UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

CONTEXTUAL PATTERN MATCHING ON GRAMMAR-BASED SELF INDEXES

TESIS PARA OPTAR AL GRADO DE
MAGÍSTER EN CIENCIAS, MENCIÓN COMPUTACIÓN

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN COMPUTACIÓN

DIEGO ORTEGO PRIETO

PROFESOR GUÍA:
Gonzalo Navarro

MIEMBROS DE LA COMISIÓN:
PROFESOR 2
PROFESOR 3

Este trabajo ha sido parcialmente financiado por:
Anid Chile
IMFD Chile

SANTIAGO DE CHILE
2025

RESUMEN DE TESIS PARA OPTAR AL
GRADO DE MAGÍSTER EN CIENCIAS

POR: DIEGO ORTEGO PRIETO
FECHA: 2025
PROF. GUÍA: Gonzalo Navarro

CONTEXTUAL PATTERN MATCHING ON GRAMMAR-BASED SELF INDEXES

TODO: Traducir el abstract

ABSTRACT OF THESIS SUBMITTED FOR
THE DEGREE OF MASTER OF SCIENCE

BY: DIEGO ORTEGO PRIETO
DATE: 2025
ADVISOR: Gonzalo Navarro

CONTEXTUAL PATTERN MATCHING ON GRAMMAR-BASED SELF INDEXES

The Contextual Pattern Matching query of pattern P on text T with context length L returns the position in the text of a single occurrence for every unique "context" substring XPY in T, where $|X| = |Y| = L$. The problem definition can be further constrained to return the first occurrence for each context. It is this constrained Contextual Pattern Matching problem that we address here.

In highly repetitive text collections like DNA sequences and large version histories of code and web articles, it is not unusual for a substring to appear many times in the text while being surrounded by the same or similar context. This query allows the user to search specifically for the unique ways in which a pattern shows up in the text. In DNA collections in particular it would be useful to know which introns can surround a specific exon, and the same can be said for the unique instructions that might surround a specific function call in a code version history.

The problem was first proposed by Navarro in 2020, along with a storage impractical solution on the R-Index, a classic text self-index format. This solution was later improved, fixing the storage issue in exchange for slightly slower searching, by Abedin, Chubet, Gibney, and Thankachan.

A Grammar-Based self-index constructs a context-free grammar that uniquely generates the text, and stores said grammar in the text's place, together with some additional data structures that enable substring decompression and pattern matching. Pattern matching in these indices requires grammar traversal, and considering the fact that any rule which generates P will necessarily have an ancestor rule that generates XPY, it would seem like these indexes are ideal for implementing the Contextual Pattern Matching query.

This master's thesis addresses this claim. We analyze the known pattern matching algorithm in grammar-based self-indexes, explain its intricacies and suggest a solution that might reduce the amount of processing necessary to report all contextual occurrences. This solution relies on pruning grammar traversal paths that would lead to already found contexts during occurrence retrieval, and later filtering any lingering repetitions from the pruned results.

We implement the solution for two differently engineered grammar-based indexes. Additionally we implement arbitrary substring hash calculation for one of these indexes.

Finally we discuss the effectiveness of our approach by analyzing whether path pruning makes any difference, and evaluating the cost of filtering through extracted substring comparisons versus hash comparisons.

We conclude pruning makes a great difference for small context length values but gets decreasing benefits fast. The filtering of repeated contexts is by far the most expensive section of the algorithm, and our implementation of hash retrieval, even if fast, does not make much of a difference.

The code is freely available at github.com/Gedoix/contextual_grammar_indexes

Agradecimientos

Table of Contents

Index of Figures

Chapter 1

Introduction

As new technologies like LLMs and Gene Sequencing evolve, there is a growing necessity for larger data collections to be both stored and processed in a compressed form. Text collections with high substring repetition especially waste a lot of space if stored as-is. It is desirable, then, to place resources on developing formats that get nearer to the data's information entropy.

Text collections can be compressed using a wide range of technologies, generally under two categories: Those designed for regular text and those designed for highly repetitive text, text where substrings are often repeated many times with little to no changes across occurrences. Compression methods for highly repetitive text can achieve far superior compression rates on such collections.

Of these technologies, Straight Line Programs or SLPs are a form of Grammar-Based compression founded on the idea of assigning repeated substrings a unique symbol that can replace them in the text, reducing its length. These methods therefore construct a Context-Free Grammar that uniquely generates the text when deriving its rules.

These compression methods can then be augmented into Text Self Indexes[1, 2] through the addition of complementary data structures that allow for random substring access and efficient pattern matching queries, directly on the uncompressed data structure. Adding a Grammar Tree with additional information on how the grammar parsed the text can easily result in Random Access, while the data structures for Pattern Matching are more complex but can be made to enumerate results in amortized logarithmic time.

These indexes work great for solving classic Pattern Matching queries, but inside highly repetitive text collections, many occurrences of a pattern usually appear surrounded by the same, repeated context. Would it not be useful to sometimes limit this search to occurrences surrounded by unique contexts only? Or to be able to efficiently compare the different contexts in which a pattern shows up?

An alternative pattern matching query could be, then, Contextual Pattern Matching: The query receives a grammar-compressed text T, a pattern P and an additional integer L, and returns the first occurrence of P for every unique *context* it appears in within T . The *context* is defined here as the two substrings of length L to the left and right of every occurrence, meaning we are looking to report each unique X_iPY_i substring of T where $|X_i| = |Y_i| = L$. This query was first proposed by Gonzalo Navarro in 2020 [3], along with an efficient but storage-impractical algorithm built on a classic non-grammar index, the R-Index.

This thesis proposes an algorithm that solves the Contextual Pattern Matching problem for Grammar-based Text Self Indexes, implements solutions for two grammars with different structural properties, and compares the efficiency of each.

The first part of the thesis focuses on explaining the relative advantages of grammar structures for matching contextual occurrences while cutting down on unnecessary processing of repeated contexts, and explains an algorithm that solves Contextual Pattern Matching for these indexes.

These advantages come from leveraging the fact that a rule can always be trusted to represent a substring that is longer than that of its children. While looking for contextual occurrences, when a rule holding the pattern is found, we can climb the parse tree to eventually find a rule that contains the pattern and its context. We also have the assurance that any parent of this second larger rule can not contain a context that is different, so we can skip checking the rule's possible ancestors.

The second part of the text focuses on the implementation of the algorithm on a Locally Consistent Grammar-based Self Index. First, properties of the index are discussed, as well as reasons why it might be especially suited for this query. Second, the implementation details are explained.

The third part focuses on the implementation of the algorithm on a RePair Relaxed Grammar-based Self Index. Once again, properties of this index are discussed first, with special considerations given to its Relaxed nature. Then, the implementation is explained with an eye on how performance measurements can be adjusted for a more accurate comparison with the first implementation.

The fourth part of the thesis is a discussion of the comparison between both indexes' performances, and the implementation difficulties of each solution.

Chapter 2

Basic Concepts

We introduce some data structures relevant to the topic of this thesis.

2.1. Compact Data Structures

2.1.1. Rank and Select Support

The text self-indexes we modify make heavy use of many compact data structures, the most elementary and essential being bit-vectors and other sequence-storing schemes built upon them, that support rank and select operations. We assume here that positions in the text are 1-based. A data structure encoding the sequence S of length n over alphabet Σ of size σ supports:

- $rank_S(c, i)$ counts the appearances of symbol $c \in \Sigma$ in the subsequence $S[1..i]$. We also define $rank_S(c, 0) = 0$, it will be convenient later.
- $select_S(c, j)$ finds the smallest i such that $rank_S(c, i) = j$, meaning it finds the position of the jth occurrence of symbol $c \in \Sigma$ in the sequence. $select_S(c, 0) = 0$ and $select_S(c, m) \geq n + 1$ if $rank_S(c, n) < m$.
- $access_S(i)$ retrieves $S[i] \in \Sigma$.

For the binary alphabet $\Sigma = \{0, 1\}$ the standard data structure is called a Bit Vector [4, 5], and it has three relevant variants for this thesis:

- A general solution exists that solves all three queries in $O(1)$ operations while storing $n + o(n)$ bits.
- For sparse sequences[6] such that $m << n$ where $m = rank_S(1, n)$, the space requirement can be reduced to only $m \log \frac{n}{m} + O(m) + O(n \log\log n / \log n)$ bits.
- When we only require $select_S(1, i)$ queries, the $o(n)$ space term can be improved to $O(\log\log n)$.

For larger alphabets, the solution landscape is more diverse; the standard data structure is called a Wavelet Tree, and it is built on top of Bit Vectors.

The Wavelet Tree comes in the following variants:

- The general solution achieves all queries in $O(\log\sigma)$ operations using $n\log\sigma + o(n)\log\sigma$ bits of space.
- Multiary wavelet trees improve query operations to $O(1+\frac{\log\sigma}{\log\log n})$ within the same space.
- For large alphabets, a solution exists that occupies the same asymptotic space and offers two trade-offs[7, 8]:
  - The first can answer *select* in $O(1)$ operations, and both *rank* and *access* in $O(\log\log\sigma)$ operations.
  - The second can answer *access* in $O(1)$ operations, *select* in $O(\log\log\sigma)$ operations and *rank* in $O(\log\log\sigma\log\log\log\sigma)$ operations.

2.1.2. Compact Tree Representations

it is known that any arbitrary unlabeled tree can be represented one-to-one as a balanced parentheses sequence. A Depth-First Unary Degree Sequence or DFUDS[9] is a Depth-First ordering balanced parentheses representation of a tree that, when stored in a rank/select structure, allows for fast and ergonomic tree traversal in compact space.

Its depth-first nature means that node data is stored in the same order as a left-to-right DFS algorithm might traverse the tree. Unary degree stands for the way the amount of children in any given child is represented. This means each node, in order, is represented as a sequence of 1 bits that represent its degree in unary, followed by a 0.

Because of this scheme, each node $n \in [1..N]$ in depth-first order can be uniquely identified by the position p_n of its trailing 0 in the sequence, which allows for the structure to support the following operations over its bit vector B:

- $pos_B(n)$ calculates p_n from n in $O(1)$ operations.
- $ord_B(p)$ calculates n from p_n in $O(1)$ operations.
- $root_B()$ calculates p_{root} in $O(1)$ operations.
- $degree_B(p)$ calculates the degree of node p_n by counting the 1s in its segment of the sequence in $O(1)$ operations.
- $first\_child_B(p)$ calculates the position of the first child of p_n in $O(1)$ operations.
- $next\_sibling_B(p)$ calculates the position of the next sibling of p_n in $O(\log_2 N)$ operations.
- $parent_B(p)$ calculates the position of the first ancestor of p_n in $O(\log_2 N)$ operations.

This scheme can be augmented with two additional compact structures [10] for tracking sub-tree sizes and boundaries. This reduces the operation cost of $next\_sibling_B(p)$ and $parent_B(p)$ to $O(1)$ with only an $O(N) + o(N)$ bit overhead.

2.1.3. Permutation Data Structures

Mäkinen and Navarro[11] devised a way to use a wavelet tree to represent a permutation π of $[1..n]$ in $O(n \log n)$ bits, supporting the operations:

- $\mathit{direct}_\pi(i)$ permutes i into $\pi(i)$ in $O(\log n)$ operations.
- $\mathit{reverse}_\pi(j)$ reverse permutes $j := \pi(i)$ into i in $O(\log n)$ operations.
- $\mathit{range}_\pi(j_1, j_2)$ enumerates all values of i whose $\pi(i) \in [j_1..j_2]$, where $1 \leq j_1 \leq j_2 \leq n$, in $O((j_2 - j_1 + 1) + \log n)$ operations.

This was later extended by Claude[12], where given a sequence S of length n over alphabet Σ of size σ, its wavelet tree Π can support the operations:

- $\mathit{range\_count}_\Pi(i_1, i_2, j_1, j_2)$ counts all values in the subsequence $S[i_1..i_2]$ that are in the range $[j_1..j_2] \subseteq \Sigma$, in $O(\log \sigma)$ operations.
- $\mathit{range\_report}_\Pi(i_1, i_2, j_1, j_2)$ enumerates the values counted by $\mathit{range\_count}_R(i_1, i_2, j_1, j_2)$ in $O((i_2 - i_1 + 1) \log \sigma)$ operations.

2.1.4. Labeled Binary Relation Data Structures

Claude, Navarro, and Pacheco[2] build upon the above data structure and extend it for compact storage and operation of a labeled binary relation between two sets of integers. Given a relation $R \subseteq A \times B$ and a labeling function $F_L : R \longrightarrow L$, where $A := [1..n_A]$, $B := [1..n_B]$, and $L := [1..n_L]$, the structure supports the following operations:

- $\mathit{nRowsForCol}_R(b)$ calculates $|(A \times \{b\}) \bigcap R|$, the amount of labeled points with b as column.
- $\mathit{nColsForRow}_R(a)$ calculates $|(\{a\} \times B) \bigcap R|$, the amount of labeled points with a as row.
- $\mathit{nPointsForLabel}_R(l)$ calculates $|\{(a, b) \in R \mid F_L((a, b)) = l\}|$, the amount of points labeled by l.
- $\mathit{ithRowForCol}_R(b, i)$ retrieves the ith row a_i with a point on column b, or 0 if $i > nRowsForCol_R(b)$.
- $\mathit{ithColForRow}_R(a, i)$ retrieves the ith column b_i with a point on row a, or 0 if $i > nColsForRow_R(a)$.
- $\mathit{ithPointForLabel}_R(l, i)$ retrieves the ith point (a_i, b_i) with label l, or 0 when $i > nPointsForLabel_R(l)$.
- $\mathit{labelAtPoint}(a, b)$ retrieves the label l of point (a, b), or 0 if $(a, b) \notin R$.
- $\mathit{labelsInRange}_R(a_1, a_2, b_1, b_2)$ retrieves $\{l := F_L((a, b)) \in L \mid a \in [a_1..a_2] \wedge b \in [b_1..b_2]\}$, the labels of all points in that range of columns and rows.

2.2. Grammar Self Indexes

A Straight-line Program or SLP, is a Context-free Grammar $G := X, S, R, \Sigma$ that uniquely generates a single string $T[1..u]$ over some alphabet $\Sigma := [1..\sigma]$ of terminal symbols. It must therefore contain a set of rule symbols $R \notin \Sigma$, a set of rules $X \subset (R \times \{\longrightarrow\} \times (R \cup \Sigma)^+)$ of size n that translate rule symbols into sequences of rule and alphabet symbols, and a starting symbol $S \in R$.

The set X can be stored in a number of ways, we will go in depth for only two of these as they are relevant to this research. The first applies to the general case where rules can be of any length, the second applies to binary rules specifically, like the ones created in a RePair Index.

2.2.1. A DFUDS Traversable Representation

A Parse Tree P for the SLP G is an ordered representation of the set of rules X that includes information relating to the place the rules' phrases occupy in the original text $T[1..u]$.

The parse tree is constructed by recursively expanding the leftmost appearance of the rules in X starting from S, and storing said expansion in a lower level of the tree. The example in Figure 2.1 shows in black all the symbols that are actually stored in memory, while the gray sections show what a full expansion of the grammar would look like.

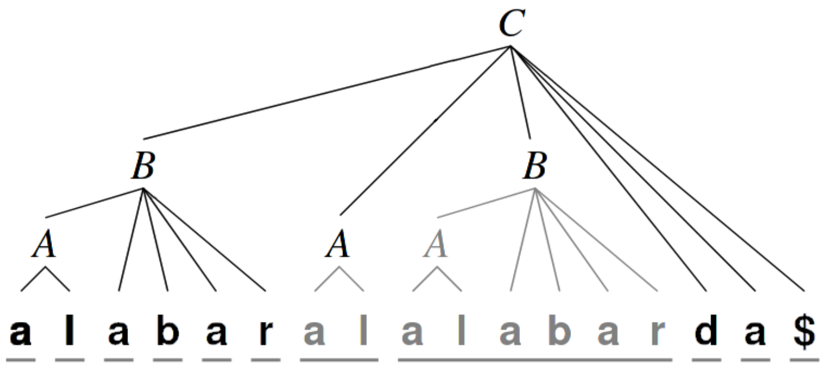

Figure 2.1: Example parse tree for a grammar that generates "alabaralalabarda$" with stored data in black and implied data in gray. Original image taken from Navarro's Indexing Highly Repetitive String Collections Survey[1], manually upscaled.

Díaz-Domínguez[13] uses a DFUDS data structure in combination with a rank/select supporting sequence of symbols to store the parse tree in compact space while keeping it traversable. His scheme involves storing the raw tree structure in the DFUDS and placing the symbols of each node into the sequence in depth-first order.

The representation enables a node's depth-first index to be used as a key to access the node's symbol in the sequence, while the sequence's rank and select capability allows for easy traversal between nodes that share the same symbol. especially $select_S(r, 1)$ for a rule symbol $r \in R$ returns the depth-first index of the non-terminal node that expands the rule, so that the node's *degree* is equivalent to the rule's length, and the node's children become the right-hand symbols of the rule.

This translates to the following operations for DFUDS B and sequence S (on top of the already discussed DFUDS operations for depth-first indexes denoted n):

- $sym_S(n)$ calculates $r_n \in R$ from n in $O(log|R|)$.
- $rank_S(n)$ calculates which appearance of r_n in the tree has depth-first order n, in $O(log|R|)$.
- $ord_S(r, i)$ calculates the depth-first order n of the ith appearance of rule r_n in the tree, in $O(log|R|)$.

Therefore, a node's properties can fall into either of the following 3 categories:

- If the node's $rank_S(n)$ is not 1 then its $degree_B(p_n)$ must be 0, it is a leaf of the tree representing a symbol in some rule's fight-hand side.
- If the node's $rank_S(n)$ is 1 and its $degree_B(p_n)$ is 0 then it must be a terminal of the grammar, and therefore $sym_S(n) \in \Sigma$.
- If the node's $rank_S(n)$ is 1 and its $degree_B(p_n)$ is not 0 then its children's symbols encode r_n's fight-hand side.

2.2.2. A Binary Relation Representation for RePair

When a grammar G is binary, such as inside RePair indexes, meaning the set of rules is $X \subset (R \times \{\longrightarrow\} \times (R \cup \Sigma)^2)$, it can be convenient to store rules as binary relationships, where the two right-hand symbols in the rule are related through their parent as a label.

This relation R_X can help encode further information about the intersection of the child rules, information that will become useful later when we address pattern matching algorithms. For this, although the nodes in the relation can, in principle, be indexed in any order, interesting properties arise when choosing the following scheme:

- For any rule $x_i \in X$, we assign it an index ds_i after sorting all rules according to lexicographic order of their expanded text.
- For any rule $x_i \in X$, we assign it an index rs_i after sorting all rules according to lexicographic order of their reversed expanded text. it is important to note that this order is different from the reverse of ds_i and can not be computed from it, meaning $rs_i \neq |R| - ds_i + 1$.

We can use this to construct a binary relation for pairs (rs_i, ds_j), labeled $F_L((rs_i, dr_j)) := ds_k$ for every $(x_k, \longrightarrow, x_i, x_j) \in X$.

With this scheme, we can then use the previously discussed Labeled Binary Relation structure in Section 2.1.4 to query for the string at the intersection of both rules, enabling the following operations given a pattern $P[1..v]$:

- For any suffix $P[c..v]$ we can find a range of rules $ds_{b_1}..ds_{b_2}$ that begin their expanded text with that suffix, by binary searching in the $ds_1..ds_{|R|}$ set and comparing the suffix to the expanded text of its rules.
- For any prefix $P[1..(c-1)]$ we can find a range of rules $rs_{a_1}..rs_{a_2}$ that end their expanded text with that prefix by binary searching in the $rs_1..rs_{|R|}$ set and comparing the prefix to the reversed expanded text of its rules.

- With this information, $labelsInRange_R(rs_{a_1}, rs_{a_2}, ds_{b_1}, ds_{b_2})$ returns the set of all parent rules containing the pattern in the intersection of their children's expanded texts, where said intersection happens at the cutting position c within the pattern.

Further still, the operations $ithRowForCol_R$, $ithColForRow_R$, $ithPointForLabel_R$, and $labelAtPoint$ allow for easy traversal of the grammar, keeping in mind that in this scheme all labels in the relation are unique and therefore $nPointsForLabelR(l)$ will always return 1 if the rule with $ds_k = l$ exists in the grammar and is not a terminal.

2.2.3. Pattern Matching Algorithm

Pattern Matching in the index means retrieving all occurrences $O := \{o_0..o_c\} \subseteq \{1..u\}$ of a pattern $P[1..v]$ in the text $T[1..u]$, including overlapping occurrences. The full algorithm will be explained later in sections 4.1 and 5.1, as part of the detailed descriptions of both indexes, but a simplified and somewhat abstracted version is presented here as well.

To come to understand the algorithm, a core distinction is necessary, that of "primary" and "secondary" occurrences[14, 15]. This distinction was first formalized by Kärkkäinen and Ukkonen [14]. A primary occurrence is an occurrence that falls on top of a phrase cut in the parse tree, and that could therefore be found easily by using the scheme in Section 2.2.2. A secondary occurrence occurs elsewhere in the parse tree, and is therefore contained in the expansion of a secondary mention of a rule.

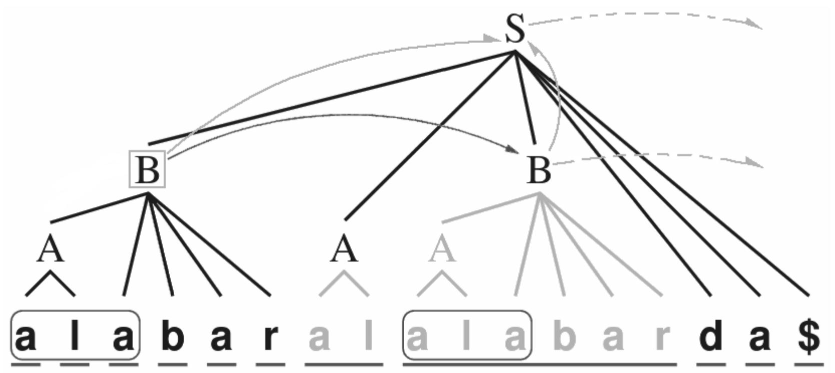

Figure 2.2: Example parse tree for a grammar that generates "alabaralala-barda$" with stored data in black and implied data in gray. Tree traversal for locating secondary occurrences of the pattern "ala" are shown. Fruitless paths are shown in broken lines. Original image taken from Navarro's Indexing Highly Repetitive String Collections Survey[1], edited with tree traversal lines and manually upscaled.

