Technical University of Denmark

Department of Applied Mathematics and Computer Science, DTU Compute
MASTER THESIS

Exploring Anchored Text Indexing

Andreea Matei

Supervised by:
Philip Bille and Inge Li Gørtz

June 2025

Abstract

The text indexing problem [1, 18] is a central topic in computer science, highly addressed by many scientists and professionals throughout time. It plays a critical role in many modern systems, especially in the field of databases, which need to handle and retrieve tons of information on a daily basis. The problem essentially deals with building efficient data structures for strings that would allow searching for online patterns. The ideal solution would show full efficiency by taking up both little space and building effort, while supporting fast operations.

The current thesis proposes three new algorithms for the text indexing problem starting from the index structure of Ayad et al. [2], which uses anchors to come closer to the desired end goal. The first variant followed a naive strategy, the second used the benefits of a kd-tree to improve the query times, and the last one adjusted the internal structure of the compact trie. The experimental evaluation indicates that searching times improve, but the space complexity remains more or less the same across the three algorithms.

Most results of the thesis are similar to the ones of the index solution of Ayad et al. [2]. Specifically, the space of the data structure decreases as the number of anchors gets higher. The significance of these findings can be found in the benefits that anchors bring to the text indexing problem.

Keywords: Algorithms, Text Indexing, Pattern Matching, Locally Consistent Anchors, Experimental Evaluation

Contents

1 Introduction

1.1 Background and Motivation

In the growing era of digital text, looking for and fetching data in an efficient manner becomes an essential issue. Domains, such as databases [49, 62] and bioinformatics [32, 51], greatly benefit from fast retrievals, as information of different characteristics are queried everyday.

Text indexing [1, 18] is the problem that deals with this. According to its definition, a string S is given for preprocessing into a data structure that could support fast search operations for online patterns coming one at a time [18].

A straightforward approach to this problem would be to naively store the original text as a whole and compare the query string against it character by character [25]. However, this method becomes quickly inefficient as the string can significantly grow in length.

To address this limitation, Ayad et al. [2] presented an index structure that achieves efficiency in terms of space, time, and building aspects. Their solution uses an anchor-based technique to select and store only the most relevant positions in the text, thus reducing the space and searching complexity.

1.2 Contributions of the Thesis

Starting from the solution of Ayad et al. [2], this thesis proposes new algorithms for the text indexing problem via an experimental approach. The idea is to design and apply new and enhanced data structures while maintaining the functionality of their anchored index [2].

As the main contribution, the study presents three variants, which use augmented compact tries and *kd*-trees. These algorithms add to the field of computer science by providing insight into the practical value of anchors. Additionally, the thesis tries to experimentally validate the results obtained by Ayad et al. [2].

1.3 Thesis Organization

The thesis is structured in ten parts. The first two chapters introduce the main topic of the paper and present an overview of its related work. The third section explains the base algorithm of Ayad et al. [2], which serves as the starting point. Sections four, five, six present the developed algorithms for this thesis. Chapter seven assesses the variants from a practical perspective using experiments while the eighth chapter draws the final conclusions. Finally, the last two sections, nine and ten, discuss the challenges and limitations faced throughout the study while advising upon potential future work.

2 Related Work

2.1 Classic Data Structures

The suffix tree [18, 25, 43, 47] is a fundamental instrument used in text indexing and string algorithms, specifically in pattern matching. Formally, for a string S, the suffix tree of S is the compact or compressed trie over all of its suffixes [18, 47, 55]. Searching then requires following the branches of the tree alongside with the characters in the query pattern.

Another widely-accepted structure used in text indexing is the suffix array [36, 43], which is strongly connected to the suffix tree. The suffix array of a given string S is an array storing lexicographically the starting positions of all suffixes in S [36, 43]. Pattern matching in the suffix array is then done by binary searching through its elements and narrowing down the search space until the pattern is found.

The FM-index [13, 31] represents an additional data structure which relies on compression to answer the indexing problem. Unlike before, this structure combines the Burrows-Wheeler Transform (BWT) [5, 27, 30] technique with other data structures, such as a C array [27, 31] and a rank data structure [30, 31], to support fast queries while taking up relatively small space.

2.2 Sampling Techniques and Anchors

In computer science, sampling [28] refers to storing a subset of the available data rather than the entire information to optimize the space usage and improve querying times. There are multiple approaches when it comes to selecting such bits of data out of everything. The most common one is based on lexicographical order, but other methods which, for example, rely on randomization, also exist [28].

Winnowing [53] represents such a sampling technique, which selects certain hashes from a document. In short, it computes the fingerprints for k-fixed length portions of the given text and keeps only the smallest ones in every overlapping window [29, 53].

Just like the selected hashes in *winnowing*, *anchors* are carefully selected and representative positions in a text. There are various types of anchors, such as minimizers [51], bidirectional anchors (bd-anchors) with their reduced version [34] and their variants [21, 22, 23]. Anchors become powerful tools in sampling, as they allow data structures like the suffix array to store fewer locations in the text instead of its entire content [19]. Hence, the full string does not need indexing anymore, but rather only its most relevant positions. For this to work, the anchors must be locally consistent. That is, anchors must satisfy two properties: (1) they must uniformly cover the string – in other words, any substring of a fixed minimum length must contain at least one anchor, and (2) they must maintain their relative position across identical substrings [2].

Roberts et al. [51] introduced for the first time the notion of minimizers as anchors within the context of DNA sequences. Minimizers are the smallest substrings of length k (in a lexicographical order) chosen out of a *window* containing w such consecutive overlapping substrings in a larger text S, for some integers w, k [2, 20, 28, 35, 51]. Figure 1 illustrates the idea of selecting minimizers in a window T, which spans over $w + k - 1$ characters in S.

Variations stemming from minimizers include mod-minimizers [23], which integrate the modulus operator in the sampling process, or the open-closed minimizers [21], which skip the first substring of length k within a window.

$$S = a\,b\,a\,b\,c\,b\,a\,b\,c\,a\,b,\ |S| = 11$$
$$\overline{a\,b\,a\,b\,c}\,b\,a\,b\,c\,a\,b,\ T = a\,b\,a\,b\,c$$
$$a\,\overline{b\,a\,b\,c\,b}\,a\,b\,c\,a\,b,\ T = b\,a\,b\,c\,b$$
$$\ldots$$
$$a\,b\,a\,b\,c\,b\,\overline{a\,b\,c\,a\,b},\ T = a\,b\,c\,a\,b$$

Figure 1: Illustration of a sliding window T of length $w+k-1 = 5$ over a string $S = ababcbabcab$, when $w = k = 3$. The sliding window T is depicted by the overline markup and the chosen minimizer within each window is highlighted in red.

In 2021, Loukides and his research group [34] proposed an innovative solution for the text indexing problem using an improved version of minimizers called (reduced) bidirectional anchors for sampling, which offered better overall coverage and alignment in both forward and reverse directions [34]. Bd-anchors represent the smallest rotations (lexicographically speaking) of every substring of length l within a text S, for some integer $l, l \leq |S|$ [2, 20, 34, 35]. Figure 2 depicts the selection process of the minimal rotation for such substrings of length l in S.

$$S = a\,b\,a\,b\,c\,b\,a\,b\,c\,a\,b,\ |S| = 11$$
$$\overline{a\,b\,a\,b\,c}\,b\,a\,b\,c\,a\,b,\ \text{Rotations: } a\,b\,a\,b\,c, b\,a\,b\,c\,a, a\,b\,c\,a\,b, b\,c\,a\,b\,a, c\,a\,b\,a\,b$$
$$a\,\overline{b\,a\,b\,c\,b}\,a\,b\,c\,a\,b,\ \text{Rotations: } b\,a\,b\,c\,b, a\,b\,c\,b\,b, b\,c\,b\,b\,a, c\,b\,b\,a\,b, b\,b\,a\,b\,c$$
$$\ldots$$
$$a\,b\,a\,b\,c\,b\,\overline{a\,b\,c\,a\,b},\ \text{Rotations: } a\,b\,c\,a\,b, b\,c\,a\,b\,a, c\,a\,b\,a\,b, a\,b\,a\,b\,c, b\,a\,b\,c\,a$$

Figure 2: Demonstration of selecting the minimal rotations within each substring of length $l = 5$ from $S = ababcbabcab$. The considered substrings are overlined and both their smallest rotations and bd-anchors are highlighted in red.

Reduced bd-anchors are similar to bd-anchors except that they only look at the first $l - r$ rotations in the substrings of S, for some integers l, r with $l \leq S$ [2, 20, 34]. Figure 3 indicates the rotations which are evaluated to determine the minimal one for substrings of length l within S. Unlike bidirectional anchors (figure 2), we stop at the $(l-r)^{th}$ rotation and do not continue any further.

$$S = a\,b\,a\,b\,c\,b\,a\,b\,c\,a\,b,\ |S| = 11$$
$$\overline{a\,b\,a\,b\,c}\,b\,a\,b\,c\,a\,b,\ \text{Rotations: } a\,b\,a\,b\,c, b\,a\,b\,c\,a, a\,b\,c\,a\,b, b\,c\,a\,b\,a$$
$$a\,\overline{b\,a\,b\,c\,b}\,a\,b\,c\,a\,b,\ \text{Rotations: } b\,a\,b\,c\,b, a\,b\,c\,b\,b, b\,c\,b\,b\,a, c\,b\,b\,a\,b$$
$$\ldots$$
$$a\,b\,a\,b\,c\,b\,\overline{a\,b\,c\,a\,b},\ \text{Rotations: } a\,b\,c\,a\,b, b\,c\,a\,b\,a, c\,a\,b\,a\,b, a\,b\,a\,b\,c$$

Figure 3: Example of calculating the reduced bd-anchors for the running example string $S = ababcbabcab$, for parameters $l = 5$ and $r = 1$. As opposed to figure 2, only the first $l - r = 5 - 1 = 4$ rotations are now considered. As before, the substrings are overlined while the reduced bd-anchors and their rotations represent are highlighted in red.

