Gauging the usefulness of minimal absent words for efficient similarity search on early music

Jelle Mulyadi (j.mulyadi@students.uu.nl)

August 2021

Master thesis
40 ECTS
Project supervisor (first examiner):
dr. F. (Frans) Wiering
Second examiner:
Prof. dr. R.C. (Remco) Veltkamp

Utrecht University

Abstract

Minimal absent words (MAWs) are subsequences that do not occur in a sequence, but their longest proper prefixes and longest proper suffixes do occur in the sequence. Previous research has shown potential for MAWs to be used for sequence comparisons.

In this thesis, the usefulness of MAWs for similarity search is gauged for early music datasets. The goal of similarity search is to obtain entries from a dataset similar to a query. The datasets consists of pages of music, encoded as interval sequences. A similarity search algorithm based on MAWs (SSMAW) is compared to several algorithms, among which: BLAST, Pass-Join and PIVOTALSEARCH. This thesis provides an extensive overview of this comparison.

We utilize a ground truth that was established in earlier research. This ground truth consists of known pages for three different query tasks: 1. Finding duplicate pages, 2. Finding pages that contain the same music and 3. Finding pages that contain related music.

The algorithms have been compared based on their retrieval performance, interpretability and scalability. Retrieval performance is measured by the mean rank of the relevant query results, for each of the query tasks. The score distributions of the algorithms for the query tasks have been calculated, using which interpretability is gauged. In addition, query times and memory usage were recorded for increasingly sized data sets. All data is provided.

The research has demonstrated that SSMAW is the best performing algorithm among the algorithms in terms of retrieval performance and scalability.

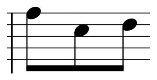
The research has provided a library of the algorithms, and configurations of their parameters. In addition, based on the results of the comparison, a decision tree was constructed to aid selection of an algorithm for similarity search.

Contents

1 Introduction

As a result of digitization, large libraries of digitized music scores have arisen (e.g. IMSLP [1]). These libraries consist of scanned pages of sheet music. The pages are in image format, which means that they can't be searched based on their content, but only on connected metadata. Developments in optical music recognition (OMR) have made it possible to encode the page-images (for example in the music encoding initiative (MEI) format [2], see Figure 1 for an example of this format). Therefore, we now have access to big datasets of machine-readable music scores and their matching page-images, and can search and analyze the data in terms of its musical content: pitches, duration, etc. However, the OMR process is not flawless and introduces errors and noise [Pugin and Crawford, 2013]. Because of this, there is a need to be able to search the datasets in a way that allows for these errors. In addition, it is interesting to find page-images in the dataset, which are related to the query page-image (that contain part of the query, or are musically related to the query) [Crawford et al., 2018]. In the research field of information retrieval (IR), this problem has been called approximate string matching [Ukkonen, 1985], or string similarity search [Wandelt et al., 2014]. It is concerned with finding strings in the dataset that match the query pattern approximately. That is: they are similar to the query pattern. The algorithms for strings are also applicable to music. The data that is produced by OMR tools, can be represented as strings. In fact, pages of music are long sequences of notes, and patterns in these sequences are melodies. This is illustrated in Figure 1.

To make sense of the large amounts of noisy, unstructured data, musicologists need tools to retrieve information from the data, and aid them with their analysis. The research field concerned with the development of these tools is called music information retrieval (MIR). Developing error-robust and efficient searching methods is key for the development of MIR systems.



(a)

```
<note pname="f" oct="5" dur="8" stem.dir="down"></note>
<note pname="c" oct="5" dur="8" stem.dir="down"></note>
<note pname="d" oct="5" dur="8" stem.dir="down"></note>
```

(b)

```
f5 c5 d5
```

(c)

Figure 1: Music in (a) sheet music, (b) MEI and (c) sequence representation

[1] www.imslp.org
[2] www.music-encoding.org

Bioinformatics, like music, also deals with long sequences, which represent the structure of molecules. Therefore, inspiration can also be drawn from this research field, and there is potential in applying its methods to music [Bountouridis et al., 2017]. For pairwise sequence comparison of biological sequences, algorithms like FASTA [Lipman and Pearson, 1985] and BLAST [Altschul et al., 1990] have been developed. However, these classical sequence comparison methods, based on alignments, are being replaced by measures that look at sequences as compositions of patterns. Minimal absent words (MAWs) are an example of such patterns, and have recently been used in the development of alternative, alignment-free sequence comparison algorithms [Crochemore et al., 2016, Barton et al., 2014, Chairungsee and Crochemore, 2012]. The volume of the data sets that are presently being dealt with, asks for alignment-free algorithms, as these are less computationally complex.

In [Crawford et al., 2018] research is done on the usage of MAWs for the purpose of searching in a database of sheet music page-images. The demo provided for their algorithm allows real-time searching of page-images on a collection of approximately 32k page-images. With that, it has proven to be a promising indexing tool for MIR systems. However, the authors recommend further testing on bigger datasets and comparison of this new method and alternative methods.

The research described in this thesis aims to provide insight into the retrieval performance, interpretability and scalability of MAWs for use in MIR systems. This will be done experimentally, by comparing it to alternative methods from the IR and bioinformatics research fields. In section 2, a literature study is described, during which a context for this paper is created, and MAWs and its alternatives are explained. In section 3, the methods for evaluating and comparing MAWs and its alternatives are described, the data that is used is introduced, and the hypotheses of the research are defined. In section 4, the details of the implementation of the algorithms are given. In section 5, the results of the evaluation are presented, and these are discussed in section 6. Lastly, in section 7 the thesis is concluded, and recommendations for future work are given.

2 Literature study

In this section, results from relevant papers are described, which creates a context for the research. It ensures that the reader of this paper has the level of knowledge to understand this research. In section 2.1 basic concepts are defined and notation is established. In section 2.2 the concept of MAWs is explained and it is shown how to calculate them. In section 2.3 the main problem of the research, string similarity search, is defined. In section 2.4 various sequence comparison methods are discussed. In section 2.5 existing string similarity search algorithms are given. Finally, in section 2.6 we will look at string similarity search in the context of MIR systems. Existing MIR systems are given and problems in these systems are identified.

2.1 Preliminaries

A main theme in this research is the comparison of two sequences. **Sequences (strings)** will be denoted using capital letters. Sequence A of length m is defined as a concatenation of m characters, where $A[i]$ is the character on the i_{th} position in the sequence: $A = A[0] + \ldots + A[m-1]$. Characters $A[i]$ come from a fixed-sized alphabet Σ. **Subsequences (substrings)** are denoted as $A[i, j]$, for i and j within the range $[0 : m-1]$. $A[i, j]$ is the subsequence that starts at position i and ends at position j in the sequence. With ε we denote the empty sequence (sequence of length 0). A **prefix** of a sequence is given by the subsequence starting at 0 and ending at some $j < m-1$: $A[0, j]$. The **suffix** of a sequence is given by the subsequence starting at some $i > 0$ and ending at $m-1$: $A[i, m-1]$. In our sequence comparisons, for sequence 1 letter A will be used and for sequence 2 letter B. For the length $|A|$ of A, m will be used, and for length $|B|$ of B, n will be used.

Matrices will be denoted using Mat. $Mat[i, j]$ indicates the element of the matrix on the i_{th} row and the j_{th} column.

Suffix arrays are an important data structure for indexing strings. A suffix array of a string, contains the starting positions of its lexicographically ordered suffixes. We will use the notation SA for the suffix array. $SA[i]$ gives us the starting position of the i_{th} suffix in the sorted suffix list. Longest common prefix (**LCP**) arrays give the length of the longest common prefix between two subsequent elements in the SA. $LCP[i]$ gives the longest common prefix between the suffix starting at $SA[i-1]$ and the suffix starting at $SA[i]$. In [Manber and Myers, 1993] an algorithm is given which constructs the suffix array in $O(n)$ expected time and $O(n)$-space.

2.2 Minimal absent words

The concept of minimal absent words was first introduced in [Béal et al., 1996]. Given a sequence A, a minimal absent word (**MAW**) B is a word (subsequence) that itself doesn't occur in A, but its longest proper prefix and longest proper suffix do occur in the sequence. The set of MAWs is a subset of the set of absent words (all words not occurring in a sequence) and superset of the set of shortest absent words (all absent words of shortest possible length). MAWs are formed from patterns within the sequence (the longest proper prefix and longest proper suffix), and negative information (characters that don't occur as left neighbour of the longest proper suffix, and characters that don't occur as right neighbour of the longest proper prefix). Therefore, the set of MAWs gives information about the sequence's composition. The set of absent words grows exponentially, and the set of shortest absent words is often too sparse. The number of MAWs a sequence has, grows at most linearly in the sequence size, and is richer than the set of shortest absent words, making it more manageable and applicable [Pinho et al., 2009, Crochemore et al., 2016].

[Pinho et al., 2009] provide an algorithm with $O(n^2)$-time and $O(n)$-space to generate the set of MAWs. Their approach constructs a suffix array and computes the LCP intervals. [Barton et al., 2014] improve upon this and present the *MAW* algorithm, which also uses a suffix array and LCP intervals. The algorithm is based on two facts:

Lemma 1.
- *Let A be a sequence, and let B be a subsequence of A. Let B_1 be the longest proper suffix of B, and let $LN_1(B_1)$ be the set of all left neighbouring letters of B_1, and let $LN_2(B_1)$ be the set of all left neighboring letters of the longest proper prefix B_2 of B_1.*
- *B is a MAW if and only if $B[0]$ is an element of $LN_2(B_1)$ and NOT of $LN_1(B_1)$.*

Lemma 2.
- *Let $PLS_1(i) = A[SA[i], SA[i] + LCP[i]]$ and $PLS_2(i) = A[SA[i], SA[i] + LCP[i+1]]$. (we use PLS standing for potential longest proper suffix)*
- *For every sequence A of length m and one of its MAWs B with length n, there exists a position i in A for which either PLS_1 or PLS_2 is the longest proper suffix of B.*

Proofs for these lemmas are given in [Barton et al., 2014]. By making use of these two lemmas, the algorithm can examine $PLS_1(i)$ and $PLS_2(i)$ for every i in the SA of sequence A, and record the left neighbours LN_1 and LN_2. As per lemma 1, the differences between sets $LN_2(PLS_1(i))$ and $LN_1(PLS_1(i))$, and sets $LN_2(PLS_2(i))$ and $LN_1(PLS_2(i))$ for every i, will give the list of all MAWs for sequence A. An example of a MAW and its identification is given in Figure 2.

The *MAW* algorithm makes two passes over the SA and LCP arrays, and runs in $O(n)$-time and $O(n)$-space. The *pMAW* algorithm in [Barton et al., 2016] manages to accelerate the *MAW* algorithm by a factor of two, by using multiprocessing.

2.3 String similarity search problem

The problem solved in [Crawford et al., 2018] is called the string similarity search problem (alternatively approximate string matching, or alignment search

i	0	1	2	3	4	5	6	7
A	X	X	Y	X	Y	X	Y	Y

i	0	1	2	3
B	X	X	Y	Y
B_1/PLS(5)		X	Y	Y
B_2		X	Y	

(a)

$LN_1(B_1)$	Y
$LN_2(B_1)$	X, Y
$LN_2(B_1)$ \ $LN_1(B_1)$	X

(b)

Figure 2: Example of MAW $B = XXYY$ for sequence $A = XXYXYXYY$. Table (a) shows the MAW B, its longest proper suffix B_1, and the longest proper prefix B_2 of B_1. B_1 would be examined as potential longest proper suffix at position 5: $PLS_1(5)$. Table (b), shows the the left neighbors of B_1 and B_2, and the difference between the sets. X is found as the only element, and this confirms that B is a MAW of A. We can easily manually verify: $XXYY$ does not occur in A, while its longest proper suffix XYY (position 0) and longest proper prefix XXY (position 5) do.

in the bioinformatics research field). This problem is about finding approximate matches of a given query string in a database of strings. This extends the exact search problem, because no exact matches are needed. Therefore, errors and inconsistencies in the data are tolerated [Yu et al., 2016]. A similar related problem is the string similarity joins problem. This problem is about finding approximately matching string pairs between two string sets. String similarity joins techniques can be adapted to be applied for string similarity search [Jiang et al., 2014].

Strings need to be compared in a pairwise manner. To determine how close a string is to the given query string, a similarity function is used. In [Crawford et al., 2018] the number of common MAWs is used as a similarity measure. Many different types of similarity functions exist. Token-based similarity functions calculate a set of tokens for each string, and then measure similarity based on the overlap (or other metric based on overlap) of the token sets of strings [Yu et al., 2016]. In addition to the overlap metric ($|r \cap s|$), there are: Jaccard ($\frac{|r \cap s|}{|r \cup s|}$), Cosine ($\frac{|r \cap s|}{\sqrt{|r|*|s|}}$) and Dice ($\frac{2|r \cap s|}{|r|+|s|}$). The method used by [Crawford et al., 2018] can be classified as a token-based similarity function. Character-based similarity functions, measure similarity based on the number of different characters in strings [Yu et al., 2016]. An example of such a function is one based on the edit distance, which is the minimal amount of single character edit operations needed to transform one string into another. Hybrid-based similarity functions combine the aforementioned methods [Yu et al., 2016]. They calculate a set of tokens for each string, and then use character-based similarity functions to measure similarity between tokens.

String similarity search algorithms can also differ in the way the results are selected. Threshold-based algorithms select all the strings which have a similarity to the query string above a certain similarity threshold T. Top-k similarity search algorithms select the k strings most similar to the query string. A flow chart of the string similarity search problem is given in Figure 3. In the next two subsections, first, different sequence comparison methods are discussed. After that, the different algorithms for string similarity search are explained.

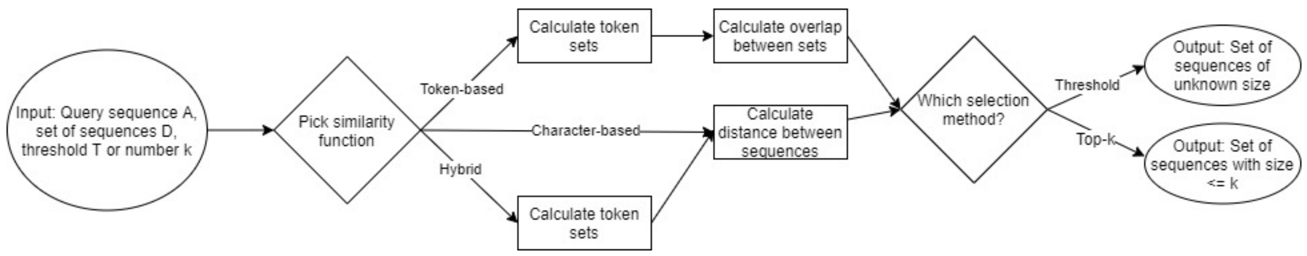

Input: Query sequence A, set of sequences D, threshold T or number k
Pick similarity function
Token-based
Character-based
Hybrid
Calculate token sets
Calculate overlap between sets
Calculate distance between sequences
Calculate token sets
Which selection method?
Threshold
Top-k
Output: Set of sequences of unknown size
Output: Set of sequences with size <= k

Figure 3: Flow chart of the string similarity search problem

2.4 Sequence comparison

As we saw in the previous subsection, an important step of the string similarity search problem is pairwise comparison of a sequence A and a sequence B. In this subsection the different methods for sequence comparison are discussed. The methods can be divided into two categories: alignment-based methods (use a character-based similarity function) and alignment-free methods (use a token-based similarity function).

2.4.1 Alignment-based

For biological research, methods are needed to compare biological sequences. Finding similarities between sequences is important, because it can signify evolutionary relationships. In [Needleman and Wunsch, 1970] the **Needleman-Wunsch** algorithm is presented for comparing two amino acid sequences of proteins. This is done by finding the best alignment of the two sequences. The aim is to find the maximum match, which is the largest number of matched amino acids between the two sequences. This form of alignment is called global alignment, because it tries to align the complete sequences. To better align the sequences the algorithm can introduce gaps in either one of the sequences. This results in three possible cases for an alignment: a match, a mismatch and an introduced gap. Arbitrary scoring systems can be used for these three cases.