This distinction can be illustrated easily by looking at the two occurrences of the pattern "ala" in Figure 2.2. The first occurrence lies at the intersection between A and a inside B, while the second occurrence is entirely contained within a secondary mention of the rule B. Hence, all occurrences fall into one of these two sets, and the sets do not intersect.

Moreover, the set of all "secondary" occurrences can be found by traversing the parse tree once the set of "primary" occurrences has been computed, as shown in the figure by following the gray arrows from node B. Once we know that an occurrence of "ala" was found inside a B node, finding the positions of all other B nodes nets us the subset of secondary occurrences that is associated with rule B and a cut c (in this case $c = 2$) of the pattern.

Since all occurrences in the text must fall inside some rule, and the parse tree explicitly represents said rule exactly once, we can trust that by finding every primary occurrence we can use them to then find every secondary one too.

Code Block 2.1: Primary Occurrence Retrieval Pseudocode.

```
def search_primary_occurrences(
        pattern: str,
        index: GrammarIndex
    ) -> List[Tuple[int, int]];
    for cut in range(1, len(pattern)-1):
        prefix: str = pattern[0:cut-1]
        rev_prefix: str = prefix[::-1]
        suffix: str = pattern[cut:len(pattern)-1]
        row0, row1 = index.grid.binSearchColRange(suffix)
        col0, col1 = index.grid.binSearchRowRange(rev_prefix)
        primary_occurrence_points: List[Tuple[int, int, int]] = index.grid.labelsInRange(col0,
      ↪ col1, row0, row1)
        primary_occurrences: List[Tuple[int, int]] = []
        for parent_rule, left_rule_reversed, right_rule in primary_occurrence_points:
            left_rule: int = index.ReverseToDirect(left_rule_reversed)
            offset_in_parent: int = index.ruleToExpandedLength(left_rule) - cut
            primary_occurrences += (parent_rule, offset_in_parent)
        return primary_occurrences

```

The pseudocode in Code Block 2.1 shows a simplification of the classic algorithm for retrieving the first set, the primary occurrences, by taking advantage of the Labeled Binary Relation structure shown in Section 2.1.4, as explained in Section 2.2.3. The algorithm retrieves at most one primary occurrence for every possible rule intersection and pattern cut $c \in [1..u]$, each associated with the symbol of the parent rule parent_rule and an offset within the rule offset_in_parent.

This part of the algorithm requires a mapping from reverse lexicographic sorting of nodes to direct lexicographic sorting, here represented by index.ReverseToDirect() which would be implemented using a permutation structure as seen in Section 2.1.3, as well as a mapping from rule symbols to the length of their expanded text.

This mapping can be achieved in many ways: It can easily be stored as a vector of integers, or it can less easily be calculated by traversing the parse tree from the rule's symbol and recursively expanding each child rule. Another alternative is to store a long sparse bit vector that marks the beginning positions of each phrase on the parse tree, an alternative that we will explore further in Section 4.1.

This step of the search is expensive in terms of operations per occurrence found, because it necessitates checking all $u-1$ cuts of the pattern and performing binary search 4 times per cut (here represented by the methods index.grid.binSearchColRange and binSearchRowRange) in order to find the appropriate query range to call index.grid.labelsInRange on the relational structure. As such, there is a vested interest in reducing the number of cuts that need to be checked, and reducing the number of points in the relation, by modifying the way the grammar is computed. This concern is addressed handily by Díaz-Domínguez by using Local Consistency Parsing for choosing phrase boundaries in his implementation.

After that, in order to find all secondary occurrences, it is only necessary to traverse the tree and recursively find all repetitions of the primary occurrences' parent rules in the parse tree.

Code Block 2.2: Secondary Occurrence Retrieval Pseudocode.

```
def search_secondary_occurrences(
        primary_occurrences: List[Tuple[int, int]],
        index: GrammarIndex
    ) -> List[int]:
    occurrence_queue = []
    results = []

    for rule, offset_in_rule in primary_occurrences:
        if index.isRoot(rule):
            results.append(offset_in_rule)
            continue
        occurrence_queue.append((rule, 0, offset_in_rule))
        def add_secondary(s_rule, s_mention):
            occurrence_queue.append((s_rule, s_mention, offset_in_rule))
        index.forAllSecondMentions(rule, add_secondary)

    while occurrence_queue:
        current_rule, current_mention, current_offset = occurrence_queue.pop(0)
        if index.isRoot(current_rule):
            results.append( current_offset )
            continue
        parent_rule, offset_in_parent = index.parentOf(current_rule, current_mention)
        occurrence_queue.append((parent_rule, 0, offset_in_parent))
        def add_parent_secondary(s_rule, s_mention):
            occurrence_queue.append( (s_rule, s_mention, offset_in_parent) )
        index.forAllSecondMentions(parent_rule, add_parent_secondary)

    return results

```

The pseudocode in Code Block 2.2 shows a simplification of the classic algorithm for retrieving this second set of occurrences, as well as translating the results of the first algorithm primary_occurrences into actual offsets within the text's bounds in results.

For this pseudocode to be possible to implement, the index needs to include a way to traverse the parse tree, specifically by jumping to a rule's parents by implementing index.parentOf(), as well as their secondary mentions by implementing grammar traversal inside index.forAllSecondMentions().

Critically, a method for finding the absolute position of a rule's first mention in the text is not strictly necessary to have. The current_offset offsets reported by the binary relation structure are relative to the phrase boundary in the rule that holds the occurrence, but on every climb step where a parent rule is computed, this offset is adapted into offset_in_parent the offset in the parent rule's phrase, by index.parentOf(), until the root is reached and the relative offset becomes absolute and can be reported to results. For index.parentOf() to be able to do this, it needs access to either the absolute position of a rule's first mention or the expanded sizes of its children; any of these two mappings will do. Díaz-Domínguez's index [13] uses the former while Claude's [2] uses the latter.

Any grammar representation that enables traversal, including the relation as presented in Section 2.2.2 and the DFUDS in Section 2.2.1 before it, can be augmented with an implementation for this second step of the algorithm. These simple additions are sufficient to implement a functioning pattern matching solution.

Therefore, the algorithm requires 3 things:

- A grid, or labeled binary relation, for finding primary occurrences.
- A traversable grammar, either in parse tree form or plainly, that allows for a rule's parents and secondary mentions to be accessed and enumerated.
- A method for computing the offset in the parent rule from the offset in the child, either by mapping rules to sized or rules to starting positions in the text.

2.3. Local Consistency

A scheme for reducing the amount of cuts of the pattern that need to be tested, which is relevant to this research, exists. Díaz-Domínguez[13] builds upon earlier work on locally consistent parsing by Navarro, Sepúlveda, and Silva [16] and leverages this property, called Local Consistency, for his phrase construction criteria. This property is defined as applying exclusively local reasoning to the computation of phrase borders, meaning that the only relevant information to any one phrase intersection is the values of the symbols that surround it.

The reason why this helps is simple: if all pattern cuts in the text occur under predictable and replicable conditions, independent of text length, then when evaluating possible cuts of a pattern that need to be processed, it is sufficient to apply the same phrase-cut logic to the relatively shorter pattern's text.

Specifically, Díaz-Domínguez's index uses a locally consistent criterion called Left-Most S-Type or LMS. Under his scheme, the index only needs to check $O(\log |P|)$ cuts of the pattern to find all primary occurrences. These cuts are computed by repeatedly parsing the pattern in a similar way to the iterative process used to build the text's grammar, and then choosing to keep each iteration's first and last border. This makes it a "hierarchical" parsing of the pattern, which he proves to be a sufficient amount of cuts to handle all edge cases and retrieve all primary occurrences.

2.4. Relaxed Grammar Self Indexes

Claude's index implements a Relaxed SLP or RSLP, which is an SLP that, instead of compressing the text into the grammar $G := \{X, S, R, \Sigma\}$ with S being the starting rule's left-hand symbol, makes S a starting *sequence* of symbols, therefore compressing the text into a grammar forest instead of a grammar tree. This variant of the SLP addresses the fact that the first rule (or rules) descending from the starting symbol represent the macroscopic structure of T and are therefore unlikely to be repeated elsewhere in the tree. This uniqueness marks them as an aspect of the traditional SLP that can be simplified, so RSLPs skip the construction of rules that would compress a substring that does not repeat by saving their phrase directly into the compressed sequence S.

Special considerations need to be taken for this scheme, though. A RePair RSLP, like Claude's, has a grammar that is binary and can in turn be represented as a binary children-to-parent relation like the one mentioned in Section 2.2.2, but the sequence S cannot be easily stored in the same relation data structure; it needs its own separate representation instead.

The way he solves this is by constructing a second relation structure for catching pattern occurrences between symbols of the sequence. This new relation need not store a parent symbol as the label; it can manage well enough by storing the offset of the symbol intersection in the text instead.

2.5. Problem Description

Given that inside a highly repetitive text, any occurrence of a pattern has a high chance of being surrounded by similar text to other occurrences of the same pattern, a need arises for isolating the different "contexts" that a pattern can occupy. This motivates the search for an efficient way to filter away context-redundant occurrences from the results of a pattern matching query.

The problem was first formalized by Navarro[3] as follows:

Definition. Given a text $T[1..n]$, a pattern P, and a context length L, report a single occurrence for every distinct substring XPY in T, where $|X| = |Y| = L$ and the text is padded by L blank characters.

Critically, this does not require said occurrence to be $the\ first\ one$ from left to right for its given context, so we have taken the liberty of constraining the problem further:

Definition. Given a text $T[1..n]$, a pattern P, and a context length L, report only the first occurrence for every distinct substring XPY in T, where $|X| = |Y| = L$ and the text is padded by L blank characters.

The first proposed solution comes from the same work as the problem itself, by Navarro. His proposed solution is built upon a classic non-grammar index called the R-Index[17], introduced by Gagie, Navarro, and Prezza, and is theoretically efficient at retrieving c contextual occurrences of the pattern P for text T in $O(\mid P \mid \log\log|T| + c\log|T|)$ operations, but is described as space-inefficient as it stores the text T using $O(\overline{r}\log\frac{n}{\overline{r}})$ bits, where $\overline{r}$ is the maximum amount of runs in the forward and backward BWT of T.

For comparison, an unaltered R-Index can retrieve c non-contextual occurrences of pattern P from text T in $O(|P| + \log|T| + c)$ operations while storing only $O(r)$ bits, where r is the amount of runs in T's forward BWT and $r \leq \overline{r}$. $\overline{r}$ has been known to sometimes be as large as $2r$, so the contextual solution can end up storing over double the bytes.

it is important to note that the absence of an L term in the operations' asymptotic cost of retrieval does not mean the context length does not play a role, as the length L will determine the amount of retrieved results c, and when $L = |T| - |P|$ the results are guaranteed to be the same as those of a regular pattern search.

Since then Abedin, Chubet, Gibney, and Thankachan[18] (DCC 2023) have introduced an improved solution that builds upon Navarro's proposal but decouples it from its dependence on a backward BWT of T, hence removing the $\overline{r}$ term. This second solution reports all c contextual occurrences in $O(\mid P \mid + c\log|P| \cdot \log\frac{|T|}{r})$ operations, and occupies only $O(r\log\frac{|T|}{r})$ bits.

This makes the second solution slightly slower than Navarro's, but effectively addresses the space bound issue.

We wish to adapt Grammar-Based Self-Indexes to solve the Contextual Pattern Matching problem because they seem to offer a native method for pruning redundant solutions. Put simply: The structure of a parse tree is beneficial for this query because it represents a substring's context in a very direct way, as an ancestry relationship between nodes in the tree.

In other words, when a Pattern Matching algorithm is climbing the parse tree and encounters a node n that contains an occurrence o of the pattern P found at offset d from processing pattern cut c of P, and some particular context X_iY_i; it is certain that no second mention of n anywhere else in the tree can contain a different context X_jY_j for the same P, c and d, and no ancestor of n can contain a different context for the same P, c and its updated d. We can then prune the search by removing ancestors and second mentions of n from the queue of nodes to be climbed when we detect that a full XPY substring fits inside n's rule, with some special considerations for border cases where either X or Y have blank characters in them from stepping over at the beginning or end boundaries of T.

Chapter 3

The Algorithm

Let us go into further detail on our modifications to the classic Pattern Matching algorithm. As seen in Section 2.2.3, the algorithm for pattern matching roughly follows the following steps:

- Choose a pattern cut position, separating the pattern into a prefix and a suffix.
- Compute ranges for the phrase intersection grid using said prefix and suffix, binary searching among the rows and columns for the nodes that contain an intersection that matches.
- Retrieve the points from the grid, these represent the primary occurrences, although for now we only have data for calculating the offset of the occurrence relative to the rule it was found in.
- For each primary occurrence, compute the offset of the pattern within its rule.
- Add all primary occurrences and their relative offsets to a queue, so their nodes can be used as starting points for the parse tree traversal.
- Process the queue by adding a node's secondary mentions and parents to the queue, climbing the parse tree. The parent nodes' offsets need to be adjusted from those of their sons on every climb operation.
- Report the node as an occurrence once we reach root, having adjusted the offset as we went along. Since the offset is now relative to root's rule, and root contains the entire text, the offset has become absolute.

The pseudocode in Appendix Code Blocks 2.1 and 2.2 illustrate a simplification of this process, for both finding all primary occurrences and then finding all secondary occurrences by climbing the parse tree.

Our addition to this foundation, then, is to prune paths of the tree that would supply redundant occurrences and then filter the results for repeated contexts.

The pseudocode in Code Block A.1, included in the Appendix, shows a simplified version of these modifications, but we will go over the details here:

- Paths in the tree are pruned by early-reporting of nodes that already hold a complete X_iPY_i substring in their expanded text.

- This early report necessitates access to a grammar tree node's absolute starting and ending positions in the text, which may or may not be natively available information in any given Grammar Index. The reason for this is that the algorithm as originally described computes this absolute offset by adjusting relative values as the tree is climbed, but we intend to avoid climbing altogether as much as possible.

- Then, a simple filter algorithm is used to remove all redundant occurrences that could not be pruned. This step requires access to substring decompression functionality from the index, which is a given for any Text Self-Index.

- Alternatively, some other substring identification method can be used if implemented, like an implementation of substring hash computation, which will be discussed further by the end of Section 4.2.

Computing whether a node contains the pattern and its context, and therefore can be reported without continuing to climb, also necessitates some manner of computing or retrieving the sizes of a rule's expansion, which is already available since it is required for adjusting offsets in the unmodified algorithm.

Put simply, given the relative 0-based offset o inside the node's rule, context length L, pattern length $|P|$ and an expanded size $|r_i|$ of the node's rule, then if the boolean condition $fits\_in\_node := o \geq L \wedge (|r_i| - (o + |P|)) \geq L$ is *true* then the context and pattern fit inside the node's rule.

This raises a question though, as according to our problem definition we need to take into account the L sized padding of blank characters around the text. So we need to calculate also, when the context does not fit in the node, whether the reason for that is that the node we are looking at is compressing a substring that is too close to the beginning or end of the text.

This calculation is also simple given access to a grammar tree node's absolute starting and ending positions in the text, which we already require for early reporting of our results. So given the length of the text is $|T|$, the starting position of the node in the text is s_i and its ending position is $e_i := s_i + |r_i|$, then we can detect a boundary context when $boundary\_context := s_i < L \vee (|T| - (s_i + o + |P|)) < L$ is *true*.

So our conditions for an early report of an occurrence are $fits\_in\_node \vee boundary\_context$, to which we add the condition of the node being root, which would mean we have taken a path that never offered us an early return and can report the occurrence as we normally would. The value we report is $s_i + o$, which trivially is equal to just o when the node is root.

Chapter 4

Solution for a Locally Consistent LMS SLP

4.1. The Index

Let us delve into the details of the pattern matching algorithm for Díaz-Domínguez's Self-Index[13]. We tackle this index first because its implementation is most similar to the algorithm as explained in Section 2.2.3, with only some small additions that, nevertheless, impact its performance and compression rate in some important key ways.

A pseudocode representation of his implementation is included in the Appendix Code Blocks A.2 and A.3, for primary and secondary occurrence retrieval respectively. We will delve into the details instead here.

First let us discuss the differences this index has when compared to the "neutral.ªlgorithm presented above:

- The index uses a DFUDS representation of the parse tree for tree traversal as well as grammar structure storage, like the one seen in Section 2.2.1.
- The index implements Locally Consistent Parsing as seen in Section 2.3, specifically using a parsing scheme called Leftmost S-Type, or LMS for short, which interprets a string as a sequence of integers that may be local maxima or local minima of their immediate character neighborhood from right to left. A character is S-type if its suffix is lexicographically smaller than the subsequent suffix, and the opposite is true for L-Type characters. It then characterizes the string as a sequence of S-Type and L-Type characters, and chooses phrase boundaries at the leftmost S-Type character in each S-Type run.
- Locally consistent parsing allows the index to compute phrase boundaries deterministically regardless of macroscopic properties of the text, which he leverages effectively by parallelizing the phrase boundary calculation step, accelerating index computation.
- Locally consistent parsing also allows it to reduce the amount of cuts that need to be evaluated in his search for primary occurrences, by parsing the pattern in the same way as the text and retrieving hierarchical cuts to handle the pattern's start and end border conditions.
- Locally consistent parsing also plays a role in the shape of the grammar, as phrases are often wide and the tree is not as tall as seen in other implementations.

- The index implements run-length compression, an extension to the native Grammar structure that allows it to represent runs of a same symbol or phrase like $AAAAA$ as simply A^5, reducing the amount of bytes that any such rules need to store in their right-hand side.

Since the index uses a DFUDS for grammar representation, it naturally operates on a parse tree node's depth-first order index, identified in the pseudocode as "prenode", as well as the position of the node's trailing 0 in the bitvector, identified as "node". Diego's index stores the following data structures:

- `grid`: A Labeled Binary Relation structure as explained in Section 2.1.4, internally stores the grid itself (A wavelet tree and a bitvector) as well as `label_to_prenode`, an integer vector mapping lexicographically indexed labels to their depth-first order index. It allows mapping of reverse lexicographically indexed labels to depth-first order index as well with the function `column_to_prenode`.

- `topology_tree`: A DFUDS tree of length $2 * |\#nodes|$, maps nodes to prenodes and allows for swift navigation of the tree in compressed space.

- `sorted_alphabet`: A vector of bytes with the alphabet of the original text, alphabetically sorted.

- `is_terminal_rule`: A sparse bitvector of length $|\#rules|$ that maps the rules to alphabet symbols, which has exactly `sorted_alphabet.size()` 1s. `rank1(r)` finds the index in the sorted alphabet for the rule's symbol, if it is a terminal rule.

- `is_firstmention_prenode`: Sparse bitvector of length $|\#nodes|$ marking a prenode as 1 if it corresponds to the first mention of a rule in the DFUDS (from left to right), or as 0 if it corresponds to a second mention, meaning there is $|\#rules|$ marked 1s. `rank1(i)` finds amount of first mention prenodes up to a certain depth-first order index i. `select1(i)` finds the depth-first order index of the ith first mention prenode. `rank0(i)` and `select0(i)` do the same for second mentions of rules.

- `rule_to_firstmention_prenode`: A reversible permutation (Section 2.1.3) of size $|\#rules|$, stores the first mention prenode of each rule symbol, as well as the rule symbol of each first mention prenode.

- `secondmention_prenode_to_rule`: Sequence of length $|\#nodes - \#rules + 1|$, stores the rule symbol of the ith second mention prenode. Uses `rank(i, r)` to find the amount of secondary appearances of rule r among the first i second mentions. Uses `select(j, r)` to find the mention, among all secondary mentions, of the jth secondary appearance of rule r.

- `is_runlen_prenode`: A sparse bitvector of length $|\#nodes|$, marking whether a prenode represents a run-length rule. `rank1(i)` finds the amount of run-length prenodes up to depth-first order index i.

- `runlen_prenode_arity`: An int vector of length $|\#runlength\_nodes|$ storing the run length of each run-length prenode.

- `is_phrase_start`: A sparse bitvector of length $|text|$ that marks the start position on the text of each phrase created by the grammar tree (same as the start position on the text of each leaf in the grammar tree). `select1(i)` finds the start position of the ith phrase. `rank1(i)` finds the amount of phrases started up to position i.

His implementation of primary occurrence search then follows this structure:

- The program computes $O(\log \sigma)$ cuts of the pattern to evaluate for primary occurrences in the grid, which can potentially be much smaller than the $O(|P|)$ cuts that are normally evaluated.
- The suffix and reversed prefix are generated and the binary searches on the columns and rows of `grid` give us a range from which to extract points.
- The points are extracted, these points are simply the depth-first order of first mention of the right node in the intersection they represent.
- Using the DFUDS we can easily find the parent of this node, which is necessarily also a first mention. We use `is_phrase_start` to find the positions of these nodes' phrases in the text and those values are used to compute the offset we need. The starting position of the parent minus that of the child, minus the cut position in the pattern, is the offset relative to the parent rule where the pattern is.
- Run length rules need special treatment here. A primary occurrence found inside a run parent node necessarily happens at every intersection of the run's children, so `run_length-1` times. These all need to be added to the set of primary occurrences with the proper offsets, calculated by adding the child's size several times to the original offset we had computed.
- We store each primary occurrence as the tuple of the parent's node `node`, prenode `prenode` and offset in the text `node_offset`, as well as the relative offset of the occurrence within the node `offset_pattern`.

