distance is a metric denoting the number of insertions, deletions, and substitutions required to convert a string T to a string S.*

1.2.2 Maximal Exact Match

One key concept of string processing is the concept of a match, a common substring found in both strings. An exact match then is match that appears as a strictly

continuous and matching substring.

Definition 2 (Exact Match) *Exact match P is a continuous, completely matching substring in strings $T[0..n-1]$ and $S[0..m-1]$.*

$$P = T[i..j] = S[i'..j'], \quad 0 \leq i \leq j \leq n-1, \quad 0 \leq i' \leq j' \leq m-1.$$

The algorithms presented in this thesis depend heavily on the number of matches. To ensure that the algorithms can be utilized in the most efficient manner possible, we want to select the matches with the least amount of redundancy and overlap. Exact matches by Definition 2 by themselves are not sufficient as the matches could be selected in manner where multiple subsequences of one match could be matched, giving a high number of redundant matches when the goal to maximize the coverage using continuous strings. Using definitions for left- and right-maximality, Definition 3, allows us to grow the matches and reduce the total amount of matches.

Definition 3 (Left- or right-maximal match) *Exact match P between strings $T[0..n-1]$ and $S[0..m-1]$, $P = T[i..j] = S[i'..j']$ is considered left-maximal if, and only if $T[i-1] \neq S[i'-1]$. Likewise, if $T[j+1] \neq S[j'+1]$ we consider the match to be right-maximal.*

For the border cases $T'[i..j]$, $i = 0$, or $j = n-1$, we define that a match is left- and right-maximal respectively.

For example, given strings "ABCDE" and "XBCDY", all possible exact matches are substrings: "B","C","D","BC","CD", and "BCD". The matches "B", "BC", and "BCD" are left-maximal as $A \neq X$, and similarly matches "D", "CD", "BCD", are right-maximal as $E \neq Y$. The match "BCD" fills both conditions and is the only match in the example that is both left- and right-maximal. Selecting the maximal exact match, by Definition 4, "BCD" provides a good coverage of the text while minimizing the number of matches.

Definition 4 (Maximal Exact Match (MEM)) *Exact match P is maximal if and only if it is both left- and right-maximal by Definition 3.*

Naturally, any exact match can be extended to cover either the whole input string or until a mismatch is found. In either case, it allows the observation, Observation 1, that any exact match must be a substring to a maximal exact match, or a maximal exact match itself.

Observation 1 *Each exact match $p = T[x..y] = S[x'..y']$ is either a maximal exact match itself $p = P = T[i..j] = S[i'..j']$, or a substring to a maximal exact match with $x \geq i$, $x' \geq i'$, $y \leq j$, and $y' \leq j'$.*

1.2.3 Suffix Array

Originally presented by Manber and Myers in 1990 [4] as a space-efficient alternative to a suffix tree. Suffix array, as suggested by name, stores indexed corresponding to each suffix sorted into a suffix-wise lexicographic order. We define suffix array formally as Definition 5

Definition 5 (Suffix Array (SA)) *Suffix array SA_T for text T is a lexicographically sorted array containing integers "i" denoting beginning indexes of suffixes $T[i..n-1]$, $0 \leq i \leq n-1$, where $T[n-1] = \$$. With $T[SA_T[i]..n-1] < T[SA_T[i+1]..n-1]$, for all $0 \leq i < n-1$.*

Notable property of the suffix array is that the first element $SA_T[0]$ will always correspond to the last suffix (symbol) of the original string: $SA_T[0] = n-1$ and $T[SA_T[0]] = T[n-1..n-1] = \$$.

Pattern matching with a suffix array often employs the Longest Common Prefix array (LCP array) as an auxiliary data structure to improve the efficiency of the search [4]. An LCP array stores the lengths of all LCP values between two adjacent suffixes in the array. LCP and LCP-array formally as Definitions 6 and 7.

Definition 6 (Longest common prefix (LCP)) *$LCP(T, S)$ is a function that returns the longest matching prefix between strings T and S.*

$$LCP(T, S) = max\{n : (T[0..n] = S[0..n])\}.$$

The border case $T[0] \neq S[0]$ is defined as $LCP(T, S) = 0$.

Definition 7 (Longest Common Prefix array (LCP array)) *Let SA_T be a suffix array of text T and LCP_T its LCP array, then $LCP_T[0] = 0$, and for each $LCP_T[i], 1 \leq i \leq n-1$ it is defined that $LCP_T[i] = LCP(T[SA_T[i]], T[SA_T[i-1]])$, where LCP is a function defined in Definition 6.*

1.2.4 Rank and Select

Rank and select queries are succinct techniques allowing efficient lookups often used with bitvectors, see Definitions 8, and 9 respectively. The operations and the ideas behind them are, however, implemented on other data structures as well.

Definition 8 (Rank) *Let $A[0..n-1]$ be a vector of length n, with alphabet of Σ. We define $rank_c(i)$ as a function returning the total count of symbol $c \in \Sigma$ contained in $A[0..i]$.*

Definition 9 (Select) *Let $A[0..n-1]$ be a vector of length n, with alphabet of Σ. We define $select_c(i)$ as a function returning index containing the i-th occurrence of $c \in \Sigma$ in the vector A.*

After initialization, both the rank and select queries can be implemented in $\mathcal{O}(1)$ time complexity for bitvectors by making use of the precomputed values and a constant time random access. With larger alphabets this becomes less feasible, and the time complexity grows linear to the size of the alphabet $\mathcal{O}(\sigma)$. The constant time complexity of the queries is a result of precomputing all values in a linear time relative to the size of the array, using $o(n)$ bits of extra space [5]. In our implementations we import and use the rank and select data structures provided by *SDSL* [6].

1.2.5 BWT - BDBWT

Burrows-Wheeler Transform as originally presented by Burrows and Wheeler in 1994 [7] is a powerful data structure originally intended for data-compression. More recently, Burrows-Wheeler Transforms has also proved to have useful properties for pattern matching. We denote Burrows-Wheeler Transform built on text T by BWT_T in the remainder of this thesis and define it formally in Definition 10.

Definition 10 (Burrows-Wheeler Transform) *Let $T[0..n-1]$ be a string such that $T[n-1] = \$$. Burrows-Wheeler Transform BWT_T is created by first taking all rotations R_i of the string, where*

$$R_i = \begin{cases} T[i..n-1].T[0..i-1], & i > 0 \\ T[0..n-1], & i = 0 \end{cases}$$

The index is created from the lexicographically sorted rotations $BWT_T[i] = R_{lex(i)}[n-1], \quad 0 \leq i \leq n-1$ where $lex(i)$ is the lexicographically $i-th$ index.

The lexicographic order for rotations follows the same logic as the suffixes of the suffix array, as can be seen in Table 1. We note the relationship between the two data structures by observing that each sorted rotation $R_{lex(i)}$ has a prefix corresponding to a suffix $T[SA_T[i]..n-1]$ indexed on the same position.

Table 1: BWT_T built on text $T = GCA\$$. We can observe that as long as the text is suffixed by the special character \$, the longest common prefix between the sorted rotations and the suffix string on same index i matches the suffix exactly.

R_i	$R_{lex(i)}$ Sorted	BWT_T	$T[SA_T[i]..n-1]$	$SA_T[i]$
GCA\$	\$GCA	A	\$	3
CA\$G	A\$GC	C	A\$	2
A\$GC	CA\$G	G	CA\$	1
\$GCA	GCA\$	\$	GCA\$	0

Observation 2 (BWT and Suffix Array) *BWT and SA indexes can be translated to and from each other as the prefixes of the BWT rotations match the suffixes indexed by SA. BWT can be constructed from SA with a linear scan over the array as follows:*

$$BWT[i] = \begin{cases} T[SA[i]-1], & \textit{iff } i > 0 \\ T[n-1] = \$, & \textit{iff } i = 0 \end{cases}$$

Observation 2 directly gives us an efficient way to translate each index of the BWT index into the corresponding suffix (the position where it begins) found in the original text by reversing the equation. Combining this notion with a known length of the substring, exact pattern matching can be accomplished by BWT.

As written by Ferragina and Manzini [8], pattern matching with BWT can be further be augmented with LF-mapping. Using the LF-mapping, suffix-wise backtracking can be implemented for the index, allowing efficient reconstruction of a substring in T.

LF-mapping maps the position of the ith suffix to the position of the preceding ((i-1)-th) suffix. To compute the mapping efficiently, we need to know the number of occurrences of each symbol $c \in \Sigma$ beforehand. If the number of occurrences is not easily retrievable from the index, then the occurrences can be easily computed through a linear scan of the index. Algorithm 1 outlines the implementation as a pseudocode for a linear timeconstruction, see Observation 3.

Algorithm 1: LF-Mapping

Input : $BDBWT$ index idx of size n, and a boolean value fwd.
Output: An array LF containing LF-mapping for the index idx.

1 Initialize $C_l[c] = 0, c \in \Sigma$.
2 Initialize $C_g[c], c \in \Sigma$ as an array containing the number of times c appears in the input
3 Initialize LF as an empty array of length n.
4 **for** $i \in [0..n-1]$ **do**
5   Initialize c as a symbol
6   **if** fwd **then**
7         $c = idx.BWT_T[i]$
8   **else**
9         $c = idx.BWT_{T^R}[i]$ `// For reverse text`
10     $LF[i] = C_g[c] + C_l[c]$
11     $C_l[c] = C_l[c] + 1$ `// Increment number of symbols` c `seen so far.`
12 **return** LF

This approach is a common way to make use of the concept [8] to transform a BWT index into an LF-mapping array. The approach utilizes knowledge of the total number of occurrences each symbol has in the index (C_g) and keeps track (C_l) of how many times they have occurred during the computation.

Observation 3 (LF-mapping) *LF-mapping for BWT_T of size n can be constructed in a linear time $\mathcal{O}(n)$ scan over the index relative to the size of the index.*

Recalling the similarity between BWT and SA from Observation 2. We define `leftExtend`, Definition 11, as a function that takes a BWT interval $([i..j])$ corresponding to the longest common prefix P and an extending character c as an input and returns an interval $([i'..j'])$ corresponding to the longest common prefix $c.P$. Definition 12 returns all possible symbols left-extension can be accomplished with as an output.

Definition 11 (leftExtend) *A function $leftExtend(([i..j]), c)$ takes a BWT interval $([i..j])$ corresponding to suffixes with the longest common prefix P, and a symbol c as a parameter. The function returns a BWT interval $([i'..j'])$ corresponding to a SA interval where each suffix shares the longest common prefix $c.P$, or a negative interval if the BWT interval $([i..j])$ does not contain the symbol c.*

Definition 12 (enumerateLeft) *We define $enumerateLeft([i..j])$ as a function that returns each distinct character $BWT_T[k], i \leq k \leq j$ from the interval in a lexicographically increasing order. The function takes a BWT interval $([i..j])$ as an input.*

Pattern matching using BWT is limited to only towards the beginning of the string. The limitation becomes critical when considering the problem of determining all maximal exact matches, as maximal exact matches require being that each match is both left- and right-maximal.

A way around the limitation is accomplished with Bi-Directional BWT index in a paper by Schnattinger, Ohlebusch, and Gog in 2012 [9]. The paper describes Bi-Directional traversal in a BWT index by synchronizing extensions between BWT_T and BWT_{T^R}. The synchronization is based on an observation that any left extension in the BWT_T (forward index) can be defined as a right extension in BWT_{T^R} (reverse index). In the remainder, we will refer to Bi-Directional Burrows-Wheeler Transform as $BDBWT$.

Synchronization between the forward and backward indexes of $BDBWT$ is implemented through the realization that they must share the same set of characters, and that the i-th occurrence of a character c in the forward index is also the i-th occurrence of c in the reverse index. Thus, even if the actual transform is different, knowing the symbol counts for one index allows us to arithmetically determine the interval in the other index. Formally the synchronization is defined in Definition 13.

Definition 13 (*BDBWT* synchronization) *Synchronization between forward and reverse intervals in BDBWT interval pair* $([i..j],[x..y])$ *can be done by extending from* $([i..j])$ *to* $([i'..j'])$ *with symbol c, and defining the extension from interval* $([x..y])$ *to* $([x'..y'])$ *arithmetically through the known intervals.*

$$([x'..y']) = \begin{cases} x' = x + d \\ y' = x' + j' - i' \end{cases}$$

Where d is the count of characters smaller than c within the BWT interval $([i..j])$*.*

A more complicated, constant time synchronization for *BDBWT* is possible [10], requiring auxiliary data structures and memory compared to the approach used in this thesis, justified to be used as our used in this thesis is small. The synchronization is still efficient and bounded by the size of the alphabet, see Lemma 1.

Lemma 1 (*BDBWT* synchronization can be done efficiently) *Any left extension from* $([i..j])$ *to* $([i'..j'])$ *on* BWT_T*, has a matching right-extension from* $([x..y])$ *to* $([x'..y'])$ *on* BWT_{T^R}*. This synchronization can arithmetically be done in* $\mathcal{O}(\sigma)$ *time by making use of the ordered nature, rank queries, and ability to enumerate symbols from a given interval.*

Proof Consider a BWT_T index built on text T and BWT_{T^R} from its reverse T^R. Assume that the BWT data structure is equipped with a rank-function as well as a function to enumerate all distinct characters appearing on a given interval on the index. Lastly, recall the relationship defined between SA and BWT.