Matches and mismatches can be given fixed scores. However, in the context of proteins the scores can be improved by basing them on a substitution matrix. These matrices indicate the likelihood of one amino acid being substituted by another. To perform the alignment, dynamic programming is used, where the sub-problems are the prefixes of the sequences. A matrix Mat of $m + 1 \times n + 1$ is constructed with sequence A of size m on the y-axis, and sequence B of size n on the x-axis. Using dynamic programming it is filled in an iterative manner. Element $Mat[0, 0]$ is initialized with value 0. The value of each other element can come from its left or upper neighbor (insertion of a gap) or from its upper left neighbor (match or mismatch). From these three values the highest value is chosen, and a reference is made to the element it originates from (if there are multiple highest values, multiple references are made). When the matrix is completed, these references can be traced back to form the optimal alignment. In Figure 4, an example of a completed matrix is given for aligning sequences ECDF and CDFG. In Figure 5 the found alignment is shown. The figures are taken from the Needleman-Wunsch algorithm demo by Bert Lievrouw [3]. The alignment that is found in the example has score 1, this is the value of the last element $Mat[m, n]$ in the matrix. The higher the score, the better the alignment is and the more similar the two sequences are. Computing every element of the matrix takes $O(1)$ time, so the time complexity is $O(mn)$. The matrix is $m \times n$ so the space complexity is also $O(mn)$.

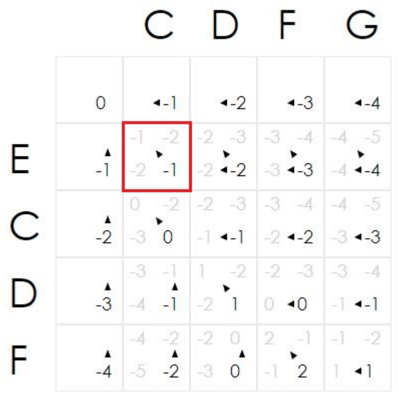


Figure 4: Matrix filled by the Needleman-Wunsch algorithm. The scoring system {match +1, mismatch -1, gap -1} was used. In each element the three possible values are written in grey, and the highest value is written in black. In the red marked element the three possible options are: mismatch (E and C), introducing a gap instead of E, introducing a gap instead of C. Mismatching E and C gives the highest value and is therefore chosen as its value.

In [Smith et al., 1981] the **Smith-Waterman** algorithm is given for finding a pair of segments in two molecular sequences that has the highest similarity.

[3] `https://blievrouw.github.io/needleman-wunsch/`

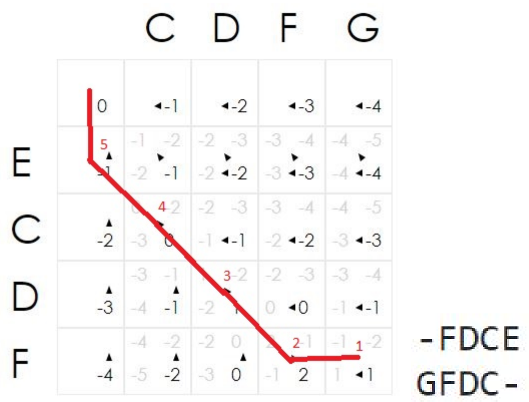


Figure 5: The path is traced back using the arrows from the end of the matrix to the beginning, and the resulting alignment is shown. Step 1 of the trace back is an alignment of a gap with G, step 2 aligns F with F, step 3 aligns D with D, step 4 aligns C with C and step 5 aligns E with a gap.

This algorithm does not look at the similarity between sequences as a whole, but rather at local similarities within the sequences. This type of alignment is therefore called local alignment. The algorithm uses dynamic programming to fill a matrix. The elements of the first row and column of the matrix are initialized as 0. The other elements are given a value similarly to the Needleman-Wunsch algorithm, with the exception that the similarity score cannot be negative. The pair of segments with highest similarity is then found by finding the element in the matrix with the highest value and tracing back until an element with value 0 is encountered. Like the Needleman-Wunsch algorithm the time complexity and space complexity are $O(mn)$.

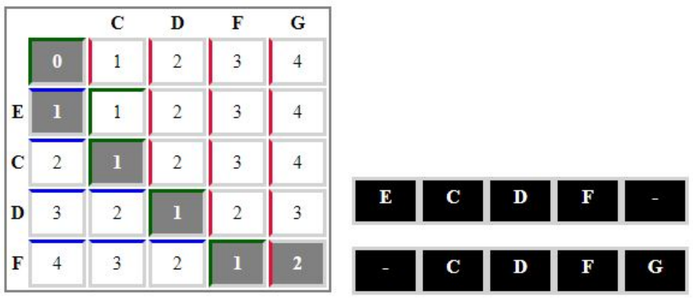
To improve the scoring of gaps in alignments, [Gotoh, 1982] introduces the Gotoh algorithm. This algorithm allows usage of affine gap penalty. This gap penalty function allows assigning different costs to opening a gap and extending a gap. The opening of a gap in an alignment is penalized more heavily than extending a gap. This makes intuitive sense, because a lot of gap openings means that there are less contiguous segments that match, which causes more dissimilarity between segments. Once a gap has been opened however, extending it won't interrupt a contiguous segment. Therefore, extension is less costly, because it impacts the similarity between segments less. The Gotoh algorithm achieves this by modifying the Needleman-Wunsch and Smith-Waterman algorithms. Instead of keeping for every element Mat[i, j] only the best score for an alignment ending in [i, j], two extra scores are kept: Ix (the best scoring alignment ending in [i+1, j] with a gap insertion in A) and Iy (the best scoring alignment ending in [i, j+1] with a gap insertion in B). These best scoring alignments are obtained by opening a new gap, or extending a previously opened gap. [Cameron et al., 2004]

For information retrieval in the context of strings, there is a need for algorithms that can find similar strings to a query string. This is important when searching in a database in which data is possibly corrupted, or for example for automatic spelling correction [Hall and Dowling, 1980]. Most of these algorithms compare strings using the edit distance. The edit distance is the minimal amount of edit operations required to turn string A into string B. There are many different variants of edit distance, but the most commonly used one is the Levenshtein distance [Levenshtein, 1966, Navarro, 2001]. The possible edit operations consist of: deleting a character, inserting a character, or changing/replacing a character [Ukkonen, 1985]. If every edit operation has cost 1, it is called the simple edit distance. When operations have different costs, it is called the general edit distance [Navarro, 2001]. In [Wagner and Fischer, 1974] the **Wagner-Fischer** algorithm is presented which calculates the edit distance using dynamic programming. Similarly to the earlier presented sequence alignment algorithms, the sub-problems are the prefixes of the strings. A matrix Mat of $m+1 \times n+1$ is created with string A of length m on the y-axis, and string B of length n on the x-axis. Each element can get its value from its left neighbor, which indicates an insertion of a character, upper neighbor, which indicates a deletion of a character, or from its upper left neighbor which indicates that the characters already match or that one has to be replaced. Element $Mat[0,0]$ is set to 0. The other elements of the matrix are filled in an iterative manner, leaving traces to the element that the value came from. When the matrix is completed, the traces can be traced back to form the optimal sequence of editing operations. In Figure 6 an example is given of a matrix filled by the Wagner-Fischer algorithm for the sequences ECDF and CDFG, and the found sequence of editing operations. The figures are taken from the Wagner-Fischer algorithm demo by Johann-Mattis List [4]. The edit distance found in the example is 2, this is the value of the last element $Mat[m,n]$ in the matrix. The lower the score, the smaller the distance between the sequences is, and therefore the more similar the two sequences are.

In [Kondrak, 2005] a family of word similarity measures is defined that is based on n-grams. N-grams are substrings of length n. They define two new measures for string comparison: n-gram distance and n-gram similarity. The n-gram distance is a generalization of edit distance, which allows for looking at the sequence in terms of its n-grams instead of only single characters. It is shown that the edit distance is a special case of n-gram distance, namely $n = 1$. The given algorithms use dynamic programming to calculate the measures and have $O(mn)$ time complexity and space complexity.

As can be seen, the sequence alignment and edit distance algorithms are similar. Sequence alignment aims to maximize the similarity score, and the edit distance algorithms aim to minimize the edit distance. The in [Hirschberg, 1975] presented space improved version of the Needleman-Wunsch algorithm, the **Hirschberg algorithm**, can therefore also be applied to all the presented dynamic programming algorithms so far. The Hirschberg algorithm uses the

[4]`http://digling.org/wagnerfischerdemo/`



		C	D	F	G
	0	1	2	3	4
E	1	1	2	3	4
C	2	1	2	3	4
D	3	2	1	2	3
F	4	3	2	1	2

Figure 6: Matrix filled by Wagner-Fischer algorithm. The simple edit distance is calculated (cost 1 for every edit operation). The elements marked grey form the optimal sequence of editing operations. To turn ECDF into CDFG, first the E has to be deleted, C, D and F already match, and G has to inserted. The edit distance between the two strings is 2 (value of element $Mat[m, n]$).

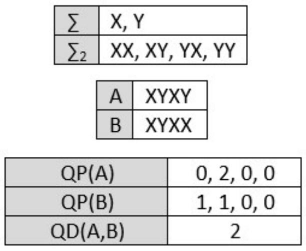
fact that only the current and previous row of the matrix have to be stored to compute the optimal alignment score. The space complexity can therefore be reduced to $O(\min\{m, n\})$. However, since the matrix is not stored completely, the list of alignment/editing steps cannot be obtained.

In [Ukkonen, 1985] two algorithms are presented, which improve upon the Wagner-Fischer algorithm. The algorithms limit the amount of elements of the matrix that have to be evaluated. The matrix elements that have to be evaluated lay in a diagonal band that is dependent on the edit distance s. Their algorithm for calculating the general edit distance, which we will call **Ukkonen ED 1**, has time and space complexity $O(s \cdot \min(m, n))$. Their algorithm for calculating the simple edit distance, which we will call **Ukkonen ED 2**, has time complexity $O(s \cdot \min(m, n))$ and space complexity $O(s \cdot \min(s, m, n))$. In addition, if a threshold t is given for the edit distance, the time complexity is $O(t \cdot \min(m, n))$ and the space complexity is $O(\min(t, m, n))$.

2.4.2 Alignment-free

The methods for sequence comparison discussed so far have been alignment-based. These methods require dynamic programming for comparing the sequences based on its characters, and are therefore quite expensive in terms of time complexity [Ukkonen, 1992]. For this reason, alignment-free methods have been proposed, which do not look at the sequence in terms of its characters, but instead calculate a set of tokens for the sequences, and compare the sequences using these sets.

In [Ukkonen, 1992] the set of q-grams (same as the earlier defined n-grams) is calculated for comparing strings, using an algorithm which we will call **Ukonnen Q-gram**. The comparison is done by calculating the q-gram distance between the strings (note that this is a different q-gram distance than the one described by [Kondrak, 2005]). Let A be a string. Let Σ_q be the set of size v, containing all sequences of length q, from alphabet Σ, in lexicographical order. Let array $QP(A)$ be the q-gram profile of string A. $QP(A)[i]$ is the amount of occurrences of q-gram Σ_q[i] in A. The q-gram distance between string A and B is then $QD(A,B) = |QP(A) - QP(B)|$. The q-gram distance can be calculated using suffix automata in O$(m|\Sigma| + n)$ time, and using $O(m|\Sigma|)$ space. In Figure 7 an example is given of the 2-gram distance (q-gram distance for q = 2) for sequences XYXY and XYXX.



Σ	X, Y
Σ₂	XX, XY, YX, YY

A	XYXY
B	XYXX

QP(A)	0, 2, 0, 0
QP(B)	1, 1, 0, 0
QD(A,B)	2

Figure 7: Example of the q-gram distance between A = XYXY and B = XYXX for q = 2. The bottom table shows the 2-gram profiles and the 2-gram distance is shown to be 2.

In [Crochemore et al., 2016] the sets of MAWs of A and of B are used for sequence comparison of sequences A and B, by calculating the MAW similarity measure. Set $\mathcal{M}_A$ (resp. $\mathcal{M}_B$) represents the MAWs of A (resp. B) as a set of tuples $\langle a, i, j \rangle$, where a is the first character of the MAW, and A[i, j] is the longest proper suffix of the MAW. The idea of their algorithm, **scMAW**, is to construct a suffix array and LCP array for the concatenation ab of the sequences a and b. Using these data structures, sets $\mathcal{M}_A$ and $\mathcal{M}_B$ can be sorted lexicographically, and elements of sets $\mathcal{M}_A$ and $\mathcal{M}_B$ can be compared. To compare sequences A and B the following strategy can be applied to calculate the symmetric difference LW $(\mathcal{M}_A, \mathcal{M}_B)$: Initialize LW $(\mathcal{M}_A, \mathcal{M}_B)$ as 0, and i and j as 0.

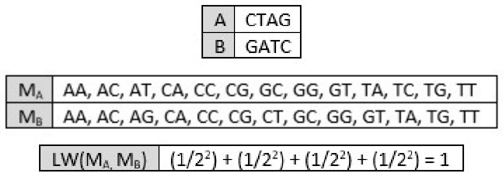
- if $\mathcal{M}_A[i] < \mathcal{M}_B[i]$ increase similarity by $\frac{1}{|\mathcal{M}_A[i]|^2}$ and increment i
- if $\mathcal{M}_A[i] > \mathcal{M}_B[i]$ increase similarity by $\frac{1}{|\mathcal{M}_B[i]|^2}$ and increment j
- if $\mathcal{M}_A[i] = \mathcal{M}_B[i]$ increment i and j

An example of scMAW can be seen in Figure 8. scMAW has a time and space complexity of $\mathcal{O}(m + n)$.

In Table 1 an overview of the sequence comparison algorithms, their similarity measures, and their time and space complexities is given.

2.5 String similarity search algorithms

As we have seen in the previous subsection, various methods exist for comparing sequences. These methods have different time complexities, ranging from linear



A	CTAG
B	GATC

M_A	AA, AC, AT, CA, CC, CG, GC, GG, GT, TA, TC, TG, TT
M_B	AA, AC, AG, CA, CC, CG, CT, GC, GG, GT, TA, TG, TT

$LW(M_A, M_B)$	$(1/2^2) + (1/2^2) + (1/2^2) + (1/2^2) = 1$

Figure 8: Example of the MAW distance between A = CTAG and B = GATC for MAW size = 2.

Year	Algorithm	Similarity measure	Time complexity	Space complexity
1970	Needleman-Wunsch	Global alignment	$O(mn)$	$O(mn)$ or $O(min\{m, n\})$*
1974	Wagner-Fischer	General edit distance	$O(mn)$	$O(mn)$ or $O(min\{m, n\})$*
1981	Smith-Waterman	Local alignment	$O(mn)$	$O(mn)$ or $O(min\{m, n\})$*
1985	Ukkonen ED 1	General edit distance	$O(s * min(m, n))$	$O(s * min(m, n))$ or $O(min(s, m, n))$*
1985	Ukkonen ED 2	Simple edit distance	$O(s * min(s, m, n))$	$O(s * min(m, n))$ or $O(min(s, m, n))$*
1992	Ukkonen Q-gram	Q-gram distance	$O(m\|\Sigma\| + n)$	$O(m\|\Sigma\|)$
2005	Kondrak N-gram 1	N-gram similarity	$O(mn)$	$O(mn)$ or $O(min\{m, n\})$*
2005	Kondrak N-gram 2	N-gram distance	$O(mn)$	$O(mn)$ or $O(min\{m, n\})$*
2016	scMAW	MAW similarity	$O(m + n)$	$O(m + n)$

Table 1: Overview of the shown sequence comparison algorithms.
*If list of alignment/editing steps is not needed. [Hirschberg, 1975]

to $O(mn)$ time. String similarity search algorithms are being performed on increasingly bigger data sets, increasing the need for efficient algorithms. Many different algorithms have been proposed for use in various situations. Different methods are used to improve efficiency of the algorithms. The following three strategies can be distinguished [Fenz et al., 2012, Jiang et al., 2014]:

- Heuristics: Some algorithms use techniques to focus the computation on certain areas of the problem. Although this does not guarantee an optimal solution, it lowers the time needed for sequence comparisons, and the speed of the algorithm can be improved.