Secondary occurrence search then follows as such:

- The program appends all primary occurrence tuples to a queue of data that will be processed in a loop, as well as all secondary mentions of these nodes' rules.
- To do this `secondmention_prenode_to_rule`'s `select(j, r)` method is essential. We retrieve the primary occurrence's rule with `rule_to_firstmention_prenode`'s reverse permutation and use said rule symbol to select all secondary mentions, and retrieve their node and prenode values. Offsets are maintained and need no modification.
- Then, for every item in the queue, we check whether it is the root node, equivalent to checking whether it is prenode value is 1. These nodes can be popped from the queue and reported as occurrences, adding the `offset_pattern` value to our set of results.
- Non-root nodes will be climbed in the tree, we use `topology_tree` to find their parent node, and `is_runlen_prenode` with `runlen_prenode_arity` to check if said parent is a run length node.

- Run length parent nodes need special handling, as they technically contain `run_length` occurrences within. The child rule's expansion length needs to be calculated in the same was as was done during primary occurrence computation. This value is then used to update the `offset_pattern` value for every repetition of the occurrence. Each second mention of the parent also needs to be appended `run_length` times to the queue, using the same process as above.

- Regular parent nodes do not need such special treatment, and only need to be appended to the queue by themselves along with their secondary mentions.

- While adding the parent nodes to the queue we need to update `offset_pattern` to reflect the occurrence's location within the parent rather than the child. It is for this reason that we have kept the `node_offset` value around. This value is the child's offset in the text, and by calculating the parent's offset in the text `parent_offset` in the same way, we can skip having to consider each child's size when updating the offset. Simply, the new offset becomes `new_off = offset_pattern + (node_offset - parent_offset)`.

- For run length parent nodes `parent_offset` and `node_offset` have the same value, nonetheless the parent's size can be divided by its run length to compute the child's size, and for every $i \in [0..run\_length - 1]$ occurrence in the run the offset becomes `new_off = offset_pattern + child_length * i`.

4.2. Implementation

The implementation for this index neatly adapts the logic from Code Block A.1, where the conditions checked are: whether the pattern and context substring $X_i P Y_i$ fits inside the node being processed, as well as whether the X_i or Y_i substrings could not possibly fit because they "spill over" the edges of the text.

A pseudocode representation of these modifications implementation is included in the Appendix Code Block A.4. The modifications are calculated as mentioned in Section 3:

- We require a way to compute the offset in the text where a node's phrase ends, to do this we leverage the parse tree's capacity to find a node's "next sibling", as mentioned in Section 2.1.2. We can then define a node's end offset as the offset in the text of their next sibling. The root's end offset is an exception, since root has no siblings, so it is end offset is the text's length.

- We compute whether the left context is contained inside the node by checking the value of `offset_pattern >= context_length`.

- We do the same for the right context, using the node's end offset, by checking `node_end - (node_offset + offset_pattern + pattern_length) >= context_length`.

- We compute whether the context would spill over the text's left border by checking if `node_offset + offset_pattern < context_length`.

- And once again we do the same for the right border by checking if `text_length - (node_offset + offset_pattern + pattern_length) < context_length`.

- We can early return if both contexts are contained in the node, or if the node is root, or if the context spills over on any of the two sides.

- The return value here is `node_offset + offset_pattern`, which differs from the original `offset_pattern`. The reason for this is that we can no longer guarantee that `node_offset` is 0 when the node is not root.

The pruning modification to the algorithm cuts down on the redundant results being reported, but will not remove them entirely. Because of this, it is necessary that we implement a simple filtering loop.

This loop is represented in pseudocode in the Appendix Code Block A.5, which simply extracts each occurrence's X_iPY_i substring and builds an associative data structure of each substring pointing to the offsets that find it. We then report the smallest offset for every unique substring.

This last bit of code is trivially simple but exceedingly costly as the extraction of a substring from the index requires traversing the tree's entire height and jumping between branches often to retrieve every relevant terminal symbol; this motivates a search for an alternative way to represent and retrieve contexts.

4.2.1. Adding Substring Hash Computation

The fact that this index stores a traversable representation of the Parse Tree, where each node stands for a rule that expands to some substring, and where each node's phrase has an easily retrievable starting position in the text, serves as further motivation. It is well known that modular algebra allows for hashes of substrings to be operated together with ease in constant time (as long as modular division is not necessary).

Because of this, we can devise a scheme for storing hash information alongside the existing data structures, and use it for arbitrary substring hash calculations. Specifically, we store the additional structures:

- `fingerprint_base`: Base of the hash algorithm, larger than the alphabet's size and co-prime to `fingerprint_prime`.

- `fingerprint_power`: Power of the Mersenne prime below, needs to be sufficiently large for us to find a valid base smaller than the prime. There are not many options, however.

- `fingerprint_prime`: Mersenne prime in the form $2^k - 1$, allows the hashing algorithm do perform faster inverse calculation (which we will need when dealing with run-length nodes), and faster modularization as well.

- `prenode_to_sibling_prefix_fingerprint`: An integer vector of length $|\#nodes+1|$ storing the cumulative hash of the expansions of a node's left siblings, `hash(sibling_prefix)`. Meaning the value for any leftmost child of a node is `0`. For run length nodes this stores the hash of the node itself.

- `prenode_to_sibling_prefix_base_power`: An integer vector of length $|\#nodes + 1|$ storing the value of `fingerprint_base^len(sibling_prefix) % fingerprint_prime` for each node. Leftmost children store `1`. For children of run length nodes this stores the base power of the child `fingerprint_base^len(node_expansion) % fingerprint_prime`.

- `runlen_prenode_base_power_predecessor_inverse`: A $|\#runlen_nodes{+}1|$ sized integer vector storing a run-length node's `(fingerprint_base^{len(repeated_child_str)} - 1)^{-1}`, the map's key is the node's rank among all other run-length nodes. This vector allows for $O(1)$ composition of run-length hashes.

We use this information to efficiently calculate the Rabin-Karp Hash of any prefix of the text. Any two such hashes can then be modulo-subtracted to retrieve the hash of any other arbitrary substring. This enables fast context string comparisons when filtering that is, critically, independent of pattern and context length. This contrasts with the algorithm for extracting substrings from the index, which necessitates retrieval of every character, and thus necessarily is at least $O(end - start)$. Hash comparisons are $O(1)$ while retrieval is $O(h)$, the Parse tree's height.

The initialization of these structures is presented as pseudocode in the Appendix Code Block A.6. It follows these steps:

- We first linearly compute the hashes of all text prefixes. We will soon operate on these values to form our other vectors.
- We then iterate over all nodes and retrieve their parent node's information. We require the parent's offset, end offset and run length values specifically.
- For run length parent nodes, the child's size can be computed using the parent's size divided by the run's length. We compute `prenode_to_sibling_prefix_base_power` with this size, and calculate `runlen_prenode_base_power_predecessor_inverse` using that same base power .
- The modular inverse calculation, even for Mersenne primes, is expensive. Thankfully it is only necessary for run length nodes.
- For regular parent nodes, the child node's offset minus that of its parent gives us the "sibling prefix"'s length, for computing `prenode_to_sibling_prefix_base_power`'s value for this node.
- The value of `prenode_to_sibling_prefix_fingerprint` for the node does require our prefix hashes. We retrieve the hash `composed_fp` of the prefix before the child node's offset, and do the same for the parent node's offset to get `parent_fp`. Then as long as the parent's offset is actually greater than 0, meaning as long as the parent's phrase does not border the left edge of the text, we compute the sibling prefix fingerprint by hash-subtracting the hashes, thus calculating the hash of the substring that begins at the parent node's offset and ends at the child's offset. If the parent does border the text's edge, then `composed_fp` is already the value that we need.

Fingerprint retrieval then utilizes the structures as follows:

- When looking for the hash of the substring beginning at some `idx` with a particular `length`, we first compute the hash of the prefix ending at position `idx` as well as the one ending at position `idx+length`. We will then hash-subtract these two prefix hashes to compute the answer we need. Thus we only need worry about computing a prefix's hash as our base case.
- For the base case of a prefix ending at position `length`, a path is followed from the root node of the parse tree all the way to the terminal symbol node at position `length-1`. To do this we will need to make several jumps from any second mention nodes we encounter, to their first mentions.

- During this traversal we keep four mutable values: `result` holds the hash we are computing, `mut_idx` holds the position of the final node we will visit, as far as we know for now, which is initialized to `length-1`, and `node` and `prenode` which are initialized to root's node and prenode values and will store our currently visited node.
- The invariant here is that `result` always stores the hash of the prefix up to the start of the current node's phrase. On every descending step of the loop we hash-add the child node's left sibling's fingerprint, which is saved at `prenode_to_sibling_prefix_fingerprint`, to `result`. Thus we are "sweeping" the text and constructing the prefix's hash by descending the tree.
- The traversal loop begins by asserting if `node` identifies a leaf in the tree, this simply queries whether the `topology_tree` DFUDS stores a 0 at position `node`. Leaves can either be terminals or second mentions of some other node, while non-leaf nodes are always necessarily non-terminal first mentions.
- The non-terminal first mention case requires us to choose which child to visit next and adjust `result` accordingly. For regular nodes we perform a binary search, calculating the starting position of the phrase of the children and comparing it to `mut_idx`.
- Run length nodes technically only store a single child in the parse tree, and therefore we will always jump to that first child, but still need to adjust `mut_idx` by calculating which child would have held our destination `((mut_idx - offset) // child_size) + 1`.
- If the child in question was already the first child, `mut_idx` does not need to change and nor does `result` need adjusting.
- If, instead, `mut_idx` points to some other child, we adjust it to `offset + (mut_idx - offset) % child_size`, the new target position. We then need to hash-add to result the fingerprint of the child's left siblings, but this time the value is not stored inside `prenode_to_sibling_prefix_fingerprint`. it is here that our vector for run-length base powers, `runlen_prenode_base_power_predecessor_inverse`, becomes useful as we will need it to compute the hash of a run of substrings.
- We retrieve The child's hash `child_hash` and base power `child_base` from their respective vectors, alongside the base power's predecessor inverse `child_basepredinv`. The hash of the child's left `sibling_amount` siblings can be computed with the formula $left\_siblings\_hash := modulus(child\_hash \cdot (child\_base^{(sibling\_amount-1)} - 1) \cdot child\_basepredinv)$. We can then hash-add `left_siblings_hash` to `result` and continue our descent.
- Terminal second mention nodes require us to jump to their first mention counterpart. We simply need find this node using `secondmention_prenode_to_rule` to retrieve the rule in question, and compute the prenode as `index.is_firstmention_prenode.select1(rule_to_firstmention_prenode[rule]) + 1`. Then we adjust `mut_idx` by subtracting the difference between the second mention node's offset and the first mention node's offset.
- The terminal first mention node case is our end condition, here we need to extract the symbol using `index.sorted_alphabet[index.is_terminal_rule_rank1(rule)]`, and then hash-add it to the result hash.

- Once the terminal is reached we can return `result` for the base case.

This process is best illustrated in the appended Appendix Code Block A.7. Of note here is that we still need to do at most one binary search or one costly modular algebra operation per step in our descent.

After choosing a good base that guarantees no collisions, we can rewrite the filtering loop from section 4.2 to compute the hashes of the context substrings instead of extracting the strings. The resulting pseudocode is also available in Appendix Code Block A.8.

There is a price though, as the stored integer vectors are large indeed. Fingerprints need to be stored in large integer types to account for the large amount of combinations of substrings for large texts with large alphabets. Because of that, each of the numbers stored can be as large as $\log_2 fingerprint\_prime/8$ bytes for $fingerprint\_power = 61$, so 7.625 bytes each.

Of the three vectors, though `prenode_to_sibling_prefix_base_power` would be easiest to remove, as the `fingerprint_base^len(sibling_prefix) % fingerprint_prime` within can be calculated on the fly during tree traversal and a single exponentiation is not as costly as the other two vectors' data.

Chapter 5

Solution for a RePair RSLP

5.1. The Index

Now we move on to addressing Alejandro Pacheco's RePair Self-Index[2]. Again a pseudocode representation of his implementation is included in Appendix Code Block A.9.

This index's differences to the neutral algorithm and to Diego's LMS index are as follows:

- This index uses RePair as its phrase cutting criteria, and therefore constructs a binary nodes in its parse tree.
- This allows the index to represent its grammar through the same grid that would enable pattern matching, as seen in Section 2.2.2.
- The tree is not completely binary though, as the index actually constructs a relaxed grammar, as seen in Section 2.4. Meaning there is a "root sequence" that stores the root nodes of a binary forest of parse trees. This sequence's symbols are a result of the iterative parsing of the text, which stops when no repeated phrases are found in the sequence. For RePair, this means in practice that the intersecting pairs of nodes in the sequence do not repeat anywhere in the forest.
- Since these sequence's symbols do not really obey the binary node relation of the grid as we know it, a second grid that represents the intersections between sequence symbols has been added, meaning that primary occurrences found between sequence symbol rules are now found on this second structure.
- The guarantee of the pairs of symbols in the sequence not repeating anywhere else allows us to conclude that this second grid will represent unique rule intersections, meaning that the primary occurrences found through this grid are a subset of the total primary occurrences that can not be found through the first grid in any way, and meaning that they do not have any associated secondary occurrences in the binary forest. This means there is a set of primary occurrences that need not trigger parse tree climbing.
- For convenience sake we will coin the term sequence-phrase to denote the substrings that are extracted from sequence symbols. Since the sequence is a compressed representation of the text, the text is a concatenation of these sequence-phrases appended in the sequence's order. We can also say that the second grid stores data about the intersections of sequence-phrases.

- This index implements a system for randomly sampling precomputed results during index construction. Since this system would weaken the comparison with Diego's LMS index, we have decided to avoid interfacing with it. Thus we set the amount of precomputed samples to 0 and will be ignoring this part of Pacheco's implementation.

In order to implement pattern matching, the index stores the following data structures:

- `text_length`: Simply stores the original text's length.
- `compressed_sequence_length`: Stores the amount of symbols in the compressed sequence.
- `terminal_rule_amount`: Stores the amount of terminal rules, here equivalent to the size of the alphabet of the original text.
- `is_ascii_terminal`: Bit vector of length 256 marking the symbols in ASCII that appear in the text. `rank1(256)` returns `terminal_rule_amount`.
- `terminal_ascii_rank_to_terminal_rule`: Integer array of size `terminal_rule_amount` that maps the rank of an alphabet symbol in `is_ascii_terminal` to its terminal rule's id. So if terminal rule `t` expands to alphabet symbol `s`, these two values satisfy $t = terminal\_ascii\_rank\_to\_terminal\_rule[is\_ascii\_terminal.rank1(s)]$.
- `rule_to_expansion_length`: Integer vector of length $\#rules - \#terminals$ that maps non-terminal rule identifiers to the length of their expanded text.
- `reverse_sorted_rule_to_rule`: Integer permutation as seen in Section 2.1.3 that maps reverse (reverse-lexicographically sorted) rule identifiers to direct (lexicographically sorted) rule identifiers, for use with the first grid as described in Section 2.2.2.
- `children_to_parent_grammar_relation`: The first Labeled Binary Relation grid. It stores the grammar itself and enables pattern matching. For any binary rule $A \longrightarrow BC$, this grid allows easy suffix binary search in Right Hand Left (B) rules and easy prefix binary search in Right Hand Right (C) rules. Right Hand Left (B) rules are therefore reverse-lexicographically sorted. Labels stored at grid intersections are the rule id of the parent.
- `is_sequence_phrase_start_offset_in_text`: Sparse bit vector of length $text\_length$ that marks the starting positions of each sequence-phrase in the original text. If the original text were compressed into 3 rules of expanded sizes 10, 15 and 20, the vector will have length 45 and store a 1 in positions 0, 10 and 25.
- `intersecting_rules_to_phrase_rank_sequence_relation`: Second Labeled Binary Relation grid that stores intersections of rules in the compressed sequence, and completes our requirements for pattern matching. For sequence rules intersecting on the form $S = C_1 \dots C_i C_{i+1} \dots C_{|S|}$, this grid allows easy suffix binary search in Left (C_i) rules and easy prefix binary search in Right (C_{i+1}) rules. Left (C_i) rules are therefore reverse-lexicographically sorted. Labels stored at grid intersections are the ranks of the left phrases in the sequence, i.

- it is key to note that `intersecting_rules_to_phrase_rank_sequence_relation` has had an extra point added to it, intersecting a unique symbol U with C_1 and storing the label 0. This serves the purpose of ensuring that the column set of the grid names every single symbol in the sequence, so we gain the power to check whether a rule's symbol is in the sequence with a single query to `nRowsForCol` with the rule's reverse-sorted identifier. The label being 0 also makes it so `is_sequence_phrase_start_offset_in_text`'s `select1` will return the correct offset of 0 when queried with it.

The pattern matching algorithm is deeply shaped by these conditions. it is critical to note form the above that the `children_to_parent_grammar_relation` grid represents rules as opposed to concrete parse tree nodes, so a certain rule r can have an arbitrary amount of different "parents", whereas a node in the LMS index's parse tree could have only a single parent node. The absence of a parse tree data structure also deprives this algorithm of any way of querying a certain rule's absolute position in the text, this data is only available for sequence rules at the root, and can only be accessed by querying the second grid `intersecting_rules_to_phrase_rank_sequence_relation` and using its labels to select in `is_sequence_phrase_start_offset_in_text`, where this second grid serves as the only representation we have of the sequence.

His implementation of pattern matching is also appended as pseudocode in Appendix Code Block A.9. it is important to note that he chooses to compute primary and secondary occurrences together in the same function, since the reader must now be accustomed to the algorithm in question we take the liberty of doing so as well. The function trails the following steps:

- Unlike in Diego's LMS index, this index evaluates all $O(|P|)$ possible cuts of the pattern when looking for primary occurrences.

- The suffix and reverse prefix are generated and the binary searches of the column and row ranges of `children_to_parent_grammar_relation` are executed.

- The points are extracted, this time the grid returns a vector with the symbols of the parent rules that contain the intersection of the row and column in the range. Let us represent these rules as before, using the symbols $A \longrightarrow BC$. We can say now that the points we have acquired are A symbols.

- We use the very same grid to query for the value of B so that we can compute the offset of the occurrence within A. To do this we use the grid's `ithPointForLabel` method as mentioned in Section 2.1.4. We then use `rule_to_expansion_length` to compute B's expanded length and subtract the value of *cut*, this is our $offset$ inside A.

- we store the individual $offset$ and A pairs into a queue for processing, we will be climbing the grammar as before from these starting points.

- Inside the queue's processing loop we retrieve the rule and offset and check whether the rule is in the sequence, which is equivalent to how we checked whether a node is root in the LMS index, since the sequence is our only information source when it comes to the absolute offset of a rule in the text. To do this we use the second grid's `intersecting_rules_to_phrase_rank_sequence_relation.nRowsForCol` query to see if the rule's reverse-sorted identifier has any associated labels.

- If it does, it must mean that these points correspond to symbols in the sequence, we use the grid's `ithRowForCol` method to iterate through them and retrieve the labels at the intersection with `labelAtPoint`. The label in this grid it a sequence symbol's index in the sequence, so `is_sequence_phrase_start_offset_in_text`'s `elect1` method, when called with this index, will return the offset of the symbol in the text. We can now add that offset to the offset of the occurrence within the symbol's rule, computing the absolute offset of the occurrence in the text. This value can be added to our results vector.

- Whether or not a rule shows up in the sequence does not stop it from having parent rules, which themselves may or may not show up in the sequence. This means that we still must continue to climb from this point, and must add the rule's parents to the queue. In this binary forest, a parent node can either have A at its right child forming a $A' \longrightarrow DA$ rule, or as its left child forming a $A' \longrightarrow AE$ rule.

- For the first of the two cases we look for left siblings D of A using the first grid's `nRowsForCol` and `ithRowForCol` methods. We then retrieve the label A' stored between A and D, this is the parent we need to append to the queue. The offset also needs adjustment though, which in this case simply means adding the expansion size of D to the offset we already possess by accessing `rule_to_expansion_length`.

- The second case is similar, we look for right siblings E of A using `nColsForRow` and `ithColForRow`. The value we retrieve is not E though, as the column elements in the grid are reverse-lexicographically sorted, but this $Rrev$ is the value we need to retrieve the label A'. This time the offset does not need any adjustment and can be pushed into the queue as-is.

- Finally it is time to address occurrences found between sequence rules, to do this we must once again go through $O(|P|)$ cuts of the pattern.

- The prefix and suffix are computed once again, but this time they are used to find the appropriate range in `intersecting_rules_to_phrase_rank_sequence_relation`.

- The points extracted are indexes of the intersections in the sequence, and can be turned to offsets by selecting in `is_sequence_phrase_start_offset_in_text`.

- Once the offset is retrieved we only need to subtract the value of the pattern cut to finish computing these occurrences, which can be added straight away to the results since no further grammar climbing is necessary.