The research team further improved this novel solution, particularly paying attention to constructing the index. The efficiency of their idea relied on a semi-external memory approach, which required less space [2]. They have additionally proposed a more optimal way of calculating the bidirectional anchors stemming from minimizers [2]. Their findings showed that anchors do address certain limitations of previous solutions and data structures for the text indexing problem. The key ingredient was storing only the reduced bd-anchors of a given input string instead of the entire full text. Their experimental results also indicated a substantial improvement in terms of both the space required by the index and the time needed for pattern searching [2]. This is the paper which the thesis uses as starting point for designing algorithms for text indexing problem.

It is worth mentioning that the research group still actively tries to optimize their index structure even to this day. Specifically, in 2024, Ayad et al. [3] released another refinement to their last solution, which focuses on the computation of anchors using Karp-Rabin fingerprints [3].

2.3 Preliminaries

Tries. A trie is a tree data structure storing a set of strings, where each edge has exactly one character associated with it, and every root-to-leaf path represents one of the stored strings [18, 61]. However, this structure can become quite space-intensive, as saving only one character per edge becomes quickly expensive, especially when there is a high number of strings each with hundreds of characters. Hence, a compact trie is an improved version of the regular trie, where paths of nodes with only one child are merged under one edge combining all the "single" characters [18, 61].

Kd-trees. A *kd*-tree is a binary tree at core, which is used to efficiently store and organize a set of points in a k-dimensional space [4, 10, 57]. To do so, it recursively divides the space based on the k dimensions/coordinates [4, 10, 57]. For instance, considering the case of the 2 dimensional space, the *kd*-tree becomes a 2*d*-tree, where the root might be splitting the nodes based on the x-coordinate while its *left* and *right* child nodes further partition the space based on the y-coordinate [10]. At a high-level overview, querying the *kd*-tree for the set of points in a given range r requires traversing the tree while checking for regions which overlap with or are fully contained in r [4, 57].

Longest Common Preffix (LCP). For two given strings T and S, LCP is the longest string which is a prefix for both T and S [33]. The prefix of some string S is then defined as any substring in S which starts at its beginning and ends at any point within S or coincides with S [12]. It is mathematically expressed as $S[1...i]$, where 1 represents the first position in S, so the beginning, and i the endpoint of the prefix in S with the property that $1 \leq i \leq |S|$. As opposed to this, the suffix of a string S is the substring of S which ends in the same way as S and starts at any position prior to the last one [12]. Formally speaking, it is written as $S[i...|S|]$, where i, this time, represents the position in S where the suffix starts with $1 \leq i \leq |S|$.

3 Base Algorithm

The algorithm proposed by Ayad et al. [2] first determines the reduced bd-anchors for the given text and then uses them to build a sparse data structure for fast pattern matching. Their solution further relies on the assumption that patterns cannot have a length lower than a certain integer l, $l \geq 1$ [2]. From this point forward, the terms "reduced bd-anchors", "anchors" and "bd-anchors" are used interchangeably.

3.1 Anchors Computation

Anchors are identified in linear time based on a crucial insight: the reduced bd-anchors represent a subset of minimizers [2]. Instead of brute-force calculating the anchors, minimizers could therefore be used to reduce the search space and filter out the unwanted positions. To do so, parameters l and r must be set to $w + k - 1$ and $k - 1$, respectively [2].

3.2 Index Data Structure

The composite index proposed by Ayad et al. [2] consists of two compact tries and a 2D range data structure.

The compact tries contain the substrings defined by the previously determined anchored positions in the text. More specifically, the first compact trie holds the suffixes beginning at these anchors whereas the second one includes the reversed prefixes ending at those very same locations. Alternatively, the trie for suffixes could be labeled as *x_trie* and the trie for prefixes as *y_trie*.

The 2D range data structure is intuitively constructed starting from the set of reduced bd-anchors. Informally, it "unites" the information found in the two compact tries. The data structure stores the set of points (x_a, y_a) for all reduced anchors a, where the x-coordinate represents the lexicographical rank of the suffix starting at position a in the *x_trie*, and the y-coordinate suggests the corresponding rank of its reversed prefix ending at a in the *y_trie*. Figure 4 better illustrates this connection between the coordinates of the points in the 2D range structure and the leaves of the compact tries.

Taken as a whole, the index could be viewed as a "coordinate system" over the set of reduced bd-anchors. The compact tries (and their leaves) hence represent the x- and y-axis while the 2D range structure holds the exact set of points.

3.3 Pattern Matching

Within this setup, pattern matching becomes a straightforward task, which requires only three operations: two trie traversals followed by one 2D range query. Algorithm 1 shows the concrete steps.

The process starts by splitting the pattern P in two, P_x and P_y, around its reduced bd-anchor a (*lines 1, 2, 3*) and then searching for these two separate parts in their corresponding tries, *x_trie* and *y_trie*, respectively (*lines 4, 5*). The reduced bd-anchor a is, however, not determined for the entire pattern, but only for its first l characters (*line 1*). P_x represents the suffix of P starting at a (*line 2*), while P_y is the reversed substring ending at a in P (*line 3*). The search in the tries returns two separate ranges (*x_*range and *y_*range) "on the x- and y-axes" of where P_x and P_y might occur in the original text (*lines 4, 5*). Here, the range represents the interval of ranks for the leaves in the compact tries under the node where the traversal ends. The two individual searches are essential, as they guarantee that both preceding and succeeding positions around the anchor are considered. Figure 5 better depicts this idea.

Algorithm 1 Pseudocode for the base algorithm of Ayad et al. [2]

Require: $l, r, P, |P| \geq l$
Ensure: *occurences* are the starting positions of P in the indexed text

1. $a \leftarrow$ reduced bd-anchor of $P[1...l]$
2. $P_x \leftarrow P[a...|P|]$
3. $P_y \leftarrow \overleftarrow{P[1...a]}$
4. $x\_range \leftarrow \text{SearchRange}(x\_trie, P_x)$
5. $y\_range \leftarrow \text{SearchRange}(y\_trie, P_y)$
6. $points \leftarrow \text{RangeLookup}(x\_range, y\_range)$
7. $occurrences \leftarrow [p - a \text{ for } p \text{ in } points]$
8. return *occurences*

Finally, the 2D range query correlates the two matches $x_$range and $y_$range returned by the trie traversals in an attempt to find all occurrences of the pattern (*line 6*). In simpler words, it "connects" the two halves of the pattern by looking up points in these ranges. Once the corresponding anchored positions in the given ranges have been found in *line 6*, the starting positions of the pattern in the main text are determined by subtracting a from these locations (*lines 7*).

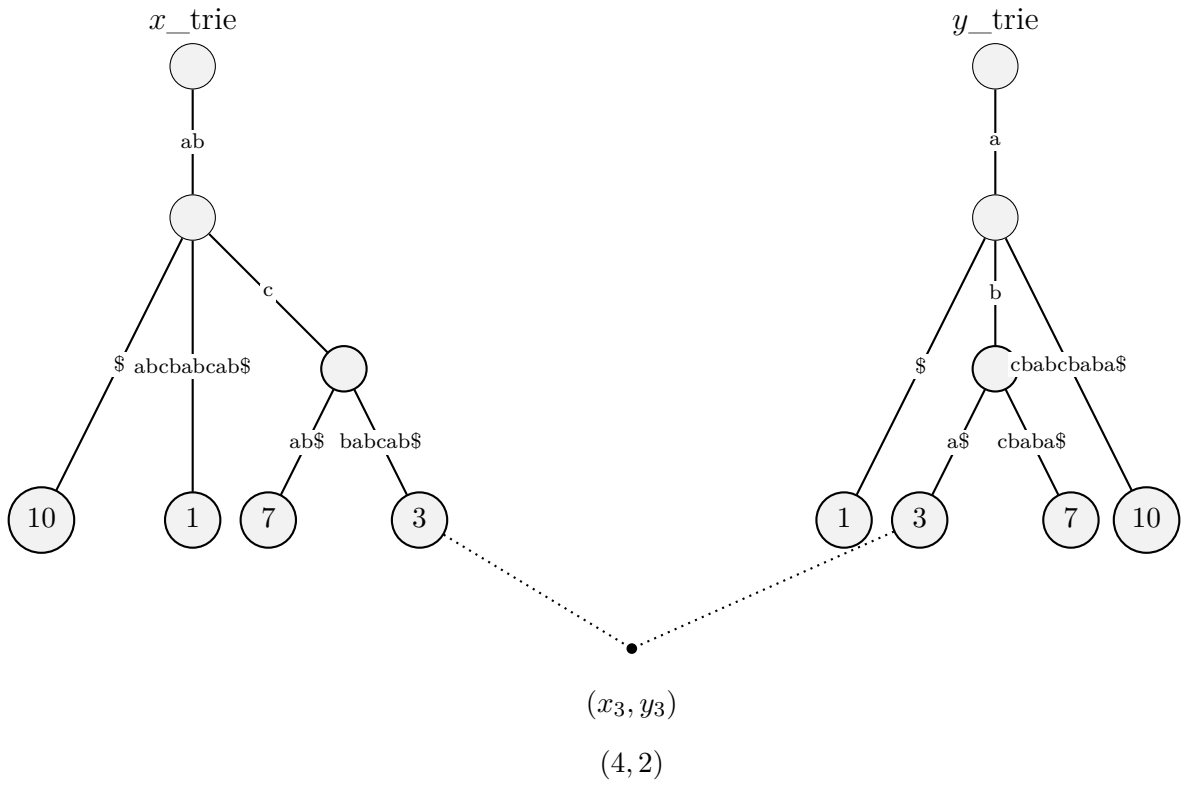


Figure 4: Correlation between the compact tries, $x_$trie and $y_$trie, and the points (x_a, y_a) within the 2D range structure. Here, the string to be indexed is "*ababcbabcab*", with $l = 5$, and $r = 1$. The reduced bd-anchors for "*ababcbabcab*" are $\{1, 3, 7, 10\}$, under the assumption that the string starts at position 1 not 0 for simplicity purposes. As it can be seen, $x_$trie holds the suffixes in "*ababcbabcab*" starting at the anchored positions (*ab*, *ababcbabcab*, *abcab*, *abcbabcab*) while $y_$trie has the prefixes ending at those positions but in reversed order (*a*, *aba*, *abcbaba*, *acbabcbaba*). Point (x_3, y_3) suggests the coordinates in the 2D range data structure for anchor 3 with $x_3 = 4$, since the suffix starting at position 3 is the fourth one in $x_$trie lexicographically speaking, and $y_3 = 2$, as the string representing anchor 3 is the second one in $y_$trie.