- Indexing: Storing the data in main memory using an efficient indexing method designed for the queries, can improve the query time. Over the years various indexes have been proposed for use with the string similarity problem. Index structures are used to divide the datasets in subsets, and during queries only certain subsets are considered [Rachkovskij, 2019]. To do this, most approaches built a search tree structure on the dataset. Index-based algorithms were shown to be outperformed by filter-and-verify algorithms [Jiang et al., 2014, Wandelt et al., 2014, Yu et al., 2016], and therefore we won't go into detail for these algorithms.

- Filter-and-verify (also called the filter-and-refine strategy in [Qin and

Xiao, 2018]): Sequence comparison has to be repeated for each sequence in the dataset. As this step is computationally expensive, to improve scalability it is beneficial to reduce the amount of times it has to be performed. Filter-and-verify algorithms are threshold-based algorithms and use the threshold to reduce the number of expensive computations. The filter-and-verify strategy consists of two steps. During the filtering step, the aim is to obtain a candidate list by filtering out dissimilar sequences. This should be done using a cheaper algorithm. Then, during the verification step, the candidates are verified using the expensive sequence comparison algorithm.

In this subsection we will look at various algorithms that use heuristic and filter-and-verify strategies, and identify the best performing ones.

2.5.1 Heuristic algorithms

In bioinformatics, research was done into improving upon the alignment-based algorithms. This resulted in heuristic algorithms for local alignment like the Wilbur-Lipman method, FASTA and BLAST. These algorithms give approximations of local alignments and do not guarantee an optimal solution, however the speed is improved.

The Wilbur-Lipman method [Wilbur and Lipman, 1983] calculates offsets between matching characters of two sequences. For each of these offsets a similarity score is calculated. The score is increased for matches and decreased for mismatches. The offsets with the highest scores represent local regions of similarity between the two sequences. The final similarity score and alignment are then calculated by looking at characters, or blocks of characters within a certain distance from each of these local regions of similarity, and calculating similarity using the Smith-Waterman algorithm.

FASTA [Lipman and Pearson, 1985] improves upon the Wilbur-Lipman method. It locates the beginning and end positions in both sequences of the five local regions of highest similarity found using the Wilbur-Lipman method. It then takes advantages of the fact that for amino acids, replacements occur more often than insertions or deletions. It gives a new score for the five highest local regions using a substitution matrix (which contains scores based on the likelihood of alignments and replacements of amino acids). Examples of amino acid replaceability matrices are PAM and BLOSUM. The score of the best scoring local region is then used as initial similarity score. This initial score is calculated for all the other sequences in the database. The database is then ranked using this initial score. For the sequences with the highest ranking similarities, the optimized scores are calculated using the Smith-Waterman algorithm.

The **BLAST** (basic local alignment search tool) algorithm [Altschul et al., 1990] improves upon the FASTA algorithm, and is an order of magnitude faster. It uses the maximal segment pair (MSP) similarity measure. The idea is that good alignments often contain these MSPs. A segment is a subsequence of any length. The similarity score of two aligned segments (one from each sequence) with the same length is the sum of similarity values for each pair of aligned characters. The MSP is the highest scoring pair of identical length segments. BLAST aims to find the locally maximal segment pairs: segment pairs that cannot be improved by extending or shortening both segments. It then uses these segment pairs to create the alignment.

The BLAST algorithm uses a technique which minimizes the time spent on regions with a low chance of having a similarity score at least S, and therefore allows for rapid approximation of the MSP score. It does this by compiling a list of high scoring words with score at least T, and using these words to seed alignments. The BLAST algorithm is as following:

1. Compile list of high scoring words (subsequences): The query sequence is broken up in its q-grams. Then, using an amino acid replaceability matrix, words of length q are found that score higher than some threshold T, and are stored in an efficient index.

2. Scan the database for hits of these words, using the index.

3. Extend the hits found to find the locally maximal segment pairs. This is done by extending the segment in both the left and the right direction, until the score falls a certain distance X below the best score found.

4. List all locally maximal segment pairs with a score higher than S, the so called high-scoring segment pairs (HSPs).

In [Altschul et al., 1997] two new versions of BLAST are introduced, Gapped BLAST and PSI-BLAST. The improvements are:

- Gapped BLAST uses a two hit method, which requires there to be two non-overlapping pairs on the same diagonal in the matrix, that are within a certain preset distance, the window size A, of each other, before extension is performed. Since extensions are an expensive operation, this improves upon the BLAST execution time.

- Gapped BLAST can create gapped alignments. When a locally maximal segment pair is found with a score that exceeds a certain threshold, a gapped extension is performed.

- PSI-BLAST allows BLAST searches to be iterated. It uses a position specific score matrix generated in the current round, in the next round.

One of the strengths of the BLAST algorithm, is that it lends itself to statistical analysis. In [Karlin and Altschul, 1990], methods are given which allow calculation of the probability of finding a segment pair with a score greater or equal to the score S, when comparing two random sequences. This probability can be used to calculate the statistical significance of the encountered HSPs.

2.5.2 Filter-and-verify algorithms

In [Jiang et al., 2014] a survey is done of threshold-based string similarity joins algorithms. Various algorithms are described, using both token-based and character-based similarity functions. Filtering is done by calculating a set of signatures for a string (many different signatures exists such as n-grams, partitions, length of the string, neighborhood of the string, etc), and then using one (or multiple) of the following filtering techniques:

- Count filtering: If two strings are similar, they share at least T common signatures. T is based on the similarity threshold.
- Length filtering: Length difference cannot be larger than a certain length threshold, which is based on the similarity threshold.
- Prefix filtering: Calculate token sets of the strings. Order the tokens, and select the first p tokens as the signature (this is denoted by the prefix). p is based on the similarity threshold. Then, calculate the overlap between the prefixes. If there is no overlap, the sequence can be filtered.
- Partition-based filtering: Given a threshold τ, which is based on the similarity threshold, partition a string A in $\tau + 1$ segments. If a string B is similar to string A, then B has to contain a sub string which is equal to one of the segments in A.

In [Wandelt et al., 2014] an overview of the state of the art in string similarity algorithms is made, by conducting a benchmark in the form of a competition. Multiple teams submitted their programs, and these were evaluated and compared to each other and also to existing methods. Two data sets were used for the evaluations, a set of reads from human genomes (uniform length strings with small alphabet size) and a set of geographical names (non-uniform length strings with large alphabet size). Two methods, Pass-Join and WallBreaker, were found to outperform all other submissions and the existing methods for both string similarity joins and string similarity search. Pass-Join outperformed WallBreaker in all evaluations, except for the string similarity search on the human genome data set. The Pass-Join method utilizes a partition-based filtering method. The WallBreaker method, is not a filtering algorithm, but it splits the query in subqueries with smaller distance thresholds.

In [Jiang et al., 2014] the Pass-Join method is identified as the suggested method for string similarity joins using edit distance, on sequences of medium and long lengths. In the survey performed in [Yu et al., 2016] many similarity search and similarity join algorithms are explained. Again, the Pass-Join method is identified as being a flexible algorithm with high pruning power, and no disadvantages compared to the other algorithms.

The **Pass-Join** algorithm [Li et al., 2011], uses a partition-based filtering technique. Applied for the string similarity search problem, the method would be as following:

1. For a given edit distance threshold τ, partition the query string A into $\tau + 1$ disjoint segments. Segments have size $\left\lfloor \frac{|A|}{\tau+1} \right\rfloor$ or $\left\lceil \frac{|A|}{\tau+1} \right\rceil$ (maximum difference between segment lengths is 1).

2. Make inverted indices $\mathcal{L}_l^i$ for strings in the database. $\mathcal{L}_l^i[w]$ gives the set of strings of length l, for which the i_{th} segments are w.

3. Search inverted index $\mathcal{L}_l^i$ using a set of substrings of A: $\mathcal{W}\left(A, \mathcal{L}_l^i\right)$. The strings found are candidates.

4. Verify candidates using edit distance calculations.

As can be seen in step 3, to improve the speed of the algorithm, instead of using all the substrings of A, a subset $\mathcal{W}\left(A, \mathcal{L}_l^i\right)$ is generated. To generate this subset, multi-match-aware substring selection is used. This makes use of the fact that some strings may have multiple substrings which match with segments in $\mathcal{L}_l^i$. Some of these substrings can be discarded.

For improvement of step 4, the verification, an extension-based method is used. For each matching string found in step 3 for $\mathcal{L}_l^i(w)$, an edit distance threshold $\tau_l = i - 1$ is calculated for the part of the string left to the segment w, and edit distance threshold $\tau_r = \tau + 1 - i$ for the part to the right of the segment w. Then, the edit distance is calculated for the left part, and if this edit distance is higher than τ_l, the verification is terminated. Otherwise, the edit distance is calculated for the right part using τ_r as a threshold. For the edit distance calculations, a length-aware method is used, which limits the elements of the matrix that have to be computed, by looking at the length difference between strings A and B. Lastly, early termination is used: if all elements of the current column exceed the threshold, the remainder of the elements will also exceed it. Therefore, the edit distance calculation can be terminated. This can be applied since we only need to know the edit distance if the entry is a query result.

In the survey performed in [Rachkovskij, 2019] Pass-Join is identified as the best performing filtering method for long strings. In addition, they suggest that PIVOTALSEARCH combined with the pigeonring principle, may possibly be the fastest available algorithm. Although, this seems to be speculative: no comparison is performed in the survey, and therefore there is a need to compare Pass-Join and PIVOTALSEARCH.

The pivotal prefix-based similarity search (**PIVOTALSEARCH**) algorithm [Deng et al., 2014], orders the sequences' q-grams. Then, for a given edit distance threshold τ, it takes the prefix pre() of size $q\tau + 1$, and then takes a set of $\tau + 1$ q-grams, called the pivotal prefix, piv(). The pivotal prefix filter is then based on two lemmas:

Lemma 3. • *If strings A and B are similar, then $piv(A) \cap pre(B) \neq \phi$ or $pre(A) \cap piv(B) \neq \phi$. ($\phi$ is the empty set).*

Lemma 4. • *If strings A and B are similar, and last(pre()) is the last q-gram in the pre() set, we have:*

- *if* $last(pre(A)) > last(pre(B))$, $piv(B) \cap pre(A) \neq \phi$
- *if* $last(pre(A)) \leq last(pre(B))$, $piv(A) \cap pre(B) \neq \phi$

Using these lemmas the algorithm checks for query string A and string B in the dataset, if $piv(B) \cap pre(A) \neq \phi$, and it performs the second check $last(pre(A)) > last(pre(B))$ if its true. Otherwise, it checks if $piv(A) \cap pre(B) \neq \phi$ and performs the second check $last(pre(A)) \leq last(pre(B))$ if its true. Only if these checks come out as true, string B has to be verified. PIVOTALSEARCH uses a special technique to select a high-quality pivotal prefix. For each q-gram in the pivotal prefix, the more strings it appears in, the more computations have to be performed. Therefore, each q-gram is given a weight based on the amount of strings it appears in, and the pivotal prefix is selected that minimizes the total amount of weight. To improve the verification step, one addition filter, the alignment filter, is applied. This filter is based on the fact that the pivotal prefix filter only effectively filters out dissimilar strings with non-consecutive differences to the query. Strings with consecutive differences can be part of the candidate set. Using the alignment filter, these can be filtered out. Only after the alignment filter, the verification is performed.

The Pass-Join and PIVOTALSEARCH algorithms are both based on the pigeonhole principle [Li et al., 2011, Deng et al., 2014]. For filtering, the pigeonhole principle states: *If no more than n items are put into m boxes, then at least one box must contain no more than n/m items [Qin and Xiao, 2018].* In the context of the Pass-Join and PIVOTALSEARCH algorithms we have a threshold τ and $\tau + 1$ boxes (segments/pivotal prefixes). Applying the pigeonhole principle means that there must be at least one box (one segment/pivotal prefix) for which the edit distance is 0. This only constrains items in a single box, and it is therefore a rather weak condition. The pigeonring principe [Qin and Xiao, 2018], creates a stronger condition, by putting a constraint on multiple boxes. Therefore it places the boxes in a ring. The pigeonring principle states: *If no more than n items are put into m boxes, then there exists at least one box such that, for every* $l \in [1 \ldots m]$, *starting from this box going clockwise,* l *consecutive boxes contain no more than* $l \cdot n/m$ *items.* Such a chain of boxes is called a prefix-viable chain. The pigeonring principle was shown to improve the filtering power, and query time for the PIVOTALSEARCH algorithm. If the chain length is set to the size of the pivotal prefix list, this pigeonring filter is equal to the alignment filter of [Deng et al., 2014].

2.6 Similarity search for MIR systems

In this subsection, we will take a look at string similarity search in the context of MIR systems. First, we will take a closer look at the data that is outputted by an OMR system, Aruspix, and how to prepare this data for string similarity search. Lastly, we will look at existing/proposed MIR systems for similar data, and the techniques they use. In addition, problems in these systems are identified.

2.6.1 Optical music recognition output data

A lot of the large libraries of music scores consist of page-images. In order to create data that is suitable for use in the string similarity search algorithms, an OMR tool has to be used. An example of an OMR tool is Aruspix [5]. This tool was developed to encode early music prints. The early music, stemming from the 16th and 17th century, was printed using movable typefaces. This printing technique made the printing process commercially sustainable. Therefore, there was a dramatic increase in printed scores in this time. During this old printing process, errors were introduced [Pugin and Crawford, 2013]. In [Pugin and Crawford, 2013] Aruspix was evaluated on the early music online (EMO) collection [6]. This collection, consisting of about 35,000 pages from more than 320 volumes of 16th century music, is a valuable source for musicologists, musicians and historians [Pugin and Crawford, 2013]. A characteristic of this music is that different voices of a piece are distributed amongst multiple books, the sheet music is therefore polyphonic. A page-image contains a variable amount of notes. Aruspix creates files in MEI format, which can be seen as sequences of notes. It was shown that 80% of the collection could be processed using the system, with a recall rate between 85% and 100% for three quarters of the pages [Pugin and Crawford, 2013]. An example of data from the EMO collection and its Aruspix output is given in Figure 9.

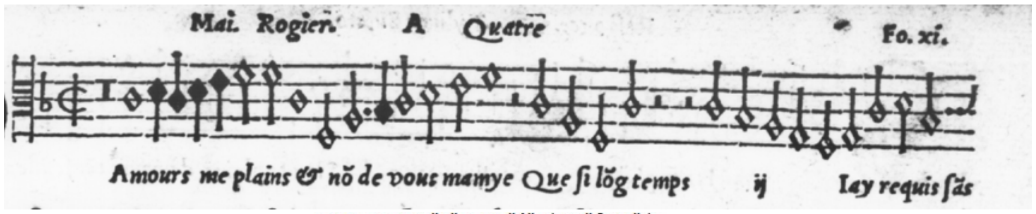
In [Crawford et al., 2018] the sequences are encoded as linear sequences of diatonic intervals. This is done to deal with certain errors and noise introduced during the OMR process. Sometimes an error is made with the recognition of a clef or key-signature, which has an effect on all the notes of a staff (e.g. pitching them all a third to low) [Crawford et al., 2018]. Encoding the notes by looking at the intervals between notes, greatly reduces this problem. For an example, see Figure 10. As can be seen the intervals between notes are calculated, and these are encoded using letters. Lower-case letters indicate a negative interval, and upper-case letters indicate a positive interval. In addition, '-' is used when the note is repeated. The size of the alphabet Σ is therefore 53. The resulting dataset consists of 31721 sequences, with minimum length = 9, maximum length = 506, average length = 137.308 and standard deviation $\approx$ 59.5287.

2.6.2 Music information retrieval systems

In [Typke et al., 2005] a survey is performed of MIR systems for content-based music searching. They distinguish between two types of MIR systems: for audio and for symbolic music notation. Hybrid systems combining the two also exist: these first convert the audio signal into a symbolic notation. As we have seen in the previous subsection, we are dealing with symbolic music data, and therefore we will focus on MIR systems for this type of data.