Consider a string $P[0..k-1]$ of length k, as the longest common prefix to suffixes indexed on an interval $([i..j])$ in SA_T. Similarly, on T^R consider $P^R[0..k-1]$ as the longest common prefix to suffixes indexed on interval $([x..y])$ on SA_{T^R}.

Let c be a character in BWT_T interval $([i..j])$ such that there exists an index "a" where $BWT_T[a] = c, \quad i \leq a \leq j$. Let $([i'..j'])$ be interval in BWT_T extended left with c from $([i..j])$, thus corresponding to a SA_T interval with the longest common prefix of $c.P$. For the reverse, $P^R.c$, there has to be an interval $([x'..y'])$ with $P^R.c$ as the longest common prefix in SA_{T^R}.

Naturally, the longest common prefix between $P^R.c$ and P^R equals to P^R, and since the former corresponds to the interval $([x..y])$, we conclude that the interval $([x'..y'])$ of $P^R.c$ must be a subset to the interval $([x..y])$ of P^R, $([x'..y']) \subseteq ([x..y])$.nd

For simplicity and clarity, we define `Range(Index,i,j,c)` [1] as a function returning the number of symbols lexicographically smaller than c on the interval given as a parameter in time bounded above as with the length of the interval. The time required to precompute all values for the range function is dominated by and can be charged to the construction of the indexes.

[1] Not defined as function in the implementation, only for clarity within the lemma.

By Definition 8, Rank query is a $\mathcal{O}(1)$ operation, given that the supporting data-structure is initialized. Computing the local counts with `Range` will then result in time complexity of $\mathcal{O}(\sigma)$ time, where σ is small, typically four with DNA. Or $\mathcal{O}(1)$ after precomputing all possible values a, b once in a linear can over the index in $\mathcal{O}(n)$ charged to construction of the index.

Utilizing the information got from the `Range`-function as well as the knowledge that $([x'..y']) \subseteq ([x..y])$, we know that there exists $d = range(BWT_T, i, j, c)$ suffixes lexicographically smaller than $P^R.c$. The starting index of the right-extended reverse interval can thus be defined as $x' = x + d$.

The number of occurrences of $c.P$ and $P^R.c$ must be the same due to the symmetry between a string and it's reverse. To define the ending index y' of the reverse interval $([x'..y'])$, we add the length of the extended forward interval $([i'..j'])$ to the beginning index of the reverse interval $y' = x' + (j' - i') - 1$.

Finally, extending the interval $([i..j])$ to the left with character c, we get synchronization between BWT_T and BWT_{T^R} by defining the resulting reverse interval $([x'..y'])$ through a function taking the original reverse interval $([x..y])$ and the resulting forward interval as parameters:

$$([x'..y']) = \begin{cases} d = Range(BWT_T, i, j, c) \\ x' = x + d \\ y' = x' + (j' - i') - 1 \end{cases} \qquad \square$$

For $BDBWT$, we define that `leftExtend` returns an interval pair $([i..j], [i'..j'])$ where the forward interval is computed analogously to a regular BWT and the reverse with the synchronization defined above. The function `rightExtend` is defined same as `leftExtend`, but on the reverse interval synchronized into the forward one.

The function `enumerateLeft` is defined by taking the interval pair and enumerating the distinct values on the forward index. Similarly, `enumerateRight` as the enumeration from the reverse index.

Additionally, we define functions `isLeftMaximal` and `isRightMaximal` as extension to `enumerateLeft` and `enumerateRight` respectively, returning a boolean value 'true', if and only if the enumeration contains two or more distinct symbols.

1.2.6 K-mers and Minimizers

Within the field of bioinformatics, k-mers are continuous substrings of length k extracted from an input text T, see Definition 14. Construction of a k-mer array can be accomplished by a simple linear scan over the input text. Taking each continuous substring of exactly k symbols that can be extracted from the text.

Definition 14 (k-mers) *An array of k-mers K_i from a text $T[0..n-1]$ is an array containing all k-long continuous substrings of T.*

$$K_i = T[i..i+k-1], \quad 0 \leq i \leq n-1, \ \textit{and} \quad 0 < k \leq n-1.$$

Clearly, given any two sufficiently long, matching substrings, they must produce the same k-mers. This observation allows the facilitation of k-mers to determine both the exact and maximal exact matches between two strings.

Minimizers, defined in Definition 15 are a subset of k-mers, with the function of reducing the amount of k-mers required to be stored in memory [11]. The choice of minimizers is arbitrary, but the scheme dictating how the choices are made has to remain consistent, clear, and unchanging.

Definition 15 (Minimizers) *Minimizer array M is a subset of the k-mer array $K[0..l-1]$ where $K[i] = T[i..i+k-1], i \leq l$. Using a window of size w and minimizer scheme $minScheme$.*

$$M[i] = minScheme(K[i..i+w-1])$$

For each window, a single minimizer is selected between the k-mers $K[i..i+w-1]$ using the rules defined in $minScheme$.

Using minimizers, minimizers with identical value $K[i] = K[j]$ are both contained in M, but duplicated minimizer choice of index $K[i]$ can be omitted.

A simple minimizer scheme is one that for each window ($K[i..i+w-1]$ in Definition 15), chooses the first lexicographically smallest k-mer as the minimizer. In the rest of this thesis, we assume the minimizer scheme to be lexicographic.

Computing minimizers from any collection of k-mers using a window of size $w \leq k$ ensures that the minimizers cover all symbols in respect to the input text [11]. Storing duplicates of each minimizer can be omitted in case the same minimizer from same starting index is selected in two different windows as demonstrated in Table 2 between the second and third minimizer, where the last, third, minimizer would be selected multiple times due to being lexicographically greater than both "bca" and "cab" that overlap it.

Table 2: Minimizers from string "aabcabcd", using lexicographic minimizer scheme, and k-mer and window sizes of $k = w = 3$. First window comparing k-mers $\{"aab","abc",and"bca"\}$, second k-mers $\{"abc","bca",and"cab"\}$, and third $\{"bca","cab","abc"\}$.

String	a	a	b	c	a	b	c	.	.	.
k-mers	**a**	**b**	**b**							
		a	**b**	**c**						
			b	c	a					
				c	a	b				
					a	**b**	**c**			

2 Finding Maximal Exact Matches

We present three different approaches for finding maximal exact matches, each with their pros and cons. All presented approaches give their output in the same format which will later be used in the chaining algorithm identically.

We begin by discussing the implementation using two Bi-Directional Burrows-Wheeler ($BDBWT$) indexes to produce all MEM matches longer than a specified threshold in a memory efficient way. The algorithms implemented are based on the Algorithms 11.3, and 11.4 described in Genome-Scale Algorithm Design [5].

The first approach finds intervals on $BDBWT$ indexes which are then translated into intervals relative to the original text by implementing a batched locating algorithm (Algorithm 9.2) from the same book [5].

The second approach utilizes minimizers. The approach implemented is relatively naive but highly efficient using a lexicographic minimizer scheme. The minimizers are matched together to form seeds for possible exact matches of length k, and further extended into MEMs by 'growing' the matches into maximal ones.

The third and final approach for finding maximal exact matches is a hybrid between both the $BDBWT$ and the minimizer implementation. The Hybrid solution uses the minimizer computation to find good seeds and instead of growing the seeds into maximal exact matches using the method described in the minimizer approach, converts them into $BDBWT$ intervals and finalizes the matches analogously to the $BDBWT$ method.

The results of the hybrid implementation bear similarity to the minimizer-based approach. While significantly faster than the $BDBWT$ method alone, the implementation suffers from the added complexity to extend the minimizer seeds into MEMs through the necessary additional data-structures.

2.1 Using BD-BWT

A high-level abstraction as an introduction to the algorithm can be mirrored to right-to-left matching between two strings. For example, assume a pair of string X = "$Wheeler$", and Y = "$Euler$" with all character belonging to an alphabet {"e", "h", "l", "r", "u", "w"}, omitting the letter case without loss of generality for the sake of conciseness.

We start searching with an arbitrarily chosen character "r", verifying that there exists a mismatch between the strings to the right of at least one occurrence of the character (considering the end of the string to be a mismatch). With only one occurrence of the character in both strings, the number of possible extensions limited to a single matching character, "e", on both strings. Similarly, both texts contain only one occurrence of "er", preceded by letter "l". The match "ler", however, cannot be extended further left without introducing a mismatch. By having no more possible extensions to either direction. We note that "ler" is a MEM.

The search continues repeating the same steps for each symbol of the alphabet. In the case of multiple occurrences, extensions yielding a match and a mismatch can both be possible. In such a case, we store the extension causing a mismatch as a MEM and continue searching for a longer match with the matching extension.

2.1.1 Implementation

In this chapter we consider two Bi-Directional Burrows-Wheeler Transform indexes $idxS$ and $idxT$ for texts T and S respectively and recall that a $BDBWT$ index is a combination of two BWT indexes.

To explain the working principle of the algorithm, we first outline the methods and data structures used. The most important concept that we have not defined yet is what we refer to as MEM candidates, formally Definition 16. A MEM candidate C is a representation of a match between T and S, such that each occurrence of a maximal exact match $P = C$ can be derived from the candidate, see Figure 2. More precisely a MEM candidate is a match between T and S such that, for the substring making up the match, at least a single MEM exists.

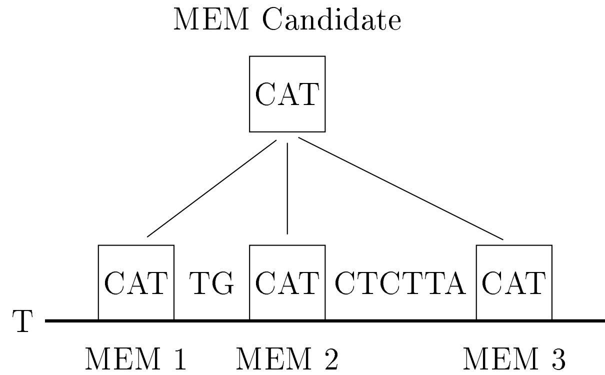

Figure 2: MEM Candidates.

Definition 16 (MEM Candidate) *A MEM candidate is a tuple*

$$\{([i_0..j_0][i'_0, j'_0]), ([i_1..j_1][i'_1..j'_1]), d\}$$

of interval pairs corresponding to matches $T' = S'$ in $BDBWT$ indexes $idxT$ and $idxS$ and length d of the match respectively, where at least one of the intervals is both left- and right-maximal on their respective index.

$$T' = T[SA_T[x]..SA_T[x] + d], i_0 \leq x \leq j_0$$

$$S' = S[SA_S[y]..SA_S[y] + d], i_1 \leq y \leq j_1$$

An important distinction about the candidates and maximal exact matches is that not all occurrences of a candidate represent a maximal exact match, but each maximal exact match is represented by exactly one candidate.

To define a MEM candidate as maximal, we say that a candidate is right-maximal if, and only if it has at least one right-maximal occurrence (the interval on the reverse index contains at least two distinct characters or the special character \$). Lastly, we note that for any string that is right-maximal, it holds that the string is right maximal on all its suffixes, formally as Lemma 2.

Lemma 2 *All suffixes of a right-maximal string are right-maximal.*

Proof Given a right-maximal match P in $T[0..n-1]$, an interval $([i..j])$ in BWT^R corresponding to P^R as the longest common prefix contains at least 2 distinct symbols. Claiming that there exists a prefix P_x of P^R that is not right-maximal, yields a contradiction as either P_x spans the interval $([i..j])$ corresponding to P being right-maximal, or it is not a prefix to P. Lastly, because each prefix of P^R matches the reverse of the suffix of same length from P exactly: each suffix of a right-maximal string is right-maximal □.

To translate the candidates into $BDBWT$ context, we need to consider how matches are represented in BWT indexes. Recalling the Observation 2, we can refer to an interval pair $([i..j][i'..j'])$ that corresponds to the patterns P, and P^R as the longest common prefixes for the suffixes $T[SA_T[i..j]$, and $T^R[SA^R_T[i'..j']]$ respectively.

It is necessary to store the length of the pattern as d, not only allowing us to limit the search to MEMs higher than specific threshold, but to translate the matches back into intervals on the original text. The tuple $\{([i_0..j_0][i'_0..j'_0]), ([i_1..j_1][i'_1..j'_1]), d\}$ contains all necessary information about a maximal exact match.

Lemma 3 proves that all intervals considered by the algorithm are right-maximal.

Lemma 3 *Using Bi-Directional Burrows-Wheeler transform to compute the maximal exact matches in Algorithm 4, all searched intervals are right-maximal.*

Proof The $BDBWT$ search using Algorithm 3 utilizes left-extension to get all candidate intervals. By Definition 11, a left-extension of a pattern P will always correspond to a pattern $c.P$ where c is any valid extension-symbol of P. By Lemma 2 it is sufficient that the right-most (shortest) suffix of each interval selected is right-maximal, which can be ensured by constraining the initial intervals to right-maximal ones.

We define Algorithm 2 `enumDiffLeft` and respectively `enumDiffRight` as helper functions returning a boolean value *true* if and only if the enumerated intervals for the indexes $idxS$ and $idxT$ contain at least one different symbol. Usage of the function improves the running time by allowing us to return as soon as the first mismatch is detected. Definition for `enumDiffRight` is omitted, as it is implemented by replacing calls to forward index with reverse index.