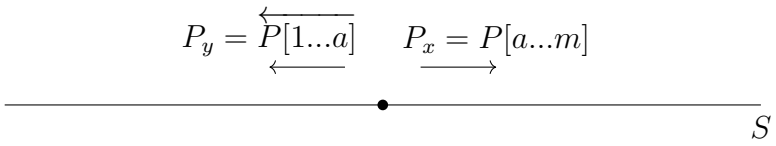


Figure 5: Visualization of individual searches for a pattern P with $|P| = m$ in S around anchor a

3.4 Correctness

Theorem 3.1 (Correctness)**.** *The algorithm proposed by Ayad et al. [2] correctly identifies all pattern appearances using anchors.*

Proof. The proof comes in two steps and follows the core properties of anchors. The first one shows that any pattern occurrence is associated with one anchor, while the second part proves that anchors maintain their (relative) arrangement.

Let S be a string and P some pattern provided dynamically such that $|P| \geq l$, for some $l \geq 1$.

Part 1. The goal here is to demonstrate that there exists at least one anchor for each appearance of P in S. To do so, consider a substring T of length l starting at position k in S where P also occurs, as indicated in figure 6.

Suppose j_k is now the reduced bd-anchor for this T. Then, according to the definition presented in section 2.2, $k \leq j_k \leq k + l - r - 1$, for some r, as j_k must be the starting position of the smallest rotation within T. However, the length of P cannot be lower than l, that is $|P| \geq l$. This relation further yields that $k + l - 1 \leq k + |P| - 1$. Putting it all together, the following inequality is reached: $k \leq j_k \leq k + |P| - 1$ (since $k + |P| - 1 \geq k + l - r - 1$), suggesting that j_k indeed lies within pattern P. Figure 6 better illustrates this idea.

Part 2. This last step aims to show that the relative positions of anchors are preserved across identical strings. Figure 7 captures this desired end goal. To demonstrate this, consider $S[k \ldots k + |P| - 1]$ as an instance of P in S. Then, for $P[1 \ldots l]$, we obtain $S[k \ldots k + l - 1]$. Next, let j be the reduced bd-anchor for $P[1 \ldots l]$, with the following property $1 \leq j \leq l - r$ according to the definition in section 2.2.

Since strings $P[1 \ldots l]$ and $S[k \ldots k + l - 1]$ are equal, their rotations must also be the same. Considering that j is the anchor for $P[1 \ldots l]$ and hence its minimal rotation, then $k + j - 1$ should correspond to the minimal rotation for $S[k \ldots k + l - 1]$. The latter means that $j_k = k + j - 1$ is the reduced bd-anchor for $S[k \ldots k + l - 1]$. □

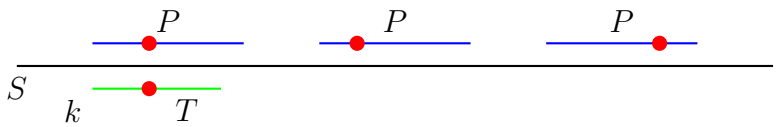


Figure 6: Part 1 of proof: each pattern appearance has at least one anchor

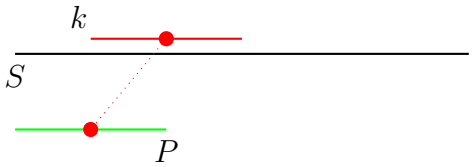


Figure 7: Part 2 of proof: anchors maintain their relative location

4 Algorithm 1

4.1 Data Structure

Algorithm 1 makes use of augmented compact tries [11] and a naive 2D range data structure [4].

In the compact trie, each node contains a dictionary for its outgoing edges, where the path label is stored as the key and the corresponding child as the value. The nodes are enhanced with additional data such as ranks and ranges, thus facilitating the search process. Ranks are optional and mostly intended for leaves, as internal nodes only store the range of the ranks found in their subtree. Figure 8 better depicts this setup.

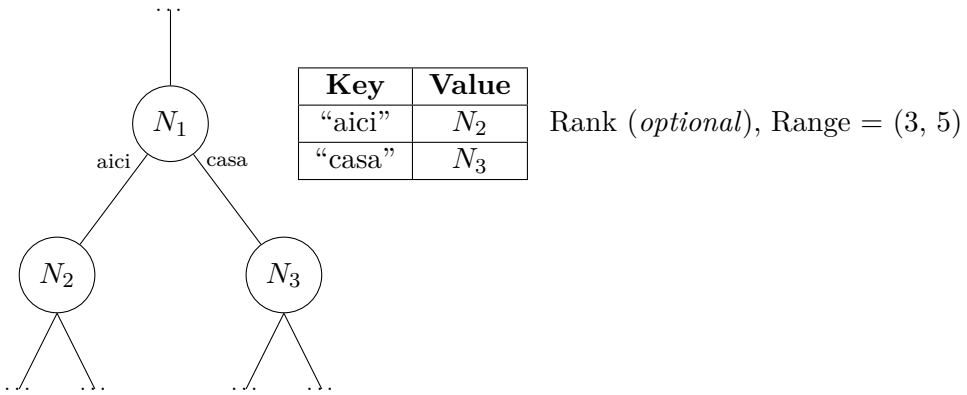


Figure 8: Illustration of a trie node N_1 for algorithm 1. Here, N_1 has two outgoing edges labeled "aici" and "casa" leading to children N_2 and N_3, respectively. The dictionary found at N_1 contains as keys the edge strings "aici" and "casa" and as values their corresponding child nodes. Besides this, node N_1 stores additional data: a Rank and a Range. The Rank is optional, since N_1 is an internal node while the Range= $(3, 5)$ shows that the subtree of N_1 contains the strings lexicographically ranked between the 3^{rd} and 5^{th} places.

The 2D range data structure consists of two suffix arrays, x_array and y_array, corresponding to leaves in the compact tries. Specifically, the arrays store the reduced bd-anchors in lexicographical order based on the substrings they represent. Figure 9 illustrates the underlying relation between the x_array and the x_trie and the y_array and the y_trie.

4.2 Preprocessing

The composite data structure in algorithm 1 can be built instinctively by first computing the reduced bd-anchors for the given text. In our case, this is done inefficiently through a brute-force approach, where each substring of length l is exhaustively checked.

Then, the suffix arrays, x_array and y_array, are built over the anchored positions by naively sorting these locations based on the suffixes and reversed prefixes they represent.

Finally, the suffix x_array and y_array are used to build the compact tries. Specifically, we insert into the x_trie the suffixes beginning at the anchors in the x_array. Similarly, we add the reversed prefixes ending at the locations found in the y_array to the y_trie.

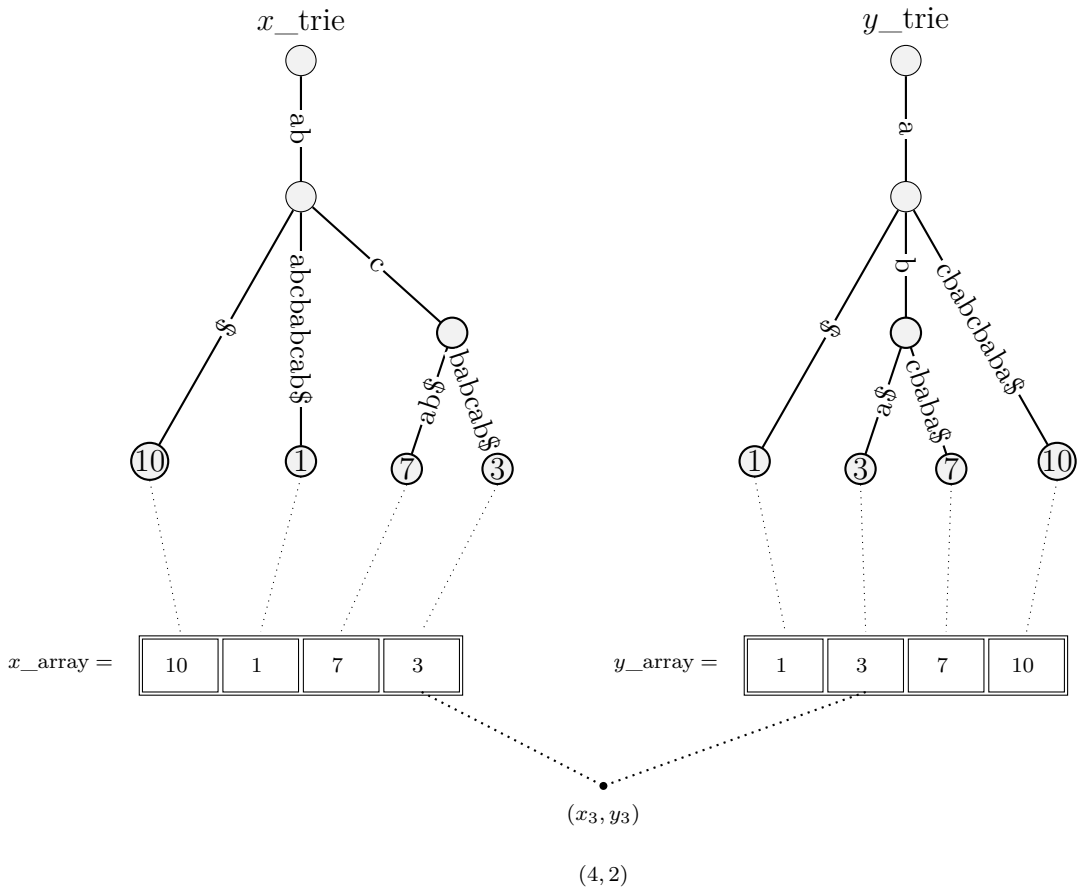


Figure 9: Connection between the compact tries, $x_$trie and $y_$trie, and the naive 2D range data structure consisting of $x_$array and $y_$array for algorithm 1. This reuses the example string "*ababcbabcab*" with variables $l = 5$ and $r = 1$ as depicted in figure 4. Here, we can better see how the 2D range structure of algorithm 1 facilitates the coordinate mapping process for the points. We can also easily determine the x-coordinate of an anchor by looking at its index in the $x_$array. For instance, anchor 3 has as x-coordinate 4, since it is the fourth leaf in the compact trie $x_$trie and the fourth position in the $x_$array. This is similar for the y-coordinate.