In [Kilian and Hoos, 2004], an algorithm based on gapped BLAST is proposed for approximate matching of symbolic music data, MusicBLAST. The

[5] `www.aruspix.net`
[6] `www.earlymusiconline.org`


Mai. Rogier. A Quatre
Fo. xi.
Amours me plains & nõ de vous mamye Que ſi lõg temps ij Iay requis ſãs

```
<note pname="a" oct="4" dur="fusa"/>
<clef line="3" shape="C"/>
<accid accid="f" ploc="b" oloc="3"/>
<mensur sign="C" slash="1"/>
<rest dur="brevis" ploc="c" oloc="4"/>
<note pname="c" oct="4" dur="semibrevis"/>
<note pname="d" oct="4" dur="semiminima"/>
<note pname="c" oct="4" dur="semiminima"/>
<note pname="d" oct="4" dur="semiminima"/>
<note pname="e" oct="4" dur="semiminima"/>
<note pname="f" oct="4" dur="minima"/>
<note pname="f" oct="4" dur="minima"/>
<note pname="c" oct="4" dur="semibrevis"/>
<note pname="f" oct="3" dur="minima"/>
<note pname="a" oct="3" dur="minima"/>
<dot ploc="b" oloc="3"/>
<note pname="b" oct="3" dur="semiminima"/>
<note pname="c" oct="4" dur="minima"/>
<note pname="d" oct="4" dur="minima"/>
<note pname="e" oct="4" dur="minima"/>
<note pname="f" oct="4" dur="semibrevis"/>
<rest dur="minima" ploc="c" oloc="4"/>
<note pname="c" oct="4" dur="minima"/>
<note pname="a" oct="3" dur="minima"/>
<note pname="f" oct="3" dur="minima"/>
<note pname="c" oct="4" dur="minima"/>
<rest dur="minima" ploc="c" oloc="4"/>
<rest dur="minima" ploc="c" oloc="4"/>
<note pname="c" oct="4" dur="minima"/>
<note pname="b" oct="3" dur="minima"/>
<note pname="a" oct="3" dur="minima"/>
<note pname="g" oct="3" dur="minima"/>
<note pname="f" oct="3" dur="minima"/>
<note pname="g" oct="3" dur="minima"/>
<note pname="c" oct="4" dur="minima"/>
<note pname="d" oct="4" dur="minima"/>
<note pname="b" oct="3" dur="minima"/>
<custos pname="c" oct="4"/>
```

Figure 9: An example of Aruspix input (snippet of "D'amours me plains" page-image) and output (snippet of simplified MEI file).

Diatonic pitch sequence	E4	D4	F4	E4	D4	D4	C4
Diatonic interval sequence	-1	+2	-1	-1	0	-1	
Encoded diatonic interval sequence	a	B	a	a	-	a	

Figure 10: An example of conversion from MEI diatonic pitch sequence, to encoded diatonic interval sequence.

MusicBLAST algorithm uses a substitution matrix based on musical data. In their implemented system they use the absolute pitch distance between two notes (intervals smaller than three semitones have a positive cost, and intervals larger than three semitones have a negative cost), but they suggest other features of notes or chords can be used (e.g. rhythmic features like duration and inter-onset interval). The note with the highest similarity score is used as starting point for the bidirectional extension. They evaluated MusicBLAST on sets of MIDI files, and this showed promising retrieval and efficiency results. However, the size of their dataset is not mentioned, and they mention the need for more detailed evaluation of their algorithm.

The large amount of sheet music page-images that are being converted to a machine readable format by OMR, can be compared to the situation for books and other text documents [Hankinson et al., 2012]. Digitization resulted in a lot of page-images from scanned books and text documents. Optical character recognition (OCR) was developed in order to recognize the characters in the text documents and encode them in a machine readable format. One way to continue after the page-images have been encoded, is to store the encoded documents, and discard the images. This results in a database of encoded documents. Another option is to keep both the encoded document and the matching page-image [Hankinson et al., 2012]. This way page-images can be retrieved using the encoded documents, creating a searchable digital library of page-images. This kind of information retrieval system is called a document image retrieval system. It is even possible to link page-image coordinates to the encoded word, and thereby retrieve specific locations in the page-images. Since the OCR and OMR tools do not give perfect encodings, it makes sense to use the data in this way. The OCR and OMR output is good enough to make searching possible, but in the end the user still has access to the real, human readable data contained in the image itself. An example of a document image retrieval system is the Google Books digital library.

In [Hankinson et al., 2012] the design of a music document image retrieval (MDIR) system for the Liber Usualis is described. The Liber Usualis is an important source for chant melodies, consisting of 2340 pages. This system was made as part of the SIMSSA [7] (Single interface for music score searching and analysis) project. They've made a scanned version of this book searchable based on its content. Queries can retrieve page-images and locations of the query on the page-image. The book was processed using OMR systems, and the MEI output is indexed using the following features: n-grams on pitches (note sequences), interval sequences, semitones, contours (direction of the intervals: up, down or repeat), and neumes. In addition, the text on the page-images is indexed. N-grams varying from $n = 2$ to $n = 10$ are used. N-grams are stored in the index along with information about its location on the page-image. Their system allows users to search the Liber Usualis pages, by inserting queries of pitch sequences or any of the other aforementioned features. A restriction of this system is that the maximum length of the search queries is 10 (as stated on

[7] `https://simssa.ca/`

[8]). The authors recommend further development of more complex query and analysis systems.

For searching approximate matches in a database of EMO page-images scanned with OMR, [Crawford et al., 2018] suggests using MAWs as tokens for token-based similarity search. They have implemented this approach in their MDIR system F-Tempo [9]. For every page, the list of MAWs is generated using the algorithm proposed in [Barton et al., 2014]. Approximate matching is done by calculating the Jaccard distance between the sets of MAWs. The algorithm (which we will call SSMAW) is evaluated by comparing it to a similarity search algorithm that uses n-grams as tokens. Three different retrieval tasks are performed: Finding duplicate images within the collection (named the 'dupl' task), finding pages containing substantially the same music as in a query page (named the 'same' task), and identifying pages which have non-identical but related or closely relevant music content (such as in different sections or voice parts than the query) (named the 'relv' task) [Crawford et al., 2018]. For each of these tasks a ground truth has been established for a number of page-images: 49 pages for dupl, 430 pages for same, and 128 for relv. Different MAW and n-gram lengths (3-10) are evaluated. Amongst these, lengths between 4 and 8 characters were found to perform best. The number of MAWs that is being generated becomes smaller as the MAW length increases, which causes there to not be enough data at the higher lengths. Therefore, a mixed length database of (4-8 characters) is used, which provides enough data for matching. The retrieval performance between MAWs and n-grams is shown to be similar, however, the use of MAWs comes with a speed improvement and a smaller index size. The authors recommend comparing the SSMAW algorithm with BLAST. Also, they mention the need to test the scalability of the algorithm using bigger datasets.

[8]`https://ddmal.music.mcgill.ca/research/omr/Search_the_Liber_Usualis/`
[9]`https://f-tempo.org/`

3 Methods

In this research, the usefulness of MAWs for similarity search on page-images of early music, will be gauged using an experimental approach. In the literature study, various similarity measures and algorithms for string similarity search have been presented. We have seen five main similarity measures: edit distance, global alignment, local alignment, MAW distance/similarity and MSP score. For each of these similarity measures an algorithm will be implemented:

- Wagner-Fischer for edit distance
- Needleman-Wunsch for global alignment
- Smith-Waterman for local alignment
- SSMAW for MAW distance
- BLAST for MSP score

In addition, we have seen that three different categories of string similarity search algorithms can be distinguished: heuristic alignment, index-based and filter-and-verify. Based on the findings in the literature study, three algorithms were identified as the best performing in terms of speed, and will be compared to the SSMAW algorithm:

- BLAST
- PIVOTALSEARCH using the pigeonring principle
- Pass-Join using the pigeonring principle

Between Pass-Join and PIVOTALSEARCH, which are both edit distance algorithms, it is not clear yet which one is the faster algorithm. Therefore, both will be implemented and a comparison between them will be performed.

Altogether, seven algorithms will be implented: Wagner-Fischer, Needleman-Wunsch, Smith-Waterman, SSMAW, BLAST, PIVOTALSEARCH and Pass-Join. The algorithms will be implemented in C++. The Strategy design pattern will be used to create an interface for the algorithms. Tests will be performed with all algorithms to gauge retrieval performance, interpretability and scalability.

3.1 Research question, hypotheses and goals

The research question is: "Are MAWs useful for efficient similarity search in the context of early music?" Specifically we will be researching if the SSMAW algorithm improves upon the other algorithms for similarity search in the context of early music. To answer the research question the following hypotheses will be tested:

- Hypothesis 1: "The SSMAW algorithm scores higher than the other algorithms in retrieval performance for the 'dupl' task (finding duplicate images within the collection)".
- Hypothesis 2: "The SSMAW algorithm scores higher than the other algorithms in retrieval performance for the 'same' task (finding pages containing substantially the same music as in a query page)".
- Hypothesis 3: "The SSMAW algorithm scores higher than the other algorithms in retrieval performance for the 'relv' task (identifying pages which have non-identical but related or closely relevant music content)".
- Hypothesis 4: "The results of the SSMAW algorithm are better interpretable than that of the other algorithms".
- Hypothesis 5: "The SSMAW algorithm shows better scalability than the other algorithms".

If at least one of the hypotheses 1-5 is true, then the SSMAW algorithm will be labelled as useful.

In addition to answering the research question, the goals of the research are the following:

- Create a library of optimized versions of the algorithms based on findings in the literature study and make an interface for them.
- Provide source code of the implementations so that future work can benefit from it.
- Empirically obtain parameters for the algorithms in the context of early music.
- Create a decision tree, which helps decide which algorithm to use in which situation.

The answer to the research question and the decision tree can help other researchers with their choice of similarity search algorithms in the context of early music. The source code and obtained parameters will aid researchers with the design/improvement of MIR systems. In addition, the interface makes the algorithms easily applicable in other research contexts, which allows for expanding its use beyond the borders of the early music and MIR context.

3.2 Hardware

The experiments will be performed using a Intel Core i5-8250u CPU @ 1.80GHz (up to 3.40GHz), 8GB of RAM, and Intel HD 620 integrated GPU.

3.3 Data

The EMO collection used by [Crawford et al., 2018] will be used for the experiments. This collection only contains 31,721 pages, which is insufficient for the scalability tests. Hence for the scalability tests, 10 more datasets will be created, by shuffling each sequence a number of times. The original dataset will be called EMO. The new datasets will be called $EMO1$ containing 317,721 pages, 10 times the data of EMO, $EMOn$ for $n \in [2 : 10]$ will contain n times the pages of $EMO1$. The size of the datasets will therefore be ranging from EMO = 31,721 to $EMO10$ = 3,172,100. Finally, a dataset $EMO11$ will be created, which contains 7,613,040 pages. This number was chosen since it surpasses the 1 billion (10^9) note mark, which makes for an interesting milestone.

3.4 Retrieval performance

For retrieval performance, the ground truth for dataset EMO must be established. The ground truth based on expert knowledge found through the work of [Crawford et al., 2018] will be used. This ground truth contains results for 49 dupl queries, 430 same queries and 128 relv queries. The implemented algorithms will be tested against this ground truth. Like in [Crawford et al., 2018] the mean rank of the known relevant page-images will be calculated. The lower this mean rank is, the higher an algorithm scores in terms of retrieval performance. For each algorithm different parameters will be tested. The parameters will be tuned using the ground truth. Empirically the best performing configuration will be found.

3.5 Measuring interpretability

Each of the three tasks (dupl, same and relv) will be performed using the different algorithms. This way for every similarity measure the distribution of the scores for each of the tasks will be obtained. The histograms of the scores for each task will be combined in a single figure. In addition, normal distributions will be fitted and combined in a single figure. In terms of interpretability, in the optimal situation, the histograms and normal distributions for the different tasks are easily distinguishable in these figures. The plots would show clear upper and lower bounds for each of the tasks. From these bounds, score thresholds can then be obtained for similar sequences. Observations about the interpretability will be made based on the separation between the score distributions, and the possibility to infer a score threshold.

3.6 Testing scalability

The implemented algorithms will be tested on data sets $\{EMO1...EMO10\}$. The best parameters obtained during the retrieval performance tests will be used. A number of queries will be performed. The time and space requirements, in milliseconds and megabytes respectively, will be measured, and the average query time and indexing time will be calculated. The results will be plotted in graphs.

4 Implementation

In this section, details of the implementations are given. In section 4.1 the implementations for the dynamic programming algorithms: Wagner-Fischer, Needleman-Wunsch and Smith-Waterman, are described. Then in sections 4.2-4.5, the implementations of SSMAW, BLAST, PIVOTALSEARCH and Pass-Join are described. Design choices and deviations from the literature are given and motivated. Lastly, in section 4.6, the interface for the algorithms is elaborated. Combining this section and the cited literature, allows the reader to reproduce and use our implementations. The source code of the implementation is also available to be downloaded from: `https://github.com/Jelmul/Music-Similarity-Search`. In this section, parameters are given in italics and datastructures are given in bold.

4.1 Dynamic programming algorithms

Our implementations of the dynamic programming algorithms simply repeatedly apply the algorithm to every database entry, in order to obtain all scores, and then sort the entries based on their scores.

To optimize the **Wagner-Fischer** algorithm the modifications by the Hirschberg algorithm were used to decrease the space complexity.

Our implementation of the **Needleman-Wunsch** and **Smith-Waterman** algorithms, uses the Gotoh algorithm to allow for affine gap penalties. Again, the modifications by the Hirschberg algorithm were used to decrease their space complexities. For both algorithms as a parameter a *substitutionMatrix* is taken, to allow for different scoring schemes. Also a gap opening parameter d and gap extension parameter e is taken, to allow for different gap costs.

The list of entries is sorted in ascending order based on their edit distances, and is returned. If a top-k result is required, the list is first shortened according to the k parameter.

4.2 SSMAW

Our implementation of the SSMAW algorithm takes two parameters:

- *min*: The minimum MAW size to be found.
- *max*: The maximum MAW size to be found.

The implementation of the SSMAW algorithm consists of three parts:

1. Indexing: For every database entry, the suffix array and longest common prefix array are constructed. These arrays define a set of proper factors of the database entry. The set of left neighbours of these proper factors and the set of left neighbours of their longest proper prefixes are calculated by making a top down and bottom up pass over the arrays. These neighbours come from an alphabet Σ of size 53. Therefore, the neighbour lists are implemented as bitsets of size 53. Bitsets allow for use of the bitwise

XOR operator, which can be used to calculate the difference between the neigboursets efficiently. This difference yields the set of MAWs. Using the min and max parameters, the desired MAWs are selected, and an index is built by inserting them in a Trie datastructure **MAWsTrie**. The Trie takes the MAW and the index of the entry in the database, and stores the entries in a list in its leaf nodes. This way the Trie allows us to obtain a list of all the entries that contain the MAW. In addition, the number of MAWs for every entry is saved in an array **MAWcounts**. The implementation of the indexing part is based on [Barton et al., 2014].

2. Searching: The MAWs of the query sequence are calculated as above. Every MAW is used as a key to search in the **MAWsTrie** index, and for every entry which contains the MAW, a score is incremented in a score array. Using this score array, and the **MAWcounts** array, the Jaccard distance can be calculated for every entry. The implementation of the searching part is based on [Crawford et al., 2018]. In [Crochemore et al., 2016] an alternative sequence comparison algorithm is given. However, in order to accord to [Crawford et al., 2018], we have chosen not to implement this algorithm.

3. Sorting: The list of entries is sorted in ascending order based on their MAW distances, and is returned. If a top-k result is required, the list is first shortened according to the k parameter.

4.3 BLAST

Our implementation of the BLAST algorithm takes five parameters:

- T: A threshold for high scoring words. Words with score $>$ T will be considered high scoring words.
- A: The window size in which two hits are to be found, before extension is initialized.
- X: The drop off score.
- *subtitutionMatrix*: This matrix is used to score substitutions of notes.
- q: The q-gram size.

For the BLAST algorithm, instead of building an index on the database, an index is built for every query. Therefore, the implementation of the BLAST algorithm only has the searching part. Searching in the BLAST algorithm consists of five steps:

1. Generating an index of high scoring words: First, at the initialization of the BLAST algorithm, the list of q-grams of a query is generated. Then, the list of of all words of length q is generated, that have a score higher than T when compared to one of the q-grams using the *substitutionMatrix*.