Algorithm 2: enumDiffLeft

Input : $BDBWT$ indexes $idxS$ and $idxT$, and interval pairs $ip0$, and $ip1$ respectively.

Output: Boolean $true$, if intervals enumerate exactly same symbols, $false$ otherwise

```
1  ([a, b]) = ip0, ([c, d]) = ip1
2  if (b − a) = 0 and idxS.forward[a] = $ then return True
3  if (d − c) = 0 and idxT.forward[c] = $ then return True
4  Initialize S as a set
5  foreach c = idxS.forward.at(i), i ∈ ip0.forward do S.insert(c)
6  for c = idxT.forward.at(j), j ∈ ip1.forward do
7      if c ∉ S then return True
8  return False
```

Algorithm 3: BDBWT-MEM. The algorithm is heavily based on the Algorithm 11.3 in Genome-Scale Algorithm Design.

Input : $BDBWT$ Indexes $idxS$, and $IdxT$ with length of n a and m respectively.

Output: An array of tuples of (i, j, d) corresponding to MEMs in $BDBWT$ indexes, where i and j are indexes in $IdxT$ and $IdxS$ respectively, and d the length of the match.

```
1  Initialize stack S with interval pairs ([0, n − 1][0, n − 1], [0, m − 1][0, m − 1])
2  while S is not empty do
3      (ip0, ip1, d) = S.pop
4      ([i_0..j_0][i'_0..j'_0], [i_1..j_1][i'_1..j'_1]) = (ip0, ip1)
5      if (j_0 − i_0 + 1 < 1 or j_1 − i_1 + 1 < 1) then continue
6      if enumDiffLeft(idxS, idxT, ip0, ip1) then sub.push(tuple(ip0, ip1, d))
7      I = ∅
8      foreach c ∈ Σ do
9          i1 = idxS.leftExtend(ip0, c)
10         i2 = idxT.leftExtend(ip1, c)
11         I.push(i1, i2)
12     foreach i ∈ I do
13         (j1, j2) = i
14         if idxS.isRightMaximal(j1) or idxT.isRightMaximal(j2) or
             enumDiffRight(idxS, idxT, ip0, ip1) then
15             S.push(tuple(j1, j2, d + 1))
16 Initialize R as an array containing arrays of tuples
17 Initialize Ret as array containing tuples
18 for m ∈ sub do R[threadNumber].append(Subroutine(IdxT, idxS, m))
19 for c ∈ R do
20     for d ∈ c do Ret.append(d)
21 return Ret
```

We define that an empty string corresponding to the full interval is right-maximal. And looking at the Algorithm 3, we can observe that each interval is subjected to either being right maximal or enumerating into single, different symbols (filling the condition for corresponding to at least one right-maximal match).

If the interval on either index is right-maximal, then there exist at least two different, valid extensions, and at least one right-maximal match exists. Similarly, if only one extension is possible on each index, but the extensions differ (or equal to the special end character $), then the match is still right-maximal under our definition. If an interval does not fill any of the conditions for right-maximality, it is not iterated on further □.

Figure 3 illustrates possible extensions for a "MEM", where left and right extensions are shown within vertical rectangles respectively.

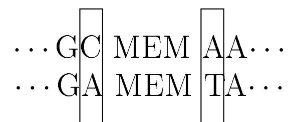

Figure 3: Considering the following illustration where the rectangle on the right side represents the forward interval, and left rectangle the reverse interval. Each interval can is considered maximal as they contain two unique symbols.

Reasoning for the right-maximal intervals resulting in a MEM (after extension to left-maximality) comes from the observation that if a candidate points to multiple occurrences of the match, having multiple unique right-extension possible, then a right-side condition of MEM can be fulfilled, Lemma 4 proves that the amount of such intervals is linear. The same arguments can be made symmetrically for left-maximality.

Lemma 4 *The number of right-maximal matches is at most linear.*

Proof Suffix trees are a similar data structure as suffix arrays, with the main difference being how the suffixes are stored. Suffixes from the tree are constructed by following a path from the root to a leaf node. The number of right-maximal suffixes is rigorously bounded from above as $2n - 2$, using suffix trees as well as with suffix links where n is the length of the input string [5].

The cited proof holds for our lemma by reduction, thus proving that for any strings S, there exists at the most linear number of right-maximal matches. [5, 12] □.

We begin the execution of the algorithm with an interval pair tuple encompassing the entirety of each $BDBWT$ index with the length d set to zero $\{([0..n-1]), ([0..n-1]), ([0..m-1]), ([0..m-1]), 0)\}$, corresponding to an empty string. On the first iteration the algorithm resolves into each right-maximal single-character suffix of the index, giving a starting ground for the iterative search for each MEM candidate.

From the Algorithm 3, we can deduce that for each interval, we compute and perform all possible extensions that can be done symmetrically to the left between the indexes $idxT$ and $idxS$. That is, for each interval corresponding to a pattern P in texts T and S with all possible extensions c, we find the intervals corresponding to the patterns $c.P$ for each c. If the extension is possible on both indexes, then the pattern must be a substring to both texts. As each maximal exact match is a case where no further extensions can be made, each interval must be unique, as proven in Lemma 5, and each candidate must be found, Lemma 6.

Lemma 5 *Each interval considered is unique.*

Proof Starting from the empty string of length $d = 0$, the algorithm extends the intervals on both indexes symmetrically to the left to yield all, at most σ new intervals corresponding to strings of length $d = 1$. As each interval is extended at most once by each symbol in the alphabet, each interval must be unique. The same argument can be followed for extensions from every subsequent interval through induction, as each interval is extended at most once by each symbol in the alphabet, and only left extensions are used □.

Lemma 6 *Algorithm 3 finds all MEM candidates longer than a specified threshold.*

Proof Assume that there exists a MEM candidate C_f that is not returned by the above algorithm. To be a candidate, C_f must have at least one occurrence that is both left- and right-maximal match. For right-maximality, the statement contradicts with the observation of the algorithm. The algorithm extends with each symbol in the alphabet, starting from the empty string it results in intervals corresponding to single symbols, resulting in first right-maximal intervals from which each subsequent interval is derived from. Thus, a suffix of C_f, C_f itself is found or C_f does not exist as a MEM candidate.

With left-maximality we observe a contradiction that for each right-maximal suffix found, the algorithm attempts to perform symmetrical left-extensions between the indexes until all valid extensions are exhausted, storing a candidate tuple at each left-maximal interval. Thus, either C_f is not left-maximal, or it does not exist. □.

With knowledge of all the MEM candidates we can return occurrences of the maximal exact matches. Each MEM candidate C corresponds to a $BDBWT$ interval pair that is both left- and right-maximal. To determine all occurrences of the candidates in the original text, we first need to identify them by each unique left-and right extension pair. We extend each candidate interval to the left with all possible symbols a to get all matches $a.C$, then enumerate all possible extensions to the right of the extended interval corresponding to $a.C.b$, obtaining extension pairs (a, b). Performing the steps for both indexes separately.

Recalling the Definition 4 of a maximal exact match, and given the extension pairs (a, b), and (c, d), we note that for two occurrences to result in a MEM it must hold

that $a \neq c$ and $b \neq d$. Thus, the valid set of occurrences corresponding to $MEMs$ becomes $\{(a,b,c,d) : (a \neq c), (b \neq d)\}$. See Algorithm 4 for details on the algorithm.

Algorithm 4: BDBWT-MEM-OCC. The algorithm is heavily based on the Algorithm 11.4 of Genome-Scale Algorithm Design

Input : $BDBWT$ indexes $idxS$, and $idxT$
Input : Tuple $\{([i_0..j_0][i'_0..j'_0], [i_1..j_1][i'_1..j'_1]), d\}$ as {ip0,ip1,depth}
Output: Tuples (i,j,d), where i,j correspond to match locations within $BDBWT$ indexes and d to the length of the MEM.

/* We refer to the int ervals with $([a,b][c,d])$, and to depth with 'depth' */
Initialize a and b as empty arrays containing a pair of characters
Initialize ret as vector storing tuples {int,int,int}
$(ip0, ip1) = ([i_0..j_0][i'_0..j'_0], [i_1..j_1][i'_1..j'_1])$

foreach $a \in enumleft(idxs, ip0)$ **do**
  $lex = idxS.leftExtend(ip0, a)$
  foreach $b \in enumerateRight(idxs, lex)$ **do** $a.push(a, b)$
foreach $cinenumerateLeft(idxt, ip1)$ **do**
  $lex = idxt.leftExtend(ip1, c)$
  foreach $b \in enumerateRight(idxs, lex)$ **do** $b.push(c, d)$
foreach $(a,b,c,d) \in \{a \bigotimes b\}$ **do**
  $r1 = idxS.leftExtend(ip0, a)$; $r2 = idxS.rightExtend(r1, b)$
  $r3 = idxT.leftExtend(ip1, c)$; $r4 = idxT.rightExtend(r3, d)$
  $([i_0..j_0][i'_0..j'_0], [i_1..j_1][i'_1..j'_1]) = (r2, r4)$
  foreach $i \in [i_0..j_0]$ **do**
    foreach $j \in [i_1..j_1]$ **do**
      $ret.push(tuple(i, j, depth))$
return ret

In the Algorithm 4, we first get valid extension pairs for each occurrence by extending left (with symbol a_i) on each interval and then from each extended interval, enumerating all symbols b_j to the right, resulting in all possible extension pairs (a_i, b_j). Computing the extensions (c_j, d_j) similarly on the other text. With all pairs $A = (a_i, b_j)\forall i, j$ and $B = (c_i, d_j)\forall i, j$, we are able to output all combinations of the occurrence-endpoints in time linear to the number of pairs $\mathcal{O}(|A| + |B| + |A| \otimes |B|)$ as defined in Lemma 11.4 of Genome-Scale Algorithm Design. The algorithm works by iteratively selecting each pair from A, and B, and constructing all compatible pairs $A \otimes B = \{(a,b,c,d) \mid \quad (a,b) \in A, (c,d) \in B, (a \neq c), (b \neq d)$.

Lastly, we extend the intervals given originally as an input to the subroutine first to the left and then to the right on both indexes using the compatible pairs $A \otimes B$. Each extended interval then corresponds to patterns $a.MEM.b$ on the first texts, and $c.MEM.d$ on the second text, where $a \neq c$ and $b \neq d$.

The tuples the subroutine produces each denote a location of a single MEM on the forward interval of both $BDBWT$ indexes and the length of the match. The tuples, however, still need to be translated back into positions corresponding to the

original texts T and S before they can be used for the chaining algorithm. Lemma 7 describes efficient way to accomplish the translation.

Lemma 7 *MEM tuples can be translated back into text positions in* $\mathcal{O}(n \log \sigma + occ)$ *time per BDBWT index for a total time of* $\mathcal{O}((n+m)\log \sigma + occ)$.

Proof We implement a batch locating Algorithm 9.2 from Genome-Scale Algorithm Design [5] with time complexity of $\mathcal{O}(n \log \sigma + occ)$, and call it separately for both indexes. The batch algorithm translates location occurrences from a single interval index BWT_T into the corresponding locations in the text T. The algorithm requires a bitvector "*marked*" that contains all integer positions that are to be translated as an input, as well as an array with respective LF-mapping for the respective text.

The algorithm considers each location set to the bitvector and utilizes LF-mapping to translate each position from the BWT index to the corresponding position on the text T.

Recall that the subroutine finds matches of form $x.MEM.y$, where the actual MEM is surrounded by the unique mismatching symbol used to determine the text position of the match to the left and right. With the batch locating done, and the matches $x.MEM.y$ translated to the text indexes.

The substrings can be trimmed into MEMs by remembering that the output tuple contains the length of the match d, and that there exists exactly one *extra* symbol on both sides. Thus, for each tuple (i, j, d) corresponding to substrings $T[i..i+d-1]$ and $S[j..j+d-1]$ as an output of the batch, we shift the indexes i and j to the right by one and get the proper MEM tuples $(i+1, j+1, d)$. Shifting the indexes takes at most $\mathcal{O}(occ)$, proving the time complexity □.

2.1.2 Analysis on the implementation

Lemma 8 *Using BDBWT, we are able to return maximal exact matches between strings* $T[0..n-1]$ *and* $S[0..m-1]$ *in time* $\mathcal{O}((m+n)\log \sigma + occ)$*, where occ is the count of occurrences.*

Proof The insertion to a set is known to have the worst case $\mathcal{O}(\log n)$ to the size of the container [13]. Furthermore, the set data-structure is restricted to unique keys and the internal ordering is retained upon insertion. The complexity of appending a value to a vector instead is defined as having an amortized constant time complexity.

The time complexity of `leftExtend`, and `rightExtend` are dominated by the size of the alphabet Σ and bounded to $\mathcal{O}(\log \sigma)$, and the synchronization bound is given in Lemma 1 to be $\mathcal{O}(\sigma)$. Thus, a synchronized extension can be computed in $\mathcal{O}(\log \sigma + \sigma) = \mathcal{O}(\sigma)$ time.