4.3 Query

The data structure is queried in a similar manner as the one described for the base algorithm in section 3.3.

The reduced bd-anchor for the prefix $P[1...l]$ is first calculated for a pattern P provided online at runtime. Then, P is broken down into two parts, P_x and P_y, respectively, as previously described in section 3.3. The first part P_x is used for searching in the compact $x_$trie for the corresponding range $x_$range. Similarly, P_y serves as input for traversing the $y_$trie looking for the $y_$range.

Querying the compact trie essentially involves comparing the given string against the available paths. Choosing a branch to follow includes iteratively going through a node's dictionary and each time performing a LCP query to determine if the string matches that branch. If the LCP query returns a match, we proceed along that path and repeat the process with the corresponding child node. Otherwise, if no match is found, the pattern is then not included in the trie and we return the range $(-1, -1)$. When the search ends on an internal node, we simply return its associated range. If it ends at a leaf node, then we return the rank of that leaf as an interval of one point.

The range lookup is done *"on the spot"* by:

1. **Searching the x-coordinate:** collecting the points in $x_$array that are within the $x_$range
2. **Searching the y-coordinate:** collecting the points in $y_$array that are within the $y_$range
3. **Combining the results:** returning the set of points that lie in both $x_$range and $y_$range ranges

Example. Let $x_$range= (x_1, x_2) and $y_$range= (y_1, y_2) be the ranges returned by the trie traversals. Then, step 1 involves looking in the suffix $x_$array for anchors with their ranks falling within (x_1, x_2). The second part similarly selects the anchors in $y_$array with their y-coordinate in (y_1, y_2). Finally, the last step looks at the anchors from phases 1 and 2, which lie in both ranges, $x_$range and $y_$range, respectively.

4.4 Complexity Analysis

Space. The space complexity of algorithm 1 is influenced by the space of the $x_$trie and $y_$trie as well as of the 2D range structure consisting of $x_$array and $y_$array, as indicated in equation 1.

$$\mathcal{O}(x\_trie) + \mathcal{O}(y\_trie) + \mathcal{O}(x\_array) + \mathcal{O}(y\_array) \tag{1}$$

The compact tries contain a number of strings equal to the one of reduced bd-anchors. Let S be the text to be indexed and A the number of anchors determined for S, for some l, r. Then, a compact trie for A substrings contains A leaves and one root node [18]. Informally, a tree with A leaves has $\mathcal{O}(A)$ nodes [18]. Since an internal node corresponds to an edge splitting to other children, then the compact tries have at most $\mathcal{O}(A)$ edges. The combined dictionaries found at each node take $\mathcal{O}(A)$ space, because the number of entries in all dictionaries is equal to the number of all edges in the trie. In addition to this, the augmented information (ranges and ranks) requires constant space per node. Since the dictionaries explicitly store all edge strings as keys, we also need to consider the lengths of all such edge labels, which is $\mathcal{O}(|S| \cdot A)$. That is because we insert A suffixes and their lengths cannot exceed the length of the main text S which they come from. Summing it all up, the space complexity of one compact trie becomes $\mathcal{O}(|S| \cdot A + A) = \mathcal{O}(|S| \cdot A)$.

The $x_$array and $y_$array are built over the set of anchors, and hence have a length equivalent to A.

Substituting the above complexities in the equation in 1, the data structure of algorithm 1 has a space complexity of $\mathcal{O}(2 \cdot |S| \cdot A + A) = \mathcal{O}(|S| \cdot A)$.

Time. As for pattern matching, the time complexity is simplified as shown in expression 2.

$$\mathcal{O}(2 \cdot \text{trie traversal}) + \mathcal{O}(\text{2D range lookup}) \tag{2}$$

One traversal in $x_$trie needs $\mathcal{O}(|P_x| \cdot c \cdot min(e_x, |P_x|))$ time. Here, c is the maximum children's number per node, e_x the longest edge label length, and P_x the query pattern. Algorithm 1 traverses through all child nodes of a given node using its dictionary, calculating the LCP for each dictionary entry. The LCP operation then involves comparing the current edge label against pattern P_x until a mismatch is found. Since the LCP is called for up to c children per single node, then the total time spent on a node is $\mathcal{O}(c \cdot min(e_x, |P_x|))$. This process is repeated approximately $|P_x|$ times, as the entire pattern needs traversing and, in the worst

case, it advances with one character per iteration. Returning the associated range of a node requires constant time, as it is a simple lookup. Compact trie y_trie is traversed in a similar manner in $\mathcal{O}(|P_y| \cdot c \cdot min(e_y, |P_y|))$ time.

For the range query, finding the points within a given range $x\_range = (x_1, x_2)$ on the x-axis and $y\_range = (y_1, y_2)$ on the y-axis requires $\mathcal{O}(x_2 - x_1 + 1)$ and $\mathcal{O}(y_2 - y_1 + 1)$ time, respectively. The intersection of these points then takes $\mathcal{O}(A^2)$, as we need to go through each one of them and check whether they lie in both ranges using a nested loop. In the worst case, all points in the $x_$array and $y_$array are contained in both ranges, whose lengths correspond to the number of anchors.

By inserting all components into equation 2, we derive the subsequent time complexity for algorithm 1:

$$\mathcal{O}(|P_x| \cdot c \cdot min(e_x, |P_x|)) + \mathcal{O}(|P_y| \cdot c \cdot min(e_y, |P_y|)) + \mathcal{O}(A^2)$$

$$= \mathcal{O}(|P_x| \cdot c \cdot min(e_x, |P_x|) + |P_y| \cdot c \cdot min(e_y, |P_y|) + A^2)$$

An internal node can have a maximum number of children bounded by the number of different characters that the alphabet holds [61]. If the alphabet size is considered to be constant, then we have $c = \mathcal{O}(1)$. Considering this, the final time complexity becomes:

$$\mathcal{O}(|P_x| \cdot min(e_x, |P_x|) + |P_y| \cdot min(e_y, |P_y|) + A^2)$$

. When e_x and e_y are quite large, the equation is reduced further to

$$\mathcal{O}(|P_x|^2 + |P_y|^2 + A^2)$$

.

5 Algorithm 2

5.1 Data Structure

The data structure for algorithm 2 consists of the two augmented compact tries, x_trie and y_trie, and a kd-tree [4, 44] (with parameter $k = 2$) as the main 2D range structure. The compact tries are the same as the ones described in section 4.1 for algorithm 1 and the $2d$-tree is built over the same points (x_a, y_a), as explained in section 3.2.

5.2 Preprocessing

The data structure is built in a similar manner as the one in algorithm 1. Essentially, x_array and y_array are first computed for the reduced bd-anchors of the given text to be indexed. Then, they are used in inserting strings in the compact tries x_trie and y_trie, respectively. This is all better explained in section 4.2. Apart from this role, algorithm 2 uses these arrays to also compute the set of points (x_a, y_a) required for the $2d$-tree. Specifically, for each anchor a located at x_a in x_array, preprocessing requires finding the corresponding position y_a of a in y_array. Once the set of points are determined, we simply insert them one by one in the $2d$-tree.

5.3 Query

Algorithm 2 performs search queries as presented in 3.3 by first traversing the compact x_trie and y_trie using P_x and P_y to retrieve their corresponding ranges, x_range and y_range. This process is also the same as the one in section 4.3.

However, what changes this time is the naive 2D range lookup, which no longer filters out the points individually on the x- and y-axes and then combines the results. Instead of this, algorithm 2 traverses the $2d$-tree in search of the points located within the x_range and y_range. The traversal is done as explained in the preliminaries section 2.3.

5.4 Complexity Analysis

Space. The space complexity remains defined by the equation in 1 from section 4.4. The space complexity of the compact tries stays the same as in algorithm 1, which is $\mathcal{O}(|S| \cdot A)$, where A is the number of reduced bd-anchors for the text S. The kd-tree requires $\mathcal{O}(n)$ space, where n is nothing else but the number of points stored in it [4, 16, 44]. Since the number of points in our $2d$-tree corresponds to the number of reduced bd-anchors, the total space complexity for the composite data structure in algorithm 2 becomes $\mathcal{O}(|S| \cdot A)$.