5.2. Implementation

Modifications to the index are similar to those done to LMS, but we will detail some key differences here. The pseudocode of our solution is once again included as Appendix Code Block A.10 and differs from the base pattern matching algorithm as follows:

- During the processing of the occurrence queue and the climbing of the tree, we use `rule_to_expansion_length` to calculate whether the context is contained in the rule we are evaluating, and if it is we use the opportunity to report the result early, but only if we have a way to do so. This is because, unlike our implementation for the LMS index, we really have no other wat to turn the relative offset into an absolute offset other than through `is_sequence_phrase_start_offset_in_text`

- Context fits when the relative offset satisfies $offset \geq context\_length$, as well as when $offset + len(pattern) + context\_length \leq rule\_length$ where the rule's length is retrieved as $rule\_length := index.rule\_to\_expansion\_length[rule]$.
- Whether the rule shows up in the sequence, as mentioned, is returned by a query of `nRowsForCol` in `intersecting_rules_to_phrase_rank_sequence_relation`.
- If either the context does not fit or the rule is not on the sequence, we must ensure that traversal continues, as there are still bigger rules that may hold the pattern and context string. So in this case we compute the parent rules and append them to the queue for processing.
- If the rule is in the sequence, we report the resulting offset just as we did before.
- This means that we can only safely prune, by skipping the parent rule computation, when we detect that both the context fits in the rule and said rule is in the sequence at the same time.
- The question of how would this address the issue of contexts that spill over the edges of the text has a simple answer: These occurrences will never be found in a rule that holds the context and pattern substring, because no such rule exists, so the context fitting condition is always false for them. This means that the algorithm works for them in the same way it did before our additions, it will continue traversing until it reaches the sequence symbol that holds them and report the result like any other.
- These results from the secondary occurrence loop are unlike the results from LMS in one important way, these results do not necessarily contain the first offset for every distinct XPY string, therefore even after filtering the occurrences for unique context and extracting the minimal offset of each, we would not be returning results that match the restricted Contextual Pattern Matching definition as presented in Section 2.5. The reason for this is that when we prune paths in this index, in the absence of a parse tree structure, we have no assurance of these path's locations in the tree with respect to the occurrence we report. In other words, we inevitable are pruning paths that would lead to the first occurrence in the context.
- We are now presented with two alternatives, to carry on with these results and simply report the first we have for each unique context, or to look for the actual first appearance of each context after filtering. Since the locate algorithm would only need to be executed once per contextual result, and since this would strengthen our comparison with the LMS implementation, we choose the second alternative.
- What follows, then, after retrieval of the second batch of primary occurrences from `ntersecting_rules_to_phrase_rank_sequence_relation`, is another filtering loop similar to the ones in our modified LMS index. Here we extract the context of each occurrence and store them in a set data structure where each unique context is only stored once.

- Then, in order to address the issue of finding the real first occurrence of each context, we preform a locate operation on each of these contexts and retrieve the minimum element of each result. These are our true first occurrences for every context.

- There is some nuance, though, as the substrings we are extracting can once again be at the edges of the text where the context would spill over to the L blank characters on its side. For this reason we must compute appropriate $start := max(0, offset - context\_len)$ and $end := min(index.text\_length, offset + len(pattern) + context\_len)$ offsets for the extraction.

- The same consideration applies for the locate call on the retrieved context, when we evaluate $len(context) < pattern\_len + 2 * context\_len$ as true we know that it spills outside the text, and then if $first\_offset <= context\_len$ we identify the case where the spilling is over the text's start position. So the locate call's result must be adjusted as $result := first\_offset + len(context) - len(pattern) - context\_len$. Otherwise when none of these special conditions apply we can report $results := first\_offset + context\_len$.

Again it is important to note that locate is only being called once per unique context, which means that the calls to extract still remain the most expensive operations in this filtering loop, just like in the LMS implementation.

5.2.1. Unimplemented Addition of Substring Hash Computation

That said, if extract remains so expensive, the motivation for implementing a hash computation scheme is present here as well. We lacked the time to implement a solution for this in this index, but we take the liberty to suggest one:

- A simple way to ensure we can compute the hash of any prefix in the text would be to store two integer vectors with precomputed hashes.

- The first vector, `phrase_start_to_prefix_hash`, would store the hash of the text up to every sequence-phrase intersection, this vector would therefore have the same length as the sequence.

- The second vector, `rule_to_hash`, would store the hash of every rule's expanded string. It would have length equal to the amount of rules in the grammar.

- To compute the hash of any prefix $T[0..i]$ of the text, we search for the sequence-phrase that contains position i and set our result variable to the hash of the prefix up to said phrase, retrieved from `phrase_start_to_prefix_hash`. Before continuing we subtract the amount of text we have already hashed from i so that it remains an offset relative to the scope of our search, by using `select1` on `is_sequence_phrase_start_offset_in_text` to see the size of the prefix that ends at the sequence-phrase we have selected.

- We then descend into the binary tree created from the rule at the phrase, adjusting our value of i as we descend in order to reach the last terminal in the substring.

- On each step of descend we evaluate whether we will continue through the left or right child of a node. If we choose left then the offset and result need not be updated. If we choose right we hash-add to the result the hash of the left sibling as retrieved from `rule_to_hash` and adjust the offset by subtracting that same sibling's length from i, which is stored in `rule_to_expansion_length`.

- When we finally reach the terminal node at i we simply hash-add the normalized value of its symbol.
- In this way we would manage to compute the hash of any arbitrary prefix of T, and can then operate two prefix hashes of different lengths to compute any arbitrary substring's hash in time proportional to the depth of the fores plus the time required to select the appropriate sequence-phrase.
- For sequence-phrase selection, `is_sequence_phrase_start_offset_in_text.rank1` lets us find the phrase's order in the sequence, which happens to be the label in grid `intersecting_rules_to_phrase_rank_sequence_relation` that stores the intersection of the rule we are looking for with its predecessor. We can then extract the rule's reverse-lexicographically sorted id using `thPointForLabel` on the grid, and turn the reverse id to a direct one using the permutation `reverse_sorted_rule_to_rule`.

Chapter 6

Experimental Comparison

6.1. Setup

The setup here is really simple, we have programmed another function on both indexes that simply executes the contextual filter on a set of non-contextual pattern matching results, with a given *context_length* and added a means to both call non-contextual pattern matching followed by filter as well as to filter externally generated results. We define the following interface for both programs:

- `index <text_file>`: Sub-command that instructs the program to compress a text fie into a self-index. This sub-command was already present before our modifications.
- `access <pos> <selfindex_file>`: Sub-command that instructs the program to extract a single symbol $T[pos]$ from an already compressed self-index. This sub-command was already present before our modifications.
- `range <start> <end> <selfindex_file>`: Sub-command that instructs the program to extract a substring $T[start \dots (end - 1)]$ from an already compressed self-index. This sub-command was already present before our modifications.
- `fingerprint <start> <end> <selfindex_file>`: Sub-command that that instructs the program to compute a substring hash $H(T[start \dots (end - 1)])$ from an already compressed self-index. This sub-command was added by us exclusively to Diego's LMS index because of time constraints.
- `search <pattern> [-L context_length [--non-pruned] [--fingerprint]] <selfindex_file>`: Pattern matching both contextual and not contextual. When `-L context_length` is supplied the contextual pattern matching functions we have added are executed. When `--non-pruned` is supplied the index executes normal pattern matching and then filters it contextually, without pruning paths during tree traversal. When `--fingerprint` is supplied the filtering loop uses hashes instead of substring extraction, but only on Diego's LMS index as discussed.
- `filter -L <context_length> [--fingerprint] <selfindex_file> "<occurrence_0>[ <occurence_i>]*"`: Executes the occurrence filtering loop on a list of pattern matching results with a given `context_length`. When `--fingerprint` is supplied the filtering loop uses hashes instead of substring extraction, but only on Diego's LMS index as discussed.

Other than that we developed the following extra tools for testing the correctness of our results:

- `pointer_generator -n <amount> <text_length>`: An external binary that generates at most $\frac{text\_length \times (text\_length-1)}{2}$ random `<start> <end>` pairs. A specific amount can be requested.

- `pattern_sampler (-n <amount> | --all) -l <length> <text_file>`: An external binary that samples a text file for patterns of a given length. Starts by scanning the file and constructing the set of all substring of said length that exist in the text, then samples randomly in that set. If `--all` is supplied, the program returns the entire set.

- `linear_range_accessor <start> <end> <text_file>`: An external binary that extracts a substring $T[start \ldots (end-1)]$ from the uncompressed text. Uses a simple call to C++'s `substring` method.

- `linear_fingerprint_computer <start> <end> <text_file>`: An external binary that linearly computes the hash $H(T[start \ldots (end-1)])$ of a substring of the uncompressed text. Tested exhaustively for correctness.

- `linear_search_filter -L <context_length> <text_file> "<occurrence_0>[ <occurence_i>]*"`: An external binary that filters a pattern matching result with a given context length, equivalent to the filtering loops we have implemented. Uses the uncompressed text and C++'s `substring` method to compare contexts. Exhaustively tested for correctness as well.

- `linear_contextual_searcher <pattern> [-L context_length] <text_file>`: An external binary that performs pattern matching on the uncompressed text using C++'s `find` method. If `-L context_length` it will call the same filter function as `linear_search_filter` to filter for unique contexts.

We used these tools to exhaustively test our algorithms and implementations for correctness using small text files of up to `5 MB` in size, comparing pre and post-filter results of pattern matching executions for both indexes against this benchmark. `pattern_sampler`'s `--all` option was used to ensure all possible patterns inside a compressed text returned correct pattern matching results. `linear_fingerprint_computer` was also used to confirm correctness of LMS's hashing implementation.

For our experiments we choose to evaluate results using https://pizzachili.dcc.uchile.cl/'s collections of highly repetitive text, there we choose to use the biggest file from each collection. We have replaced all newlines with the `~` symbol for better integration with the programs, which assume newline separators on batch input data. These files' names, descriptions, sizes and alphabet sizes (σ) are:

- Einstein (`446MB`, $\sigma = 140$): English natural language text collection of wikipedia articles about Albert Einstein and other related topics.

- Coreutils (`196MB`, $\sigma = 237$): GNU coreutils versioned source code collection.

- Kernel (`246MB`, $\sigma = 161$): Linux kernel versioned source code collection.

- Escherichia Coli (`107MB`, $\sigma = 16$): Complete genome sequence of Escherichia coli strain K-12, split into overlapping fragments.
- Influenza (`148MB`, $\sigma = 16$): Concatenated Hemagglutinin (HA) gene sequences from thousands of influenza virus strains (primarily H5N1).
- Para (`409MB`, $\sigma = 6$): Mitochondrial DNA sequences from parasitic protists.
- Cere (`440MB`, $\sigma = 6$): Complete genome sequence of Saccharomyces Cerevisiae (baker's yeast).

During self-index construction we measure the following metrics:

- `text_size`: Size of the original text before compression, in bytes.
- `index_size`: Size of the compressed self-index, including all data structures within, in bytes.

During experiment execution we will measure the following metrics:

- `total_patterns`: Size of the batch of patterns searched.
- `total_occurrences`: Amount of occurrences returned.
- `total_search_time`: Total millisecond time of execution.
- `total_nodes_traversed`: Total amount of nodes/rules traversed in the grammar during the whole execution.
- `primary_occurrences`: Amount of occurrences retrieved from grid data structures.
- `primary_occ_time`: Total millisecond time spent retrieving primary occurrences.
- `secondary_occurrences`: Amount of occurrences retrieved from grammar traversal.
- `secondary_occ_time`: Total millisecond time spent retrieving secondary occurrences.
- `filtered_occurrences`: Amount of occurrences that got context-filtered before returning. When this metric applies it satisfies $total\_occ = prim\_occ + secd\_occ - filt\_occ$.
- `filtered_occ_time`: Total millisecond time spent context-filtering occurrences.

We set up the following experiment execution loop:

- For every pair $(idx, text) \in \{\text{LMS}, \text{SLP}\} \times \{\text{eins}, \text{core}, \text{kern}, \text{coli}, \text{infz}, \text{para}, \text{cere}\}$ we compute the appropriate compressed self-index of the text file.
- For every *pattern_len* where $pattern\_len \in \{5, 10, 20, 40\}$ we sample 100 random patterns from the *text* using `pattern_sampler`.
- We then execute a non-contextual pattern matching *search* query with *idx* and store the results and metrics.
- For every *context_len* where $context\_len \in \{10, 50, 100\}$ we execute between 2 and 4 additional queries on *idx* depending n whether it has hashing support.

- First we execute $filter$ on the results of our non-contextual search and store results and metrics. This ensures we have a filtered search point of comparison where no branch pruning was ever executed during secondary occurrence computation.
- Second we execute *seach* again from scratch, but this time with the contextual argument `-L context_len` and store results and metrics. This call executes branch pruning during secondary occurrence computation.
- Third, when $idx = LMS$, we execute $filter$ as before but this time with the `--fingerprint` argument, so that the filtering loop uses hashes instead of substring extraction. We store results and metrics.
- Fourth, when $idx = LMS$, we execute *search* also as before, this time with the `--fingerprint` argument. We store results and metrics.

We then use the metrics we have stored along the way to generate plots for data comparison. We have set up the following data retrieval and plotting logic:

- We begin by parsing all log files for their metrics data.
- We create plots that compare the original text size v/s compressed self index size in bytes across all $(text, idx)$ pairs, using the logs from indexing.
- For every *text* we generate:
  - A `total_occurrences` plot: Compares total pattern matching occurrences for every pattern length and context length combination, including non-contextual pattern search. Aggregates pruned and non-pruned approaches since they report the same amount of results. This plot allows easy visualization of the difference in amount of occurrences reported between non-contextual pattern matching and contextual pattern matching for different lengths.
- Then for every distinct $(text, idx)$ combination we generate 4 additional plots:
  - A `secondary_occurrences` plot: Serves a similar function to the above plot, allows easy comparison between the amount of occurrences that emerge from grammar traversal in non-contextual and contextual pattern matching calls. Also evaluates every pattern length and context length combination as well as the non-contextual case. Aggregates non-pruned results to the non-contextual case, since these have the same value also.
  - A `total_nodes_traversed` plot: Same as the two cases above, presents the amount of nodes or rules traversed during secondary occurrence retrieval. Contrasts every pattern length and context length combination as well as the non-contextual case. Also aggregates non-pruned results to the non-contextual case.
  - A `filtered_occurrences` plot: Presents the amount of occurrences that had to be filtered before results could be reported, contrasting pruned to non-pruned approaches. there is no need to include non-contextual data here, since it does not filter any results. The plot allows for easy visualization of how effective the pruning of paths is to the overall amount of context filtering that needs to happen.

  - A `total_search_time` plot: Also includes the values of `primary_occ_time` and `secondary_occ_time`, here added into the new metric `traversal_time`. This allows us to see the difference between the non-filtering time `traversal_time` and the total search time for every distinct run type. We include non-contextual runs here as well even though their traversal time and total time are the same. This plot also allows us to compare total search times across the pattern length and context length dimensions.

- These plots are logarithmically scaled for improved readability, because the difference in sheer magnitude between many of their values would push certain results too far to the edges of the plot's range. The values of the bars are annotated though, so that the amounts can be known without needing to look at the edges of the plot.

- Since the total search time plots show a comparison between the time spent retrieving occurrences and the time spent filtering them, and both of these times are plotted logarithmically, we can not quite compare the amounts linearly visually. To solve this failing of relative logarithmic comparison, we include an annotation of what percentage of total search time is spend retrieving occurrences. This annotated percentage is shown on top of both bars.

- The non-pruned cases presented in the total search time plot require some extra arithmetic, since filtering and pattern matching were executed in different processes. The total pattern matching time from the non-contextual result is set as traversal time, and added to the filtering time from the filter process to compute the total search time of these points.

- Finally, for every distinct ($pattern\_length, context\_length$) combination, including the non-contextual case, we generate two plots:

  - We compute the set of collections with large alphabets to be [$einstein, coreutils, kernel$], where as the set of collections with small alphabets is [$escherichia\_coli, para, cere$].
  - For each of these two sets we plot `total_search_time` and `traversal_time` much like in the plot mentioned above, but this since $pattern\_length$ and $context\_length$ are fixed we gain the ability to compare execution times between combinations of ($idx, pruning, filter$). We sort idx on highest priority , followed by $filter$ and finally $pruning$.
  - We also compute the proportion between each non-pruned total and its pruned counterpart and add a vertical bracket labeled by this proportion as a percentage. This bracket, placed on top of the smallest of the two bars, allows us to see how much time is saved by pruning, despite the inconvenience presented by the logarithmic scaling of the Y axis.

6.2. Results

In the code we launch these processes in parallel batches of two, making use of the fact that the benchmarking hardware possesses 4 cores. Said hardware is comprised of a single Thinkpad T480s Laptop running Manjaro Linux kernel version 6.15.3-1, on a 4 × Intel® Core™ i5-7300U CPU @ 2.60GHz processor. The processes were launched on optimal priority

(-20 Niceness) while the system processed no other loads, and swap memory usage was disabled. 11.6 GiB of RAM proved sufficient for such a task considering compressed text sizes.

All figures, 1 for size comparisons, 4 per $(idx, text)$ for internal search metrics, 1 more per *text* with total occurrences, and 2 per $(pattern\_length, context\_length)$ for execution times, can be found in Appendix Section B.

It is from these figures that we make the following observations:

- The size comparisons plot B.1 shows the LMS index taking considerable amounts of storage space more than SLP, most likely as consequence of the additional data structures we have introduced for fingerprint computation. The storage space for the Escherichia Coli collection in particular stands out, as the self-index's storage surpasses that of its original text.
- The total occurrences plot of every *text* collection shows how pattern length is inversely correlated to occurrence amount of both contextual and non-contextual queries, while context length has the opposite relation. Plot B.2 serves as illustration.
- The secondary occurrences plot of every $(idx, text)$ shows a similar relationship as the total occurrences plot, this time aggregating non-pruned results to the non-contextual case. Of special note for the pruned case is the fact that the context length also correlates directly with the amount of secondary occurrences. It is also important to note that the difference between pruned and non-pruned metrics gets starker for lower pattern lengths. Plots B.3 and B.7 serve as illustration.
- The total nodes traversed plot of every $(idx, text)$ displays data in nearly the same proportions as the secondary occurrence plot, as expected considering grammar traversal happens exclusively during secondary occurrence computation. The same two noteworthy points show as well. Plots B.4 and B.8 serve as illustration.
- The filtered occurrences plot of every $(idx, text)$ displays:
  - An inversely proportional correlation between pattern length and amount of filtered occurrences, and a direct correlation instead between context length and filtered occurrences.
  - The difference between pruned and non-pruned filtered occurrences almost always remains within a single order of magnitude.
  - The difference is starker for the SLP index than it is for LMS on all collections. The difference is also starker on collections with large alphabets: Kernel, Coreutils and Einstein.
  - Comparing plots B.5 and B.9 illustrates the index choice's effect, and comparing plots B.5 and B.59 illustrates the alphabet size's effect.
- The total search time plot of every $(idx, text)$ is the most dense in information, it displays the following:
  - There is very meaningful difference in magnitude between traversal time and total search time for both pruned and non-pruned executions.

  - Comparing pruned and non-pruned traversal times, we find the same trend as the filtered occurrences plot, where differences get starker with increasing collection alphabet size.
  - Sometimes a very small difference between pruned and non-pruned traversal time can translate to a much larger difference in total search time, as see in plot B.28.
  - On LMS, the plot also shows how fingerprint-based filtering is almost always slower than string-based filtering. The einstein collection is the only one where this tendency is consistently inverse, suggesting a relationship to alphabet size. Plots B.51 and B.6 illustrate this well.

- The total search time comparison plots of every $(pattern\_length, context\_length)$ display:
  - SLP is generally the faster index, this is especially noticeable on smaller $context\_length$ values where the difference is near order-of-magnitude. Plot B.65 illustrates this relationship.
  - Fingerprint-based filtering is generally slower than string-based filtering, even for very large context and patten lengths. Some combinations do allow the fingerprint-based filter to surpass its counterpart though, as seen in plots B.69 and B.77, where context length is large and pattern length small.
  - The best predictor of pruning effectiveness is the size of the collection's alphabet. Larger alphabets exhibit a larger difference between pruned and non-pruned execution times. Smaller pattern and context lengths also contribute to pruning effectiveness to a lesser extent. This is best exemplified by plots B.81 and B.82.

Chapter 7

Discussion

Longer patterns reduce occurrence counts, while longer contexts increase occurrence counts. These correlations come down to statistics and the nature of the query. Longer patterns must recur less often and eventually converge to a single occurrence of P when $P = T$. Longer contexts mean there are more characters to differentiate between occurrences, and eventually all occurrences become distinct as L approaches $|T|$.

We measured increased pruning effectiveness on smaller pattern/context lengths, which is a consequence of the grammar's structure. As we climb the grammar, rule expansion lengths increase and the difference between child expansion length and parent expansion length also grows. Because of this, larger input parameters require deeper traversal before the pruning condition is satisfied, increasing computational cost.

Notably, filtered occurrences were measured to follow the same trend as total occurrences. This means that the amount of results responds more rapidly to changes in pattern length and context length than pruning effectiveness does.

The most noteworthy feature of total time plots remains the overwhelming difference between traversal and total time values for the same execution. Traversal time for small patterns stays near 0.5 % of the total, and only ever reaches past 10 % for very large patterns on LMS. This means that the filtering loop dominates runtime, and reaffirms the hypothesis that pruning is critical to tackle this problem.

Unexpectedly, sometimes large decreases in traversal time do not correlate to equally large decreases in total time. We attribute this to CPU caching: Filtering repeated contexts increases the chance that the filtering loop will traverse the same paths in the grammar in close proximity, which increases the chance of cache hits and compensates for the greater amount of contexts that bypassed pruning.

Another noteworthy finding is that the modified SLP index is overall faster than its LMS counterpart despite the lack of local consistency, and this difference is seen in total time, traversal time but also on the total amount of filtered occurrences for each run type. This speed is therefore a consequence of these two factors:

- Grammar shape: The grammar not being binary, and the way it is constructed where rule creation aligns with the most repeated substring of each parsing round, might result in wider rules that catch pruning condition evaluations early.

- Efficiency of rule parent access: The SLP grammar is traversed directly, evaluating distinct rules on each step. LMS evaluates nodes in the parse tree instead, and multiple parse tree nodes represent different mentions of the same rule. As such the amount of nodes traversed on LMS is expected to be larger than the amount of rules traversed on SLP.

A node in LMS's parse tree only ever has a single parent node, otherwise it would not be a "tree". To find the parents of the rule of any node we need to iterate over all of its mentions first, each having a single parent. On SLP we instead traverse rules directly, and every rule has direct access to all of its parents through the grid.

This means in practice that when pruning condition gets evaluated to true on SLP, we skip processing its parents in a single step. On LMS we need to skip the parents of every secondary mention of the same rule, and must therefore still traverse these non-terminal leaf nodes.

A solution that evaluates the pruning condition of a node before it and its secondary mentions are added to the processing queue might fix this issue on LMS.

We measured increased pruning effectiveness for large alphabet, natural-language-like collections, and especially Einstein among them. The effect is reflected in total time, traversal time and filtered occurrence metrics. The size of repeated substrings in these collections is bigger which leads to fewer unique contexts. Einstein and Kernel achieve exceptionally good compression ratios, which confirms this reality and suggests a large amount of rules that are not root but are still large enough to satisfy the pruning condition. This allows the algorithm to prune earlier and more often.