Q-grams scoring higher then T are high scoring words (HSW). HSWs are a tuple (word score, position in the query sequence). An index is built by inserting them in a Map data structure, **indexHSW**, with the word as key and a list of HSWs as value.

2. Scanning the database for hits: Using every q-gram of the database entries as a key, the index is searched. If they are found in the index, a hit is recorded. An array is kept which records, for every diagonal of the dynamic programming matrix, the most recent hit found. If two hits are found on the same diagonal and within distance A of each other, an ungapped extension is started of the most recent hit. This implementation is based on [Altschul et al., 1997].

3. Ungapped extension: The selected hits in step 2 are extended to the left and the right, using ungapped alignment. Hits are extended for as long as the score is greater or equal to the best score found so far minus the drop off score X. This way, for every hit a high-scoring segment pair (HSP) is obtained. This implementation is based on [Altschul et al., 1990].

4. Scoring: For every HSP found, the highest score found so far is updated. After all hits for an entry have been extended, the highest score found is returned as its MSP score.

5. Sorting: The list of entries is sorted in descending order based on their MSP scores, and is returned. If a top-k result is required, the list is first shortened according to the k parameter.

4.4 PIVOTALSEARCH

Our implementation of the PIVOTALSEARCH algorithm takes three parameters:

- q: The q-gram size.
- *threshold*: The threshold for the edit distance. Query results have to have an edit distance $\leq$ the threshold.
- *chainLength*: The length of the pigeonring chain.

The implementation of the PIVOTALSEARCH algorithm is largely based on [Deng et al., 2014]. It consists of three parts:

1. Indexing: For PIVOTALSEARCH the database needs to be indexed. First, the database is sorted on length. Then, we iterate over the database and for every entry the q-grams are generated. For every unique q-gram string a frequency is kept, using a Map datastructure, **qgramFrequency**, with as key the q-gram string and as value the frequency.

   We then iterate over the database again. For every entry, the list of q-grams, which are tuples (q-gram string, q-gram position in entry, q-gram frequency), is generated and it is sorted on frequency. We then generate its prefix q-grams, which are the first (q * *threshold* + 1) q-grams, and insert them in a double Map datastructure, **indexPrefixes**, with the length of the entry as the first key, the q-gram as the second key and a list of tuples (entry index, q-gram position in entry) as value. In addition, an array **lastPrefixesFrequency** is kept with the frequency of the last prefix for every entry.

We then sort the prefix q-grams by their positions and generate the pivotal q-grams, which we do by taking the first (*threshold* + 1) non-overlapping prefix q-grams. In [Deng et al., 2014] an optimal selection of pivotal q-grams is defined, which finds the set of pivotal q-grams with minimal summed frequency. However, we found empirically that the gain in speed didn't weigh up against the overhead for finding this set. Therefore, this optimal selection method was not used. The pivotal q-grams of every entry are saved in an array, **pivotals**. The pivotal q-grams are also inserted in a double Map datastructure, **indexPivotals**, with the length of the entry as the first key, the q-gram as the second key and a list of tuples (entry index, q-gram position in entry) as value.

2. Searching: The PIVOTALSEARCH algorithm searching part consists of two steps:

    - Filter: For the query sequence, the prefix and pivotal q-grams are generated as during the indexing part. To filter the entries the pivotal prefix filter is applied, which consists of multiple filters:
        - First, a length filter is applied. To do this, a minimum and maximum length are calculated using the *threshold*. Entries that differ more than *threshold* in length need not be considered, as the edit distance will be greater than *threshold*, and can therefore be filtered. These minimum and maximum lengths are used to iterate over **indexPivotals** and **indexPrefixes**.
        - The prefix q-grams are used to search in **indexPivotals**. The found entries are further filtered by checking if the frequency of the last prefix of the query is greater than the frequency of the last prefix of the entry. Then, the pivotal q-grams are used to search in **indexPrefixes**. The found entries are further filtered by checking if the frequency of the last prefix of the query is lesser than or equal to the frequency of the last prefix of the entry.
        - Entries are furthered filtered by checking if the absolute distance between the positions of matched q-grams is within the threshold. The entries that remain, pass the pigeonhole principle, and are candidates for the next filtering step.
        - Lastly, a final filtering step based on the pigeonring principle is applied on the remaining entries. This replaces the final alignment filter used in [Deng et al., 2014]. This filtering step tries to

find a prefix-viable chain, starting with the q-gram found, using the pigeonhole principle.

- Verify: For the selected entries, the edit distance is calculated based on the dynamic programming algorithm presented in [Wagner and Fischer, 1974], to verify if they are query results. To improve the time complexity of this edit distance calculation, a length-aware edit distance algorithm, with early termination based on [Li et al., 2011] is implemented. The time and space complexity are further improved using the optimisation in [Hirschberg, 1975]. Entries succeeding the verification step are inserted into the query result list, and an array, **checked**, is kept to avoid inserting the same entry multiple times.

3. Sorting: The list of entries is sorted in ascending order based on their edit distances, and is returned.

4.5 Pass-Join

Our implementation of the Pass-Join algorithm takes two parameters:

- *threshold*: The threshold for the edit distance. Query results have to have an edit distance $\leq$ the threshold.
- *chainLength*: The length of the pigeonring chain.

The implementation of the Pass-Join algorithm is largely based on [Li et al., 2011]. It consists of three parts:

1. Indexing: For indexing the database is sorted on length. Then, entries of the same length are ordered in alphabetical order. For every entry we split it up in a number of segments equal to (*threshold* + 1). We iterate over the segments and insert them in a double Map datastructure, **index**. The length of the entry is used as the first key, then the number of the current segment is used as index for the array, after which the segment string is used as a second key, and the value is the index of the entry in the database.
2. Searching: The searching part of the Pass-Join algorithm consists of two steps:
   - Filter:
     - First a length-based filter is applied. Entries that differ more than *threshold* in length can be ignored.
     - Then we iterate over all the segments of the entries. For every segment a portion from the query is selected, to which the segment needs to be matched. This portion of the query sequence is limited using the multi-match aware method in [Li et al., 2011]. For every substring in this portion, the index is searched. If a match is found, the entry is considered a candidate, as per the pigeonhole principle.

  - Candidates are then further filtered using the pigeonring principle. We try to find a prefix-viable chain, starting with the matching segment found using the pigeonhole principle. This implementation of the pigeonring principle for the Pass-Join algorithm is novel to the best of our knowledge.

- Verify: The selected entries are verified using the same way as for PIVOTALSEARCH. In [Li et al., 2011], an extension-based verification method is used to speed up the verification process. This method calculates separate thresholds for the parts of the sequences before the matching segment, and for the part after the matching segment. It then calculates the edit distance for these parts separately, first the left part and if it passes, the right part. However, we found that this method slowed down our algorithm, so we chose not to use it.

3. Sorting: The list of entries is sorted in ascending order based on their edit distances, and is returned.

4.6 Similarity search interface

This research provides a library of similarity search algorithms. In figure 11 a diagram of the interface for this library can be seen. Every algorithm is a type of similarity search algorithm, and is therefore derived from the base class SimilaritySearch. Every class derived from SimilaritySearch therefore inherits a Database object which contains all the sequences, and a SearchSequenceID function, which simply obtains a sequence from the given ID and then performs the SearchSequence function. Every derived class therefore has their own implementation of the SearchSequence function and its own set of parameters, using which a list of results can be returned as described in the previous sections.

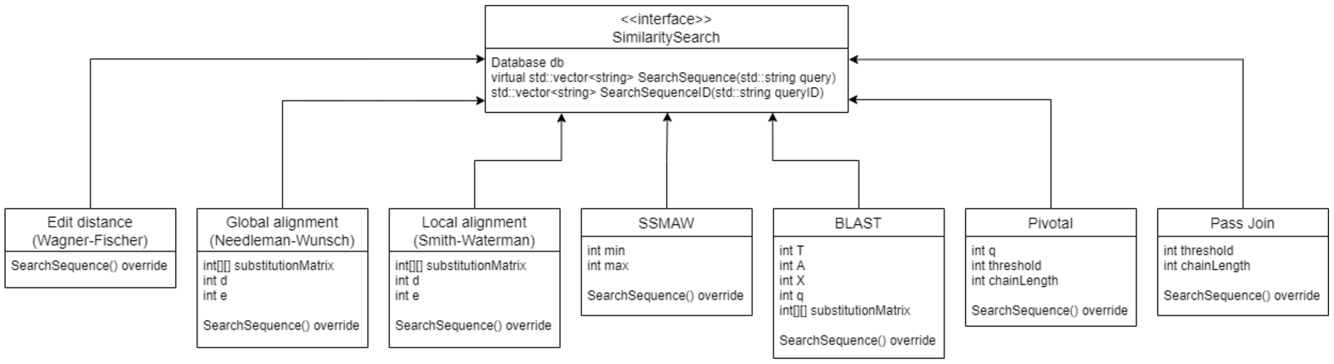

<<interface>>
SimilaritySearch
Database db
virtual std::vector<string> SearchSequence(std::string query)
std::vector<string> SearchSequenceID(std::string queryID)
Edit distance (Wagner-Fischer)
SearchSequence() override
Global alignment (Needleman-Wunsch)
int[][] substitutionMatrix
int d
int e
SearchSequence() override
Local alignment (Smith-Waterman)
int[][] substitutionMatrix
int d
int e
SearchSequence() override
SSMAW
int min
int max
SearchSequence() override
BLAST
int T
int A
int X
int q
int[][] substitutionMatrix
SearchSequence() override
Pivotal
int q
int threshold
int chainLength
SearchSequence() override
Pass Join
int threshold
int chainLength
SearchSequence() override

Figure 11: Interface diagram

The library of implementations can be downloaded from the Github repository: `https://github.com/Jelmul/Music-Similarity-Search`. By including the Music-Similarity-Search.h header file, a user can add the implementations to their own projects. All that has to be provided by the user to start using the algorithms, is a database, and the desired parameters. At the moment of writing, only music datasets are supported, which consist of sequences that have been encoded as diatonic interval sequences like described in section 2.6.1. This is because some of the algorithms are hardcoded to use the alphabet used by these encodings. Future iterations of the library may be altered to allow any type of sequence and database.

5 Results

In this section, the results are given of the experimental comparison between the Wagner-Fischer, Needleman-Wunsch, Smith-Waterman, SSMAW, BLAST, PIVOTALSEARCH and Pass-Join algorithms. Three main comparisons were performed: Retrieval performance, Scalability and Interpretability.

First, in section 5.1 an overview is given of the chosen parameter configurations for the different algorithms. Then, in section 5.2 the results of the retrieval performance experiments are described. In section 5.3 the results of the interpretability experiments are described. Section 5.4 gives information on the parameter configurations for the PIVOTALSEARCH and Pass-Join algorithms, and describes the results of the comparison between the two. Lastly, in section 5.5 the results of the scalability experiments are described.

As each algorithm represents a similarity measure, throughout the following sections, the algorithms and their similarity measures are sometimes used interchangeably. For example, to improve readability in figures, the Wagner-Fischer, Needleman-Wunsch and Smith-Waterman algorithms are referred to by ED, GA and LA respectively. By ‘edit distance algorithms’, Wagner-Fischer, PIVOTALSEARCH and Pass-Join are denoted. In addition, Smith-Waterman and BLAST may be grouped as the ‘local alignment algorithms’, since both calculate a form of local alignment.

5.1 Parameter configurations

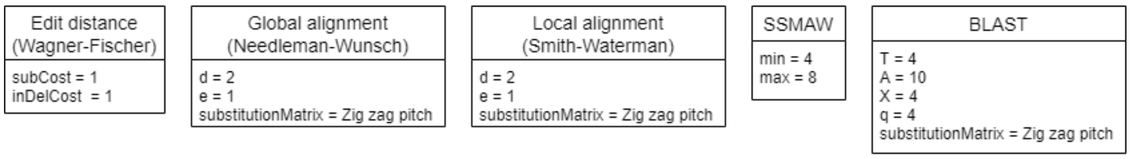

Edit distance (Wagner-Fischer)
subCost = 1
inDelCost = 1
Global alignment (Needleman-Wunsch)
d = 2
e = 1
substitutionMatrix = Zig zag pitch
Local alignment (Smith-Waterman)
d = 2
e = 1
substitutionMatrix = Zig zag pitch
SSMAW
min = 4
max = 8
BLAST
T = 4
A = 10
X = 4
q = 4
substitutionMatrix = Zig zag pitch

Figure 12: The used parameters for the experiments

The different algorithms have parameters, which can be tuned to optimize retrieval performance and/or speed. The parameter configurations for the Wagner-Fischer, Needleman-Wunsch, Smith-Waterman, SSMAW and BLAST algorithms can be seen in figure 12. In this section, it is described how these parameter values were chosen.

For Wagner-Fischer the simple edit distance was used, as this is in accordance with the PIVOTALSEARCH and Pass-Join algorithms. For Needleman-Wunsch and Smith-Waterman, a gap opening cost of 2 and a gap extension cost of 1 were used. For SSMAW, a minimum MAW size of 4 and a maximum MAW size of 8 were used, which were found in [Crawford et al., 2018].

For scoring note substitutions in the Needleman-Wunsch, Smith-Waterman and BLAST algorithms, a substitution matrix based on the zig-zag pitch function [10] is used, which scores a substitution between notes based on the distance (in semitones) between their pitches. Although the sequences used for this research are encoded as interval sequences, this distance still applies, because an interval indirectly represents a pitch (relative to the previous pitch). A score ranging from -2 to 1 was chosen. The function used can be seen in figure 13.

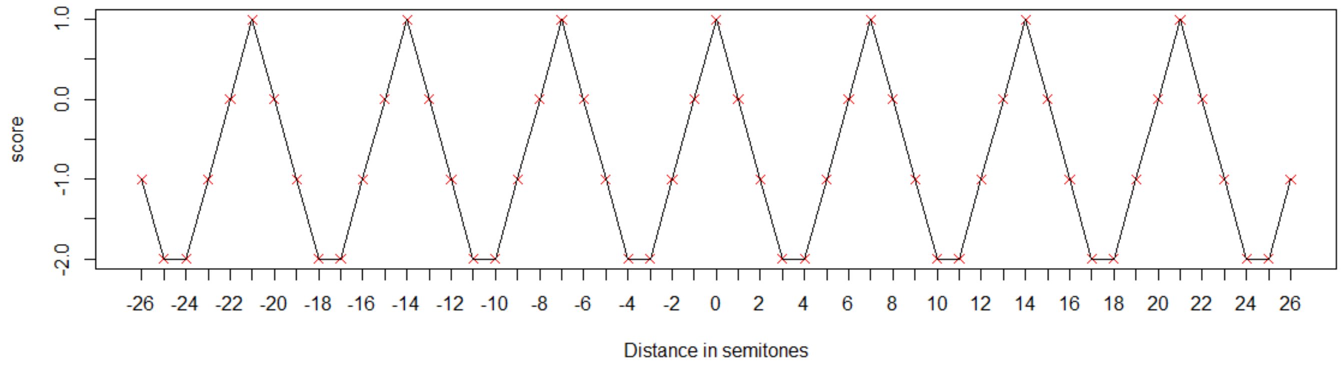


Figure 13: Zig-zag pitch scoring function

5.1.1 Tuning parameters

BLAST required some tuning of the remaining parameters, which influences its retrieval performance. Therefore, different configurations of BLAST were compared empirically. In figure 14, the different configurations, BLAST 1 - BLAST 6, are shown. For each configuration, the drop off score X, window size A, q-gram size q and high scoring word threshold T are given. The last column “Score” indicates whether the algorithm picked the best score or the last score found during extension. In figure 15, the mean rank of every configuration is given for all three tasks. The configurations are sorted based on their mean rank, max rank and query time, in that particular order. Although for every task a different configuration scored the highest, BLAST 1 is top 2 every time. The summed rankings of the algorithms indicate that BLAST 1 performed best. Therefore, a drop off score of 4, window size of 10, q-gram size of 4 and high scoring word threshold of 4 were chosen.