To enumerate all distinct values from an interval, we need to iterate over the whole interval. Let I be an interval of length N. And let Σ be alphabet of size σ containing

all possible symbols that can be contained within the interval. For each index we attempt to insert the symbol-on-point to a set. The set enforces uniqueness and order during insert and takes, in the worst case $\mathcal{O}(\log \sigma)$ time. Thus, the enumeration will at the worst case take $\mathcal{O}(N \log \sigma)$ time with a conversion into a vector in time linear to its size, at most $\mathcal{O}(\sigma)$, where σ in case of for example, DNA, is small and dominated by the length of the interval. The conversion to vector from a set saves a significant amount of time later with efficient random access and can be charged into the enumeration itself.

Similarly, the running time for answering the left- and right-maximality for an interval is bounded by a logarithmic time complexity in respect to the length of the interval in the underlying data structure using wavelet trees by a single extension taking $\mathcal{O}(\log \sigma)$, thus N extensions taking $\mathcal{O}(N \log \sigma)$ time.

As the extending functions are fast, we can save time by not enumerating all possible extension symbols in time linear to N by instead attempting to extend with each character in the alphabet Σ directly. This gives us the time complexity of $\mathcal{O}(\sigma \log \sigma)$, with synchronization charged to the extensions, yielding all extensions to one direction. Instead of first enumerating and then extending for all symbols in the interval in $\mathcal{O}(N \log \sigma)$ time with $N \log \sigma$ dominating $\sigma \log \sigma$.

We observe that with each extension from an interval the resulting interval will always be smaller or at most equal in size. And the worst case would result from having to observe each interval in both of the indexes, bounding the upper limit of the intervals computed by the amount of the input indexes $n + m$.

The time complexity of the main algorithm becomes $\mathcal{O}((n+m) \log \sigma + subroutine)$, as also stated in Genome-Scale Algorithm Design [5]. The Lemma 4 further supports the time complexity by proving that the number of right-maximal intervals must be linear to the size of the index.

For the subroutine we can observe that each interval is extended to the left twice, and once to the right. And each interval and all their possible left-extensions are enumerated on both left and right side. The reverse interval is computed symmetrically to the left one, and thus analysis of only the forward is sufficient for details.

The subroutine takes as an input a single interval of length n, each having at most σ unique extensions to the left can thus be bounded by amortized bound of $\mathcal{O}(n + \sigma \log \sigma)$, where n generally is smaller than σ The intervals pushed into the subroutine are expectedly small, and directly attempting to extend with each character is not efficient. Furthermore, enumerations to the right side cannot exceed the enumerations to the left and are as such bounded by the interval length.

Computing the Cartesian product also proved to be bounded by the possible extensions, and thus is bounded by the size of the alphabet as proven in Lemma 11.4 of Genome-Scale Algorithm Design [5]. This finalizes the result that the subroutine as a whole is bounded from above by the interval size.

This results in the subroutine being bounded by the size of the alphabet and its running time is thus dominated by the number of calls, linear to the number of

MEM candidates.

The final time complexity becomes $\mathcal{O}((n+m)\log\sigma + occ)$, where occ represents the amount of occurrences of maximal exact matches, proving Lemma 8 $\square$.

2.1.3 Optimization - Parallelization

Running time for the MEM search utilizing Bi-Directional Burrows-Wheeler Transform can be sped up by a parallelization scheme. Parallelization of the algorithm can be achieved in multiple ways, such as by splitting the input between the possible first extensions for the index. In the case of DNA alphabet, this would mean splitting the input between the symbols $\{A, C, G, T\}$, splitting the size of the problem into four.

We make use of OpenMP [14], to implement the backbone of the parallelization described in Algorithm 5. Prior to the actual MEM search, we construct a vector with an index reserved for each thread that could be running, and during the searching, insert the returned result into the index denoted by the thread number unique to it. Avoiding a data-race conditions where the threads could try to insert values into the array at the same time, causing the program to fail through exception.

Algorithm 5: BWT Parallelization

Initialize array Arr[number of threads]
Initialize ret as array
`// Parallelized loop.`
1 **foreach** $c \in \Sigma$ **do**
2 | $arr[threadNum] = bwt - mem(index1, index2, c)$
3 **foreach** $tc \in arr$ **do**
4 | **foreach** $i \in tc$ **do**
5 | | $ret.append(i)$

To extend the parallelization scheme for a thread count greater than σ, we would need to determine starting intervals for each interval instead of only focusing on the starting character. To parallelize between two-to-n-character-suffixes we would first have to perform ℓ extensions, each giving up to $\ell \cdot \sigma$ total intervals. First extension giving up to σ intervals, with σ possible extensions each.

It can be observed that, by creating too many starting seeds, intervals for parallelization, the parallelization itself begins to act like the actual algorithm, causing the efficiency to become questionable. We further explore the possibility of parallelizing the $BDBWT$ search in the hybrid solution.

2.2 Using Minimizers

Compared to the $BDBWT$, using minimizers to find MEMs between T and S is a relatively simple approach to implement. The approach first computes k-mers

and their minimizers for each text separately, and then efficiently matching the minimizers to yield a good set of exact matches (recall Observation 4 on lexicographic minimizer scheme limitation). The matches found are not yet maximal, so they lastly extended into ones. The largest caveat of the minimizer approach is the slightly higher memory usage compared to using $BDBWT$ due to the requirement to store k-mers and minimizers as strings on top of their numerical positions.

The intuition for extending MEMs from each minimizer match can be thought back to the Observation 1 by which, each exact match is a substring (or equal to) a maximal exact match.

2.2.1 Implementation

The k-mers and minimizers are computed using a linear time scan over the text and storing each substring (k-mer) $K_i = T[i..i+k-1]$. For every group of $k = w$ k-mers $K[i..i+w-1]$, choosing the lexicograhically smallest, leftmost k-mer as the minimizer. This process is implemented in the Algorithm 6 as follows:

Algorithm 6: Minimizers

Input : Text $T[0..n-1]$, integer k, and integer w denoting k-mer and window sizes respectively.

Output: Set of minimizers tuples $(T[j..j+k-1], j)$ sorted lexicographically

1 Initialize ret as vector containing pairs $(string, integer)$
2 Initialize kmers as vector containing pairs $(string, integer)$
3 **for** $i \in [0..n+1-k]$ **do**
4     $kmers.append((T[i..i+k-1], i))$ **if** $i \geq w$ **then**
5         $l = |kmers|$
6         $minimizers.append(smallest(kmers[l-w..l-1]))$
7 **return** *Sort(minimizers)*

After finding the minimizers from both texts, each minimizer from the first text is matched with each possible matching minimizer from the second text, covering all valid combinations. For an efficient implementation, consider Lemma 9 below.

Lemma 9 *Given arrays A and B of minimizer occurrence indexes. Cartesian product containing all valid occurrence pairs $A[i] = B[j]$ can be computed in $\mathcal{O}(N \log N)$ time, N is the size of the output, and the number of matches between the minimizer occurrences.*

proof Let arrays A and B contain all minimizers from two given input texts respectively. Initialize a map "*minimap*" with each minimizer as key. Iterating over both arrays A and B, for each pair $(a, i) \in A$ and $(b, j) \in B$ insert the values i and j into "*minimap*" with key a and b respectively. Creating a map of arrays listing all the indexes where each minimizer (key) starts in both texts. Consider the first part of Algorithm 7 for details.

Algorithm 7: Minimizer Tuples

Input : Minimizer arrays A, and B, where for $(a, x) \in A$ and $(b, y) \in B$, a, b corresponds to the minimizer and x, y to its index.
Output: Arrays $RetA$, $RetB$ where $\{(a, i) \in A : a \in A \cap B\}$,$\{(b, j) \in B : b \in A \cap B\}$

```
1  Initialize minimap as a map ["Minimizer"] → (occ_A, occ_B)
2  smaller = (Min(|A|, |B|))
3  for i ∈ [0..smaller] do
4  |   (a, x) = A[i]
5  |   minimap[a].appendA(x)
6  |   (b, y) = B[i]
7  |   minimap[b].appendB(y)
8  if |A| > smaller then
9  |   for i ∈ [smaller..|A|] do
10 |   |   (a, x) = A[i]
11 |   |   minimap[a].appendA(x)
12 else if |B| > smaller then
13 |   for i ∈ [smaller..|B|] do
14 |   |   (b, y) = B[i]
15 |   |   minimap[b].appendB(y)
16 for c ∈ minimap do
17 |   (occ_A, occ_B) = minimap[c]
18 |   for i_1 ∈ occ_A do
19 |   |   for i_2 ∈ occ_B do
20 |   |   |   retTuple.insert((i_i, i_2, |c|))
21 return retTuple
```

In the above Algorithm 7, on lines 3 to 16, we cluster all occurrences of each minimizer and on the last part (lines 16 to 20) we get every possible combination of them and append tuples $(i_1, i_2, |c|)$, where i_1 and i_2 are occurrence locations and $|c|$ the length of the minimizer in question, to the output.

To find all valid occurrence pairs, we have to consider a linear time scan over all minimizers returned from Algorithm 6 and store their occurrences on both strings into an array with the string value of the minimizer as a key. If n is the total number of all minimizers, then the time to taken to iterate through all minimizers and append them to an array becomes $\mathcal{O}(n)$.

With n total minimizers each having a occurrences on the first text and b on the second, the total number of possible matches becomes $a \times b$. Assume without loss of generality that $a \geq b$, then the number of matches can be bounded from above as a^2. Further, the number of occurrences of any minimizer cannot exceed the total number of minimizers for a given text, giving us the bound of $\mathcal{O}(n^2)$ for amount possible matches.

Insertion to a set is bounded logarithmically by its size, in this case the total number

of matches $N = n^2$. With N insertions, the time complexity of the algorithm becomes $\mathcal{O}(N \log N)$ where N is the number of matches dominating over the time required to map all minimizers in the first part of the algorithm $\square$.

To extend the exact matches to MEMs, we recall the Observation 1 and choose to naively extend the matches to both directions one by one until mismatches are found on both sides with Algorithm 8. For each minimizer, we first attempt to extend left until a mismatch $T[i-x..i+d-1] \neq S[j-x..j+d-1]$ is introduced, and afterwards performing the same steps to the right side. The resulting substring of the text is guaranteed to be a MEM.

Algorithm 8: Minimizer extension

Input : Texts T and S, as well as a set $M[0..n-1]$ of integer tuples (i, j, d), where i and j correspond to a minimizer starting position on texts T and S respectively and d denoting the length of the minimizer.

Output: Set of integer tuples (i, j, d) similar to input, where each tuple corresponds to a MEM.

```
1  Initialize ret as array containing integer tuples
2  foreach m_i ∈ M do
3  |   (i, j, d) = m_i
4  |   Initialize x = y = 0
5  |   while T[i − x] = S[j − x] do
6  |   |   x = x − 1
7  |   |   d = d + 1
8  |   while T[i + y] ≠ S[j + y] do
9  |   |   y = y + 1
10 |   |   d = d + 1
11 |   ret = ret ∪ {(i − x, j − x, d)}
12 return ret
```

The primary drawback of this method is that multiple occurrences of the same maximal exact match can be found from different minimizers, if and only if two minimizers correspond to the same MEM. This could be solved by keeping track of all the locations where a MEM has already been found, and not extending minimizers that overlap an area where an extension has been done before.

The minimizers tuple values are already on correct index (the text index), thus further transformations are not required before the matches could be used for the chaining algorithm.

A downside to the lexicographic minimizer scheme is that it does not hold a guarantee of outputting minimizers for each exact match of length $k = w$. For example, looking to the Table 3 below using the same example as Table 2 earlier. Consider a case where the string "*bca*" would have been an exact match, the match would have been missed as the selected minimizers were {"*aab*", "*abc*", "*abc*"}, see Observation 4

Table 3: K-mer overlap. Visualizing how 2k-1 k-mers overlap any chosen k-mer (a k-mer overlaps itself). If the k-mer "bca" represented an exact match, it would not be chosen under a lexicographic minimizer scheme as lexicographically smaller k-mer "abc" exists within a window of size $w = k = 3$ on both sides.

String	a	a	**b**	**c**	**a**	b	c	.	.	.
k-mers	a	b	**b**							
		a	**b**	**c**						
			b	**c**	**a**					
				c	**a**	b				
					a	b	c			

Observation 4 *Lexicographic minimizer scheme can only guarantee finding exact matches longer than* $2k - 1$.

Consider that there exists an exact match that is composed solely of characters corresponding to the character with the greatest lexicographic value in the alphabet. Then, for this match to result from a minimizer computation using lexicographic scheme, it must hold that for all windows surrounding the match, no character within it can have value smaller than within the match.

For an exact match of length k, it holds that the match can be overlapped by at most $2k - 1$ k-mers, such that the k-mer covers at least one symbol of the match. Of these $2k - 1$ k-mers, it can hold that only one corresponds to an exact match, independent of the selected k-mer being part of the exact match, illustrated in Table 3.

Thus, unless all $2k-1$ k-mers are part of the same exact match, there is a chance that a minimizer does not correspond to the specific match. This limitation, however, is easy to take into an account when initially choosing the k-mer size and the threshold for the size of the maximal exact matches.