Time. The query time is dictated by the equation in 2 in chapter 4.4. Once again, since the augmented trie structures, x_trie and y_trie, did not undergo any changes, their traversal times remain $\mathcal{O}(|P_x| \cdot c \cdot min(e_x, |P_x|))$ and $\mathcal{O}(|P_y| \cdot c \cdot min(e_y, |P_y|))$ in algorithm 2, where parameters c, e_x, e_y are explained in the early section 4.4. Performing a 2D range query now involves a $2d$-tree traversal using ranges x_range and y_range. The kd-tree is queried in $\mathcal{O}(\sqrt{n} + occ)$ time, for $k = 2$ and with n as the number of points stored in the kd-tree, and occ as the number of points found lying in the given range [4, 44, 57]. Putting it all together, we obtain a total of $\mathcal{O}(|P_x| \cdot c \cdot min(e_x, |P_x|) + |P_y| \cdot c \cdot min(e_y, |P_y|) + \sqrt{A} + occ)$ time for pattern matching. If the alphabet size is constant, we further obtain $\mathcal{O}(|P_x| \cdot min(e_x, |P_x|) + |P_y| \cdot min(e_y, |P_y|) + \sqrt{A} + occ)$. Finally, if the maximal string lengths e_x and e_y of an edge are significantly large, we can simply the equation further to $\mathcal{O}(|P_x|^2 + |P_y|^2 + \sqrt{A} + occ)$.

6 Algorithm 3

6.1 Data Structure

The composite data structure of algorithm 3 includes again the two augmented compact tries and a kd-tree (with $k = 2$) over the set of reduced bd-anchors.

The nodes in the tries still store the same satellite data (ranks and ranges) as presented in algorithms 1 and 2, but with a minor modification in the way the dictionary holds its elements. Rather than using the entire edge string as the key (as presented in algorithms 1 and 2), the dictionary now stores only the first character. Furthermore, the values are no longer limited to only child nodes. Additionally, the values in the dictionary store the remaining edge strings without their first character. Figure 10 shows this idea.

The $2d$-tree remains the same as the one in algorithm 2 from section 5.1.

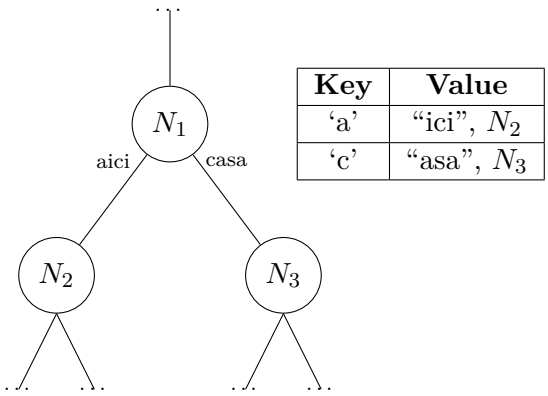


Figure 10: Changes in the dictionary found in the trie node N_1 for algorithm 3. This serves as an extension of figure 8 from algorithm 1. For clarity reasons, the satellite data, Rank and Range, are omitted now in the current figure. The dictionary of N_1 now stores as key the first characters 'a' and 'c' of its edge strings, while the dictionary values store the rest of the edge labels, "ici" and "asa", besides the corresponding children, N_2 and N_3.

6.2 Preprocessing

Preprocessing the data structure of the third algorithm does not change much from the process presented for algorithm 2 in section 5.2. The only difference lies in the way the compact tries store internally the edge information.

Once again, it implies computing the reduced bd-anchors and creating the x_array and y_array by sorting these selected positions. The arrays are then used as input not only to create the x_trie and y_trie, but also the set of points required for the $2d$-tree.

6.3 Query

Algorithm 3 queries the data structure by first traversing the tries x_trie and y_trie, respectively, followed by a traversal of the $2d$-tree.

Querying the compact tries x_trie and y_trie requires following a path from top to bottom which coincides with the online patterns P_x or P_y. However, in comparison with algorithms 1 and 2, the next branch to follow in the trie is chosen based on one dictionary lookup for the first character. This quick retrieval helps determine whether there even exists a (partially)

matching path to follow, thus sparing us from exhaustively checking all potential branches to take. The dictionary lookup for the first character returns the remaining edge string and its child node. Then, a LCP query is performed on the remainder edge string and the currently considered pattern. Based on the value returned by LCP, we either continue down the path and repeat the process on the child node, or we conclude that the pattern is not included in the compact trie and simply return $(-1, -1)$ as range. If the pattern is fully matched against the branches of the trie, then we return the range found at the node where the search ends (either $x_$range or $y_$range based on the trie).

Using the found ranges, algorithm 3 performs a $2d$-tree traversal for the set of points within $x_$range and $y_$range. This is done by consecutively narrowing down the search area using intersection and recursion [4], as indicated in chapter 2.3.

6.4 Complexity Analysis

Space. Following equation 1 in section 4.4, the total space complexity of the data structure in algorithm 3 is $\mathcal{O}(|S| \cdot A)$, where A is the number of anchors calculated for the input text S.

Although the internal structure of the compact tries is changed, the edge labels are still explicitly stored in each dictionary entry. The difference is that they do not represent keys anymore, but values. Therefore, we still need to take into consideration the total length of all suffixes in the compact tries. Just like in algorithm 1, this corresponds to $\mathcal{O}(|S| \cdot A)$, as the A suffixes cannot be longer than their "parent" string S. The augmented data still requires $\mathcal{O}(1)$ per node, and there are $\mathcal{O}(A)$ edges in the compact tries as well as dictionary entries. Hence, the total space for a compact trie remains $\mathcal{O}(|S| \cdot A)$, as in the previous algorithms 1 and 2.

The $2d$-tree has the same structure as in algorithm 2, with a space complexity of $\mathcal{O}(A)$.

Time. The query time of the data structure in algorithm 3 continues to be dictated by equation 2 in section 4.4. The main idea shares similarities with the approach discussed in algorithm 1. However, the main change lies in the compact trie traversal. This modification stems from the internal dictionary no longer storing the entire edge string explicitly as a key, but rather using its first character instead, which greatly simplifies pattern matching.

We present next the case for $x\_$ trie. The trie search operation performs one dictionary lookup and one LCP query per node. Unlike before, it now checks only one edge per node instead of all children. The dictionary lookup for the first character is constant and the LCP query between the edge string and the remaining suffix of the query pattern P_x takes $\mathcal{O}(min(e_x, |P_x|))$, where e_x is the maximum length of any edge string and P_x the pattern to be queried. We must repeat this process for each node on the path while all characters in P_x are matched. Returning the ranges found at the nodes where the search ends at requires constant time. In total, querying $x_$trie takes $\mathcal{O}(|P_x| \cdot min(e_x, |P_x|))$ time. This is analogous for compact trie $y_$trie.

The $2d$-tree traversal does not change and still takes $\mathcal{O}(\sqrt{A} + occ)$ time. Here, parameter occ is the number of points located within ranges $x_$range and $y_$range.

Putting it all together, we get the following time complexity for our data structure:

$$\mathcal{O}(|P_x| \cdot min(e_x, |P_x|)) + \mathcal{O}(|P_y| \cdot min(e_y, |P_y|)) + \mathcal{O}(\sqrt{A} + occ)$$

If e_x and e_y are again quite large, then the above equation could be simplified to

$$\mathcal{O}(|P_x|^2) + \mathcal{O}(|P_y|^2) + \mathcal{O}(\sqrt{A} + occ) = \mathcal{O}(|P_x|^2 + |P_y|^2 + \sqrt{A} + occ)$$

7 Experimental Evaluation

7.1 Setup

The tests were performed on a computer with a 2.70 GHz processor and 16.0 GB RAM. The data was collected in the following manner: the query times were recorded using the Stopwatch class [54] provided by C# due to its high accuracy. The time was measured in milliseconds (ms) from the moment the pattern was split around its anchor until the starting positions of all occurrences were returned.

As for the index space, the memory usage was measured in megabytes (MB) by storing the data structures in separate json files [26] and logging only the logical sizes with the amount of data saved. This decision was influenced by the limited choices of libraries that would support such functionality in C#. Available options included *Marshal.SizeOf<T>()* [38] or *GC.GetTotalMemory(true)* [15] functions for tracking the memory usage of a *struct* type or of the heap size, respectively. However, the first approach was not entirely suitable for this study, as the prototypes only used public classes and not *structs* while the second one did not clearly distinguish between instance size and additional memory usage. Even though measuring the logical file sizes could appear to be a bit imprecise at first glance, it still presents a pretty good estimation of the actual index space. Therefore, this workaround was still found to be a suitable choice for the thesis.

For testing, 1000 patterns were randomly generated for each different value of the parameter l. This variable represented the minimum length of a pattern and was hence set to $100, 200, 500, 700, 1000$ and 1200 for each dataset. As indicated in the paper, r was further determined by the equation $\lceil 4 \cdot log(l)/log(\sigma) \rceil$ [2], where σ is the size of the alphabet.

7.2 Datasets

The datasets chosen for this thesis corresponded to the first three sets used in the original paper by Ayad et al. [2], namely DNA, PROTEINS, and XML [14, 48]. This selection allowed for a fair comparison between the results of this study and the ones obtained in the latter research paper.

However, due to the extremely large size of the test data and the existing time constraints, the datasets were adjusted accordingly to facilitate a smooth experimental phase. Specifically, their lengths, and implicitly their alphabets, were reduced to accommodate the hardware and computational limitations, as shown in table 1.

Dataset	Size	Alphabet Size
DNA	100 KB	4
PROTEINS	100 KB	20
XML	100 KB	84

Table 1: Overview of datasets used in experiments

7.3 Implementation Details and Technological Stack

The algorithms for this study were implemented in C# programming language [6] using Visual Studio Code [58] as the main editor due to several benefits. To begin with, they are widely spread tools whose popularity has increased tremendously over the past few decades [7, 56, 59]. Considering this, there is a large body of documentation available online for troubleshooting purposes [6, 40].