[10]`https://www.projects.science.uu.nl/monochord/risma2/about`

Configuration	Drop off X	Window size A	q-gram size	Threshold T	Score
BLAST 1	4	10	4	4	Last
BLAST 2	10	10	4	4	Last
BLAST 3	50	10	4	4	Last
BLAST 4	4	30	4	4	Last
BLAST 5	4	10	4	4	Best
BLAST 6	4	10	4	3	Last

Figure 14: Different configurations of BLAST

	Duplicate	Mean rank	Min rank	Max rank	Time (ms)
1	BLAST 1	2	2	5	645,102
2	BLAST 4	2	2	5	671,673
3	BLAST 5	2	2	5	869,408
4	BLAST 6	2	2	5	3928,49
5	BLAST 2	2	2	13	696,306
6	BLAST 3	1410	2	21350	769,571

(a)

	Same	Mean rank	Min rank	Max rank	Time (ms)
1	BLAST 2	3	2	270	600,34
2	BLAST 1	3	2	423	569,363
3	BLAST 5	4	2	454	614,816
4	BLAST 4	5	2	877	579,498
5	BLAST 6	10	2	1994	3723,75
6	BLAST 3	3202	1	31717	695,5

(b)

	Relevant	Mean rank	Min rank	Max rank	Time (ms)
1	BLAST 4	879	2	11450	602,883
2	BLAST 1	996	2	30150	688,453
3	BLAST 5	1096	2	30150	608,938
4	BLAST 6	1183	2	18717	3911,27
5	BLAST 2	2812	2	31533	623,562
6	BLAST 3	9420	2	31533	698,922

(c)

	Overall	Sum
1	BLAST 1	5
2	BLAST 4	7
3	BLAST 5	9
4	BLAST 2	10
5	BLAST 6	13
6	BLAST 3	18

(d)

Figure 15: Results of the comparison between different configurations of BLAST for the (a) dupl (b) same (c) relv tasks. (d) The overall order of the configurations

5.2 Retrieval performance

	Duplicate	Mean rank	Min rank	Max rank	SD
1	SSMAW	2	2	3	0,202031
2	GA	2	2	4	0,319438
3	LA	2	2	4	0,451754
4	BLAST	2	2	5	0,451754
5	ED	2	2	5	0,451754

(a)

	Same	Mean rank	Min rank	Max rank	SD
1	BLAST	3	2	423	23,6396
2	SSMAW	10	1	2276	113,166
3	LA	28	1	7746	409,083
4	GA	556	1	28638	2866,35
5	ED	3374	1	31679	8165,44

(b)

	Relevant	Mean rank	Min rank	Max rank	SD
1	SSMAW	95	2	3497	459,514
2	LA	733	2	13213	2036,09
3	BLAST	996	2	30150	3240,48
4	GA	2240	2	29703	5379,02
5	ED	5440	2	31717	9627,57

(c)

	Overall	Sum
1	SSMAW	4
2	LA	8
3	BLAST	8
4	GA	10
5	ED	15

(d)

Figure 16: Results of the comparison of query ranking for the (a) dupl (b) same (c) relv tasks. (d) The overall order of the algorithms in terms of retrieval performance

Figure 16 shows the retrieval performance of the algorithms. The results show the mean rank, min rank, max rank and standard deviation for the ground truth given by the algorithms. The algorithms were sorted by mean rank, max rank and standard deviation, in that particular order.

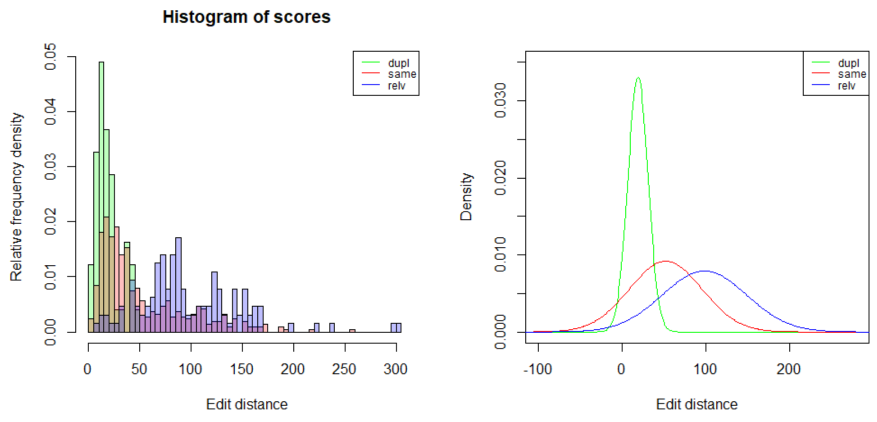
Results for the 'dupl' task were very close amongst all algorithms. For the alignment algorithms and ED, this was to be expected, since these compute global or local alignments. In the case of very similar sequences, the local alignments become global alignments. Interestingly, SSMAW performs best for the dupl task, with a near perfect score: the duplicates were always ranked as the best or second best match.

For the 'same' task, the BLAST algorithm was the best performing, with SSMAW as a close second. For this task the global alignment and edit distance algorithms start to lose their usefulness. For the 'same' task, parts of the sequences may be similar, but in terms of the sequences as a whole, they become less similar. Therefore, it is not surprising that the local alignment and alignment-free algorithms score best for this task. What is surprising, is that the implementation of BLAST scores better than local alignment. Apparently, using the MSP score as a similarity measure instead of the full local alignments, gives us a better result for the 'same' task.

For the 'relv' task, once again SSMAW is the best performing algorithm. This time the local alignment algorithm comes in second place (although not a close second place). As expected, the global alignment algorithms have lost their value.

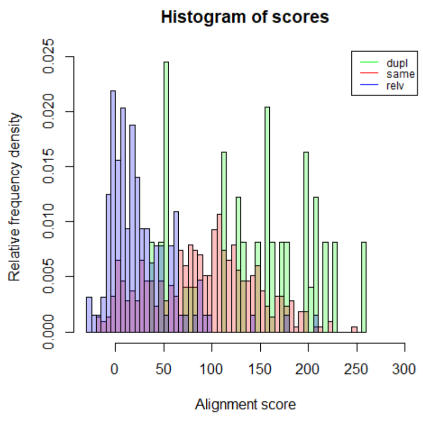
Overall, SSMAW is the clear winner in terms of retrieval performance. Local alignment and BLAST are in a shared second place.

5.3 Interpretability

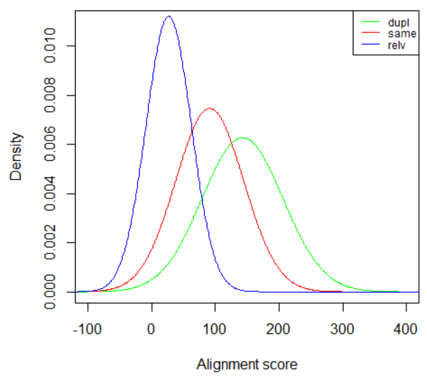


(a) Wagner-Fischer normal distributions (b) Wagner-Fischer histograms

Figure 17: Wagner-Fischer score plots

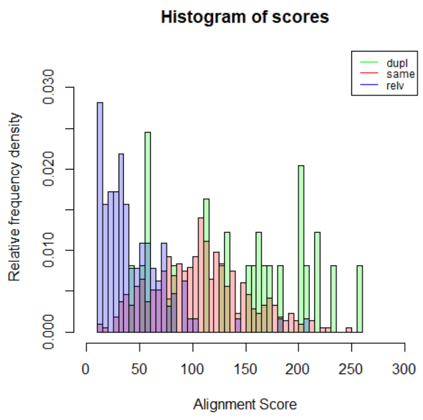


(a) Needleman-Wunsch normal distributions

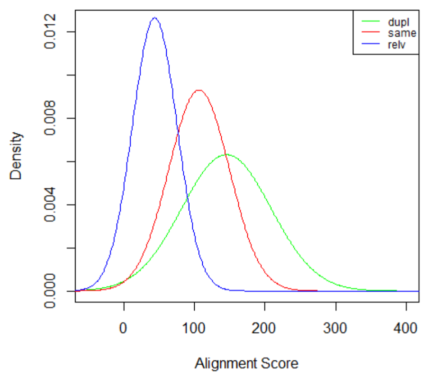


(b) Needleman-Wunsch histograms

Figure 18: Needleman-Wunsch score plots

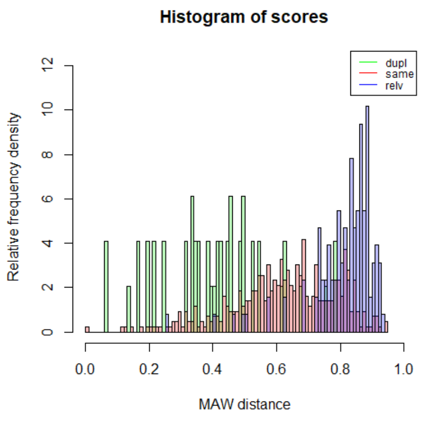


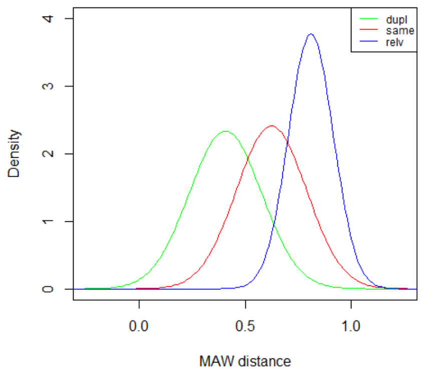
(a) Smith-Waterman normal distributions



(b) Smith-Waterman histograms

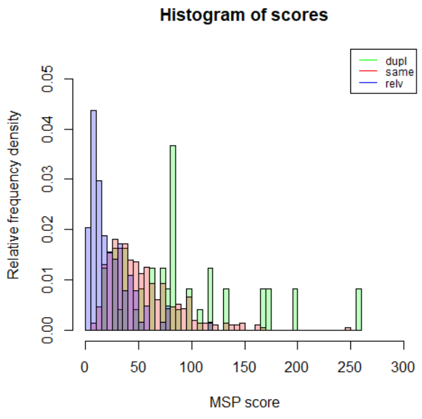
Figure 19: Smith-Waterman score plots

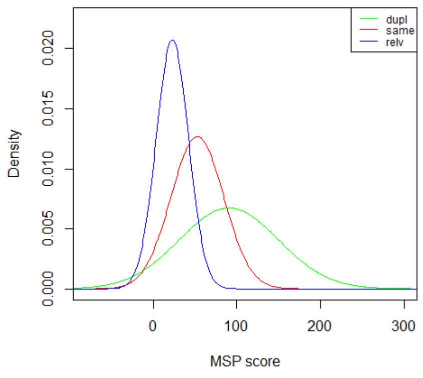




(a) SSMAW normal distributions (b) SSMAW histograms

Figure 20: SSMAW score plots





(a) BLAST normal distributions (b) BLAST histograms

Figure 21: BLAST score plots

The plots in figures 17-21 show, for every algorithm, the histograms of the scores, and normal distribution curves fitted to the distributions of the scores of the dupl (green), same (red) and relv (blue) tasks. For each of the algorithms there is a big overlap between their histograms and curves. This indicates that there is no particular clear classification of a database entry possible, solely based on its score. However, two interesting observations can be made:

- The distinction between the scores seems to be the clearest for SSMAW, although there is still a lot of overlap. Also, there seems to be somewhat of an indication of the range of scores for the tasks. In the normal distribution curves, the peak for the dupl task is at 0.42 MAW distance, the peak for the same task at 0.63 MAW distance, and the peak for the relv task is at 0.82 MAW distance. Combining this information with the histograms, would allow inference of some approximate thresholds of 0.5 for dupl, 0.8 for same and 0.9 for the relv task.

- Also interesting, is the clear concentrated area of duplicate entry ED scores. This provides a clear area of interest, and can give a threshold for use with the threshold-based PIVOTALSEARCH and Pass-Join algorithms. It gives an interesting use case for these algorithms, because they are very fast for low edit distance threshold values. Therefore, they might be useful for finding duplicates in extremely large datasets, when other algorithms are too slow.

5.4 PIVOTALSEARCH and Pass-Join

In this section, the experiments with the PIVOTALSEARCH and Pass-Join algorithms are described. First, it is described how the settings for the parameters of the algorithms were determined empirically. Then, the algorithms are compared. It was found that the PIVOTALSEARCH algorithm requires a substantial amount of memory for its indexes. Therefore, because of the limited hardware setup, the experiments concerning the PIVOTALSEARCH algorithm were performed on 10 smaller datasets, ranging from 31721 to 317210 entries. We will call these datasets $\{EMO1A...EMO10A\}$.

5.4.1 Tuning parameters

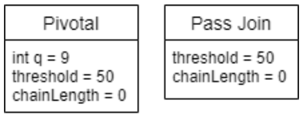
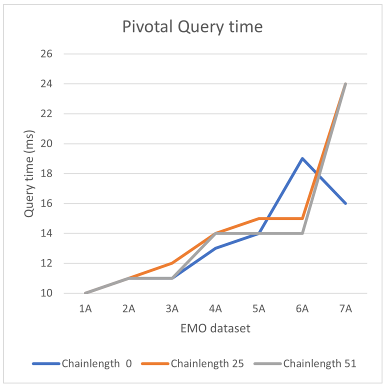


Figure 22: The used parameters for PIVOTALSEARCH and Pass-Join

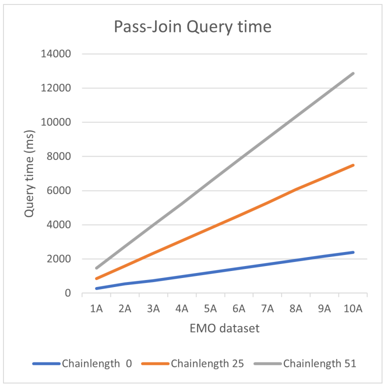
PIVOTALSEARCH and Pass-Join allow tuning of parameters which influence their speed. In figure 22, the parameters are shown which were used for the scalability experiments.

The edit distance threshold parameter was inferred from the interpretability results. The plots of the score distributions for the Wagner-Fischer algorithm, show a clear upper bound for the dupl task at an edit distance of 50. Therefore, 50 is used as the edit distance threshold for the algorithms. Since this threshold only allows for finding duplicates, the algorithms can only be used for the dupl task. This is not a problem, since in section 5.2 the edit distance similarity measure was found to be useless for the other tasks.

Configurations using different chain lengths were compared empirically. The results in figure 23 show that the benefits for Pass-Join do not weigh up against the overhead of using the pigeonring filter. The longer the chain length the higher the query time became. Therefore, the chainLength parameter is set to 0, which disables the pigeonring filter. Because of the large index size of PIVOTALSEARCH, the query times for $EMO9A - EMO10A$ show a huge increase. This can be attributed to the lack of memory in the hardware setup, which causes part of the index to be stored in virtual memory on the external storage device. Therefore, to paint a clearer image, only $EMO1A - EMO8A$ are shown. The graph still doesn't give that much clarity on whether the chain length has a negative effect on the PIVOTALSEARCH query time. Still, it was chosen to set the chain length to 0 for PIVOTALSEARCH as well.



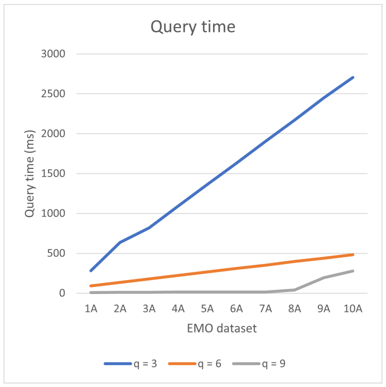
(a)



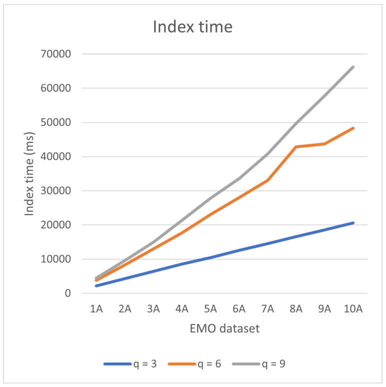
(b)

Figure 23: Scaling of (a) PIVOTALSEARCH and (b) Pass-Join for different settings of chain length.

PIVOTALSEARCH also allows tuning of the q-gram size. Configurations using different q-gram sizes were compared. The results in figure 24a show that, the greater the q-gram size, the faster the queries are performed. Therefore, the q-gram size was set to the maximal possible value, which is 9: the length of the shortest sequence in the EMO dataset. As a trade off, the indexing time increases with increasing q-gram size, because a larger prefix set is generated. This can be seen in figure 24b.