Finally, we discuss the failure of fingerprint retrieval within the filtering loop. The ≈ 4 bytes/rule storage overhead of this modification yields minimal speed gains, while making it cost-ineffective for most configurations. It only speeds up the filtering loop for very large context lengths and on very small grammars, like Einstein's. The fact that LMS implements run-length rules does not help, as the hash calculation for these nodes is much more expensive.

Chapter 8

Conclusions

Our results reveal an increase in the amount of reported contextual occurrences as pattern lengths decrease and context lengths increase. Pruning effectiveness inversely correlates with both parameters but is outpaced by their impact on total occurrences. Pruning is most effective on large alphabet natural-language-like collections, especially because of their improved compression ratios. Traversal time is always a small fraction of total search time, demonstrating that the filtering loop is by far the most expensive part of both implementations. SLP is generally faster and also features better pruning capabilities due to the properties of traversal through its data structures and the shape of the grammar. Fingerprint calculation during the filtering loop fails at reducing processing speeds in most configurations, only excepting very large context lengths, and implies an impractical storage overhead of ≈ 4 bytes/rule.

Therefore our first conclusion is that this approach presents a fundamental trade-off:

- Accepting the large superset of pruned but unfiltered occurrences, containing orders of magnitude of context-redundant occurrences, especially on small pattern lengths, but making full use of the speedup in traversal time that arises from pruning.
- Accepting the prohibitive time cost increase of context-filtering the results, only viable when filtering small contexts on collections with large alphabets, conditions that guarantee high pruning effectiveness.

Pruning demonstrably improves execution times for grammar-based indexes attempting to solve this query, but this effectiveness is heavily dependent on properties of the collection being processed, particularly its natural-language-likeness.

Our systematic exploration of alternative solutions during research concluded that context filtering is fundamentally necessary for eliminating redundant results given grammar structures as we have presented them.

There is, therefore, still work to be done:

- We propose investigating whether improved pruning condition is possible, leveraging a context-tracking data structure for substrings within the grammar.
- There is space for further exploration on how different grammar structures react to the query. A larger binary grammar with a constant expansion rate of 1 character on every step toward root could see better results at the cost of compromised compression ratios by increasing the amount of non-root nodes that satisfy the pruning condition, but any imbalance when it comes to the ratio of left-sided rule growth to right-sided rule growth

would affect execution times, and traversal would be affected by the grammar's increased size.

- The context-filtering loop of results could be further improved in ways that do not require the extraction of an entire context substring. A faster, differently precomputed hashing implementation might solve our current limitations, for example.

Bibliography

[1] Navarro, G., “Indexing highly repetitive string collections, Part II: Compressed indexes”, ACM Comput. Surv., vol. 54, 02 2021, doi:10.1145/3432999.

[2] Claude, F., Navarro, G., y Pacheco, A., “Grammar-compressed indexes with logarithmic search time”, J. Comput. Syst. Sci., vol. 118, pp. 53–74, 2021, doi:10.1016/J.JCSS.2020.12.001.

[3] Navarro, G., “Contextual pattern matching”, en String Processing and Information Retrieval (SPIRE 2020) (Boucher, C. y Thankachan, S. V., eds.), vol. 12303 de Lecture Notes in Computer Science, pp. 3–10, Springer, 2020, doi:10.1007/978-3-030-59212-7_1.

[4] Clark, D., Compact PAT Trees. Tesis PhD, University of Waterloo, 1996. PhD Thesis.

[5] Munro, J. I., Raman, R., Raman, V., y Rao, S. S., “Succinct representations of permutations”, en Automata, Languages and Programming (ICALP 2003) (Baeten, J. C. M., Lenstra, J. K., Parrow, J., y Woeginger, G. J., eds.), vol. 2719 de LNCS, pp. 345–356, Springer, 2003, doi:10.1007/3-540-45061-0_29.

[6] Raman, R., Raman, V., y Rao, S. S., “Succinct indexable dictionaries with applications to encoding k-ary trees and multisets”, en Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms, January 6-8, 2002, San Francisco, CA, USA (Eppstein, D., ed.), pp. 233–242, ACM/SIAM, 2002, http://dl.acm.org/citation.cfm?id=545381.545411.

[7] Grossi, R., Gupta, A., y Vitter, J. S., “High-order entropy-compressed text indexes”, en Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms, January 12-14, 2003, Baltimore, Maryland, USA, pp. 841–850, ACM/SIAM, 2003, http://dl.acm.org/citation.cfm?id=644108.644250.

[8] Ferragina, P., Manzini, G., Mäkinen, V., y Navarro, G., “Compressed representations of sequences and full-text indexes”, ACM Trans. Algorithms, vol. 3, no. 2, p. 20, 2007, doi:10.1145/1240233.1240243.

[9] Benoit, D., Demaine, E. D., Munro, J. I., Raman, R., Raman, V., y Rao, S. S., “Representing trees of higher degree”, Algorithmica, vol. 43, no. 4, pp. 275–292, 2005, doi:10.1007/S00453-004-1146-6.

[10] Navarro, G. y Sadakane, K., “Fully functional static and dynamic succinct trees”, ACM Trans. Algorithms, vol. 10, no. 3, pp. 16:1–16:39, 2014, doi:10.1145/2601073.

[11] Mäkinen, V. y Navarro, G., “Rank and select revisited and extended”, Theor. Comput. Sci., vol. 387, no. 3, pp. 332–347, 2007, doi:10.1016/J.TCS.2007.07.013.

[12] Barbay, J., Claude, F., y Navarro, G., “Compact rich-functional binary relation representations”, en LATIN 2010: Theoretical Informatics, 9th Latin American Symposium,

Oaxaca, Mexico, April 19-23, 2010. Proceedings (López-Ortiz, A., ed.), vol. 6034 de Lecture Notes in Computer Science, pp. 170–183, Springer, 2010, doi:10.1007/978-3-642-12200-2_17.

[13] Díaz-Domínguez, D., Navarro, G., y Pacheco, A., "An lms-based grammar self-index with local consistency properties", en String Processing and Information Retrieval - 28th International Symposium, SPIRE 2021, Lille, France, October 4-6, 2021, Proceedings (Lecroq, T. y Touzet, H., eds.), vol. 12944 de Lecture Notes in Computer Science, pp. 100–113, Springer, 2021, doi:10.1007/978-3-030-86692-1_9.

[14] Kärkkäinen, J. y Ukkonen, E., "Lempel-ziv parsing and sublinear-size index structures for string matching", en Proceedings of the 3rd South American Workshop on String Processing (WSP) (Gonnet, G. H., Baeza-Yates, R., y Ziviani, N., eds.), vol. 5 de International Series in Computing Science, pp. 141–155, Carleton University Press, 1996.

[15] Navarro, G., "Indexing text using the ziv-lempel trie", J. Discrete Algorithms, vol. 2, no. 1, pp. 87–114, 2004, doi:10.1016/S1570-8667(03)00066-2.

[16] Belazzougui, D., Navarro, G., y Valenzuela, D., "Improved compressed indexes for full-text document retrieval", J. Discrete Algorithms, vol. 18, pp. 3–13, 2013, doi:10.1016/J.JDA.2012.07.005.

[17] Gagie, T., Navarro, G., y Prezza, N., "Optimal-time text indexing in bwt-runs bounded space", en Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, USA, January 7-10, 2018 (Czumaj, A., ed.), pp. 1459–1477, SIAM, 2018, doi:10.1137/1.9781611975031.96.

[18] Abedin, P., Chubet, O. A., Gibney, D., y Thankachan, S. V., "Contextual pattern matching in less space", en Data Compression Conference, DCC 2023, Snowbird, UT, USA, March 21-24, 2023 (Bilgin, A., Marcellin, M. W., Serra-Sagristà, J., y Storer, J. A., eds.), pp. 160–167, IEEE, 2023, doi:10.1109/DCC55655.2023.00024.

Appendix

Annex A. Pseudocode Blocks

A.1. Neutral Base

Code Block A.1: Contextual Occurrence Retrieval Pseudocode.

```
def locate_contextual_occurrences(
        primary_occurrences: List[Tuple[int, int]],
        pattern_length: int,
        context_length: int,
        index: GrammarIndex
    ) -> List[int]:
    occurrence_queue = []
    results = []

    for rule, offset_in_rule in primary_occurrences:
        if index.isRoot(rule):
            results.append(offset_in_rule)
            continue
        occurrence_queue.append((rule, 0, offset_in_rule))
        def add_secondary(s_rule, s_mention):
            occurrence_queue.append((s_rule, s_mention, offset_in_rule))
        index.forAllSecondMentions(rule, add_secondary)

    while occurrence_queue:
        current_rule, current_mention, current_offset = occurrence_queue.pop(0)
        node_offset = index.positionInText(current_rule, current_mention)
        total_offset = node_offset + current_offset
        node_end = index.nodeEndOffset(current_rule, current_mention)
        is_root = index.isRoot(current_rule)
        fits_in_node = (current_offset >= context_length and
                       (node_end - total_offset) >= (context_length + pattern_length))
        fits_in_text = (total_offset >= context_length and
                       (index.textLength() - total_offset) >= (context_length + pattern_length))
        if is_root or fits_in_node or not fits_in_text:
            results.append(total_offset)
            continue
        parent_rule, offset_in_parent = index.parentOf(current_rule, current_mention)
        occurrence_queue.append((parent_rule, 0, offset_in_parent))
        def add_parent_secondary(s_rule, s_mention):
            occurrence_queue.append( (s_rule, s_mention, offset_in_parent) )
        index.forAllSecondMentions(parent_rule, add_parent_secondary)
    return results

def filter_contextual_occurrences(
        occurrences: List[int],
        pattern_length: int,
        context_length: int,
        index: GrammarIndex
    ) -> List[int]:
    context_buckets: Dict[str, List[int]] = []
    for offset in occurrences:
        start: int = max(0, offset-context_length)
        end: int = min(index.textLength()-1, offset+pattern_length+context_length-1)
        context: str = index.extract(start, end)
        context_buckets[context].append(offset)
    results: List[int] = []
    for _, offsets in context_buckets.items():
        results.append(min(offsets))
    return results

```

A.2. LMS Base

Code Block A.2: Primary Occurrence Retrieval Pseudocode for LMS.

```
def search_primary_occurrences_lms(
        pattern: str,
        index: GrammarIndex
    ) -> List[Tuple[int, int, int, int]];
    pattern_boundaries: List[int] = parseHierarchicalLMS(pattern)
    for cut in pattern_boundaries:
        prefix: str = pattern[0:cut-1]
        rev_prefix: str = prefix[::-1]
        suffix: str = pattern[cut:len(pattern)-1]
        row0, row1 = index.grid.binSearchColRange(suffix)
        col0, col1 = index.grid.binSearchRowRange(rev_prefix)
        grid_points: List[int] = index.grid.labelsInRange(col0, col1, row0, row1)
        primary_occurrences: List[Tuple[int, int, int, int]] = []
        for right_rule in grid_points:
            # Offset of the node's phrase in the text
            node_off = index.is_phrase_start.select1(index.topology_tree.rank00(node)+1)
            parent = index.topology_tree.parent(node)
            parent_prenode = index.topology_tree.node_to_prenode(parent_rule)
            parent_off = index.is_phrase_start.select1(index.topology_tree.rank00(parent)+1)
            offset_in_parent = (parent_off - node_off) - cut

            run_len = index.runlen_prenode_arity[index.is_runlen_prenode.rank1(
   ↪ parent_prenode)]
            if run_len > 0:
                child_size = parent_off - node_off
                while run_len > 1:
```

```
                primary_occurrences.append(
                    (parent_node, parent_prenode, parent_off, (parent_off - node_off) - cut)
                )
                node_off += child_size
                run_len -= 1
        else:
            primary_occurrences.append(
                (parent_node, parent_prenode, parent_off, offset_in_parent)
            )
    return primary_occurrences

```

Code Block A.3: Secondary Occurrence Retrieval Pseudocode for LMS.

```
def search_secondary_occurrences_lms(
        primary_occurrences: List[Tuple[int, int, int, int]],
        index: GrammarIndex
    ) -> List[int]:
    results = []
    occurrence_queue = []

    # Captures occurrence_queue
    def add_node_and_secondmentions(node: int, prenode: int, off_node: int , off_pattern:
    ↪ int):
        occurrence_queue.append((
            node,
            prenode,
            off_node,
            off_pattern
        ))
        rule = index.prenode_to_rule[prenode]
        secondary_mention_amount = index.secondmention_prenode_to_rule.rank(index.
    ↪ secondmention_prenode_to_rule.size(), rule)
        for counter in range(secondary_mention_amount):
            secondmention_prenode_rank = index.secondmention_prenode_to_rule.select(
    ↪ counter+1, rule)
            secondmention_prenode = index.is_firstmention_prenode.select0(
    ↪ secondmention_prenode_rank+1)+1
            secondmention_node = index.topology_tree.prenode_to_node(
    ↪ secondmention_prenode)
            secondmention_node_offset = index.is_phrase_start.select1(index.topology_tree.
    ↪ rank00(secondmention_node)+1)
            occurrence_queue.append((
                secondmention_node,
                secondmention_prenode,
                secondmention_node_offset,
                off_pattern
            ))

    for node, prenode, off_node, off_pattern in primary_occurrences:
        if prenode == 1:
            results.append(off_pattern)
            continue
        add_node_and_secondmentions(node, prenode, off_node, off_pattern)

    while occurrence_queue:
        current_node, current_pre, current_off, current_pat = occurrence_queue.pop(0)
        total_offset = current_off + current_pat
        if current_pre == 1:
            results.append(current_pat)
            continue

        # Parent resolution with topology tree
        parent_node = index.parent(current_node)
        parent_pre = index.node_to_prenode(parent_node)
        run_len = index.runlen_prenode_arity[index.is_runlen_prenode.rank1(parent_pre)]

        # Run-length parent handling
        if run_len > 0 and (parent_pre + 1 == current_pre):
            # Calculate first child length
            second_child_off = index.is_phrase_start.select1(index.topology_tree.rank00(index.topology_child(parent_node, 2))+1)
            child_len = second_child_off - current_off
            # Insert run-length expanded occurrences
            for i in range(run_len):
                new_off = current_pat + child_len * i
                add_node_and_secondmentions(parent_node, parent_pre, current_off, new_off)
        else:
            parent_off = index.is_phrase_start.select1(index.topology_tree.rank00(parent_node)+1)
            new_off = current_pat + (current_off - parent_off)
            add_node_and_secondmentions(parent_node, parent_pre, parent_off, new_off)

    return results

```

A.3. LMS Modified

Code Block A.4: Contextual Occurrence Retrieval Pseudocode for LMS.

```
def locate_contextual_occurrences_lms(
        primary_occurrences: List[Tuple[int, int, int, int]],
        pattern_length: int,
        context_length: int,
        index: GrammarIndex
    ) -> List[int]:
    results = []
    occurrence_queue = []

    def node_end_offset(node: int, prenode: int) -> int:
        if prenode == 1:
            return index.text_length
        else:
            next_sibling = index.topology_tree.next_sibling(node)
            return index.is_phrase_start.select1(index.topology_tree.rank00(next_sibling)+1)

    # Captures occurrence_queue
    def add_node_and_secondmentions(node: int, prenode: int, off_node: int , off_pattern:
     ↪ int):
        occurrence_queue.append((
            node,
            prenode,
            off_node,
            off_pattern
        ))
        rule = index.prenode_to_rule[prenode]
        secondary_mention_amount = index.secondmention_prenode_to_rule.rank(index.
     ↪ secondmention_prenode_to_rule.size(), rule)
        for counter in range(secondary_mention_amount):
            secondmention_prenode_rank = index.secondmention_prenode_to_rule.select(
     ↪ counter+1, rule)
            secondmention_prenode = index.is_firstmention_prenode.select0(
     ↪ secondmention_prenode_rank+1)+1
            secondmention_node = index.topology_tree.prenode_to_node(
     ↪ secondmention_prenode)
            secondmention_node_offset = index.is_phrase_start.select1(index.topology_tree.
     ↪ rank00(secondmention_node)+1)
            occurrence_queue.append((
                secondmention_node,
                secondmention_prenode,
                secondmention_node_offset,
                off_pattern
            ))

    for node, prenode, off_node, off_pattern in primary_occurrences:
        if prenode == 1:
            results.append(off_pattern)
            continue
        add_node_and_secondmentions(node, prenode, off_node, off_pattern)

    while occurrence_queue:
        current_node, current_pre, current_off, current_pat = occurrence_queue.pop(0)
        total_offset = current_off + current_pat
        node_end = node_end_offset(current_node, current_pre)

        # Context check conditions
        left_context = current_pat
        right_context = node_end - (total_offset + pattern_length)
        text_right_available = index.text_length() - (total_offset + pattern_length)
        is_root = (current_pre == 1)
        fits_in_node = (left_context >= context_length and
                        right_context >= context_length)
        left_fits_text = (total_offset >= context_length)
        right_fits_text = (text_right_available >= context_length)

        if is_root or fits_in_node or not (left_fits_text and right_fits_text):
            results.append(total_offset)
            continue

        # Parent resolution with topology tree
        parent_node = index.parent(current_node)
        parent_pre = index.node_to_prenode(parent_node)
        run_len = index.runlen_prenode_arity[index.is_runlen_prenode.rank1(parent_pre)]

        # Run-length parent handling
        if run_len > 0 and (parent_pre + 1 == current_pre):
            # Calculate first child length
            second_child_off = index.is_phrase_start.select1(index.topology_tree.rank00(index.topology_child(parent_node, 2))+1)
            child_len = second_child_off - current_off
            # Insert run-length expanded occurrences
            for i in range(run_len):
                new_off = current_pat + child_len * i
                add_node_and_secondmentions(parent_node, parent_pre, current_off, new_off)
        else:
            parent_off = index.is_phrase_start.select1(index.topology_tree.rank00(parent_node)+1)
            new_off = current_pat + (current_off - parent_off)
            add_node_and_secondmentions(parent_node, parent_pre, parent_off, new_off)

    return results

```

Code Block A.5: Contextual Occurrence Filtering Pseudocode for LMS.

```
def filter_contextual_occurrences_lms(
        occurrences: List[int],
        pattern_length: int,
        context_length: int,
        index: GrammarIndex
    ) -> List[int]:
    context_buckets: Dict[str, List[int]] = []
    for offset in occurrences:
        start: int = max(0, offset-context_length)
        end: int = min(index.textLength()-1, offset+pattern_length+context_length-1)
        context: str = index.extract(start, end)
        context_buckets[context].append(offset)
    results: List[int] = []
    for _, offsets in context_buckets.items():
        results.append(min(offsets))
    return results

```

Code Block A.6: Substring Fingerprint Computation for LMS.

```
def compute_rabin_karp_signatures(input_data: bytes, index: GrammarIndex):
    # Step 1: Compute cumulative fingerprints for entire text
    cumulative_fingerprints = [0]
    current_fp = 0

    def node_offset(node: int, prenode: int) -> int:
        return index.is_phrase_start.select1(index.topology_tree.rank00(node)+1)

    def node_end_offset(node: int, prenode: int) -> int:
        if prenode == 1:
            return index.text_length
        else:
            next_sibling = index.topology_tree.next_sibling(node)
            return index.is_phrase_start.select1(index.topology_tree.rank00(next_sibling)+1)

    def compose_fingerprints(prefix, suffix, suffix_length):
        modulus(prefix*suffix_length + suffix) # Real code uses fast modulus computation of
        ↪ Mersenne primes

    def base_power(exponent):
        modulus(base ^ exponent)

    def predecessor_inverse(power):
        modulus((power - 1) ^ (-1)) # Real code uses fast modular inverse computation of
        ↪ Mersenne primes

    def subtract_prefix_fingerprint(composed, prefix, suffix_length):
        modulus(composed - prefix * (base ^ suffix_length))

    for byte in input_data:
        symbol = index.normalize_alphabet_symbol(byte)
        current_fp = compose_fingerprints(current_fp, symbol, 1)
        cumulative_fingerprints.append(current_fp)

    # Step 2: Initialize storage vectors
    prenode_count = index.grammar_size()
    mut_sibling_fp = [0] * (prenode_count + 1)
    mut_base_power = [0] * (prenode_count + 1)
    mut_runlen_inverse = [0] * (len(index.runlen_prenode_arity) + 1)

    # Step 3: Populate prenode fingerprint data
    for prenode in range(1, prenode_count + 1):
        node = index.topology_tree.prenode_to_node(prenode)
        node_offset = node_offset(node)

        if index.node_is_root(node):
            mut_sibling_fp[prenode] = 0
            mut_base_power[prenode] = 0
            continue

        parent_node = index.topology_tree.parent(node)
        parent_offset = node_offset(parent_node)
        parent_end = node_end_offset(parent_node)
        parent_prenode = index.topology_tree.node_to_prenode(parent_node)
        parent_runlen = index.runlen_prenode_arity[index.is_runlen_prenode.rank1(parent_prenode)]

        # Handle run-length nodes
        if parent_runlen != 0:
            child_size = (parent_end - parent_offset) // parent_runlen
            node_offset += child_size  # Adjust for run-length expansion
            mut_base_power[prenode] = base_power(child_size)
            runlen_rank = index.is_runlen_prenode.rank1(parent_prenode)
            mut_runlen_inverse[runlen_rank] = predecessor_inverse(mut_base_power[prenode])
        else:
            mut_base_power[prenode] = base_power(node_offset - parent_offset)

        # Calculate sibling prefix fingerprint
        composed_fp = cumulative_fingerprints[node_offset] if node_offset > 0 else 0
        parent_fp = cumulative_fingerprints[parent_offset] if parent_offset > 0 else 0

        if parent_offset > 0:
            fp_length = node_offset - parent_offset
            mut_sibling_fp[prenode] = subtract_prefix_fingerprint(
                composed_fp, parent_fp, fp_length
            )
        else:
            mut_sibling_fp[prenode] = composed_fp

    # Step 4: Store final vectors
    index.prenode_to_sibling_prefix_fingerprint = mut_sibling_fp
    index.prenode_to_sibling_prefix_base_power = mut_base_power
    index.runlen_prenode_base_power_predecessor_inverse = mut_runlen_inverse