Secondly, C# is an object-oriented language at its core, which supports custom data structures through classes and objects [39]. This was particularly advantageous in the thesis' case, as it allowed for defining complex and augmented structures, like tries or trees, when building the index. Additionally, it provided useful plugins and libraries [6]. Specifically, the *kd*-tree was integrated in code using the NetTopologySuite [45] library provided by .NET, which deals exclusively with data in space. This approach was chosen over building the *kd*-tree from scratch because it represented a reliable and widely-recognized tool with high performance and fast speed [46]. Doing so helped in achieving the best query time possible as well as minimizing any potential delays in pattern matching.

Last but not least, the familiarity with C# from the author's side represented another important aspect that was considered when choosing the technological stack for this study. The existing expertise thus helped not only in speeding up the development pace but also in smoothing out the debugging process.

As far as memory allocation is concerned, the I/O model [63] falls outside the scope of this study for simplicity reasons. Thus, the algorithm implementation adopted neither an external nor a semi-external approach.

7.4 Evaluation Metrics

The thesis focused on evaluating the data structure proposed by Ayad et al. [2], leaving its preprocessing aspects outside the scope of this study. Hence, the primary metrics of focus were the index's time and space requirements. The size of the computed anchors was also noted.

7.5 Data Analysis

The collected data was first normalized and standardized to ensure consistency and accuracy. Following this, the time averages were next calculated per each unique value of l. As a last step, Excel sheets were created for each individual dataset, containing an overview of the most significant pieces of information such as the number of anchors, times and space.

RStudio version 4.4.1 [50] was then used to create meaningful graphs from the recorded data in spreadsheets to better visualize it. This contributed significantly to identifying certain areas of improvements for the upcoming iterations.

7.6 Results

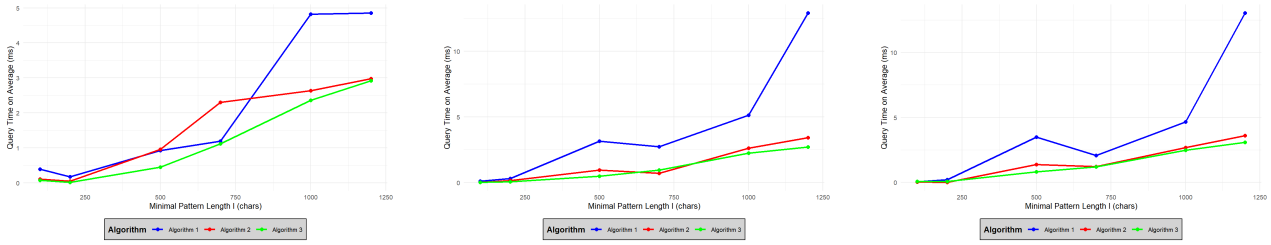
Algorithms 1, 2, and 3 (from sections 4, 5, 6, respectively) were tested against the datasets in section 7.2. Their experimental results are next presented in figures 11, 12, and 13.

Average Time for Pattern Matching We observe in figure 11 that algorithm 1 records the longest times on average for search queries regardless of the dataset, having the poorest performance. These results can be, of course, explained by the naive design of the data structures, especially for the 2D range one, which was represented by the two arrays $x_$array and $y_$array. As opposed to this, algorithms 2 and 3 present improved querying times, which are quite comparable. There are, however, few exceptions for the DNA dataset (figure 11a), where algorithm 3 registers faster times for pattern matching. Therefore, algorithm 3 exhibits the best performance out of all, while algorithm 2 falls in between with moderate efficiency. For instance, for PROTEINS dataset, when $l = 1000$, algorithm 2 presents an improvement of 49.21% in query times compared to algorithm 1. Following this, algorithm 3 had an improvement of 14.23% compared to algorithm 2 and of 56.44% compared to algorithm 1.

Space for Data Structure The data structures of algorithms 1, 2 and 3 presented extremely close space complexities in value. Figure 12 supports this idea through its overlapping graphs, indicating that the space of the three data structures did not vary by much. Besides this, we see that the space complexity decreases for all three algorithms while the value of parameter l goes up. This reduction in the data structures' space validates the findings of Ayad et al. [2], on which the thesis relied.

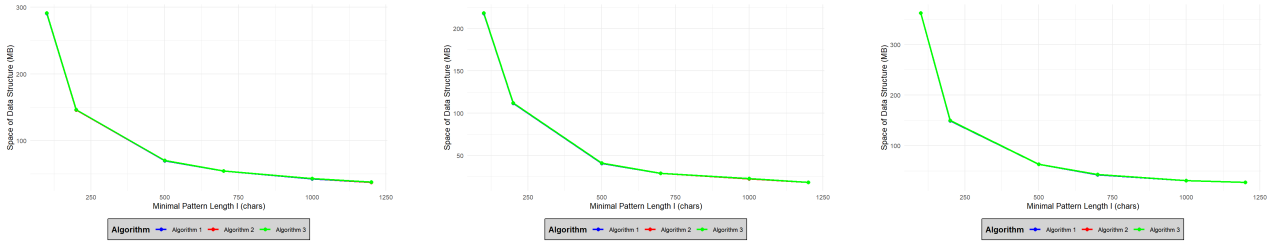
Anchors Computation We can see in figure 13 that the number of reduced bd-anchors does decrease as l increases. This is somehow intuitive, as the higher the bound l goes, the less anchors are generated by the algorithms.

Remarks The data structure in algorithm 3 required less preprocessing time than the rest, which took hours to be built. This was possible due to the internal trie structure presented in algorithm 3, which made child access constant.



(a) DNA Dataset (b) PROTEINS Dataset (c) XML Dataset

Figure 11: Results of the algorithms 1, 2, 3 for DNA, PROTEINS, XML datasets for query times considering lower bound l



(a) DNA Dataset (b) PROTEINS Dataset (c) XML Dataset

Figure 12: Results of the algorithms 1, 2, 3 for DNA, PROTEINS, XML datasets for structure space considering the lower bound l

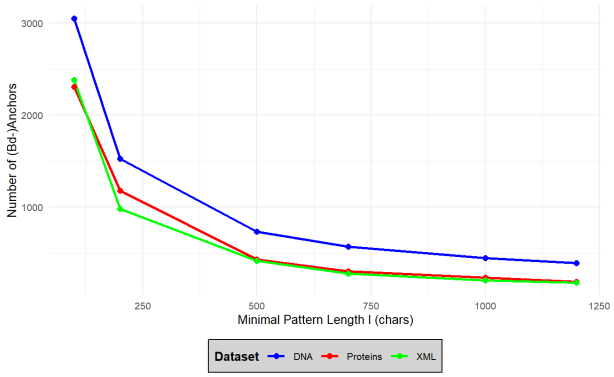


Figure 13: Number of (bd-)anchors generated by algorithms 1, 2, 3 per dataset considering the lower bound l

8 Conclusion

This thesis created and presented three algorithms using the work of Ayad et al. [2], which presented an innovative index for fast searches when the length of patterns become quite long. The first algorithm relied on augmented compact tries and on a naive 2D range data structure for queries, which proved quite slow in practice. The second algorithm maintained the compact tries, but used instead a *kd*-tree as the main 2D range structure, thus significantly reducing the search times. The last solution was similar to the second, however, with a key modification in the internal structure of the compact tries, which did not improve the performance by much. All three algorithms measured similar space complexities.

All in all, the presented variants still need improvement, and could therefore act as a new starting point for other studies or projects.

9 Limitations and Challenges

A first limitation of this thesis was the restricted number of available community projects and benchmarks in C# for the text indexing problem. This made it challenging and impossible to compare the algorithms in this thesis with other existing approaches, such as the suffix tree [18] or FM-index [13, 31].

A major challenge was also the practical evaluation on large datasets which presented some difficulties due to computational reasons. Specifically, using the original files of size $200MB$ for experiments was not entirely feasible due to the limited computer resources and existing preprocessing inefficiency.

Another limitation of this study was the programming language, which could have influenced the experimental efficiency of the three algorithms. Although C# uses a garbage collector [41] for an easy memory handling "*under the hood*", it can also create some performance overhead occasionally [8, 60]. As opposed to this, C++ requires manual memory management, which can turn to be quite beneficial in improving performance aspects [8, 60].

Finally, while the solution of Ayad et al. [2] relied on sdsl-lite library [17] for succinct data structures, the algorithms in the current thesis did not. This, of course, had a negative impact on the overall efficiency of the algorithms.

10 Future Work

The work on the current project is far from complete and still requires some future work. Hence, further studies could begin with extensively testing the presented algorithms against even larger datasets in order to be able to create a more complete "profile" of their characteristics and bottlenecks. The latter could include once again the space and time aspects of structures, sampling behavior, or perhaps memory and computational bottlenecks (especially when it comes to calculating and using anchors). The results could then show a more accurate insight into the algorithms' true performance levels.

Additionally, future research could conduct a more elaborate comparison between the algorithms of this thesis and widely-recognized solutions for the text indexing problem. This could help determine where the variants stand in relation with other indexes and even identify further areas of improvement.

Despite all efforts, the space complexity did not seem to improve too much in this project. Therefore, a great suggestion for future work could be to investigate and identify new ways of reducing the space of the structure without affecting the querying performance. One suggestion would be to store pointers to the original text in the nodes' dictionary instead of the entire edge strings to save up space.

Another interesting idea would be to replace the *kd*-tree with a range tree using the fractional cascading technique [9, 42] in order to determine whether similar practical results are reached.

Since the current thesis focused solely on the space and time of the algorithms and their data structures, a good idea for future work could be to include preprocessing within key metrics. A suitable starting point for this could be the computation of reduced bd-anchors or the building effort for the compact tries.

Moreover, the algorithms in this thesis used only reduced bd-anchors for indexing purposes. However, there are many more types of anchors at our disposal such as mod-minimizers [23], or open-closed minimizers [21]. Hence, an interesting idea would be to explore the ways these new variants of anchors could be applied to our context of text indexing.