(a)



(b)

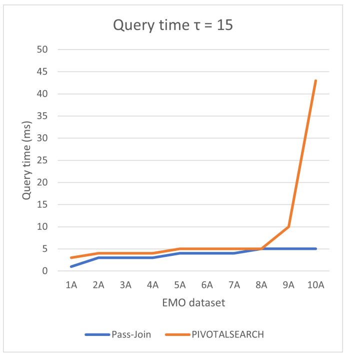
Figure 24: Scaling of the PIVOTALSEARCH algorithm for different settings of q.

5.4.2 Comparison for varying edit distance thresholds

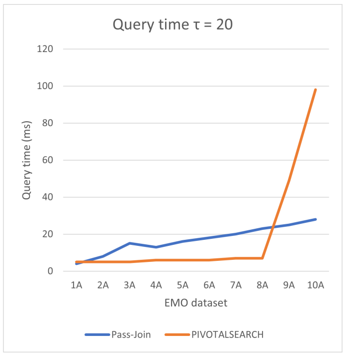
As determined before, for the scalability experiments in section 5.5, an edit distance threshold of 50 is used. However, it can be the case that for other datasets, this threshold can be set lower, for example due to improved OMR technology, which lowers the edit distance between duplicates. Therefore, a comparison between PIVOTALSEARCH and Pass-Join was performed for various edit distance thresholds. Figure 25, shows plots for the query times for various settings of τ: 15, 20, 25, 50. The results show that Pass-Join is only faster than PIVOTALSEARCH for a $\tau \leq 15$. As the threshold increases, the gap between Pass-Join and PIVOTALSEARCH becomes bigger. Once again, PIVOTALSEARCH shows a big increase in query time for $EMO9A-EMO10A$.

In general, PIVOTALSEARCH is the better algorithm in terms of query time. This does, however, come at a cost. Figure 26 shows that Pass-Join performs better than PIVOTALSEARCH in terms of time needed to construct the index and the index size. Index time is not much of a problem, since the indexes only have to be computed once, and this can happen beforehand. The index size however, poses more of a problem, and only allows usage of the PIVOTALSEARCH algorithm, if sufficient memory is available.

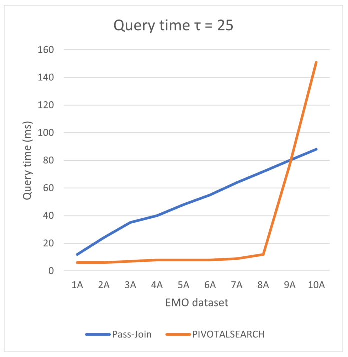
Given that enough memory is available PIVOTALSEARCH is the superior algorithm. Our hardware setup, however, has a very limited amount of memory. Therefore, it was chosen to compare only Pass-Join to the other algorithms for the scalability experiments on the bigger $EMO1-EMO10$ datasets.



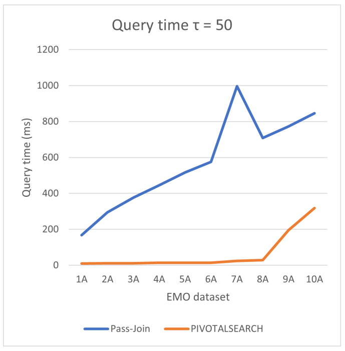
(a)



(b)

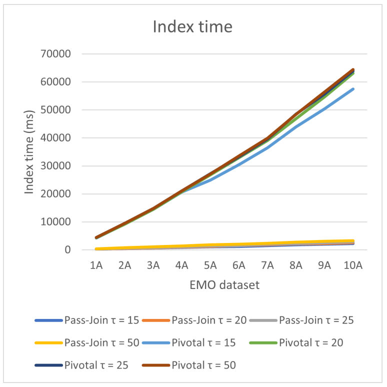


(c)

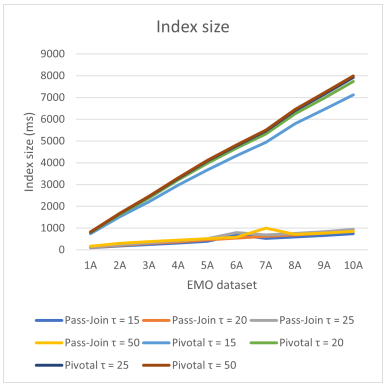


(d)

Figure 25: Scaling of the query time for the PIVOTALSEARCH and Pass-Join algorithms for (a) $\tau = 15$ (b) $\tau = 20$ (c) $\tau = 25$ (d) $\tau = 50$.



(a)



(b)

Figure 26: Scaling of (a) index time and (b) index size for the PIVOTALSEARCH and Pass-Join algorithms for different settings of τ.

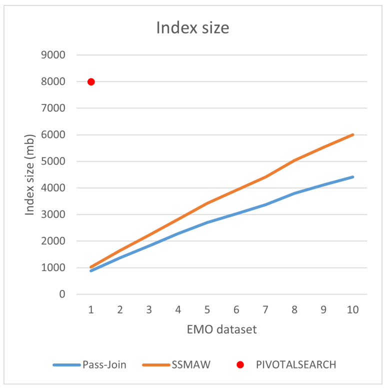
5.5 Scalability

In this section, the results of the scalability experiments are described. For every algorithm, 49 queries (from the dupl ground truth list) were performed on the $EMO1 - EMO10$ datasets. The index size, indexing time and the average query times were recorded.

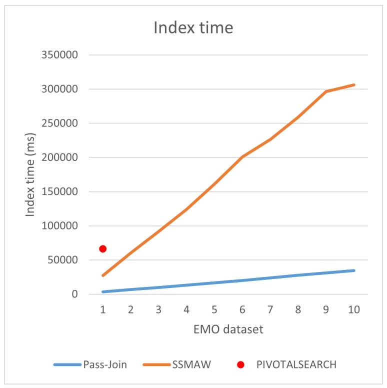
5.5.1 Indexing

Database	Number of pages	Pass-Join index size	Pass-Join index time	SSMAW index size	SSMAW index time
EMO1	317210	882	3373	1022	27360
EMO2	634420	1370	6661	1648	60388
EMO3	951630	1820	9714	2230	91828
EMO4	1268840	2279	13228	2822	123881
EMO5	1586050	2700	16679	3421	160990
EMO6	1903260	3035	19838	3921	200737
EMO7	2220470	3370	23885	4406	226318
EMO8	2537680	3797	27584	5036	258788
EMO9	2854890	4112	31005	5526	296332
EMO10	3172100	4409	34698	6007	306379
EMO11	7613040	8719	83587	13127	858435

Table 2: Table of the index sizes and indexing times of SSMAW and Pass-Join for EMO1-EMO10 and for EMO11, which is 2.4 times the size of EMO10



(a)



(b)

Figure 27: Plots of the (a) index size and (b) index time for the SSMAW and Pass-Join algorithms. For reference, the result of PIVOTALSEARCH for EMO1 is given as a single dot.

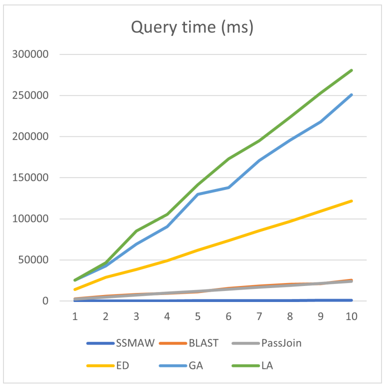
The SSMAW, PIVOTALSEARCH and Pass-Join algorithms, in contrast to the other algorithms, require the construction of an index on the entire database. The other algorithms require space only per comparison. In table 2, the size and construction time of this index for each of the datasets, EMO1-EMO10 and EMO11 is shown. In addition, for SSMAW and Pass-Join, plots of the index sizes and index construction times in relation to the database sizes are shown in figure 27. For PIVOTALSEARCH only the results for EMO1 are shown.

The plots show that the algorithms seem to have a linear space complexity. This trend is further confirmed by the index size of EMO11. The index construction times also seem to scale linearly. Even though the index constructions take a long time compared to the query times, they only have to be performed once. The indexes can therefore be precomputed and stored in memory.

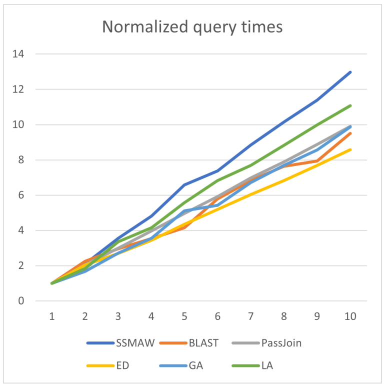
5.5.2 Querying

Dataset	SSMAW	Pass-Join	BLAST	ED	GA	LA
EMO1	68	2420	2658	14180	25430	25330
EMO2	143	4961	5999	28810	42610	46730
EMO3	242	7260	7873	38390	69130	85310
EMO4	327	9595	9416	48890	90370	105160
EMO5	448	11994	11032	61710	129790	141250
EMO6	501	14321	15394	73610	137920	173040
EMO7	601	16869	18138	85560	170850	194780
EMO8	690	19099	20295	96800	195520	223500
EMO9	774	21473	21110	109040	217810	252860
EMO10	882	23955	25247	121630	250800	280570

Table 3: Table of the average query times in milliseconds of the algorithms for EMO1-EMO10.



(a)



(b)

Figure 28: Plots of the (a) average query times in milliseconds and (b) normalized query times for EMO1-EMO10.

For testing the scalability of the query time, the 49 queries were performed on EMO1-EMO10. For the dynamic programming algorithms, the queries were performed on the smaller EMO1A-EMO10A datasets, and the average query time was multiplied by 10 to obtain the average query times for EMO1-EMO10. This was done to save time, which is possible since the dynamic programming algorithms simply repeatedly perform the sequence comparisons for every database entry.

Table 3 gives an overview of the average query times for the datasets EMO1-EMO10. Figure 28 shows a plot of the query times and the normalized query times. For normalization of the query time each query time on EMOn is divided by the query time for EMO1. The plots show that the running times of all the algorithms seem to be scaling linearly.

There is a big difference between the algorithms in terms of the query times. Even for the smallest dataset $EMO1$, the dynamic programming algorithms do not have acceptable real-time query times: 14.18, 25.43 and 25.33 seconds for ED, GA and LA respectively. SSMAW is the only algorithm that gives acceptable query times for real-time searching. We define 'real-time' as less than or around 1 second, which is realistic for an information retrieval system. Pass-Join and BLAST give 'semi real-time' query times for $EMO1$, 2.43 and 2.65 seconds respectively. However, their query times quickly reach unacceptable levels, with a query time of around 24 seconds for $EMO10$. At the largest datasets, this leaves SSMAW as the only viable option, with 0.88 second query times. The SSMAW and Pass-Join algorithms were also tested on $EMO11$, which gave an average query time of 2.62 seconds for SSMAW and 58.26 seconds for Pass-Join. These results further confirm the linear time complexity of the algorithms. The average query time of SSMAW for $EMO11$ does not continue the exact same trend as for $EMO1 - EMO10$, but this can be attributed to the large index size, which causes the index to be partially stored in virtual memory.

5.5.3 Pass-Join vs SSMAW

Various edit distance thresholds, lower than 50, were tested for Pass-Join, to see from which edit distance threshold Pass-Join becomes a competitor to SSMAW. The results are shown in figure 29. Empirically it was found that for $\tau = 25$ Pass-Join performs similarly to SSMAW. For $\tau < 25$, Pass-Join becomes faster then SSMAW.

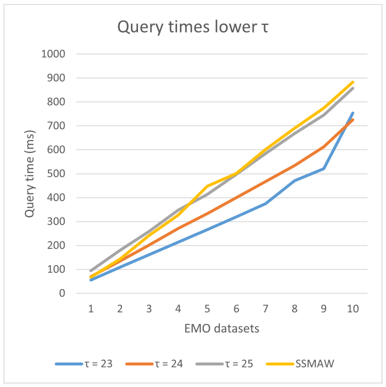


Figure 29: The average query times of Pass-Join for different settings of τ compared to SSMAW.

6 Discussion

In this section, the results of the experiments are discussed.

6.1 Retrieval performance

In terms of overall retrieval performance, SSMAW was the clear winner. The reason for this is speculated to be as follows: MAWs require there to be a subsequence (length 3-7 for min=4 and max=8) that exactly matches between sequences, for them to share a MAW. There also have to be multiple matching subsequences (length 2-6 for min=4 and max=8) within a single sequence. The presence of these common repeated subsequences/patterns, may indicate a shared melody or motif between sequences, which seems to be a good measure of musical similarity. Also, the set of MAWs gives information about the context of these motifs. It gives information on the intervals between notes going into the melody (first letter of the MAW), and intervals between notes going out of the melody (last letter of the MAW). This may represent going in and out of different sections in the music. These are just hypotheses that have to be tested by taking a closer look at the query results and the MAWs shared between the query and its results.

Local alignment and BLAST share a second place for retrieval performance. Particularly BLAST is an interesting algorithm for further research, because it allows for a lot of tuning. BLAST, like SSMAW, also requires shared exact matches of substrings. However, our implementation allows for substitutions between the same note in a different octave (because of the zig zag pitch scoring function). Alternative configurations of BLAST (especially PSI-BLAST), may increase the performance of BLAST for the relevant task. This would increase the usefulness of BLAST.

The global alignment and edit distance algorithms are clearly the least useful in terms of retrieval performance, only proving useful for the dupl task. Although global alignment overall scores higher than edit distance (but only barely), edit distance may still be more useful because of the speed that PIVOTALSEARCH and Pass-Join can provide for the dupl task, when given a threshold.

The extent to which the results are generalizable to contexts other than music data is unclear. However, it would seem as though these results apply to other musical contexts. Experiments on other types of music datasets are needed to make a judgement about the generalizability.

The parameter settings for the algorithms were found empirically. They might not be optimal: it may be possible that other configurations may result in different results. A more systematic evaluation and tuning of the parameters, may be beneficial.

6.2 Interpretability

The results of the interpretability experiment are unfortunately not very promising for all of the algorithms. Scores for each of the tasks show a large overlap, and span a large portion of the score space.

The edit distance threshold that was obtained from the Wagner-Fischer results, shows the potential of using PIVOTALSEARCH and Pass-Join for the dupl task. However, this threshold is not necessarily generalizable to other music datasets. The threshold depends on the quality of the OMR tools used and, therefore, is specific to the version of the EMO dataset that was used.

The plots for SSMAW, show a somewhat better interpretability than the other algorithms. However, the quality of the thresholds that can be obtained from them is questionable. Future work with a bigger ground truth set, may be able to give more meaningful results.

6.3 Scalability

Overall, as for the retrieval performance, the SSMAW algorithm is the clear winner in terms of scalability. The SSMAW algorithm is the only alignment-free algorithm that was tested, and the only algorithm that utilizes a token-based similarity measure. The other algorithms all require some form of alignment and use a character-based similarity measure. Therefore, these results come as no surprise.

PIVOTALSEARCH would have been a good competitor for SSMAW for the dupl task, as it seemed to be faster than SSMAW. However, this has to be tested with a better hardware setup. The Pass-Join algorithm is also a good competitor, in the case of a low edit distance threshold. These algorithms can then filter vast amounts of database entries, which causes the costs of the more expensive character-based similarity function to be neglectable.

A single test of the SSMAW algorithm on the EMO11 dataset gave a semi real-time query time of 2.62 seconds. In other words, the SSMAW algorithm allowed for semi real-time querying on a music dataset of 7,613,040 pages of sheet music, or more than 1 billion notes, on a low end consumer laptop. Obviously, server hardware on which the retrieval code will be ran, will be superior both in terms of computing power and memory. At the moment of writing this thesis, the CPU with best single thread performance currently has approximately 1.84 times the speed of the test setup. Fitting a maximum dataset size in 64gb of ram, would yield a dataset of approximately 37 million pages. This hardware setup, would give a query time of approximately 6.92 seconds on the dataset, which surpasses 5 billion notes. This theoretical dataset would be approximately 1166x the size of the Early Music Online dataset, in other words more than 370,000 volumes of printed music. Possible improvements of the implementation, such as parallelization and vectorization may be able to further increase the performance.