```

Code Block A.7: Substring Fingerprint Retrieval for LMS.

```
def fingerprint(idx: int, length: int, index: GrammarIndex) -> int:
    result = 0

    if idx + length >= index.text_length():
        return result

    if idx != 0:
        prefix_fp = fingerprint(0, idx, index)
        composed_fp = fingerprint(0, idx + length, index)
        return index.subtract_prefix_fingerprint(composed_fp, prefix_fp, length)

    # Base case: idx == 0
    mut_idx = length - 1
    node = index.topology_root()
    prenode = index.node_to_prenode(node)

    while True:
        if index.node_is_leaf(node):
            rule = index.prenode_to_rule(prenode)

            if index.rule_is_terminal(rule):
                symbol = index.terminal_rule_to_symbol(rule)
                norm_symbol = index.normalize_alphabet_symbol(symbol)
                result = index.compose_fingerprints(result, norm_symbol, 1)
                break
            else:
                # Follow first occurrence of non-terminal rule
                prenode = index.rule_first_occurrence(rule)
                old_offset = index.node_offset_in_text(node)
                node = index.prenode_to_node(prenode)
                new_offset = index.node_offset_in_text(node)
                mut_idx -= old_offset - new_offset
                continue
        else:
            run_len = index.runlen_prenode_arity[index.is_runlen_prenode.rank1(prenode)]

            if run_len != 0:
                # Handle run-length node
                offset = index.node_offset_in_text(node)
                child_size = (index.node_end_offset(node) - offset) // run_len
                parent_prenode = prenode

                # Move to first child
                node = index.topology_child(node, 1)
                prenode = index.node_to_prenode(node)

                child_num = ((mut_idx - offset) // child_size) + 1
                mut_idx = offset + ((mut_idx - offset) % child_size)

                if child_num >= 2:
                    base_fp = index.prenode_to_sibling_prefix_fingerprint[prenode]
                    base_pow = index.prenode_to_sibling_prefix_base_power[prenode]
                    run_pow_inv = index.runlen_prenode_base_power_predecessor_inverse[
                        index.prenode_to_runlen_rank(parent_prenode)
                    ]

                    run_fp = index.compose_run_fingerprints_cached(
                        base_fp, base_pow, run_pow_inv, child_num-1
                    )
                    result = index.compose_fingerprints(
                        result, run_fp, child_size * (child_num-1)
                continue
            else:
                # Binary search for correct child
                children = index.topology_children(node)
                first, last = 1, children

                while first < last:
                    mid = (first + last + 1) // 2
                    child_node = index.topology_child(node, mid)

                    if mut_idx >= index.node_offset_in_text(child_node):
                        first = mid
                    else:
                        last = mid - 1

                parent_offset = index.node_offset_in_text(node)
                node = index.topology_child(node, first)
                prenode = index.node_to_prenode(node)
                child_offset = index.node_offset_in_text(node)

                if child_offset > parent_offset:
                    result = index.compose_fingerprints_cached(
                        result,
                        index.prenode_to_sibling_prefix_fingerprint[prenode],
                        index.prenode_to_sibling_prefix_base_power[prenode]
                    )

    return result

```

Code Block A.8: Contextual Occurrence Filtering Pseudocode for LMS.

```
def filter_contextual_occurrences_lms(
        occurrences: List[int],
        pattern_length: int,
        context_length: int,
        index: GrammarIndex
    ) -> List[int]:
    context_buckets: Dict[int, List[int]] = []
    for offset in occurrences:
        start: int = max(0, offset-context_length)
        end: int = min(index.textLength()-1, offset+pattern_length+context_length-1)
        context: int = index.fingerprint(start, end)
        context_buckets[context].append(offset)
    results: List[int] = []
    for _, offsets in context_buckets.items():
        results.append(min(offsets))
    return results

```

A.4. SLP Base

Code Block A.9: Primary Occurrence Retrieval Pseudocode for SLP.

```python
def locate(pattern: str, index: RePairIndex) -> List[int]:
    results = []

    for cut in range(len(pattern) - 1):
        # Get primary occurrences from grammar relations
        prefix: str = pattern[0:cut]
        rev_prefix: str = prefix[::-1]
        suffix: str = pattern[cut+1:len(pattern)-1]
        row0, row1 = index.children_to_parent_grammar_relation.binSearchColRange(suffix)
        col0, sol1 = index.children_to_parent_grammar_relation.binSearchRowRange(
  ↪ rev_prefix)

        # Phase 1: Primary occurrences
        grid_points: List[int] = index.children_to_parent_grammar_relation.labelsInRange(
  ↪ col0, col1, row0, row1)
        occs: List[Tuple[int, int]] = []

        for lhs_rule in grid_points:
            rhsl_rule, _ = index.children_to_parent_grammar_relation.ithPointForLabel(
  ↪ lhs_rule, 1)
            rhsl_length: int = index.rule_to_expansion_length[rhsl_rule]
            offset: int = rhsl_length + 1 - cut
            occs += (lhs_rule, offset)

        # Phase 2: Secondary occurrences
        while occs:
            lhs_rule, offset = occs.pop(0)
            rev_lhs_rule = index.reverse_sorted_rule_to_rule.rev(lhs_rule)

            # Find sequence-phrase positions
            phrase_amount = index.intersecting_rules_to_phrase_rank_sequence_relation.
  ↪ nRowsForCol(rev_lhs_rule)
            for phrase_idx in range(phrase_count):
                row = index.intersecting_rules_to_phrase_rank_sequence_relation.
  ↪ ithRowForCol(rev_lhs_rule, phrase_idx+1)
                phrase_rank = index.intersecting_rules_to_phrase_rank_sequence_relation.
  ↪ labelAtPoint(row, rev_lhs_rule)
                abs_offset = index.is_sequence_phrase_start_offset_in_text.select1(
  ↪ phrase_rank) + offset
                results.append(abs_offset)

            # Find parent rules
            parents_from_right : int = index.children_to_parent_grammar_relation.
  ↪ nRowsForCol(rev_lhs_rule)
            for parent_idx in range(parents_from_right):
                parent_rhsl: int = index.children_to_parent_grammar_relation.ithRowForCol(
  ↪ rev_lhs_rule, parent_idx+1)
                parent_lhs: int = index.children_to_parent_grammar_relation.labelAtPoint(
  ↪ parent_rhsl, rev_lhs_rule)
                parent_rhsl_length: int = index.rule_to_expansion_length[parent_rhsl]
                parent_offset: int = offset + parent_rhsl_length
                occs += (parent_lhs, parent_offset)

            parents_from_left: int = index.children_to_parent_grammar_relation.
    ↪ nColsForRow(lhs_rule)
            for parent_idx in range(parents_from_left):
                parent_rhsr_rev: int = index.children_to_parent_grammar_relation.
    ↪ ithColForRow(lhs_rule, parent_idx+1)
                parent_lhs: int = index.children_to_parent_grammar_relation.labelAtPoint(
    ↪ lhs_rule, parent_rhsr_rev)
                parent_offset: int = offset
                occs += (parent_lhs, parent_offset)

    # Phase 3: Extra primary occurrences
    for cut in range(len(pattern) - 1):
        # Get primary occurrences from grammar relations
        prefix: str = pattern[0:cut]
        rev_prefix: str = prefix[::-1]
        suffix: str = pattern[cut+1:len(pattern)-1]
        row0, row1 = index.intersecting_rules_to_phrase_rank_sequence_relation.
    ↪ binSearchColRange(suffix)
        col0, sol1 = index.intersecting_rules_to_phrase_rank_sequence_relation.
    ↪ binSearchRowRange(rev_prefix)

        grid_points: List[int] = index.intersecting_rules_to_phrase_rank_sequence_relation.
    ↪ labelsInRange(col0, col1, row0, row1)
        for phrase_rank in grid_points:
            abs_offset = index.is_sequence_phrase_start_offset_in_text.select1(phrase_rank)
    ↪ + 1 - cut
            results.append(abs_offset)

    return results

```

A.5. SLP Modified

Code Block A.10: Contextual Occurrence Retrieval Pseudocode for SLP.

```
def locate_contextual(pattern: str, context_len: int, index: RePairIndex) -> List[int]:
    results = []

    for cut in range(len(pattern) - 1):
        # Get primary occurrences from grammar relations
        prefix: str = pattern[0:cut]
        rev_prefix: str = prefix[::-1]
        suffix: str = pattern[cut+1:len(pattern)-1]
        row0, row1 = index.children_to_parent_grammar_relation.binSearchColRange(suffix)
        col0, sol1 = index.children_to_parent_grammar_relation.binSearchRowRange(
    ↪ rev_prefix)

        # Phase 1: Primary occurrences
        grid_points: List[int] = index.children_to_parent_grammar_relation.labelsInRange(
    ↪ col0, col1, row0, row1)
```

```
        occs: List[Tuple[int, int]] = []

        for lhs_rule in grid_points:
            rhsl_rule, _ = index.children_to_parent_grammar_relation.ithPointForLabel(lhs_rule, 1)
            rhsl_length: int = index.rule_to_expansion_length[rhsl_rule]
            offset: int = rhsl_length + 1 - cut
            occs += (lhs_rule, offset)

        # Phase 2: Secondary occurrences
        while occs:
            lhs_rule, offset = occs.pop(0)
            lhs_length: int = index.rule_to_expansion_length[lhs_rule]
            rev_lhs_rule = index.reverse_sorted_rule_to_rule.rev(lhs_rule)

            valid_left = offset >= context_len + 2
            valid_right = (offset + len(pattern) + context_len) <= lhs_length - 2
            context_fits = valid_left and valid_right

            # Find sequence-phrase positions
            phrase_amount = index.intersecting_rules_to_phrase_rank_sequence_relation.nRowsForCol(rev_lhs_rule)
            if not context_fits or phrase_amount == 0:
                # Find parent rules
                parents_from_right : int = index.children_to_parent_grammar_relation.nRowsForCol(rev_lhs_rule)
                for parent_idx in range(parents_from_right):
                    parent_rhsl: int = index.children_to_parent_grammar_relation.ithRowForCol(rev_lhs_rule, parent_idx+1)
                    parent_lhs: int = index.children_to_parent_grammar_relation.labelAtPoint(parent_rhsl, rev_lhs_rule)
                    parent_rhsl_length: int = index.rule_to_expansion_length[parent_rhsl]
                    parent_offset: int = offset + parent_rhsl_length
                    occs += (parent_lhs, parent_offset)

                parents_from_left: int = index.children_to_parent_grammar_relation.nColsForRow(lhs_rule)
                for parent_idx in range(parents_from_left):
                    parent_rhsr_rev: int = index.children_to_parent_grammar_relation.ithColForRow(lhs_rule, parent_idx+1)
                    parent_lhs: int = index.children_to_parent_grammar_relation.labelAtPoint(lhs_rule, parent_rhsr_rev)
                    parent_offset: int = offset
                    occs += (parent_lhs, parent_offset)

            if phrase_amount > 0:
                for phrase_idx in range(phrase_count):
                    row = index.intersecting_rules_to_phrase_rank_sequence_relation.ithRowForCol(rev_lhs_rule, phrase_idx+1)
                    phrase_rank = index.intersecting_rules_to_phrase_rank_sequence_relation.labelAtPoint(row, rev_lhs_rule)
                    abs_offset = index.is_sequence_phrase_start_offset_in_text.select1(phrase_rank) + offset
                    results.append(abs_offset)

    # Phase 3: Extra primary occurrences
    for cut in range(len(pattern) - 1):
        # Get primary occurrences from grammar relations
        prefix: str = pattern[0:cut]
        rev_prefix: str = prefix[::-1]
        suffix: str = pattern[cut+1:len(pattern)-1]
        row0, row1 = index.intersecting_rules_to_phrase_rank_sequence_relation.binSearchColRange(suffix)
        col0, sol1 = index.intersecting_rules_to_phrase_rank_sequence_relation.binSearchRowRange(rev_prefix)

        grid_points: List[int] = index.intersecting_rules_to_phrase_rank_sequence_relation.labelsInRange(col0, col1, row0, row1)
        for phrase_rank in grid_points:
            abs_offset = index.is_sequence_phrase_start_offset_in_text.select1(phrase_rank) + 1 - cut
            results.append(abs_offset)

    # New Phase 4: Context filtering
    unique_contexts = set()

    # Extract context substrings
    for offset in results:
        start: int = max(0, offset - context_len)
        end: int = min(index.text_length, offset + pattern_len + context_len)
        context: str = index.extract(start, end)
        unique_contexts.add(context)

    # Validate contexts
    results = []
    for context in unique_contexts:
        # Find base offsets within context
        ctx_offsets = index.locate(context)
        first_offset = ctx_results[0]

        # Offset adjustment logic
        if len(context) < pattern_len + 2*context_len and first_offset <= context_len:
            crop_amount = (pattern_len + 2*context_len) - len(context)
            results += (first_offset + (context_len - crop_amount))
        else:
            results += (first_offset + context_len)

    # Final sorted results
    results = sort(results)

    return results
```

Annex B. Figures

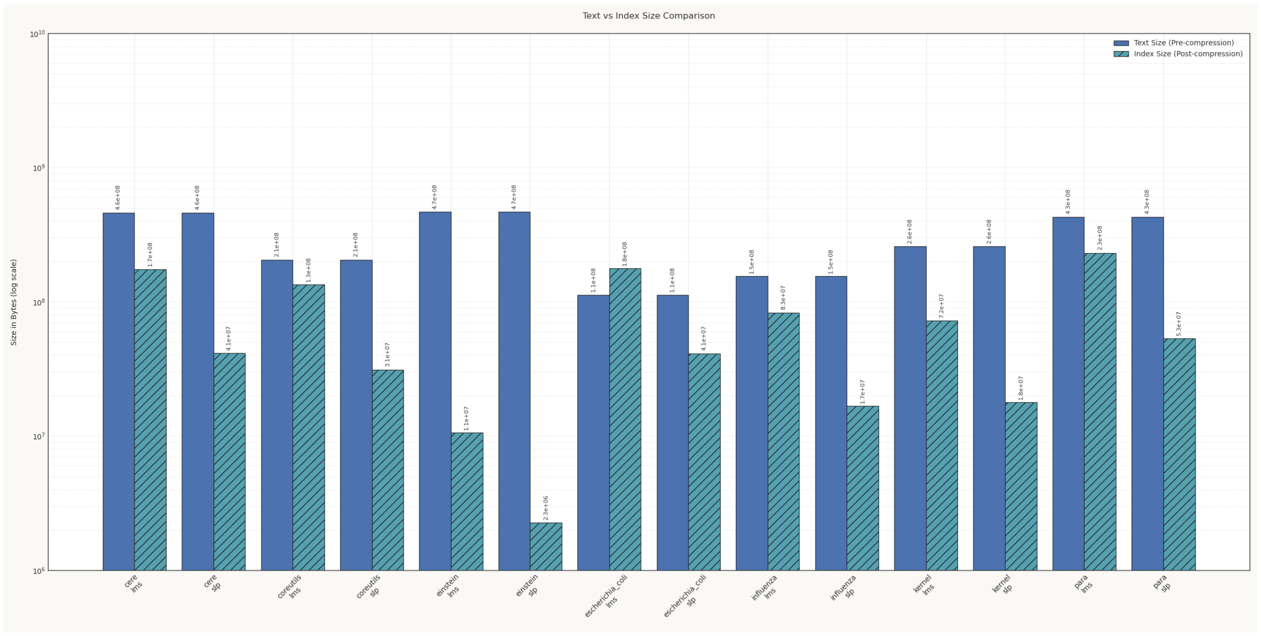

Figure B.1: Size comparison in bytes of each text to it is compressed self index, for both indexing programs LMS and SLP. Logarithmically scaled.

B.1. Pizzachili Collection Einstein

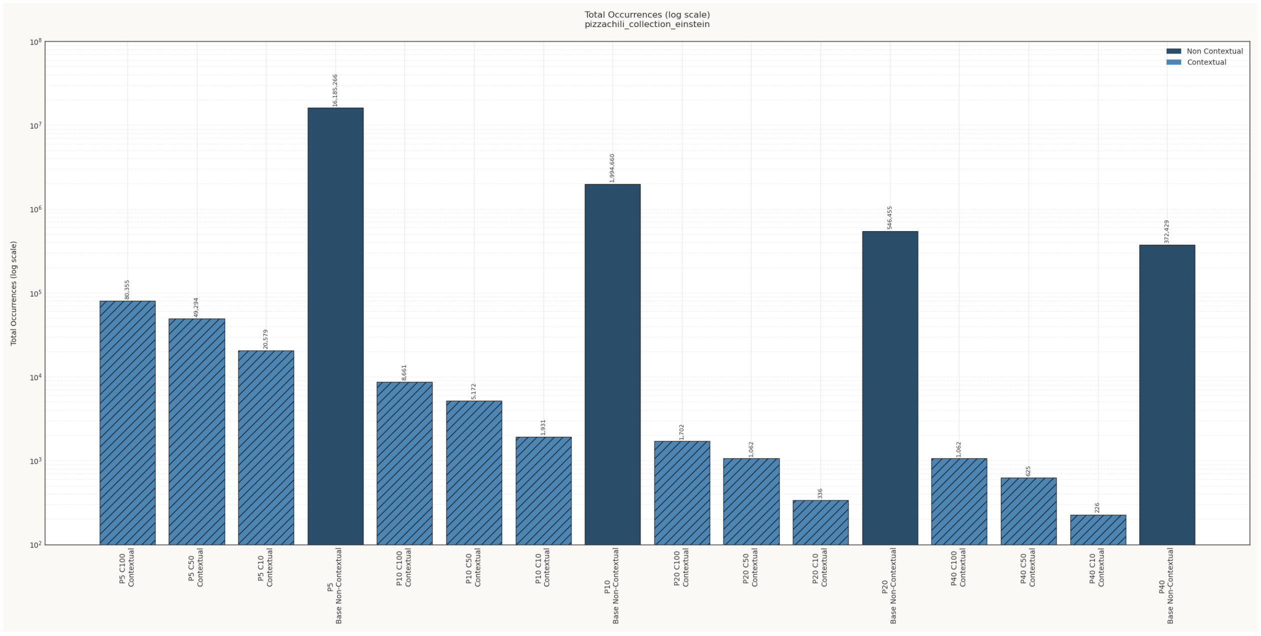

Figure B.2: Total reported occurrences on collection Einstein, for each pattern and context length including the non-contextual case. Aggregates pruned and non-pruned approaches. Logarithmically scaled.

B.1.1. LMS

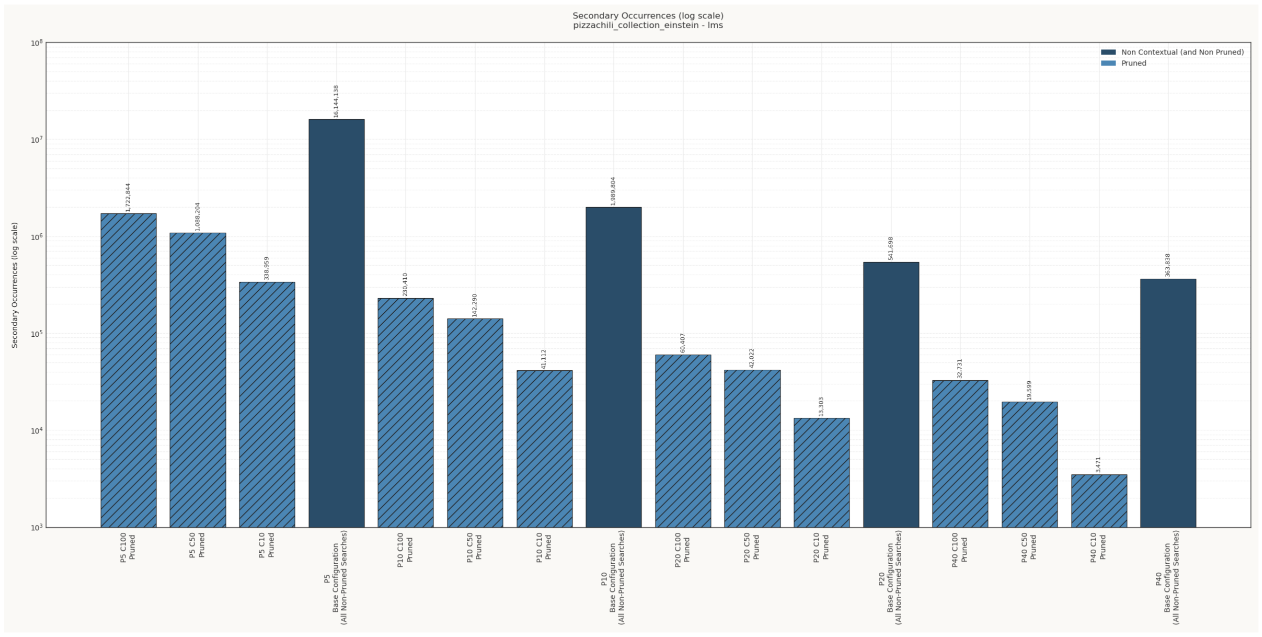

Figure B.3: Secondary occurrences found before filtering on LMS, for each pattern and context length including the non-contextual case. Aggregates non-pruned results to the non-contextual case. Logarithmically scaled.