2.2.2 Analysis

Theorem 1 (MEMs can be found efficiently with minimizers) *Using minimizers, we can output all MEMs between text* $T[0..n-1]$ *and* $S[0..m-1]$ *efficiently in time*

$$\mathcal{O}(K(NM)\log(NM))$$

time, where N, M *are the number of minimizers in* T *and* S *respectively, and* K *the length of the input text.*

proof Yielding the sets of minimizers, we only need to consider a $\mathcal{O}(n)$ linear scan over the input text $T[0..n-1]$ to obtain all k-mers for a single text. To get each minimizer during the same scan, we take the k-mer with the smallest lexicographic

value contained in each window in time $\mathcal{O}(w)$, where w is an integer value denoting the size of the window. Thus, the total time to compute all minimizers for single text becomes $\mathcal{O}(nw)$ where $n > w$.

The number of k-mers N found from the text is bounded by its length n with $N = n - k$, where k is the length of each k-mer. Further the number of minimizers is bounded by the total number of k-mers as well as by the size of the minimizer window w.

Let N be the number minimizers found from text $T[0..n-1]$, and similarly M for minimizers from the text $S[0..m-1]$. To determine each matching minimizer pair, we have to compare each minimizer in N, with the ones in M.

To match each minimizer from text T with respective minimizers from S, we use the procedure described in the Lemma 9 above for clustering the matches and output all occurrence pairs in $\mathcal{O}(MN \log MN)$ time, where M, N are the number of minimizers in text T and S respectively.

Lastly, to extend each minimizer match into a MEM, we need to do at most $K = min(n, m)$ (minus the length of the minimizer) comparisons for each pair in the worst case where both texts match completely and thus each minimizer pair must be extended to match the full text. In the worst case, each minimizer would be extended to cover the text.

The resulting time complexity to yield MEM matches from minimizers becomes $\mathcal{O}\Big(K|N|\log|N|)\Big)$, where N, M are the number of minimizers in T and S, and K the length the input text, proving Theorem 1 $\square$.

2.2.3 Optimization

The minimizer approach could be improved further by, for example, implementing k-mers with dynamic lengths, or by any means that would resolve the minimizer computation into matches more accurate to the desired maximal exact matches.

An intuition, however, says that if it was possible to know the optimal size for each k-mer during computation, then each maximal exact match between any two strings T and S could be found right-away as k-mers. This could result in an almost trivial matching of the resulting k-mers, eliminating the need for minimizers at all in the process. Thus, for a reasonable solution for variable k-mer length, one that is more efficient than the quadratic time-complexity required for naively checking each position of one text to another. An approximate function with a high confidence would appear to be the best contender.

Another possible optimization to using minimizers for finding all exact matches is a rather new concept of universal k-mer sets [15]. Universal k-mer sets would, at least in theory, allow using higher k-mer sizes (higher order of minimizer sizes) without increasing the number of distinct minimizers. After all, finding mutual minimizers-seeds for maximal exact matches between two strings becomes more and more time-consuming the more (distinct) distinct we need to compare.

The greatest improvements would result from being able to compute the mutual minimizers more efficiently and from extending the minimizer matches into proper maximal exact matches.

2.3 Hybrid Solution

The main motivation for the hybrid solution stems from noticing that the bulk of the time taken in both the $BDBWT$ and in the minimizer approaches happen at different stages. The minimizer approach is fast when it comes to determining the actual minimizers, seeds for maximal exact matches, but the bottleneck becomes apparent when those matches must be extended into maximal ones.

Similarly, $BDBWT$ approach is fast at extending a single match into a maximal one, using at most D left- and right-extend operations in $\mathcal{O}(\log \sigma)$ time, where D is the length of the longest maximal exact match that can be found from the match given as an input. Typically, the bottleneck with $BDBWT$ comes into play when a single pattern results in multiple possible extensions, of which each needs to be computed separately. With no way of knowing beforehand whether any of them correspond to a unique maximal exact match.

Thus, the primary question leading to this approach is whether it is possible, and feasible to compute the initial seeds using the minimizer-approach, and then utilize the $BDBWT$ search to finalize the maximal exact matches. Replacing the extension to maximal exact matches of the minimizer computation in prior chapter by a $BDBWT$ computation.

We present an implementation that provides the correct result, but regrettably loses efficiency compared to only using minimizers. After explaining the working principle of the solution, we also discuss possible optimizations to the method.

One of the main key components to making the hybrid solution work, is the notion that we can translate the minimizers into intervals in $BDBWT$. For this purpose, we recall that minimizer indexes are based on the proper indexes relative to their source text (starting at index i of text T), while $BDBWT$ intervals are related to indexes within a suffix array built on the same text.

2.3.1 Necessary additional data structures

Additional data structures are required to translate the minimizers into $BDBWT$ intervals efficiently. The first key data-structure is suffix array, which wasn't directly used in earlier approaches, as well as an inverse suffix array. Inverse suffix array (defined in following Chapter 2.3.3) is required to determine the intervals where each minimizer appears as a prefix to a suffix.

Next, we require a bitvector to store information where each segment of the k-long common prefixes appear in the suffix array. The bitvector can be computed efficiently with the third additional data-structure, LCP-array, which in turn, can be

computed efficiently using the so-called permuted longest-common-prefix ($PLCP$) introduced by Kärkkäinen, Manzini and Puglisi in "Permuted Longest-Common-Prefix Array" [16].

We will briefly go over the details of each additional data-structure and how to compute them efficiently. As a starting point we assume that we already have computed $BDBWT$ indexes $idxS$ and $idxT$ for texts S and T respectively.

2.3.2 Suffix- and inverse suffix arrays

Suffix array as defined in Definition 5, is a representation of all possible suffixes of a string, such that each suffix is stored into the array in a lexicographically increasing order. Recall Observation 2 noting that the forward index BWT_T of bidirectional index $idxT$ has a close relationship with the suffix array SA_T, allowing for an efficient construction.

To facilitate efficient queries for determining where a suffix starting at the ith index of the text is stored in a suffix array, we use a data structure defined as an inverse suffix array, Definition 17.

Definition 17 (Inverse Suffix Array) *Given a suffix array* $SA[i]$, $0 \leq i \leq n-1$, *an inverse suffix* $ISA[j]$ *array contains values such that* $ISA[SA[i]] = i$.

To compute the arrays space efficiently, we compute an LF-mapping [8] for the index we are turning into a suffix array, recalling that an LF-mapping allows us to backtrack the indexes of a BWT in the same order as they appear in the original text. Thus, we can compute both the suffix array and its inverse from the BWT using Algorithm 9 as follows:

Algorithm 9: BWT to SA

```
Input   : BDBWT idxS of length n
1  Initialize SA, ISA, as suffix array and inverse suffix array respectively.
2  LF = LF-mapping built from idxS
3  k = 0
4  currIndex = 0
5  while k < n do
6  |   SA[currIndex] = n − k − 1
7  |   ISA[n − k − 1] = currIndex
8  |   k = k + 1
9  |   currIndex = LF[currIndex]
10 return(SA, ISA)
```

2.3.3 Longest common prefix array

Recall that in Definition 7 we defined the LCP array to denote the longest common prefix between two adjacent suffixes in a suffix array, that is $LCPA[i] = LCP(SA[i], SA[i-1]), i > 0$, and $LCPA[0] = 0$.

Using the efficient algorithm described in Figure 1a of "Permuted Longest-Common-Prefix Array" [16]. We first compute the array ϕ as $\phi[SA[j]] = SA[j-1], \quad 1 \leq j \leq n-1$. Using the ϕ array we can proceed to computing the $PLCP$ array as Algorithm 10:

Algorithm 10: $PLCP$

Input : Array ϕ representing values in a suffix array
Output: Permuted longest common prefix array $PLCP$

1 Initialize $\ell = 0$
2 **for** $i \in [0..n-1]$ **do**
3 | $j = \phi[i]$
4 | **while** $T[i+\ell] = T[j+\ell]$ **do**
5 | | $\ell = \ell + 1$
6 | $PLCP[i] = \ell$
7 | $\ell = \max(\ell, 0)$
8 **return** $PLCP$

The resulting $PLCP$ array contains all the necessary information about the longest common prefixes for the suffix array it is built on. But we still need to translate it to an order that is the same as a regular suffix array. We can do this by considering statement and equation given in the original paper, $PLCP[i] = LCP(i, \phi[i])$ [16].

We get Algorithm 11 to convert $PLCP$ to LCP as follows:

Algorithm 11: $PLCP$ to $LCPA$

Input : Permuted longest common prefix array $PLCP[0..n-1]$, and a suffix array $SA[0..n-1]$
Output: Longest common prefix array $LCP[0..n-1]$

1 Initialize LCP as empty array
2 **foreach** $i \in [0..n-1]$ **do**
3 | $LCP[i] = PLCP[SA[i]]$
4 **return** LCP

2.3.4 Partitions and conversion

The partition bitvector is an important piece of the puzzle as it allows us to determine the location of each minimizer by use of rank- and select-queries in a constant time complexity (Definitions 8 and 9).

The values of the partition vector are dependent on the LCP array. Setting each index i on the partition bitvector to 1 if, and only if the index of the $LCP[i] < k$,

where k is equal to the size of the minimizers computed prior. The first and last index of the partition bitvector are set to 1 regardless. Formally the partition vector $B[0..n-1]$ is:

$$B[i] = \begin{cases} 1, & if\ LCP(i) < k \\ 1, & if\ i = 0, i = n-1 \\ 0, & otherwise \end{cases}$$

Converting the minimizers into a $BDBWT$ array efficiently is crucial step to make the hybrid implementation function. The theoretical implementation to the conversion using the above partitions is given as a proof to Theorem 2:

Theorem 2 (Minimizer to suffix array interval) *Given constant time rank- and select operations, minimizer strings can be translated to suffix array intervals in $\mathcal{O}(occ)$, where occ is the number of minimizers using inverse suffix arrays and partition bitvectors.*

Proof Let $SA[0..n-1]$, $ISA[0..n-1]$ be a suffix- and inverse suffix arrays built on text $T[0..n-1]$, and let M be any minimizer of length k. Given an LCP array built on SA, as well as a bitvector B

$$B[i] = \begin{cases} 1, & iff \quad LCP[i] < k, \quad 0 < i < n-1 \\ 1, & iff \quad i = 0, \text{ or } i = n-1 \\ 0, & otherwise \end{cases}$$

For any minimizer M_i, where i is the starting index of the minimizer (M_i is the ith k-mer of the text T starting at $T[i]$). Recall from the definition of suffix array (Definition 5) that all suffixes are sorted into a lexicographically increasing order, and as such, each set of suffixes prefixed by a fixed k-mer are adjacent in the interval.

From the inverse suffix array we get information where the suffix corresponding to the minimizer appears. That is, since $M_i[0..k-1] = T[i..i+k-1]$, $ISA[i]$ gives us the location in the suffix array that the minimizer corresponds to. $T[SA[ISA[i]] = T[i]$. Thus, with the inverse suffix array we get knowledge of an index where the minimizer appears as a prefix.

The partition bitvector B is constructed to have value $B[i] = 1, \quad 1 < i < n-1$, if and only if $LCP[i] < k$. The result given is that, for all positions $B[i'] = 0$, it must hold that $SA[i']$ and $SA[i'-1]$ must have a common prefix of length at least k. And similarly, if $B[j'] = 1$, then we know that $SA[j']$ and $SA[j'-1]$ do not have a common prefix of at least k symbols.

To get the whole interval, we need to find the lower and upper bound for the interval, such that the interval only contains suffixes with a common prefix of length at least k. Recall that we marked $B[i] = 1$ if and only if $LCP(SA[i], SA[i-1]) < k$. Thus, by using $rank_1(i)$, we return the number of bits assigned to value 1 before the suffix containing the desired prefix.

Further, if $B[i] = 1$ then we know that the suffix in question is already the lower bound, and thus we only need to find the upper bound. Otherwise, if $B[i] = 0$, we can find the lower bound by selecting the last bit set to 1 by $select_1(rank_1(i))$ in $\mathcal{O}(1)$ time. The upper bound can similarly be found by selecting the first following bit set that is set to 1 by $select_1(rank_1(i) + 1) - 1$ (Note we need to subtract one from the index to get the last matching index.)

The interval bounded by the queries thus results in an interval containing all suffixes with the common prefix of M. With only constant time operations used, the overall time complexity is $\mathcal{O}(1)$ for a single element and becomes bounded from above by the number of elements occ. □

The translation described in the proof, however, isn't sufficient alone for producing interval pair for $BDBWT$. We are still required to synchronize the interval to the reverse interval as well. However, this isn't computationally difficult to do, and can be performed without affecting the overall time complexity.

Given we know the length of the original text T, and the length of the minimizer M_i. Finding the starting position of the minimizer M_i^R in text T^R can be done by subtracting the index i from the length of the text, and offsetting by the length of the minimizer. Given the information on the reverse minimizer, the reverse interval is otherwise computed analogously to the forward interval. Giving us the second component for $BDBWT$ interval pair.

2.3.5 Implementation

We begin the hybrid implementation by computing minimizers the same way as in the minimizer computation defined in Chapter 2.2. Contrary to the minimizer approach, however, we also compute inverse suffix arrays for each text and their reverse at this stage. The order in which the arrays need to be computed is arbitrary and have no crossing dependencies, allowing us to parallelize the computation.