The results of the scalability experiments are generalizable to contexts other than music datasets. Therefore, the plots and data obtained through this research, give a good benchmark of the algorithms for other contexts.

6.4 Data representation

The sequences of notes were represented as sequences of diatonic intervals. This representation solves some problems regarding the OMR errors, but there is also an unwanted effect. A single error in the recognition of a diatonic pitch sequence, can cause two errors in the diatonic interval encoding. One interval error will be made between the correct note and the erronous note, then another interval error will be made when going from the erronous note to the next correct note. Although this unwanted effect does not weigh up against the positive effects that the diatonic interval encoding brings for the OMR data, it is still something to take into account. Especially, since it influences the score for some of the similarity measures, and thereby the results.

6.5 Ground truth

The results of this research depend heavily on the quality of the ground truth used. Therefore, it is only logical to look critically at this ground truth. The ground truth was obtained in the context of an evaluation of SSMAW in a MIR system [Crawford et al., 2018]. Partially it may have even been obtained from classifications performed by users of the MIR system. Therefore, it is possible that the ground truth used is somewhat biased towards the SSMAW algorithm. In addition to these flaws, the ground truth is also limited in size. However, at the time of performing this research it was the only ground truth available. For future work, it would be beneficial to use more ground truth. Preferably, this ground truth should be obtained away from the SSMAW algorithm. This does, however, require a large amount of manual labour.

6.6 Decision tree for efficient similarity search on early music

The results of the experiments revealed strengths and weaknesses of the different algorithms. Based on the context and goals of the queries, different algorithms can be recommended to be used. From the results, the following decision tree can be made:

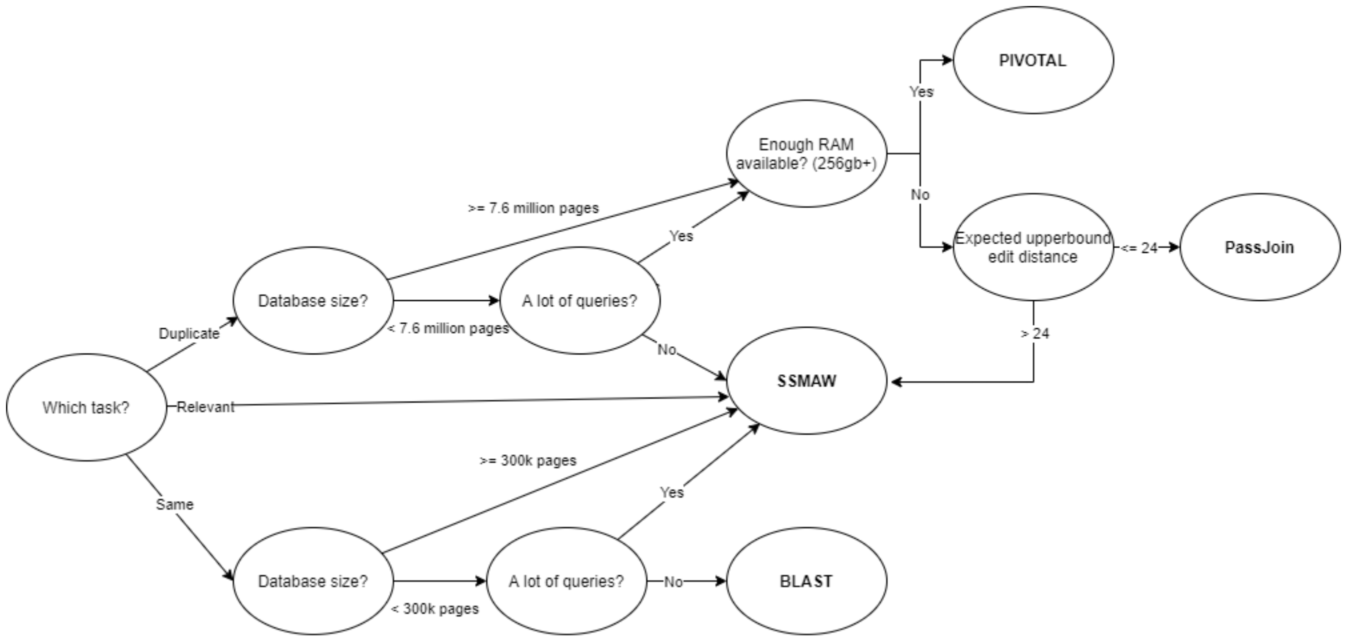


Figure 30: Decision tree

This decision tree can aid a user or designer of a music information retrieval system with the selection of the algorithm, which is best suited to their needs for making real time queries on music datasets.

7 Conclusion

In this section, the thesis and thereby the research is concluded.

To reiterate, the research question is: "Are MAWs useful for efficient similarity search in the context of early music?" To answer the question the following hypotheses have been tested:

- Hypothesis 1: "The SSMAW algorithm scores higher than the other algorithms in retrieval performance for the 'dupl' task (finding duplicate images within the collection)".
- Hypothesis 2: "The SSMAW algorithm scores higher than the other algorithms in retrieval performance for the 'same' task (finding pages containing substantially the same music as in a query page)".
- Hypothesis 3: "The SSMAW algorithm scores higher than the other algorithms in retrieval performance for the 'relv' task (identifying pages which have non-identical but related or closely relevant music content)".
- Hypothesis 4: "The results of the SSMAW algorithm are better interpretable than that of the other algorithms".
- Hypothesis 5: "The SSMAW algorithm shows better scalability than the other algorithms".

As has been discussed, the results show that SSMAW is the clear winner in terms of retrieval performance for the 'dupl' and 'relv' task. Therefore, the experiments support hypothesis 1 and 3. The SSMAW algorithm takes the second place for the 'same' task, being beaten by BLAST, therefore hypothesis 2 can be rejected. The results have shown that the SSMAW algorithm has better scalability than the other algorithms. The query times are faster than those of the other algorithms, at the cost of building an index. Therefore, the experiments support hypothesis 4. Lastly, SSMAW has shown a relatively better interpretability compared to the other tested algorithms. Therefore, hypothesis 5 is supported.

As earlier established, we would label SSMAW as useful, if at least one of the hypotheses 1-5 is true. Four of the hypotheses are supported. Therefore, we can answer the research question with: "Yes, the experiments have shown that MAWs are useful for efficient similarity search in the context of early music."

This research also has some other findings and outcomes as a by-product:

- PIVOTALSEARCH was found to be faster than Pass-Join, which to the best of our knowledge is a novel finding.
- The pigeonring filter was implemented for both PIVOTALSEARCH and Pass-Join, and it was found to be detrimental for the query time.

- A library containing the seven algorithms, and an interface to use them in other projects is provided.
- Empirically determined parameters for the SSMAW, PIVOTALSEARCH and Pass-Join algorithms have been obtained.
- A decision tree was created to aid selection of the appropriate algorithm for making real time queries.

This research also unvealed some suggestions and possible directions for future research:

- **PSI-BLAST implementation -** Papers on PSI-BLAST show that it may be a good tool for improving the results for the 'relv' task. It would be interesting to see results of the experiments for an implementation of PSI-BLAST.
- **Evaluating the output of SSMAW -** In the discussion, the hypothesis was formulated, that the SSMAW algorithm may be performing well, due to the fact that MAWs represent melodies/motives in sequences. To test this hypothesis, a closer evaluation of the SSMAW output in combination with the page-images is needed.
- **Using SSMAW with melodic queries -** It would be interesting to search music databases with SSMAW using smaller melodic queries, instead of whole pages of music. For example, the hooks of songs may be used as queries.
- **SSMAW with substitution matrix -** SSMAW currently looks for minimal absent words based on exact matches, and therefore also compares sequences based on exact matches. This may be its strength, but utilizing zig-zag pitch or another substitution matrix for SSMAW, to create a hybrid-based similarity function, would be an interesting experiment.
- **Testing PIVOTALSEARCH on a better hardware setup -** The amount of memory of the hardware setup used in this research, was found to be insufficient for properly testing the PIVOTALSEARCH algorithm. Therefore, repetition of the experiments is required for PIVOTALSEARCH, using a better hardware setup.
- **Increased size dataset and ground truth set -** The ground truth that was available for this research was only limited, and in addition, the set of real data that was available was only small. Therefore, further research in which both these sets are expanded is recommended.
- **Further optimization of algorithms -** The algorithms might lend themselves for further optimization, for example by parallelization or vectorization. Or perhaps a version using the GPU could be implemented.

- **Online music similarity search platform/application -** This research has provided the implementation of several algorithms along with an interface for them. In addition, a decision tree has been constructed, which, given the current task, points to the optimal algorithm. An interesting next step would be to implement an online platform or application, which provides utilization of these algorithms on the server side. Combined with a larger dataset, good visualization and audiation of the page-images, and an evaluation system for the results, this may prove a useful tool for musicologist.

References

[Altschul et al., 1990] Altschul, S. F., Gish, W., Miller, W., Myers, E. W., and Lipman, D. J. (1990). Basic local alignment search tool. *Journal of molecular biology*, 215(3):403–410.

[Altschul et al., 1997] Altschul, S. F., Madden, T. L., Schäffer, A. A., Zhang, J., Zhang, Z., Miller, W., and Lipman, D. J. (1997). Gapped blast and psi-blast: a new generation of protein database search programs. *Nucleic acids research*, 25(17):3389–3402.

[Barton et al., 2014] Barton, C., Heliou, A., Mouchard, L., and Pissis, S. P. (2014). Linear-time computation of minimal absent words using suffix array. *BMC bioinformatics*, 15(1):388.

[Barton et al., 2016] Barton, C., Héliou, A., Mouchard, L., and Pissis, S. P. (2016). Parallelising the computation of minimal absent words. In *Parallel Processing and Applied Mathematics*, pages 243–253. Springer.

[Béal et al., 1996] Béal, M.-P., Mignosi, F., and Restivo, A. (1996). Minimal forbidden words and symbolic dynamics. In *Annual Symposium on Theoretical Aspects of Computer Science*, pages 555–566. Springer.

[Bountouridis et al., 2017] Bountouridis, D., Brown, D. G., Wiering, F., and Veltkamp, R. C. (2017). Melodic similarity and applications using biologically-inspired techniques. *Applied Sciences*, 7(12):1242.

[Cameron et al., 2004] Cameron, M., Williams, H. E., and Cannane, A. (2004). Improved gapped alignment in blast. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 1(3):116–129.

[Chairungsee and Crochemore, 2012] Chairungsee, S. and Crochemore, M. (2012). Using minimal absent words to build phylogeny. *Theoretical Computer Science*, 450:109–116.

[Crawford et al., 2018] Crawford, T., Badkobeh, G., and Lewis, D. (2018). Searching page-images of early music scanned with omr: a scalable solution using minimal absent words. In *ISMIR*. Goldsmiths, University of London.

[Crochemore et al., 2016] Crochemore, M., Fici, G., Mercaş, R., and Pissis, S. P. (2016). Linear-time sequence comparison using minimal absent words & applications. In *LATIN 2016: Theoretical Informatics*, pages 334–346. Springer.

[Deng et al., 2014] Deng, D., Li, G., and Feng, J. (2014). A pivotal prefix based filtering algorithm for string similarity search. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 673–684.

[Fenz et al., 2012] Fenz, D., Lange, D., Rheinländer, A., Naumann, F., and Leser, U. (2012). Efficient similarity search in very large string sets. In *International Conference on Scientific and Statistical Database Management*, pages 262–279. Springer.

[Gotoh, 1982] Gotoh, O. (1982). An improved algorithm for matching biological sequences. *Journal of molecular biology*, 162(3):705–708.

[Hall and Dowling, 1980] Hall, P. A. and Dowling, G. R. (1980). Approximate string matching. *ACM computing surveys (CSUR)*, 12(4):381–402.

[Hankinson et al., 2012] Hankinson, A., Burgoyne, J. A., Vigliensoni, G., Porter, A., Thompson, J., Liu, W., Chiu, R., and Fujinaga, I. (2012). Digital document image retrieval using optical music recognition. In *ISMIR*, pages 577–582. Citeseer.

[Hirschberg, 1975] Hirschberg, D. S. (1975). A linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, 18(6):341–343.

[Jiang et al., 2014] Jiang, Y., Li, G., Feng, J., and Li, W.-S. (2014). String similarity joins: An experimental evaluation. *Proceedings of the VLDB Endowment*, 7(8):625–636.

[Karlin and Altschul, 1990] Karlin, S. and Altschul, S. F. (1990). Methods for assessing the statistical significance of molecular sequence features by using general scoring schemes. *Proceedings of the National Academy of Sciences*, 87(6):2264–2268.

[Kilian and Hoos, 2004] Kilian, J. and Hoos, H. H. (2004). Musicblast-gapped sequence alignment for mir. In *ISMIR*.

[Kondrak, 2005] Kondrak, G. (2005). N-gram similarity and distance. In *International symposium on string processing and information retrieval*, pages 115–126. Springer.

[Levenshtein, 1966] Levenshtein, V. I. (1966). Binary codes capable of correcting deletions, insertions, and reversals. *Soviet physics doklady*, 10(8):707–710.

[Li et al., 2011] Li, G., Deng, D., Wang, J., and Feng, J. (2011). Pass-join: A partition-based method for similarity joins. *arXiv preprint arXiv:1111.7171*.

[Lipman and Pearson, 1985] Lipman, D. J. and Pearson, W. R. (1985). Rapid and sensitive protein similarity searches. *Science*, 227(4693):1435–1441.

[Manber and Myers, 1993] Manber, U. and Myers, G. (1993). Suffix arrays: a new method for on-line string searches. *siam Journal on Computing*, 22(5):935–948.

[Navarro, 2001] Navarro, G. (2001). A guided tour to approximate string matching. *ACM computing surveys (CSUR)*, 33(1):31–88.

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

[Pinho et al., 2009] Pinho, A. J., Ferreira, P. J., Garcia, S. P., and Rodrigues, J. M. (2009). On finding minimal absent words. *BMC bioinformatics*, 10(1):137.

[Pugin and Crawford, 2013] Pugin, L. and Crawford, T. (2013). Evaluating omr on the early music online collection. In *ISMIR*, pages 439–444.

[Qin and Xiao, 2018] Qin, J. and Xiao, C. (2018). Pigeonring: A principle for faster thresholded similarity search. *Proceedings of the VLDB Endowment*, 12(1):28–42.

[Rachkovskij, 2019] Rachkovskij, D. (2019). Index structures for fast similarity search for symbol strings. *Cybernetics and Systems Analysis*, 55(5):860–878.

[Smith et al., 1981] Smith, T. F., Waterman, M. S., et al. (1981). Identification of common molecular subsequences. *Journal of molecular biology*, 147(1):195–197.

[Typke et al., 2005] Typke, R., Wiering, F., and Veltkamp, R. C. (2005). A survey of music information retrieval systems. In *Proc. 6th international conference on music information retrieval*, pages 153–160. Queen Mary, University of London.

[Ukkonen, 1985] Ukkonen, E. (1985). Algorithms for approximate string matching. *Information and control*, 64(1-3):100–118.

[Ukkonen, 1992] Ukkonen, E. (1992). Approximate string-matching with q-grams and maximal matches. *Theoretical computer science*, 92(1):191–211.

[Wagner and Fischer, 1974] Wagner, R. A. and Fischer, M. J. (1974). The string-to-string correction problem. *Journal of the ACM (JACM)*, 21(1):168–173.

[Wandelt et al., 2014] Wandelt, S., Deng, D., Gerdjikov, S., Mishra, S., Mitankin, P., Patil, M., Siragusa, E., Tiskin, A., Wang, W., Wang, J., et al. (2014). State-of-the-art in string similarity search and join. *ACM Sigmod Record*, 43(1):64–76.

[Wilbur and Lipman, 1983] Wilbur, W. J. and Lipman, D. J. (1983). Rapid similarity searches of nucleic acid and protein data banks. *Proceedings of the National Academy of Sciences*, 80(3):726–730.

[Yu et al., 2016] Yu, M., Li, G., Deng, D., and Feng, J. (2016). String similarity search and join: a survey. *Frontiers of Computer Science*, 10(3):399–417.