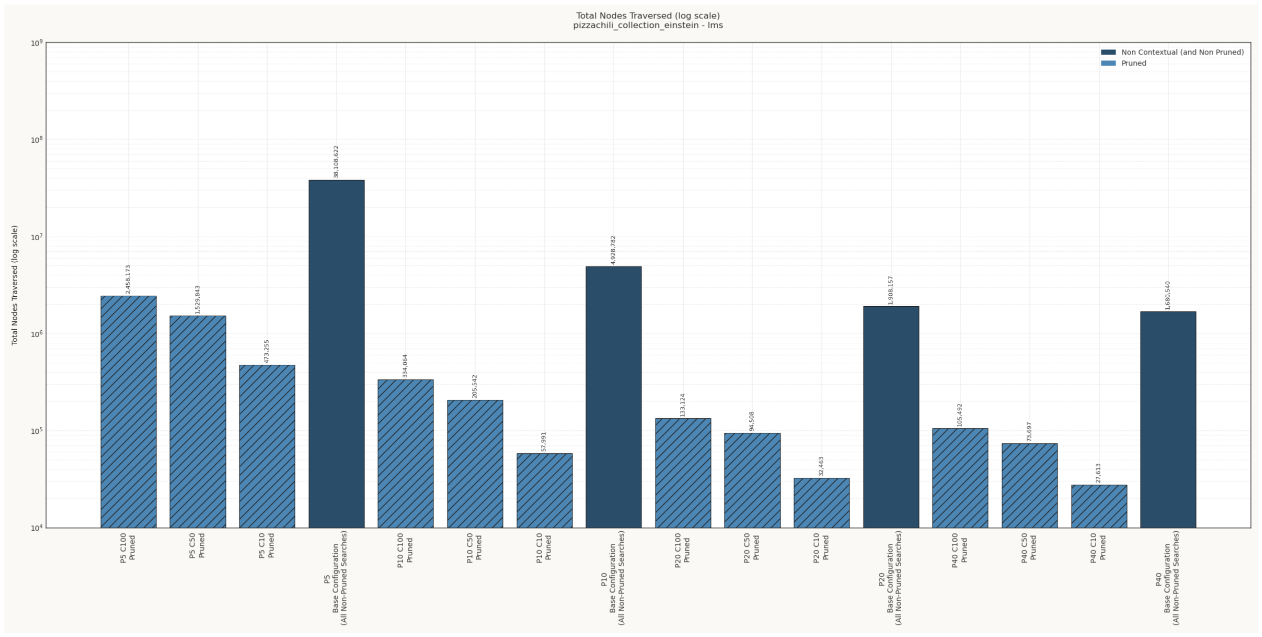

Figure B.4: Nodes traversed while computing secondary occurrences on LMS, for each pattern and context length including the non-contextual case. Aggregates non-pruned results to the non-contextual case. Logarithmically scaled.

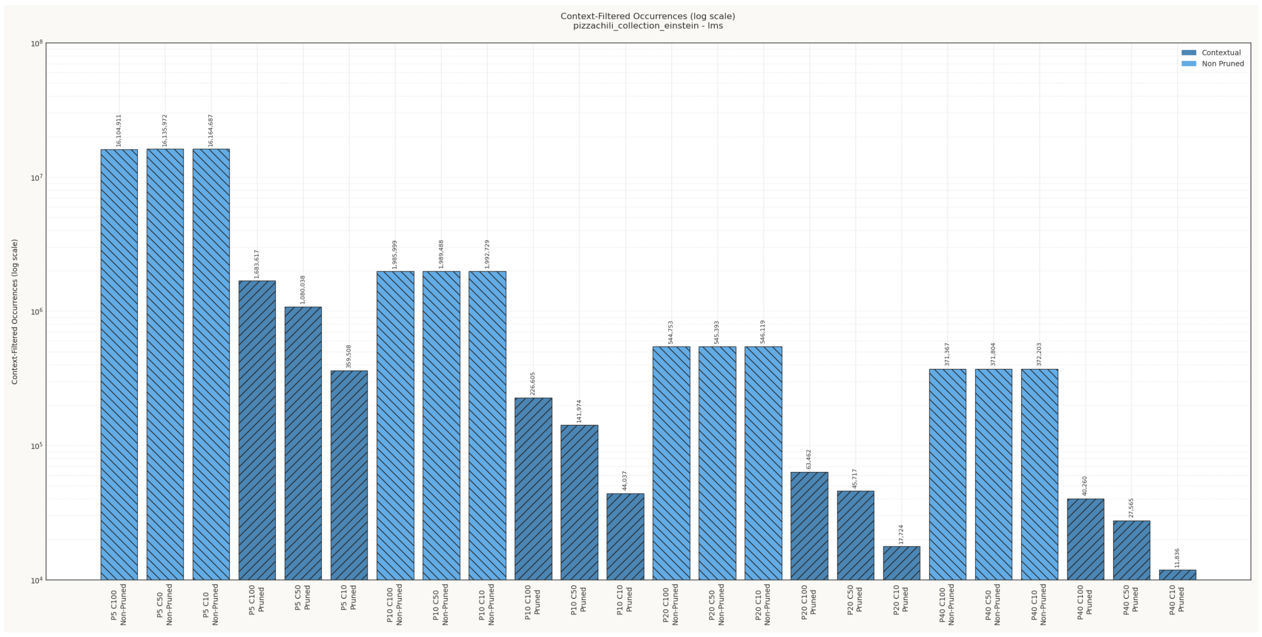

Figure B.5: Contextually filtered occurrences on LMS, for each pattern and context length and for both the pruned and non-pruned approaches. The non-contextual case is not shown since its value is 0. Logarithmically scaled.

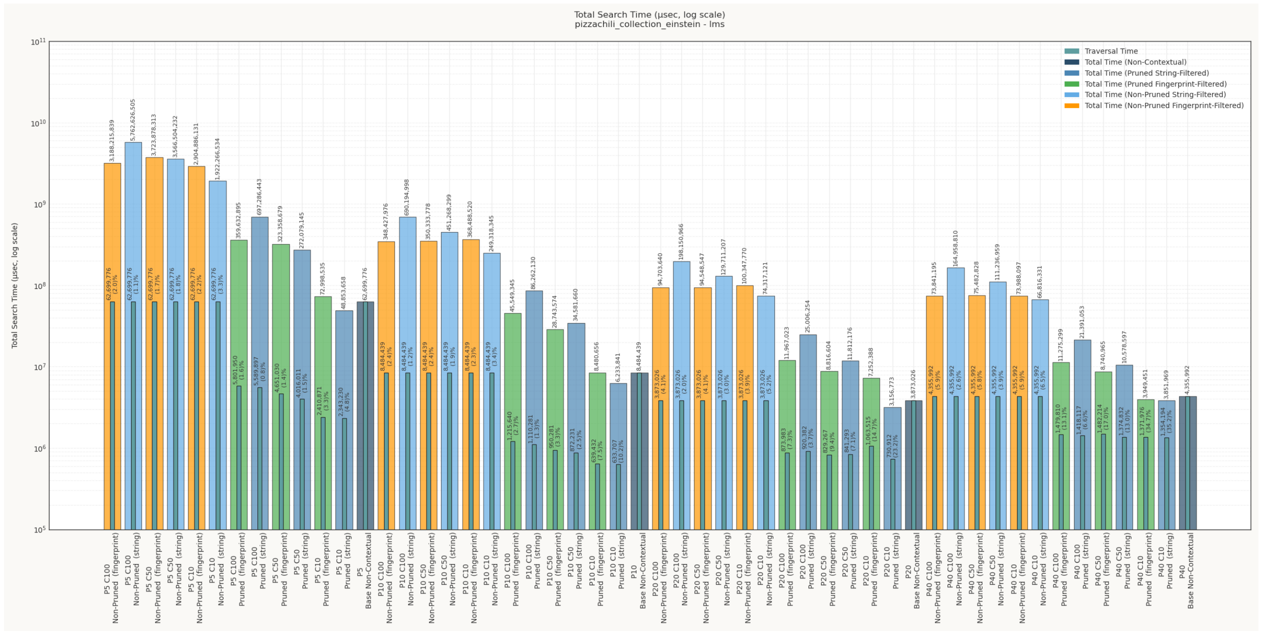

Figure B.6: Total search time, for each pattern and context length including the non-contextual case, for both the pruned and non-pruned approaches, on LMS. Additionally shows the traversal time on top of total search time for comparison, with a percentage annotation for the relationship between both values. Logarithmically scaled.

B.1.2. SLP

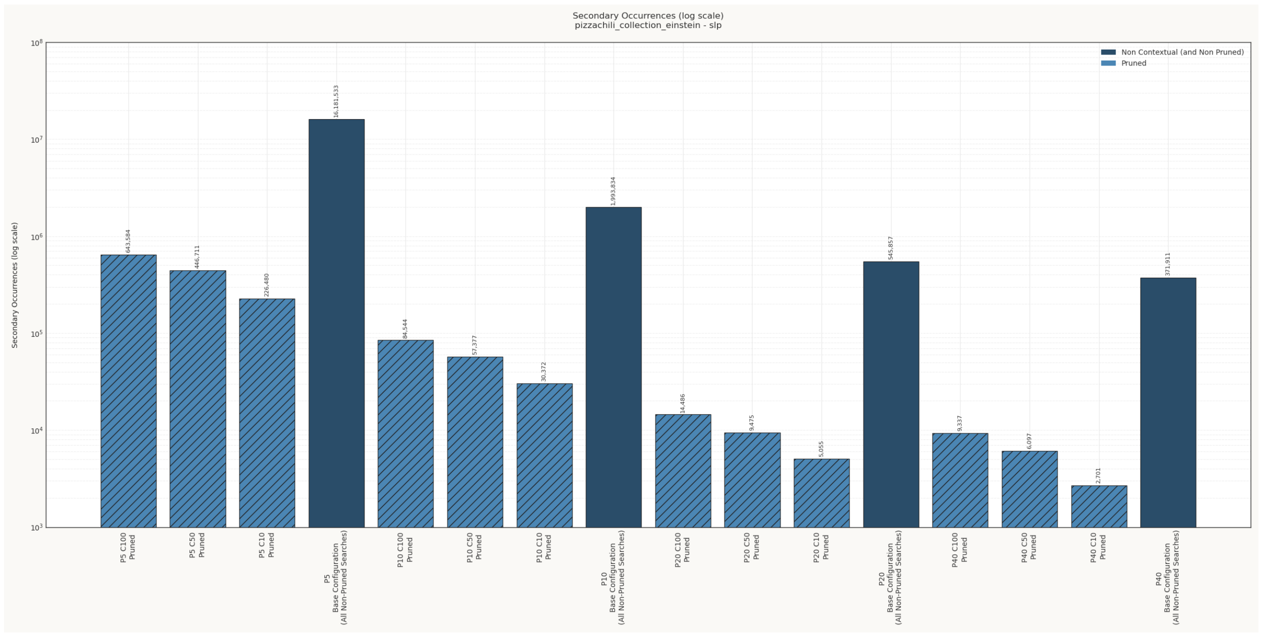

Figure B.7: Secondary occurrences found before filtering on SLP, for each pattern and context length including the non-contextual case. Aggregates non-pruned results to the non-contextual case. Logarithmically scaled.

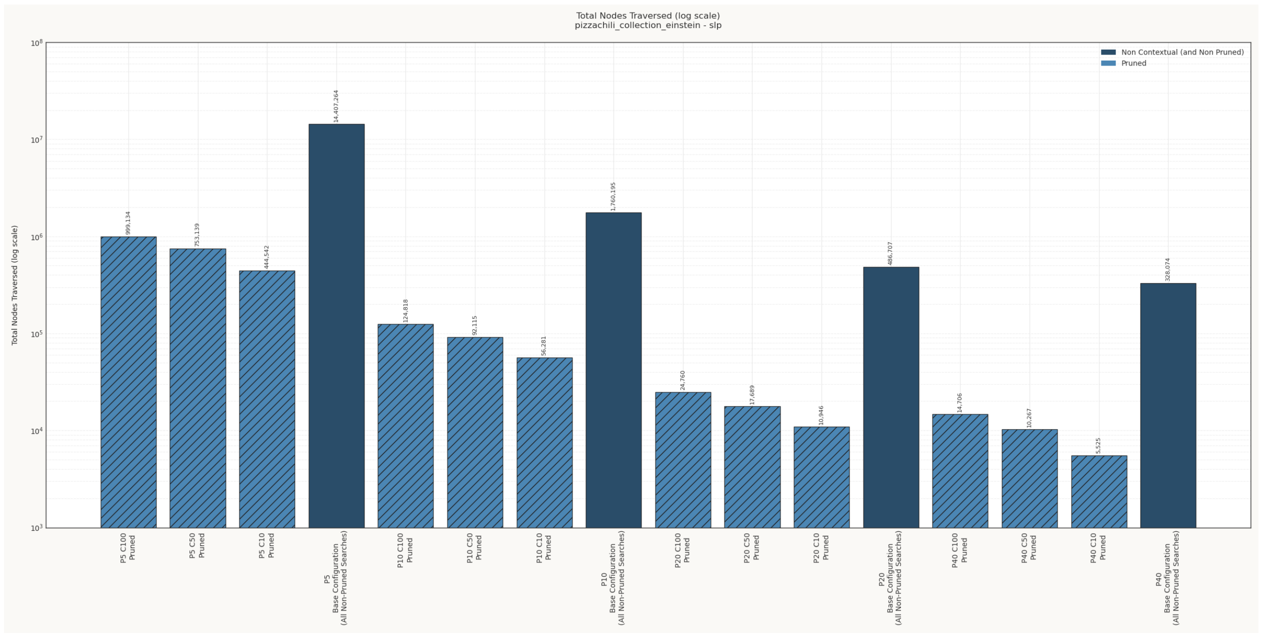

Figure B.8: Nodes traversed while computing secondary occurrences on SLP, for each pattern and context length including the non-contextual case. Aggregates non-pruned results to the non-contextual case. Logarithmically scaled.

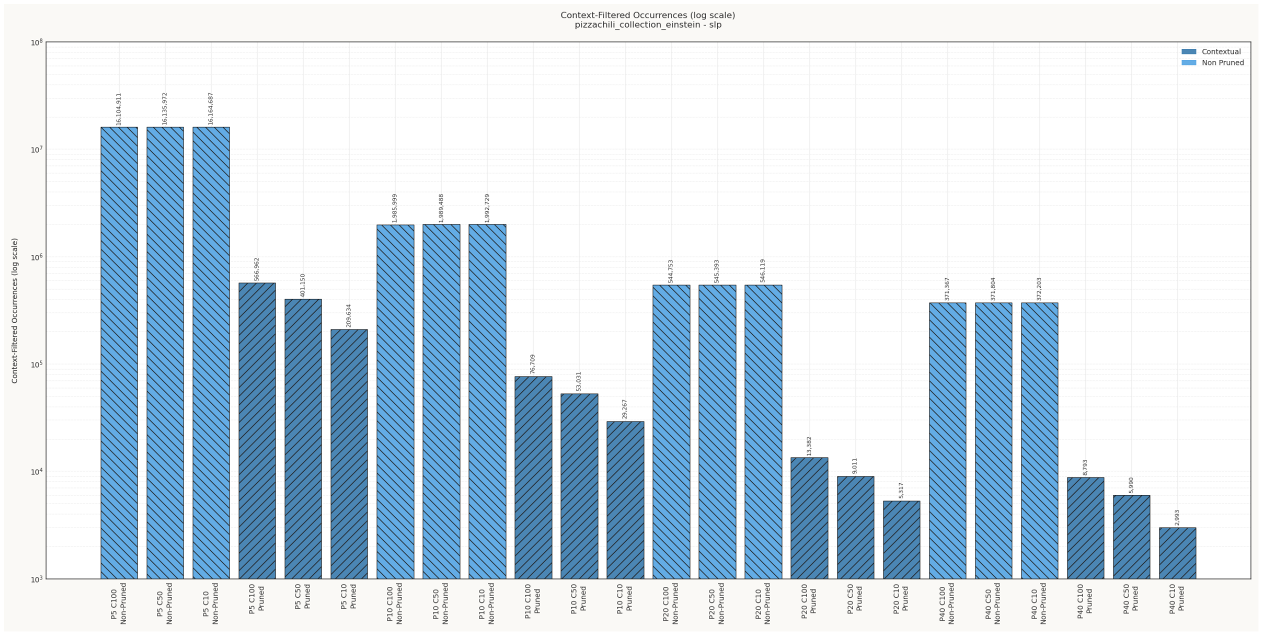

Figure B.9: Contextually filtered occurrences on SLP, for each pattern and context length and for both the pruned and non-pruned approaches. The non-contextual case is not shown since its value is 0. Logarithmically scaled.

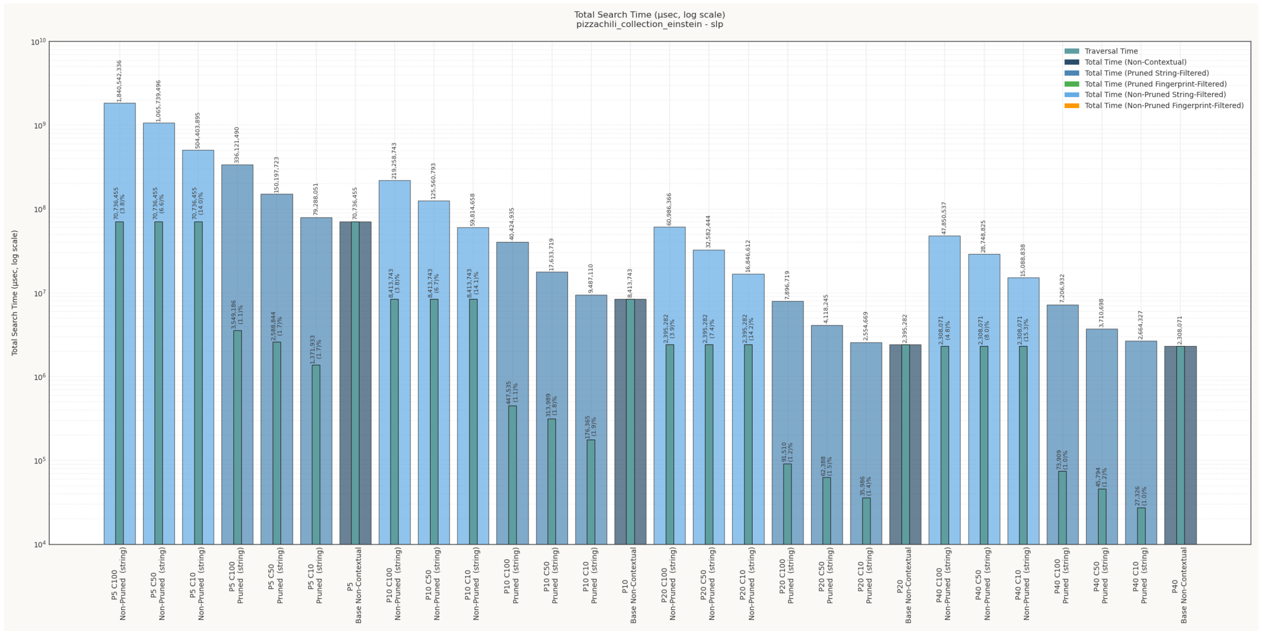

Figure B.10: Total search time, for each pattern and context length including the non-contextual case, for both the pruned and non-pruned approaches, on SLP. Additionally shows the traversal time on top of total search time for comparison, with a percentage annotation for the relationship between both values. Logarithmically scaled.

B.2. Pizzachili Collection Coreutils

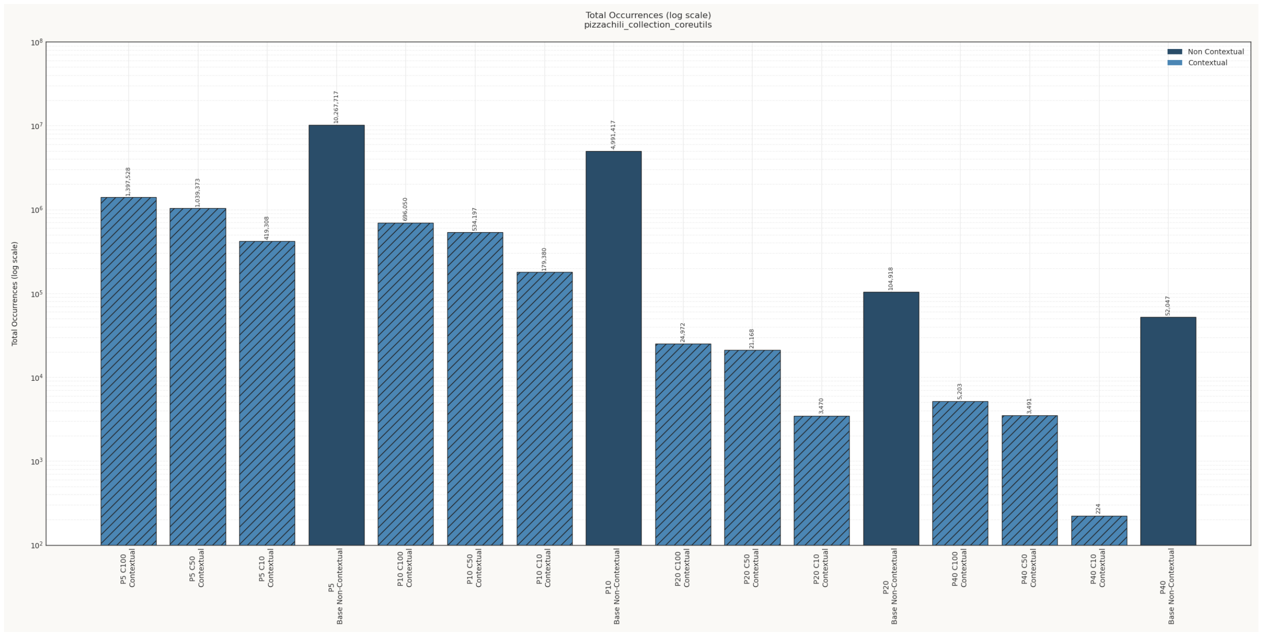

Figure B.11: Total reported occurrences on collection Coreutils, for each pattern and context length including the non-contextual case. Aggregates pruned and non-pruned approaches. Logarithmically scaled.