Lastly, the parameter r was set to $\lceil 4 \cdot log(l)/log(\sigma) \rceil$ [2] during the experimental phase, with σ being the size of the alphabet and l the minimum pattern's length. Hence, it would be interesting to see how the algorithms behave when r is set to different values.

Bibliography

[1] Srinivas Aluru. *Text Indexing (1993; Manber and Myers)*. Lecture Notes. Entry editor: Paolo Ferragina; based on Manber and Myers (1993). Dipartimento di Informatica, Università di Pisa, 1993. URL: https://pages.di.unipi.it/ferragina/Teach/InformationRetrieval/TextIndexing.pdf.

[2] Lorraine Ayad, Grigorios Loukides, and Solon Pissis. "Text indexing for long patterns: Anchors are all you need". In: *Proceedings of the VLDB Endowment* 16.9 (July 2023), pp. 2117–2131. DOI: 10.14778/3598581.3598586.

[3] Lorraine A. K. Ayad, Grigorios Loukides, and Solon P. Pissis. *Text Indexing for Long Patterns using Locally Consistent Anchors*. 2024. arXiv: 2407.11819 [cs.DS]. URL: https://arxiv.org/abs/2407.11819.

[4] Philip Bille. *Range Reporting*. Class lecture slides, 02282 Algorithms for Massive Data Sets, Technical University of Denmark. Accessed on June 8, 2025, from https://www2.compute.dtu.dk/courses/02282/2024/rangereporting/rangereporting1x1.pdf. 2024.

[5] Michael Burrows and David J. Wheeler. *A Block-Sorting Lossless Data Compression Algorithm*. Tech. rep. 124. Digital Equipment Corporation, 1994.

[6] *C# - the modern, open-source programming language | .NET*. en-US. URL: https://dotnet.microsoft.com/en-us/languages/csharp (visited on 06/16/2025).

[7] *C# Popularity, Usage, and Developer Momentum in 2025 - ZenRows*. en. URL: https://www.zenrows.com/blog/c-sharp-popularity (visited on 06/16/2025).

[8] *C++ vs C#*. en-US. Section: Difference Between. URL: https://www.geeksforgeeks.org/c-vs-c-sharp/ (visited on 06/17/2025).

[9] Bernard Chazelle and Leonidas J. Guibas. "Fractional cascading: I. A data structuring technique". In: *Algorithmica* 1.1 (Nov. 1986), pp. 133–162. ISSN: 1432-0541. DOI: 10.1007/BF01840440. URL: https://doi.org/10.1007/BF01840440.

[10] Jenny Chen. *CS 225 Spring 2023: k-d Trees*. Online. Accessed: 2025-06-16, Department of Computer Science, Grainger College of Engineering, University of Illinois Urbana-Champaign. 2023. URL: https://courses.grainger.illinois.edu/cs225/sp2023/resources/kd-tree/.

[11] Thomas H. Cormen et al. "Augmenting Data Structures". In: *Introduction to Algorithms*. Third. Cambridge, MA: MIT Press, 2009. Chap. 14.

[12] Thomas H. Cormen et al. *Introduction to Algorithms, Third Edition*. Chapter 32: String Matching, pp. 985-989. Cambridge, MA: MIT Press, 2022. ISBN: 9780262033848.

[13] P. Ferragina and G. Manzini. "Opportunistic data structures with applications". In: *Proceedings 41st Annual Symposium on Foundations of Computer Science*. 2000, pp. 390–398. DOI: 10.1109/SFCS.2000.892127.

[14] Paolo Ferragina et al. "Compressed text indexes: From theory to practice". In: *ACM J. Exp. Algorithmics* 13 (Feb. 2009). ISSN: 1084-6654. DOI: 10.1145/1412228.1455268. URL: https://doi.org/10.1145/1412228.1455268.

[15] *GC.GetTotalMemory(Boolean) Method (System)*. en-us. URL: https://learn.microsoft.com/en-us/dotnet/api/system.gc.gettotalmemory?view=net-9.0 (visited on 03/23/2025).

[16] Amit Kumar Ghosh. *K-Dimensional Tree in Data Structures*. URL: https://www.scholarhat.com/tutorial/datastructures/k-dimentional-tree-in-data-structures (visited on 06/15/2025).

[17] Simon Gog. *simongog/sdsl-lite*. original-date: 2013-02-28T22:34:07Z. June 2025. URL: https://github.com/simongog/sdsl-lite (visited on 06/17/2025).

[18] Inge Li Gørtz. *Suffix Trees*. Class lecture slides, 02282 Algorithms for Massive Data Sets, Technical University of Denmark. Accessed on June 8, 2025, from https://www2.compute.dtu.dk/courses/02282/2024/suffixtrees/suffixtrees-1x1.pdf. 2024.

[19] Szymon Grabowski and Marcin Raniszewski. "Sampled suffix array with minimizers". In: *Software: Practice and Experience* 47.11 (2017), pp. 1755–1771. DOI: https://doi.org/10.1002/spe.2481. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.2481. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2481.

[20] Koerkamp Ragnar Groot. *Notes on bidirectional anchors*. en. Section: posts. Jan. 2024. URL: https://curiouscoding.nl/posts/bd-anchors/ (visited on 06/16/2025).

[21] Ragnar Groot Koerkamp, Daniel Liu, and Giulio Ermanno Pibiri. "The open-closed mod-minimizer algorithm". In: *Algorithms for Molecular Biology* 20.1 (Mar. 2025), p. 4. ISSN: 1748-7188. DOI: 10.1186/s13015-025-00270-0. URL: https://doi.org/10.1186/s13015-025-00270-0.

[22] Ragnar Groot Koerkamp and Igor Martayan. "SimdMinimizers: Computing random minimizers, fast". In: *bioRxiv* (2025). DOI: 10.1101/2025.01.27.634998. eprint: https://www.biorxiv.org/content/early/2025/05/13/2025.01.27.634998.full.pdf. URL: https://www.biorxiv.org/content/early/2025/05/13/2025.01.27.634998.

[23] Ragnar Groot Koerkamp and Giulio Ermanno Pibiri. "The mod-minimizer: A Simple and Efficient Sampling Algorithm for Long k-mers". In: *24th International Workshop on Algorithms in Bioinformatics (WABI 2024)*. Ed. by Solon P. Pissis and Wing-Kin Sung. Vol. 312. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024, 11:1–11:23. ISBN: 978-3-95977-340-9. DOI: 10.4230/LIPIcs.WABI.2024.11. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.WABI.2024.11.

[24] Roberto Grossi and Jeffrey Scott Vitter. "Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching". In: *SIAM Journal on Computing* 35.2 (2005), pp. 378–407. DOI: 10.1137/S0097539702402354. URL: https://doi.org/10.1137/S0097539702402354.

[25] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.

[26] *JSON File - What is a .json file and how do I open it?* URL: https://fileinfo.com/extension/json (visited on 06/24/2025).

[27] Ragnar {Groot Koerkamp}. *BWT and FM-index*. en. Section: posts. Oct. 2022. URL: https://curiouscoding.nl/posts/bwt/ (visited on 06/16/2025).

[28] Ragnar {Groot Koerkamp}. *Low Density Minimizers: Theory of Sampling Schemes*. en. Section: posts. Feb. 2025. URL: https://curiouscoding.nl/posts/minimizers/ (visited on 06/16/2025).

[29] Ragnar {Groot Koerkamp}. *Minimizer papers*. en. Section: posts. Feb. 2025. URL: https://curiouscoding.nl/posts/minimizer-papers/ (visited on 06/16/2025).

[30] Ben Langmead. *Introduction to the Burrows–Wheeler Transform and FM Index*. Technical Report. Baltimore, MD, USA: Department of Computer Science, Johns Hopkins University, Nov. 2013. URL: https://www.cs.jhu.edu/~langmea/resources/bwt_fm.pdf.

[31] *Learning Resources: Compact Data Structures - FM-Index*. URL: https://docs.seqan.de/seqan/learning-resources/fm_index.html (visited on 06/16/2025).

[32] Heng Li and Richard Durbin. "Fast and accurate short read alignment with Burrows–Wheeler transform". In: *Bioinformatics* 25.14 (July 2009), pp. 1754–1760. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/btp324. URL: https://doi.org/10.1093/bioinformatics/btp324 (visited on 06/07/2025).

[33] *Longest Common Prefix (With Solution)*. Oct. 2021. URL: https://www.interviewbit.com/blog/longest-common-prefix/ (visited on 06/16/2025).

[34] Grigorios Loukides and Solon P. Pissis. "Bidirectional String Anchors: A New String Sampling Mechanism". In: *29th Annual European Symposium on Algorithms (ESA 2021)*. Ed. by Petra Mutzel, Rasmus Pagh, and Grzegorz Herman. Vol. 204. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021, 64:1–64:21. ISBN: 978-3-95977-204-4. DOI: 10.4230/LIPIcs.ESA.2021.64. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ESA.2021.64.

[35] Grigorios Loukides, Solon P. Pissis, and Michelle Sweering. "Bidirectional String Anchors for Improved Text Indexing and Top-K Similarity Search". In: *IEEE Transactions on Knowledge and Data Engineering* 35.11 (2023), pp. 11093–11111. DOI: 10.1109/TKDE.2022.3231780.

[36] Udi Manber and Gene Myers. "Suffix Arrays: A New Method for On-Line String Searches". In: *SIAM Journal on Computing* 22.5 (1993), pp. 935–948. DOI: 10.1137/0222058. eprint: https://doi.org/10.1137/0222058. URL: https://doi.org/10.1137/0222058.

[37] Udi Manber and Gene Myers. "Suffix Arrays: A New Method for On-Line String Searches". en. In: *SIAM Journal on Computing* 22.5 (Oct. 1993), pp. 935–948. ISSN: 0097-5397, 1095-7111. DOI: 10.1137/0222058. URL: http://epubs.siam.org/doi/10.1137/0222058 (visited on 03/21/2025).