Further, we don't need to find every match between minimizers between two strings. For example, in the approach using only minimizers, we would have had to find all occurrences of a minimizer from the other text. In the hybrids' case, however, since we are going to be translating each minimizer into suffix array interval, we only need to know one occurrence of each minimizer. This greatly reduces the number of minimizers in the case where the k-mer size is smaller, or the text is primarily composed of small number of repeating minimizers.

Like the approach using only minimizers, we create an intersection between the two sets of minimizers A and B, and then take the first of each unique element from A and B that are contained in the intersection.

For each minimizer M_i, we need to as well compute the index where the matching reverse minimizer M_j^R appears on the reverse text. With the starting indexes i and j, we can get an interval location within the suffix array that points to a suffix that has the prefix of M_i on the forward index and M_j^R on the reverse index.

Each minimizer is then translated to corresponding intervals in SA using the procedure described above as a proof for the Theorem 2.

The translated intervals are then combined into a tuple data structure containing interval pairs for both indexes and the minimizer length for each tuple, identical to the tuples used with the $BDBWT$ approach defined in Chapter 2.1.

This gives us Algorithm 12 for the workflow of the hybrid approach. For clarity the pseudocode contains *section* and *end* markers to denote the sections which can be executed in parallel to each other.

Algorithm 12: Hybrid approach

Input : Strings S, T, k-mer size k, window size w, $BDBWT$ indexes $idxS$, $idxT$

Output: Array *mems* containing maximal exact matches between the input indexes.

```
   // section
1  M1 = minimizers(S, k, w)
2  M2 = minimizers(T, k, w)
3  S1 = ISA(S)
4  S2 = ISA(S^R)
5  S3 = ISA(T)
6  S4 = ISA(T^R)
   // end
7  (M3, M4) = minimizerTuples(M1, M2)
   // section
8  lcp1 = LCPUsingPLCP(idxS, S, SA1, forward)
9  lcp2 = LCPUsingPLCP(idxS, S, SA2, backward)
10 b1 = partition(k, idxS, lcp1)
11 b2 = partition(k, idxS, lcp2)
12 set1 = MinimizerToBwtIntervals(b1, b2, M3, S1, S2)
   // end
   // section
13 lcp3 = LCPUsingPLCP(idxT, T, SA3, forward)
14 lcp4 = LCPUsingPLCP(idxT, T, SA4, backward)
15 b3 = partition(k, idxT, lcp3)
16 b4 = partition(k, idxT, lcp4)
17 set2 = MinimizerToBwtIntervals(b3, b4, M3, S3, S4)
   // end
18 foreach i ∈ [0..|set1| − 1] do
19 |  seeds = seeds ∪ (set1[i], set2[i], |set1[i]|)
20 Initialize mems as an array
21 foreach j ∈ seeds do
22 |  mems.append(BWT-MEM-Hyb(idxS, idxT, j))
23 return mems
```

Furthermore, to finish the MEM searching with the hybrid implementation, we

need to adjust the $BWT - MEM$ algorithm to consider all possible extensions to the left as well, since minimizers hold no guarantee of being left- or right maximal match. And the implementation using only $BDBWT$ starts with an empty string, thus finding all MEMs from right to left, without the need to extend to the right side.

We accomplish this by adding a segment to the algorithm that steps similar to the original Algorithm 3 from line 11 to line 19, but instead of extending to the left, we extend to the right. Giving us the revised algorithm, Algorithm 13 as follows:

Algorithm 13: BWT-MEM-Hyb

Input : $BDBWT$ indexes $idxS$, $IdxT$ of length n and m respectively and a tuple $seeds$

Output: Vector of tuples containing $BDBWT$ interval pairs of a MEM and its length.

```
1  Initialize stack S with interval pairs from the seeds
2  Initialize ret as the return array of tuples
3  while S is not empty do
4      (ip0, ip1, d) = S.pop()
5      ([i0..j0][i'0..j'0], [i1..j1][i'1..j'1]) = (ip0, ip1)
6      if (j0 − i0 + 1 < 1 or j1 − i1 + 1 < 1) then continue
7      if enumdiffLeft(idxS, idxT, ip0, ip1) then
8          ret.append(subroutine(tuple(ip0, ip1, d))) // Algorithm 4
9      I = ∅
10     foreach c ∈ Σ do
11         i1 = idxS.leftExtend(ip0, c)
12         i2 = idxT.leftExtend(ip1, c)
13         I.push(i1, i2)
14     foreach i ∈ I do
15         (j1, j2) = i
16         if idxS.isRightMaximal(j1) or idxT.isRightMaximal(j2) or
            enumDiffRight(idxS, idxT, ip0, ip1) then
17             S.push(tuple(j1, j2, d + 1))
18     foreach c ∈ Σ do
19         i1 = idxS.rightExtend(ip0, c)
20         i2 = idxT.rightExtend(ip1, c)
21         I.push(i1, i2)
22     foreach i ∈ I do
23         (j1, j2) = i
24         if (not idxS.isLeftMaximal(j1)) or (not idxT.isLeftMaximal(j2)) or
            enumDiffLeft(idxS, idxT, ip0, ip1) then
25             S.push(tuple(j1, j2, d + 1))
26 return ret
```

2.3.6 Optimization

The hybrid solution is not optimal, and further optimizations including dropping the reliance on having to build suffix array separately by constructing the $PLCP$ array directly from BWT could still be achieved. Pursuing more efficient implementation of the algorithm, however, seems like a dead end and out of scope for this thesis. It is hard to compete with the minimizer when its time complexity is dominated by the number of occurrences instead of the lengths of the text.

Another path to a larger optimization would be to investigate the possibility of more efficient algorithm for the $BDBWT$ portion of the solution. The algorithm used is derived from the approach using only $BDBWT$ to yield all $MEMs$, designed to only look in one direction. With the hybrid solution, however, we must look in both directions as the minimizers don't guarantee maximality on either side of the match.

2.3.7 Analysis

Theorem 3 *The hybrid approach can be used to output set of maximal exact matches in*

$$\mathcal{O}(N \log N + ((m + n) \log \sigma))$$

where N is the total number of minimizers matches, m, n the lengths of the input text, and σ the size of the alphabet.

Proof The analysis of the hybrid implementation can, for a large part be reduced to the analysis of both the minimizer and $BDBWT$ approaches. First, we require the linear time scan over each text to produce the minimizers, thus $\mathcal{O}((n + m)w)$ to result all minimizers for both texts, where n and m correspond to the input text sizes and w the size of the window for each minimizer.

The other major part of the hybrid, the $BDBWT$ extension, follows the same principle as described in Chapter 2.1, giving us the preliminary time complexity of $\mathcal{O}((m + n) \log \sigma + occ)$. However, within the hybrid implementation, the $BDBWT$ algorithm isn't used to find all MEMs at the same time, but only to extend a single interval translated from the minimizers. Based on the minimizers, the search also cannot be constrained to only to the left, and the right-side of the string must be extended in the same manner.

The right extensions required can be done by duplicating the procedure for left extensions but operating on the reverse intervals instead, as seen on the Algorithm 13. The time complexity of the right side becomes a constant factor multiple for the extensions, having no bearing on the amortized analysis.

Furthermore, in the worst case, each $BDBWT$ computation on the hybrid approach, could end up resulting in all the MEM matches, in the case that all minimizers found correspond to one with same lexicographic value. Thus, the time complexity from the $BDBWT$ Lemma 8 still holds verbatim for the worst case.

For the necessary auxiliary data structures, we know that the LF-mapping, inverse suffix array, LCP-array and partition bitvectors can be constructed with a linear time scan over the $BDBWT$ indexes. And from Theorem 2 we know that the minimizer matches can be converted to SA intervals in time linear to the number of matches.

This leaves us to determine the time complexity required to match the minimizers when only one match for each different minimizer is required to determine the intervals. The analysis is made simply by noticing that, if each minimizer is unique and non-repeating (appearing only once). The algorithm to match the minimizers performs verbatim to the one used in Chapter 2.2 for minimizer-based approach. Thus, the time complexity to match the minimizers remains $\mathcal{O}(N \log N)$ for output, where N is the number of minimizer-matches between the input strings as proven in Lemma 9

The overall time complexity then becomes the sum of matching the minimizers (charged into the output) and extending the matches using $BDBWT$ $\mathcal{O}(N \log N + ((m+n) \log \sigma))$, where N is the total number of minimizers matches, m, n the lengths of the input text, and σ the size of the alphabet Σ, proving Theorem 3 $\square$.

3 Chaining

Between any two texts that bear any similarity, there are bound to be substrings that can be found within both. Alone, these substrings, however, give us only a limited amount of information. A natural follow up to the information is the order in which the substrings are in their respective texts. If the order matching substrings is the same, or even if they follow a pattern, we can deduce that there exists a stronger relationship between the two strings than just a length of matching symbols.

3.1 Motivation

Sequence alignment based on edit distance, using traditional methods quickly becomes inefficient to compute for long reads due to its quadratic time complexity in relation to the length of the compared strings [1, 2].

The chaining algorithm presented in the paper by Mäkinen and Sahlin [2] describes an algorithm that takes a set of local alignments in the form of anchors, matches between the input strings and outputs the best possible semi-global alignment given the input anchors. In this thesis we assume each anchor is a maximal exact match.

We implement Algorithm 2, "Chaining with two-sided overlaps" described in the original paper [2]. The algorithm considers four distinct cases of overlaps between the anchors. First case is that the anchors do not overlap in either text, with second case being that there exists an overlap in one text, but not in the other one. The third and fourth case are symmetric to each other, representing cases where the overlap exists in both texts, but the overlap is greater in one of them.

The results for the chaining approach are promising with the experimental data given at the end this thesis. The results also confirm that the chaining algorithm has an approximated alignment comparable to methods guaranteeing the optimal alignment in a quadratic time.

Further, to compute the edit distance of the input strings from the chaining, we make use of the intervals that are absent from the chaining output, as it must hold that all mismatches between the strings must be contained in these substrings, allowing splitting the computation into smaller segments to implement partitioned edit distance calculation for the whole input.

On top of the implementation, we give a simple way of parallelizing the computation by splitting the bulk of the computation between logical threads during range-maximum queries.

3.2 Definition of parts

To understand the chaining algorithm, we have to define what constitutes as a chain, what is a valid output from the algorithm, and what would be the optimal result. Beginning with Definition 18 for anchors.

Definition 18 (Anchor intervals) *Given texts $T[0..n-1]$, and $S[0..m-1]$ of length n and m respectively. We define $I = \{([a..b],[c..d]), a \leq b, c \leq d\}$ to denote a set maximal exact matches used as anchors for the chain.*

A strict precedence of anchors would imply that each anchor interval, could only begin after the end of a prior interval in both input strings, preventing any overlap from occurring at all. Using strict overlap would reduce the number of anchors selected by a large margin but would likewise reduce the number of symbols covered in the text.

The chaining algorithm instead uses a relaxation of strict precedence, the weak precedence, Definition 19. In the rest of this thesis we assume that when referring to precedence with the symbol '$\prec$' we are referring to the weak precedence unless otherwise explicitly stated.

Definition 19 (Weak-precedence) *Given two intervals $I_0 = ([a..b])$ and $I_1 = ([c..d])$. A weak precedence $I_0 \prec I_1$ holds if and only if $a < c$. That is, overlap is allowed if the precedence holds for the beginning of each interval.*

Weak precedence ensures that if the anchors were to be combined into two continuous substrings, then each string would remain a (non-continuous) substring of original text. Definition for two-sided overlap can be formalized Definition 20 below.

Definition 20 (Chaining with weak precedence) *Let $C[0..\ell-1]$ be a set containing ℓ anchors $I_j = ([a..b][c..d]) \in C, 0 \leq j \leq \ell$.*

Any subset of the set of anchors C can be considered a chain (under weak precedence) if, and only if the selection of anchors can be sorted with respect to the weak precedence defined above.

3.3 Implementation

The chaining algorithm computing two-sided overlap in $\mathcal{O}(n \log n)$ time is implemented verbatim to the paper first describing it [2]. In this chapter we, instead of redefining the algorithm, focus on presenting a concrete example of the implementation.

For the example, we will assume the following two texts of 67 character each.

T = CAATTTAAGGCCCGGGGTGCGTGATCATCATTTGTGCGTGTTCATCATTTGTGCGTGATCATCATTT

S = CAAAGTAAGGCCCTCCAGTGCAAAGTGATTACCGTGCGTGATCATCATTTAGTGCGCGTGACATCTT

Further, we can assume all maximal exact matches are found by using one of the three methods described prior. We choose for the sake of the conciseness of the example to limit ourselves to maximal exact matches of four characters or longer. The Table 4 contains interval pairs given as the input vector of interval pairs A of maximal exact matches for the two texts.

Table 4: Intervals contained in the vector A at indexes $j \in [0..|A|-1]$ with a string representation of each interval.