[38] *Marshal.SizeOf Method (System.Runtime.InteropServices)*. en-us. URL: https://learn.microsoft.com/en-us/dotnet/api/system.runtime.interopservices.marshal.sizeof?view=net-9.0 (visited on 03/23/2025).

[39] Microsoft. *class keyword - C# reference*. en-us. URL: https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/class (visited on 06/16/2025).

[40] Microsoft. *Working with C# in Visual Studio Code*. Online. Accessed: 2025-06-16. Aug. 2023. URL: https://code.visualstudio.com/docs/languages/csharp.

[41] Microsoft Documentation. *.NET Garbage Collection*. Online documentation. Accessed via Microsoft Learn. Sept. 2021. URL: https://learn.microsoft.com/en-us/dotnet/standard/garbage-collection/.

[42] MIT Course 6.851. *Lecture 3: Advanced Data Structures*. Lecture slides, 6.851: Advanced Data Structures, Massachusetts Institute of Technology. Accessed on June 8, 2025, from https://courses.csail.mit.edu/6.851/spring10/scribe/lec03.pdf. 2010.

[43] MIT Scribe. *Lecture 7 Scribe Notes: Advanced Data Structures*. https://courses.csail.mit.edu/6.851/spring10/scribe/lec07.pdf. Massachusetts Institute of Technology, 6.851 Advanced Data Structures, Spring 2010. 2010.

[44] MIT Scribe. *Scribe Notes on Computational Geometry*. Accessed: May 19, 2025. Unofficial course notes, Massachusetts Institute of Technology. 2025. URL: https://courses.csail.mit.edu/6.851/spring10/scribe/lec03.pdf.

[45] *Namespace NetTopologySuite.Index.KdTree | NetTopologySuite*. URL: https://nettopologysuite.github.io/NetTopologySuite/api/NetTopologySuite.Index.KdTree.html (visited on 04/14/2025).

[46] *NetTopologySuite | NetTopologySuite*. URL: https://nettopologysuite.github.io/NetTopologySuite/ (visited on 04/14/2025).

[47] Rahul Pandey. *Suffix Trees*. 2017. URL: https://rkpandey.com/AlgorithmHelper/suffix/trees/trie/2017/02/14/suffix-trees.html (visited on 06/12/2025).

[48] Gonzalo Navarro Paolo Ferragina. *Pizza&Chili Corpus – Compressed Indexes and their Testbeds*. Online. Accessed: 2025-06-16, Department of Computer Science, University of Chile. 2025. URL: https://pizzachili.dcc.uchile.cl/.

[49] Phyo Phyo et al. "Indexing Relational Databases for Efficient Keyword Search". In: June 2011.

[50] Posit, PBC. *RStudio Desktop Download*. Online. Accessed: 2025-06-16. 2025. URL: https://posit.co/download/rstudio-desktop/.

[51] Michael Roberts et al. "Reducing storage requirements for biological sequence comparison". In: *Bioinformatics (Oxford, England)* 20.18 (Dec. 2004), pp. 3363–3369. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/bth408. URL: https://academic.oup.com/bioinformatics/article-pdf/20/18/3363/520444/bth408.pdf.

[52] Kunihiko Sadakane. "Compressed Suffix Trees with Full Functionality". In: *Theor. Comp. Sys.* 41.4 (Dec. 2007), pp. 589–607. ISSN: 1432-4350. DOI: 10.1007/s00224-006-1198-x. URL: https://doi.org/10.1007/s00224-006-1198-x.

[53] Saul Schleimer, Daniel S. Wilkerson, and Alex Aiken. "Winnowing: local algorithms for document fingerprinting". In: *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*. SIGMOD '03. San Diego, California: Association for Computing Machinery, 2003, pp. 76–85. ISBN: 158113634X. DOI: 10.1145/872757.872770. URL: https://doi.org/10.1145/872757.872770.

[54] *Stopwatch Class (System.Diagnostics)*. en-us. URL: https://learn.microsoft.com/en-us/dotnet/api/system.diagnostics.stopwatch?view=net-9.0 (visited on 03/23/2025).

[55] *Suffix tree - (Data Structures) - Vocab, Definition, Explanations | Fiveable*. en. URL: https://library.fiveable.me/key-terms/data-structures/suffix-tree (visited on 06/19/2025).

[56] TIOBE Software BV. "TIOBE Index for May 2022: C# Gains Most in Programming Language Popularity Index". In: *Visual Studio Magazine* (May 2022). Accessed: 2025-06-16. URL: https://visualstudiomagazine.com/articles/2022/05/10/csharp-gains.aspx#:~:text=Microsoft%27s%20C%23%20programming%20language%20posted%20the%20largest%2012-month,C%23%20ultimately%20failed%2C%20coming%20in%20second%20to%20Python..

[57] University of Maryland, Department of Computer Science. *Lecture 14: kd-tree range queries*. https://www.cs.umd.edu/class/fall2019/cmsc420-0201/Lects/lect14-kd-query.pdf. Accessed: 2025-06-15. 2019.

[58] *Visual Studio Code - Code Editing. Redefined*. en. URL: https://code.visualstudio.com/ (visited on 03/08/2025).

[59] "VS Code and Visual Studio Rock the 2022 Stack Overflow Developer Report -". en-US. In: *Visual Studio Magazine* (). URL: https://visualstudiomagazine.com/articles/2022/06/23/stack-overflow-2022-survey.aspx (visited on 06/16/2025).

[60] "What's the difference between C# and C++?" en. In: *Educative* (). URL: https://www.educative.io/blog/difference-between-c-sharp-and-c-plus-plus (visited on 06/17/2025).

[61] Justin Wyss-Gallifent. *CMSC 420: Tries*. PDF document, accessed on June 11, 2025, from https://www.math.umd.edu/~immortal/CMSC420/notes/tries.pdf. Mar. 2024.

[62] Xiaofeng Yang. "Improving the Relevance, Speed, and Computational Efficiency of Semantic Search through Database Indexing: A Review". In: *Optimization Algorithms*. Ed. by Mykhaylo Andriychuk and Ali Sadollah. Rijeka: IntechOpen, 2023. Chap. 9. DOI: 10.5772/intechopen.112232. URL: https://doi.org/10.5772/intechopen.112232.

[63] Norbert Zeh and Ulrich Meyer. "I/O-Model". In: *Encyclopedia of Algorithms*. Ed. by Ming-Yang Kao. New York, NY: Springer New York, 2016, pp. 943–947. ISBN: 978-1-4939-2864-4. DOI: 10.1007/978-1-4939-2864-4_190. URL: https://doi.org/10.1007/978-1-4939-2864-4_190.

A Appendix A - Project Plan

Table 2 Timeline

Week	**Activity**
Week 1	Write documents for project plan and project description
Week 2	Study the research paper "*Text Indexing for Long Patterns: Anchors are All you Need*" [2] **Thesis Work**: Describe main ideas behind algorithm from [2]
Week 3	Literature Study **Thesis Work**: Present related work
Week 4-5	Create tests and configures test suite **Thesis Work**: Describe procedure and test environment
Week 6-8	Develop and assess initial algorithmic solution (*algorithm 1*) **Thesis Work**: Present test outcomes and discuss results
Week 9-11	Develop and assess ideas for improvement (*algorithm 2*) **Thesis Work**: Present improved solution and test outcomes and discuss results
Week 12-14	Develop and assess ideas for improvement (*algorithm 3*) **Thesis Work**: Present improved solution and test outcomes and discuss results
Week 15-17	Finalize writing thesis
Week 18-19	Improve thesis based on received feedback

B Appendix B - Problem Description

Problem Definition

Consider a string S with $|S| = n$ and a pattern P with $|P| \geq l$, for some l. There exists an instance of P in S if and only if there is a position i such that $P = S[i...i + |P| - 1]$.

The text indexing problem is then defined as building an efficient data structure for S which facilitates fast search queries for an online pattern P [18]. It is a relevant and indispensable challenge in modern systems, where fast string retrievals are essential [49, 62]. A representative example is the field of bioinformatics, where text indexes are used for comparing DNA sequences in large databases [32, 51].

Previous Work

The first approaches to solving the text indexing problem include classical structures like the suffix tree and suffix array. Both structures essentially store the suffixes of the given input S in a slightly different way. The suffix tree takes $\mathcal{O}(|S|)$ space [18] and $\mathcal{O}(|P| + k)$ time [18], while the suffix array requires $\mathcal{O}(|S|)$ space [37] and $\mathcal{O}(|P| + log(|S|))$ time [37]. Here, P represents the pattern while k is the number of pattern instances in S.

Sadakane [52] improved the space of the suffix tree by proposing a compressed version storing only $\mathcal{O}(|S|log(\sigma))$ bits. In a similar manner, Roberto and Jeffrey [24] presented the compressed suffix array, which improves the space complexity.

Loukides et al. [35] introduced a new index structure for the text indexing problem, which utilizes locally consistent anchors in its construction. Unlike earlier solutions, their novel structure achieves a balance between space and time. Particularly, the index uses $\mathcal{O}(\#(anchors))$ extra space and supports pattern matching in $\mathcal{O}^{\sim}(|P| + k)$ time [35], where P is the pattern and k the number of appearances. The anchors are however built in $\mathcal{O}(|S| \cdot l)$ time, where l is a lower bound for pattern length.

Ayad et al. [2] provided two improvements to the previous work of Loukides et al. [35]. Concretely, they gave two alternative algorithms to compute the index in shorter time. The first one creates the anchors in linear time on average while the second uses a semi-external approach to "assemble the structure".

Aim of the Project

The main goal of this thesis is to build and refine new algorithms for the text indexing problem using the technique of anchors. It starts by implementing a prototype based on the idea introduced by Ayad et al. [2] and then improves it iteratively through empirical tests. The results are discussed and then compared against those in the previous paper [2] and other relevant solutions.