Index j	Anchor interval in T ([a,b])	Anchor interval in S ([c,d])	MEM string
0	([2, 6])	([46, 50])	ATTTA
1	([5, 12])	([5, 12])	TAAGGCCC
2	([16, 32])	([33, 49])	GTGCGTGATCATCATTT
3	([18, 23])	([55, 60])	GCGTGA
4	([19, 22])	([32, 35])	CGTG
5	([23, 27])	([43, 47])	ATCAT
6	([26, 30])	([40, 44])	ATCAT
7	([33, 39])	([33, 39])	GTGCGTG
8	([35, 39])	([55, 59])	GCGTG
9	([36, 39])	([32, 35])	CGTG
10	([41, 49])	([41, 49])	TCATCATTT
11	([43, 47])	([40, 44])	ATCAT
12	([50, 66])	([33, 49])	GTGCGTGATCATCATTT
13	([52, 57])	([55, 60])	GCGTGA
14	([53, 56])	([32, 35])	CGTG
15	([57, 61])	([43, 47])	ATCAT
16	([60, 64])	([40, 44])	ATCAT

As an interlude, we need to first consider a brief overview on how the algorithm functions. The first step of the algorithm is to sort all the anchor pairs (see Table 4 for an example), with order respective to the beginning index on the first text, into an array denoted as E. The array dictates the order in which the next values will be computed.

For each jth anchor, a value $C[j]$ is computed using range-maximum queries. The exact value of $C[j]$ corresponds to the value of the best chaining leading up to that specific anchor, but not including it, for that, we define $C_p[j]$. These values corresponding to the best choices of chaining are stored in the four range-maximum queries (RMQ), each representing a different way of selecting the chain at the point. No overlaps, overlap on one text but not the other one, and both variations of overlaps in both texts depending on which overlap is greater.

To implement the range maximum data structures efficiently we utilize an $\mathcal{O}(\log n)$ implementation of the data structure defined as an implementation to a paper by Kuosmanen et al [17]. The implementation is used verbatim after sorting the necessary keys into an increasing order prior to insertion into the data structure.

The algorithm requires use of four separate RMQ data structures, of which the first two are initialized with keys corresponding to the index d, the end-point index on the second text as well as with zero, each with value of a large negative integer. The other two structures are initialized with keys $c-a$, the subtraction of the beginning index in T from that in S, likewise with value of a large negative integer.

The start- and endpoints of the interval in T (a and b) are used as keys (a, j) and (b, j) for the primary loop of the algorithm, stored into an array E and sorted into an increasing order relative to a and b. The values of the E array are visualized in Table 5.

Table 5: The array E with each index i represented by pair (e_1, e_2).

i	e_1	e_2
0	2	0
1	6	0
2	5	1
3	12	1
4	16	2
5	18	3
6	19	4
7	23	3
8	23	5
⋮	⋮	⋮
33	66	16

During the loop of the algorithm, on each ith iteration, we take the ith key pair from the array E as (e_1, e_2) check if interval starting point a in $([a, b], [c, d]) = A[e_2]$ equals to e_1. If the equality holds, then we perform four range-maximum queries

$T_a(0, c-1)$, $T_b(c, d)$, $T_c(-\infty, c-a)$, $T_d(c-a+1, \infty)$, and store the maximum value out of all queries as

$$C[j] = \max \begin{pmatrix} T_a(0, c-1) \\ c+T_b(c, d) \\ a+T_c(-\infty, c-a) \\ c+T_d(c-a+1, \infty) \end{pmatrix}$$

where $C[j]$ denotes the 'score' or symmetric coverage of the chain up to, but not including the anchor.

The symmetric coverage is then computed as $C_p[j] = C[j]+b-a+1$, and the values in the T_c and T_d structures are updated for key $c-a$ into $C[j]-a$ and $C[j]-c$ respectively. Similar to $C[j]$ the $C_p[j]$ contains the symmetric coverage of the chain up to, but also including the anchor.

In the given example, we can note that the first index of the array E matches the condition, and since each query structure remains on their initialized values, it is easy to see that the maximum value $C[0]$ is gained from the $T_a(0, c)$ query with value of 0. Thus, the value $C_p[0]$ becomes $0+6-2+1=5$. We also store a reference to the index to remember which query gave the maximum value to $C[0]$ for traceback.

In the case that the equality did not hold above, the algorithm simply updates the values within all four query structures. The structures T_a and T_b are upgraded with values $C_p[j]$ and $C[j]-c$ respectively, both for key d. The structures T_c and T_d on the other hand are both set to value $-\infty$ for key $c-a$.

Using the example, we can notice that the equality fails for the second value in E, in which case the key (50) is updated to have value of $C_p[0] = 5$ as computed in the first step for T_a and the value $C[0] - 46$ for T_b. For T_c and T_d both, the key $(46-2)$ is updated to have value $-\infty$ instead.

After all values in E are iterated over, the traceback to obtain the chain can be performed by taking the largest value at the end of the C_p array and using traceback to move through values using reference to a prior index until the first index is reached and no more traceback steps are available.

As the result for the example, the following table, Table 6 holds all intervals in order for the optimal chain.

Table 6: Optimal chain for the example string T and S.

Interval in T	Interval in S	Symmetric Coverage up to the anchor
([52,57])	([55,60])	31
([41,49])	([41,49])	25
([33,39])	([33,39])	16
([19,22])	([32,35])	12
([5,12])	([5,12])	8

3.4 Edit Distance from chaining output

Recall Definition 1, edit distance is the number of insertions, deletions, and substitutions required to convert one string to another.

To compute the edit distances for the chaining alignment, we need to take into consideration instead all the intervals that are not part of the chain.

When gathering the absent intervals from the chain, it is important to note that the absents segments must be matched in order, substituting an empty segment if corresponding absent interval does not exist, see Observation 5.

Observation 5 *Edit distance between string T and S can be approximated efficiently from a chaining alignment C for the strings by using substrings absent from the chain ordered in respect to their locations in regard to the original text.*

Let strings S and T be two strings with a chain C with N anchors computed between them and A containing all absent intervals. Each segment of the chain denoting a maximal exact match.

First, we note that each absent segment has an edit distance of at least one. This must hold because by Definition 4, each MEM is both left-and-right maximal; no segment of the chain can be extended to the left or right without introducing a mismatch.

Second, each absent segment is separate and non-overlapping. The absent segments are all the substrings of the two texts that are in between the beginning of the text, each anchor, and the end of the text. Each absent segment has clear and defined boundaries.

Third, all absent segments can be sorted into same order with respect to the beginning and ending indexes such that for $A_i = \{([i_0, j_o][i'_0, j'_0]), ([i_1, j_1][i'_1, j'_1])\}$ and $A_{i+1} = \{([i_2, j_2][i'_2, j'_2]), ([i_3, j_3][i'_3, j'_3])\}$ it holds that $i_0 < i_2, j_0 < j_2$, $i_1 < i_3, j_1 < j_3$, and likewise for the indexes i', j'.

The edit distance between two maximal exact matches is naturally zero. Further, all matches longer than the threshold, are contained in the chain.

Using the absent intervals now allows us to compute independent local alignments for each absent segment. Combining the edit distances and alignments gained from the local alignments then yields a semi-global alignment between T and S.

The time complexity of aligning the absent intervals is bounded by the product of lengths of the substrings in each text [18]. Let N denote the maximum length of the occ absent intervals, the time complexity to compute all absent interval alignments is $\mathcal{O}(occ \cdot N^2)$. The computation can further be sped up by parallelizing the alignment between logical cores, each core computing a single occurrence at a time.

4 Results

To assess the results of the algorithms described in this thesis, the algorithms were run multiple times against Edlib [18], comparing the time taken and edit distances resulted. To use the test suite, we pass the tool the same configuration file as for running the tool normally combined with two additional arguments: the number of mutations to perform each step, and the number of steps to run the test for. For the first string sequence given as input, we used pre-generated [19] random DNA-sequences. The second string sequence in all tests was a sequence initially copied and mutated from the first sequence to a low percentage (at most 1%) to give a non-zero starting edit distance, a special case in which our approach performs significantly worse. For exact details on the sequences, see the linked repository in Appendix 2.

After each iteration one of the two texts given to the algorithm as an input was further mutated by an automated tool, increasing the edit distance by the amount given as input for the test suite. The mutations were accomplished by changing selected symbols to another random symbol within the alphabet. The random seed for the mutations was fixed to ensure that all results to be reproducible. If the same strings were compared with the test suite using different increment counts, the results would vary by a small margin as the randomness is advanced at a different pace.

All tests were done on the same machine running Ubuntu 18.04.5 LTS with an AMD processor with a CPU frequency of 2.5 GHz using four physical cores and 7 gigabytes of DDR3-1600 MHz memory.

The choice to test the algorithms against Edlib was natural as it is also used to compute the final edit distance for the absent intervals as described. The main result of the algorithms being a faster way to compute edit distance, comparing the method in detail to methods such as Minimap2 [20] that are specifically tailored to aligning sequences against a "large reference database", were omitted as further work to create a comparable alignment results would have been necessary to be created. Our algorithms lack ability to detect, or understand the structure of DNA sequence, and simply returns the best approximation of the edit distance. Using just the chaining alignment, a naive alignment can be achieved, but likewise, it does not hold respect to structure of the DNA, introns and exons. Future work could potentially remedy this limitation by adding further clauses and computation to how matches are chosen, and how the chaining is performed.

Initially the expectations of the results were that the algorithms would be able to give a general approximation of the edit distances between any two long strings. The initial, hopeful, expectation was that an edit distance with at most 10% divergence could be found in time faster than by using Edlib alone. The results, however, impressed with being both faster, and with most of the test samples, still optimal result in terms of edit distance. The initial expectations were also in favor of using the Bi-Directional Burrows-Wheeler transform, $BDBWT$.

Overall, the results are good but not groundbreaking. Regretfully as hinted before,

both methods utilizing the Bi-directional Burrows-Wheeler Transforms fall short of what is achievable by using minimizers alone. The minimizer approach described in the Chapter 2.2 on the other hand shows good results and should be considered the prime option for this implementation.

However, as predicted, the algorithm provides the best results when the edit distances are primarily caused by individually mismatched symbols, such as in certain cases of read-error with DNA samples. When patterns in the string are shifted, moved from one location to another or duplicated. The accuracy of the approximation suffers slightly. On the other hand, this verifies that as long as the strings are subjected to mutations to single symbols and not to shifts in longer sequences, the approximation is most of the time able to result in the optimal edit distance. The minimizer approach gives results of being able to find all minimizer matches of k length in roughly the same time as Edlib was able to align two strings in situation where the strings represented the most optimal case for it. The time gap resulted in being primarily from the time taken to implement the chaining and edit distance computation for the absent intervals.

The minimizer approach gives results that with majority of the test data are comparable to the results from the Edlib approach, even in the cases where the given input strings have a very high similarity (99.9%). As the edit distance between the input strings increases, the time taken by Edlib to guarantee optimal edit distance grows quite fast, seen in Figure 4.

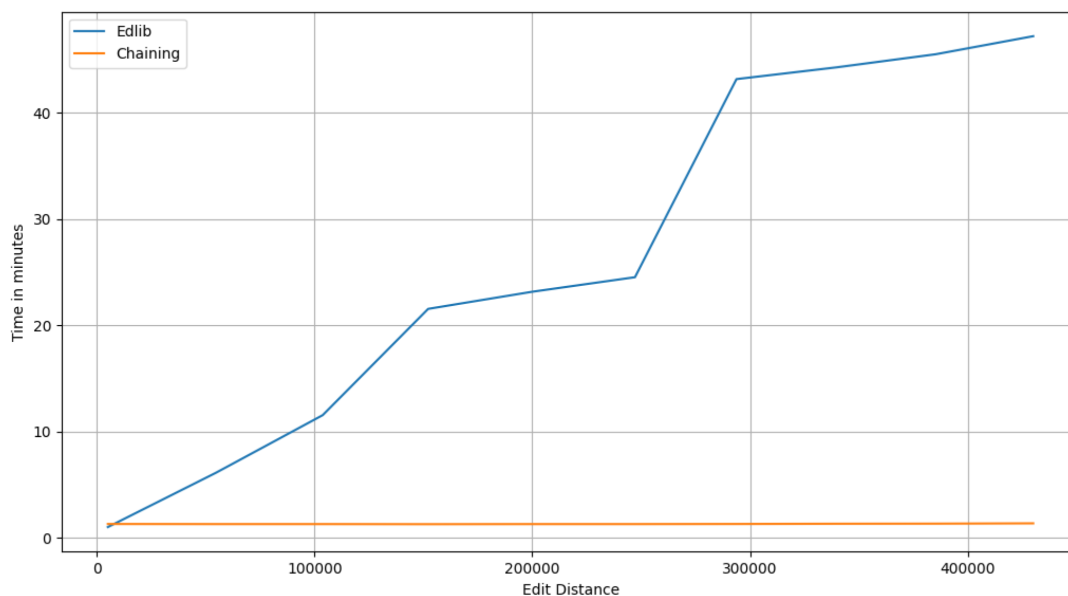

Figure 4: Comparison of speed of Edlib and Chaining approaches relative to edit distance. The chaining approach uses the minimizer approach with k-mer size of 50. The length of both input texts is 5 million symbols. Raw data can be found in the Appendix 1 Table 11.

With minimizers and chaining, the time taken grows very slowly, bounded inversely relative to the number of occurrences, as the method allows computing the edit distances only for sequences that are absent from the chain, seen by the time taken by chaining growing very slowly in Figure 4 and Appendix Table 11 with same data. The time taken by the chaining approach, especially with minimizers, does not scale off the edit distance like Edlib. Instead, it is dominated by the number of occurrences, growing significantly slower. With edit distance corresponding to just around 1% of the total length of the input text, the chaining alignment can approximate the edit distance with high accuracy (in the above plot, the approximated edit distances were exactly the optimal), and as evident, with a significant increase in speed. As the edit distance grew to 10% of the input strings, the chaining approach was able to approximate the exact edit distance in a fraction of the time.

Higher threshold (MEM minimum size, and k-mer length with minimizers) value increases the time taken to compute the edit distances for the absent intervals proportionally to the average size of each such interval. As stated in 5 above, the time complexity remains quadratic to length of each interval. As such, with higher thresholds, less matches, and less unique absent intervals can be reliably returned. This observation from the Tables 7, 8, and 9, below, can be used to come to the results that the size of the threshold is desireable to set as low as possible. Too low threshold, however, causes the algorithm to give wrong results.

In the following three Tables 7, 8, and 9, the maximal exact matches used as anchors are all computed using k-mers and minimzers using the minimizer approach described in Chapter 2.2.

Table 7: Both the accuracy of the chaining approach as well as the time required grows with greater k-mer sizes. The time taken for Edlib for the same edit distance was 1 minute and 10 seconds. The results are with strings consisting of million symbols, and the randomness seeded with same value.

k-mer size	Edlib ED	Chaining ED	Chaining time taken
15	48425	48425	00min08s990ms
20	48425	48425	00min10s931ms
50	48425	48425	00min18s082ms
100	48425	48425	00min36s415ms

Table 8: Comparing the accuracy of the methods for strings of 5 million symbols each with different k-mer lengths. The time taken by Edlib on each test was 23 minutes and 17 seconds.

k-mer size	Edlib ED	Chaining ED	Chaining time taken
25	199861	199861	01min03s564ms
50	199861	199861	01min31s121ms
100	199861	199861	02min59s967ms

Table 9: Comparing the accuracy of the approach using minimizers at different k-mer sizes. With two strings of 5 million symbols each, including randomly moving 19132 symbols as part of multiple longer sequences. It is observed that the size of the k-mer does not greatly influence the accuracy in case of shifted patterns. For comparison, time taken by edlib was 2 minutes and 31 seconds.

k-mer size	Edlib ED	Chaining ED	Chaining time taken
25	24188	24959	01min05s744ms
50	24188	24959	01min31s166ms
100	24188	24958	02min44s065ms

Further comparing the three approaches for computing the maximal exact matches, we notice that the hybrid implementation has the least predictable speed out of the three. With small edit distance, the hybrid approach is easily the slowest of the three, but with increased edit distance, it quickly becomes faster than the $BDBWT$ approach, but no quicker than the minimizer based one.

As visualized in Table 10, the $BDBWT$ approach is in most cases slowest of the three. The greatest take-away for $BDBWT$ for this type of problem is that at least with the approach and implementation given, considerations upon other methods should be taken.

Table 10: Comparison of speed between the three methods, the fastest speeds are bolded. The significiant speed up with the hybrid after the smallest edit distance can be keenly observed. Further, both the minimizer, and hybrid implementations see a slow down towards at the highest tabled edit distance. The slow-down is caused by the selected minimizer scheme not being able to discern enough minimizer matches (as noted in Observation 4) and the number of absent intervals becomes smaller, speeding up the overall edit distance computation less.

Edit Distance	2812	45650	71819
Edlib	**00min01s808ms**	00min17s984ms	00h00min33s792ms
Minimizer	00min03s010ms	**00min03s447ms**	**00h00min08s769ms**
Hybrid	21min22s561ms	00min09s100ms	00h00min13s848ms
BDBWT	03min08s040ms	02min47s935ms	00h02min37s010ms

The results give a clear indicator that the method shows clear promise and potential use. However, the choice of threshold value remains as an open problem and must be set with intuition. Good values for the threshold, however, seem to range between 20 and 100. With longer strings allowing for a larger threshold, and smaller strings for smaller. A good guideline is base it around the average size of a maximal exact match one would expect to find.

5 Discussion

The results as discussed above, even if taken into consideration that the values are still just approximations and there is no clear way of discerning how close to the optimal results they are. Further, due lack of technical limitations, the approach was, regrettably, not able to be tested against truly large strings of 10 million symbols or more as the system was not able to handle space complexity required. The main limitation of this approach, however, is that under no case, can it with certainty guarantee that the results are optimal, and it also is not able to give any approximation-ratio for the results.

A minor limitation to the approach is that the size of the threshold for a maximal exact match cannot be set too low, or the algorithm will predictably give significantly increased edit distance, to the point where the result is completely unusable. Based on testing, this has to do with the number of matches between the strings growing far too high, and either the chaining, or range-max-query algorithms not being able to handle high number of matches that are lexicographically identical. This limitation, however, was only noticed when setting a threshold to values smaller than 15 on strings with length at least million symbols each. The limitation boils down to expected number of times any sequence of k symbols can appear on the text, and presumably the size of the alphabet used.

The slowness of the Bi-Directional Burrows-Wheeler Transform, in comparison to the minimizer approach came as a minor surprise. The initial hypothesis and hunch were that it would in fact, be the faster and more reliable of the two methods. But as the results show, this is not the case. The method is certainly reliable, but it is performing slower than if using Edlib alone, makes it mostly not worth using to compute edit distances.

As mentioned earlier, I believe there are still multiple ways to improve on the implementation. For example, the choice of minimizer scheme, and the way the minimizers are extended to cover the maximal exact matches. The hybrid implementation as well, is in its current, far from being the most optimal implementation of its kind. A different way of computing maximal exact matches by using Bi-Directional Burrows-Wheeler Transform could also change the outcome more to its favor as well.

As the results show. The approach described in this thesis for approximating the edit distance between two strings works the best when the two string are originally based on the same string, but one, or both strings have undergone a level of mutation or errors. A potential use-case for this method might lie in analyzing the similarity between long strings where the accuracy of the result isn't the most crucial aspect. Or as a preliminary tool to filter strings depending on whether they might have a high similarity between them.

The latter use case especially, swift analysis of similarity in relation to edit distance is something that this approach solves extremely well. As can be seen in Figure 4, with increase in edit distance, our approach was able to return the edit distance in a fraction of the time taken. And regardless of the edit distance, the time taken was

still within a small margin of one another.

Another use could make use of the inaccuracy resulted from shifts in the patterns. The described approach could be run together with other methods and from the difference and the actual output of the chaining algorithm, insight into how the texts differ and how the patterns have shifted between the texts. The chaining alignment especially could potentially find a lot of use in finding segments that are missing from one string in relation to a reference string.

The accuracy of the method under an ideal situation, however, does give hope that with further work, more efficient algorithms to compute edit distance optimally, or with a good approximation ratio be developed. Taking minimizers for an example, accounting for shifts in the patterns, the the worst case for our algorithm, could potentially be solved by an advanced heuristic, or by a more suitable minimizer scheme. Another possible approach would be to allow a certain level of mismatch or mutation to occur within the matches, possibly allowing the algorithm to sacrifice robustness for better results in the worst case.

With thanks to all the friends and family, including those no longer with us. And to everyone else along the way.

References

1 A. Backurs and P. Indyk, “Edit distance cannot be computed in strongly subquadratic time (unless seth is false),” in *Proceedings of the Forty-Seventh Annual ACM Symposium on Theory of Computing*, STOC ’15, (New York, NY, USA), p. 51–58, Association for Computing Machinery, 2015.

2 V. Mäkinen and K. Sahlin, “Chaining with Overlaps Revisited,” in *31st Annual Symposium on Combinatorial Pattern Matching (CPM 2020)* (I. L. Gørtz and O. Weimann, eds.), vol. 161 of *Leibniz International Proceedings in Informatics (LIPIcs)*, (Dagstuhl, Germany), pp. 25:1–25:12, Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020.

3 V. I. Levenshtein, “Binary codes capable of correcting deletions, insertions, and reversals,” in *Soviet physics doklady*, vol. 10, pp. 707–710, 1966.

4 U. Manber and G. Myers, “Suffix arrays: A new method for on-line string searches,” in *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA ’90, (USA), p. 319–327, Society for Industrial and Applied Mathematics, 1990.

5 V. Mäkinen, D. Belazzougui, F. Cunial, and A. I. Tomescu, *Genome-Scale Algorithm Design: Biological Sequence Analysis in the Era of High-Throughput Sequencing*. Cambridge University Press, 2015.

6 S. Gog, T. Beller, A. Moffat, and M. Petri, “From theory to practice: Plug and play with succinct data structures,” in *13th International Symposium on Experimental Algorithms, (SEA 2014)*, pp. 326–337, 2014.

7 M. Burrows and D. J. Wheeler, “A block-sorting lossless data compression algorithm,” tech. rep., 1994.

8 P. Ferragina and G. Manzini, “Opportunistic data structures with applications,” in *Proceedings 41st Annual Symposium on Foundations of Computer Science*, pp. 390–398, Nov 2000.

9 T. Schnattinger, E. Ohlebusch, and S. Gog, “Bidirectional search in a string with wavelet trees and bidirectional matching statistics,” *Information and Computation*, vol. 213, pp. 13 – 22, 2012. Special Issue: Combinatorial Pattern Matching (CPM 2010).

10 D. Belazzougui, F. Cunial, J. Kärkkäinen, and V. Mäkinen, “Versatile succinct representations of the bidirectional burrows-wheeler transform,” in *Algorithms – ESA 2013* (H. L. Bodlaender and G. F. Italiano, eds.), (Berlin, Heidelberg), pp. 133–144, Springer Berlin Heidelberg, 2013.

11 M. Roberts, W. Hayes, B. R. Hunt, S. M. Mount, and J. A. Yorke, “Reducing storage requirements for biological sequence comparison,” *Bioinformatics*, vol. 20, pp. 3363–3369, 07 2004.

12 D. Belazzougui and F. Cunial, "Fully-Functional Bidirectional Burrows-Wheeler Indexes and Infinite-Order De Bruijn Graphs," in *30th Annual Symposium on Combinatorial Pattern Matching (CPM 2019)* (N. Pisanti and S. P. Pissis, eds.), vol. 128 of *Leibniz International Proceedings in Informatics (LIPIcs)*, (Dagstuhl, Germany), pp. 10:1–10:15, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019.

13 "cppreference." https://en.cppreference.com/. Accessed 2020-07-27.

14 OpenMP Architecture Review Board, "OpenMP application program interface version 5.0," 2018. Accessed 2020-08-23.

15 D. DeBlasio, F. Gbosibo, C. Kingsford, and G. Marçais, "Practical universal k-mer sets for minimizer schemes," in *Proceedings of the 10th ACM International Conference on Bioinformatics, Computational Biology and Health Informatics*, BCB '19, (New York, NY, USA), p. 167–176, Association for Computing Machinery, 2019.

16 J. Kärkkäinen, G. Manzini, and S. J. Puglisi, "Permuted longest-common-prefix array," in *Combinatorial Pattern Matching* (G. Kucherov and E. Ukkonen, eds.), (Berlin, Heidelberg), pp. 181–192, Springer Berlin Heidelberg, 2009.

17 A. Kuosmanen, T. Paavilainen, T. Gagie, R. Chikhi, A. I. Tomescu, and V. Mäkinen, "Using minimum path cover to boost dynamic programming on dags: Co-linear chaining extended," in *Research in Computational Molecular Biology - 22nd Annual International Conference, RECOMB 2018, Paris, France, April 21-24, 2018, Proceedings* (B. J. Raphael, ed.), vol. 10812 of *Lecture Notes in Computer Science*, pp. 105–121, Springer, 2018.

18 M. Šošić and M. Šikić, "Edlib: a C/C++ library for fast, exact sequence alignment using edit distance," *Bioinformatics*, vol. 33, pp. 1394–1395, 01 2017.

19 "Stothard p (2000) the sequence manipulation suite: Javascript programs for analyzing and formatting protein and dna sequences. biotechniques 28:1102-1104." http://www.bioinformatics.org/sms2/random_dna.html. Accessed 2020-21-12.

20 H. Li, "Minimap2: pairwise alignment for nucleotide sequences," *Bioinformatics*, vol. 34, pp. 3094–3100, 05 2018.

Appendix 1. Results data table

Table 11: Accurate values for the presented plot (strings of 5 million symbols and a treshold of 50), faster time given in bold text. Observe that the time taken by the chaining approach does not scale significantly from increaed edit distance.

Edlib Time	Edlib ED	Chaining Time	Chaining ED
01min02s976ms	5084	01min31s839ms	5084
06min13s556ms	54737	**01min31s073ms**	54737
11min56s047ms	103762	**01min30s525ms**	103762
21min56s477ms	152194	**01min30s185ms**	152194
23min17s009ms	199917	**01min31s121ms**	199917
24min53s828ms	247192	**01min30s955ms**	247192
43min17s835ms	293804	**01min32s193ms**	293804
44min28s712ms	339829	**01min33s535ms**	339829
45min52s007ms	385315	**01min35s051ms**	385315
47min21s936ms	430108	**01min38s375ms**	430108

Appendix 2. Location of implementation

Everything discussed as implemented algorithms and methods used is included within the GitHub repository https://github.com/algbio/bdbwt-mem-chaining.

The repository constains instructions on how to compile and use the implementation as well as attributions to all external libraries used. The repository also contains a simple tool to mutate string by a set percentage of it's length, and the test utility used in results to iteratively perform edits in one string to increase the edit distance and then compare our approach with Edlib.