B.2.1. LMS

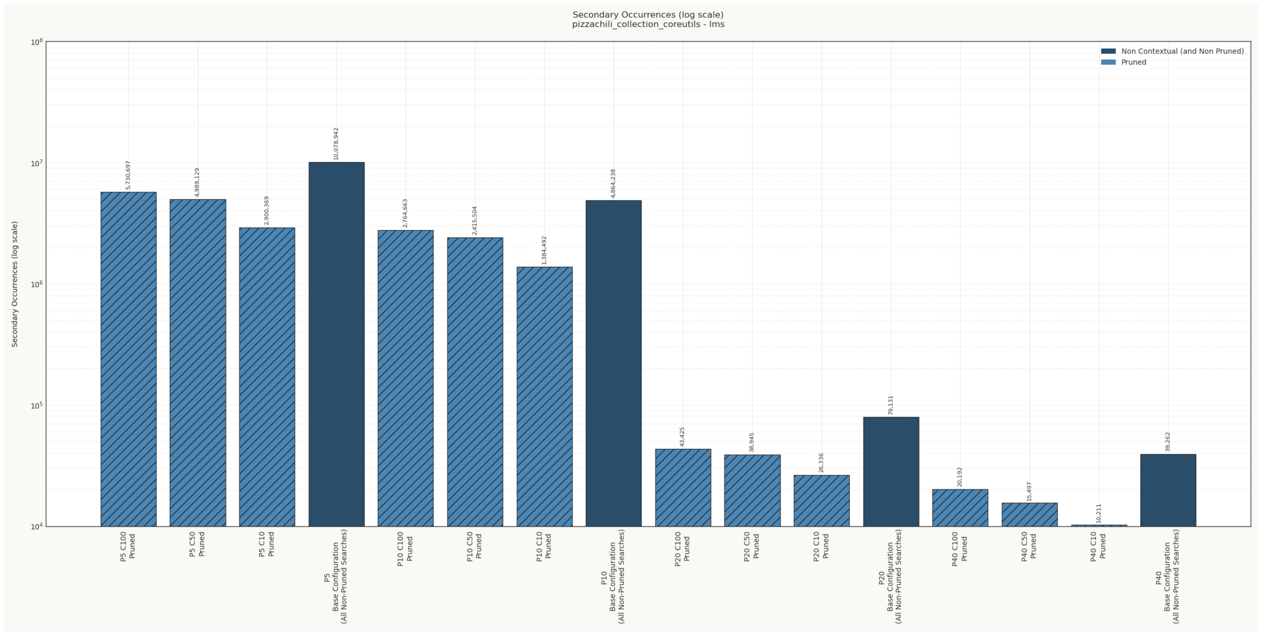

Figure B.12: Secondary occurrences found before filtering on LMS, for each pattern and context length including the non-contextual case. Aggregates non-pruned results to the non-contextual case. Logarithmically scaled.

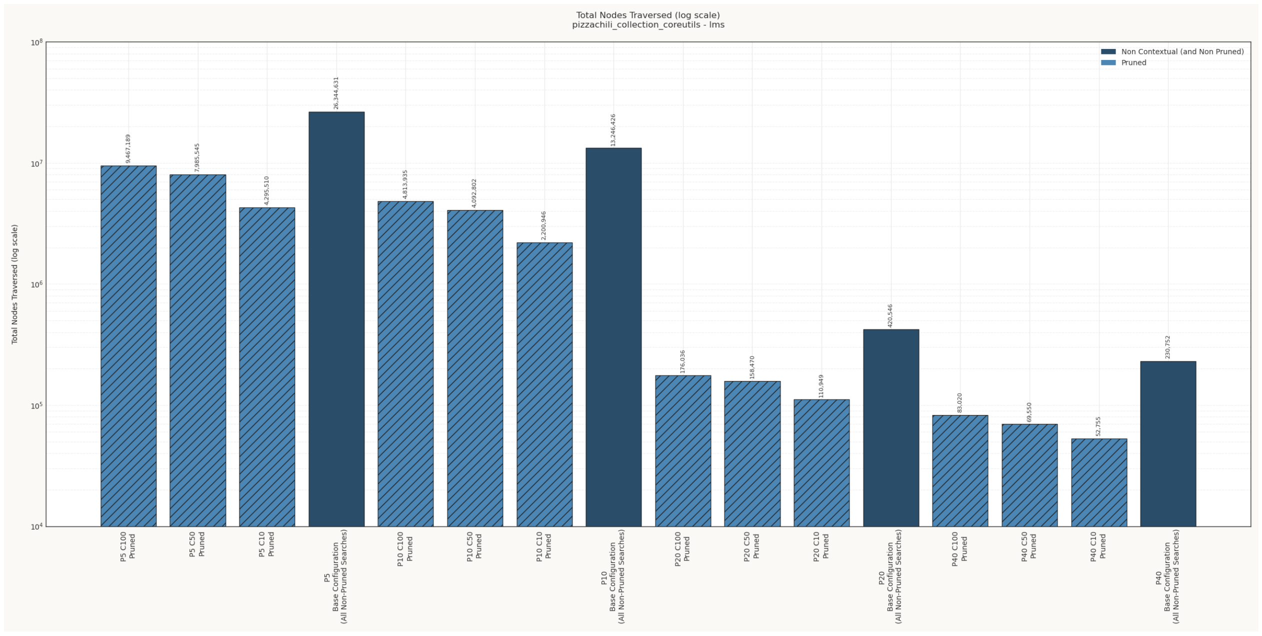

Figure B.13: Nodes traversed while computing secondary occurrences on LMS, for each pattern and context length including the non-contextual case. Aggregates non-pruned results to the non-contextual case. Logarithmically scaled.

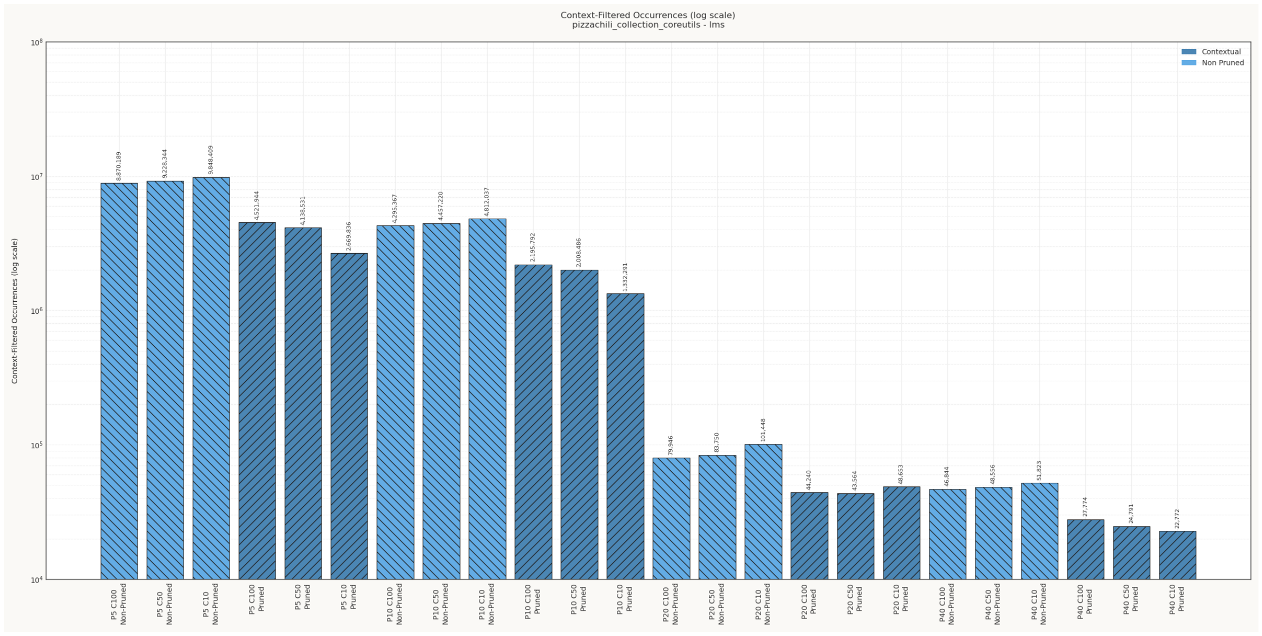

Figure B.14: Contextually filtered occurrences on LMS, for each pattern and context length and for both the pruned and non-pruned approaches. The non-contextual case is not shown since its value is 0. Logarithmically scaled.

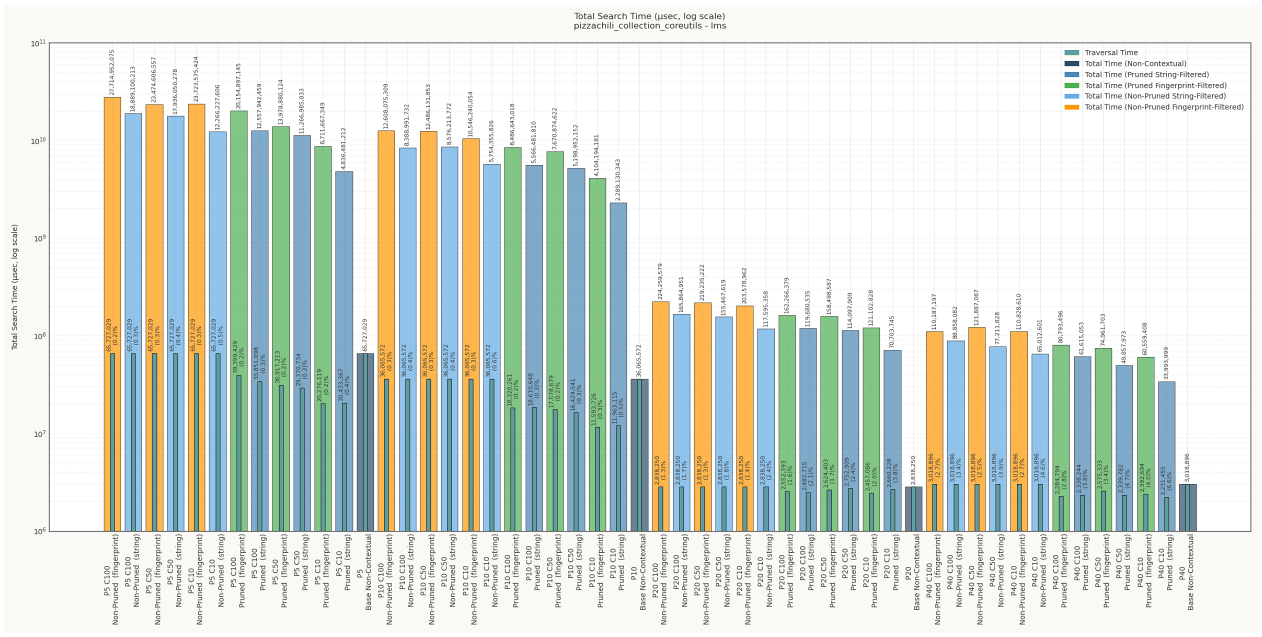

Figure B.15: Total search time, for each pattern and context length including the non-contextual case, for both the pruned and non-pruned approaches, on LMS. Additionally shows the traversal time on top of total search time for comparison, with a percentage annotation for the relationship between both values. Logarithmically scaled.

B.2.2. SLP

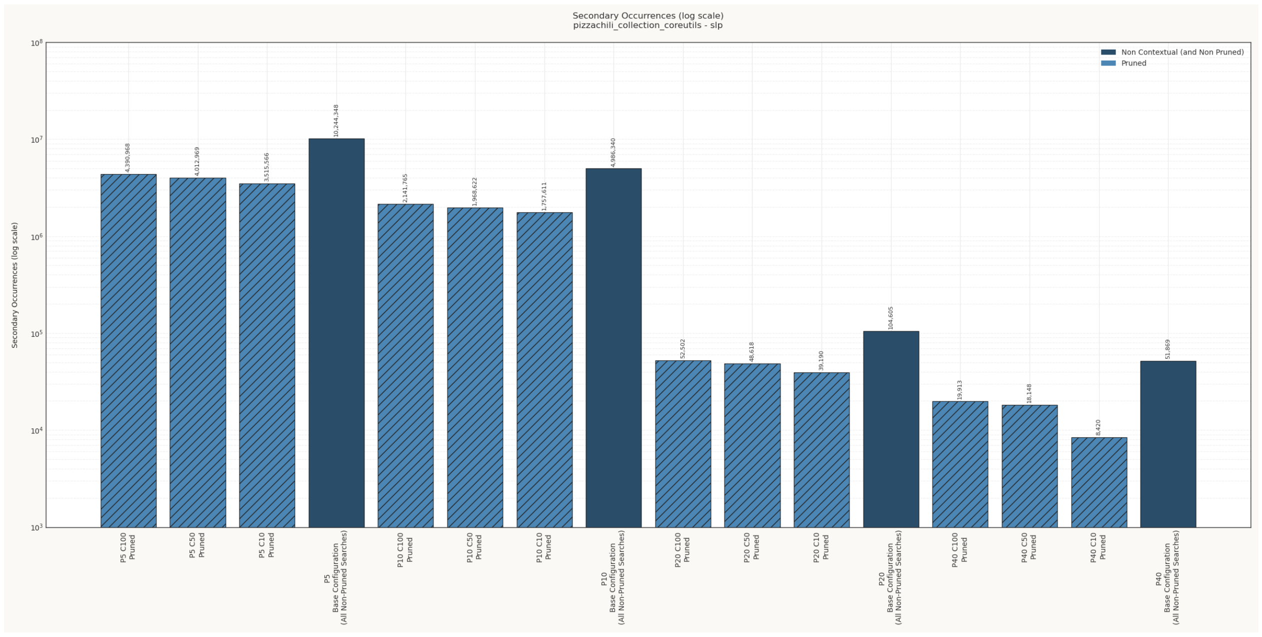

Figure B.16: Secondary occurrences found before filtering on SLP, for each pattern and context length including the non-contextual case. Aggregates non-pruned results to the non-contextual case. Logarithmically scaled.

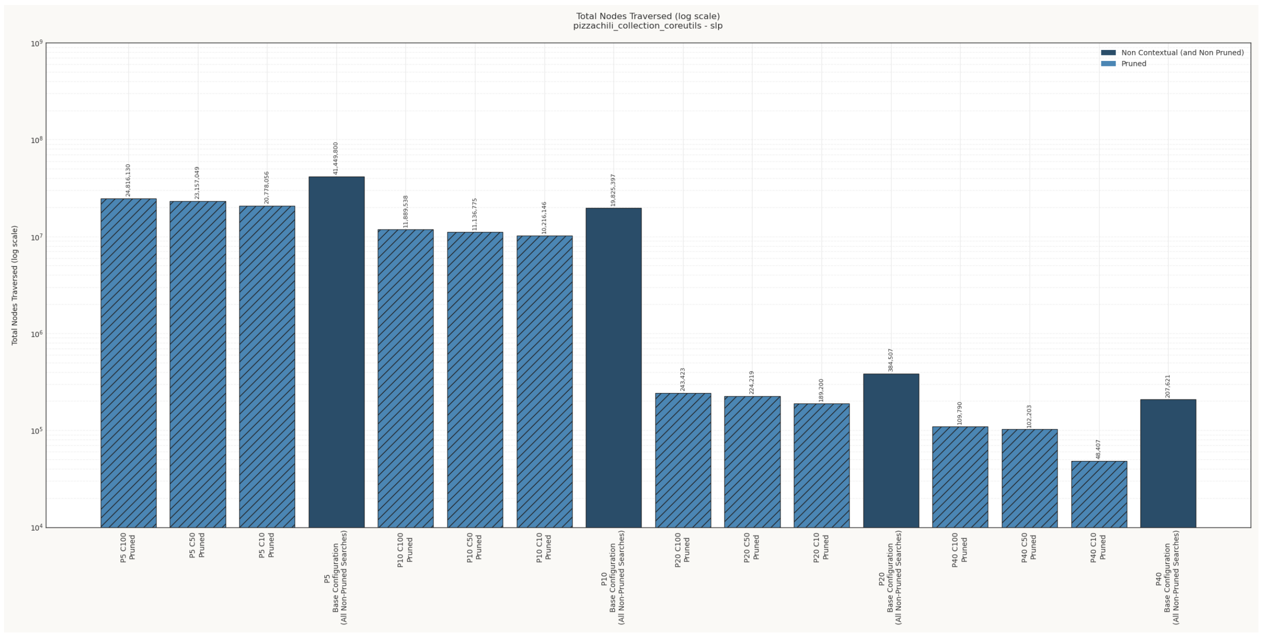

Figure B.17: Nodes traversed while computing secondary occurrences on SLP, for each pattern and context length including the non-contextual case. Aggregates non-pruned results to the non-contextual case. Logarithmically scaled.

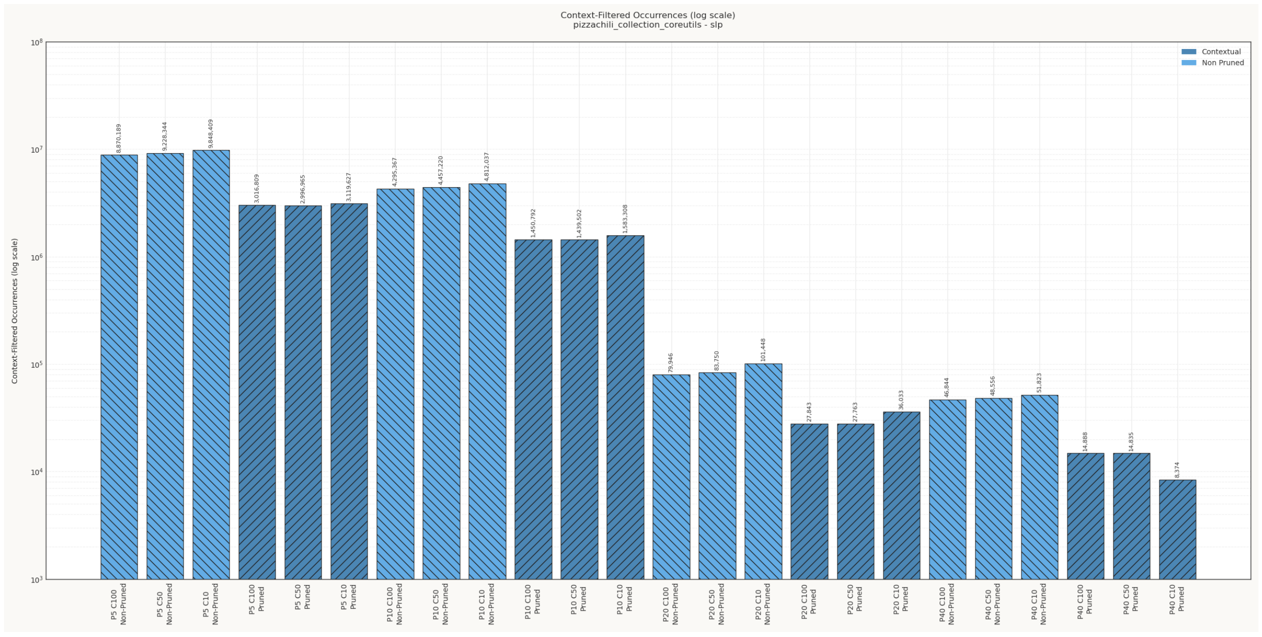

Figure B.18: Contextually filtered occurrences on SLP, for each pattern and context length and for both the pruned and non-pruned approaches. The non-contextual case is not shown since its value is 0. Logarithmically scaled.

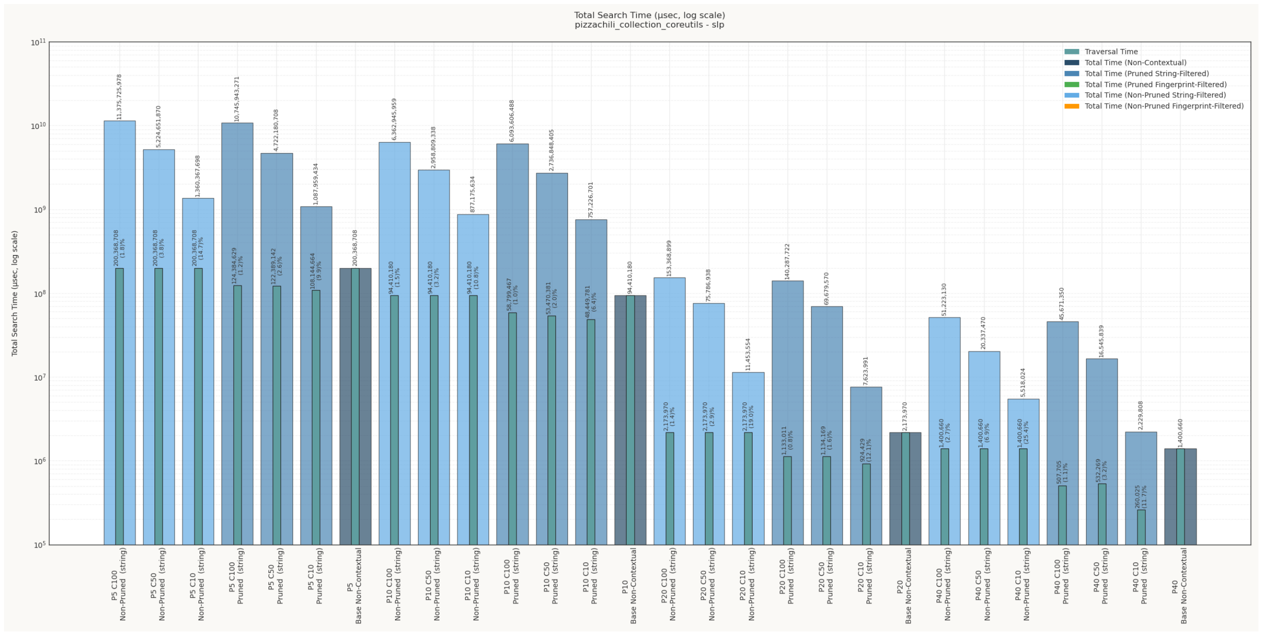
Figure B.19: Total search time, for each pattern and context length including the non-contextual case, for both the pruned and non-pruned approaches, on SLP. Additionally shows the traversal time on top of total search time for comparison, with a percentage annotation for the relationship between both values. Logarithmically scaled.

B.3. Pizzachili Collection Kernel

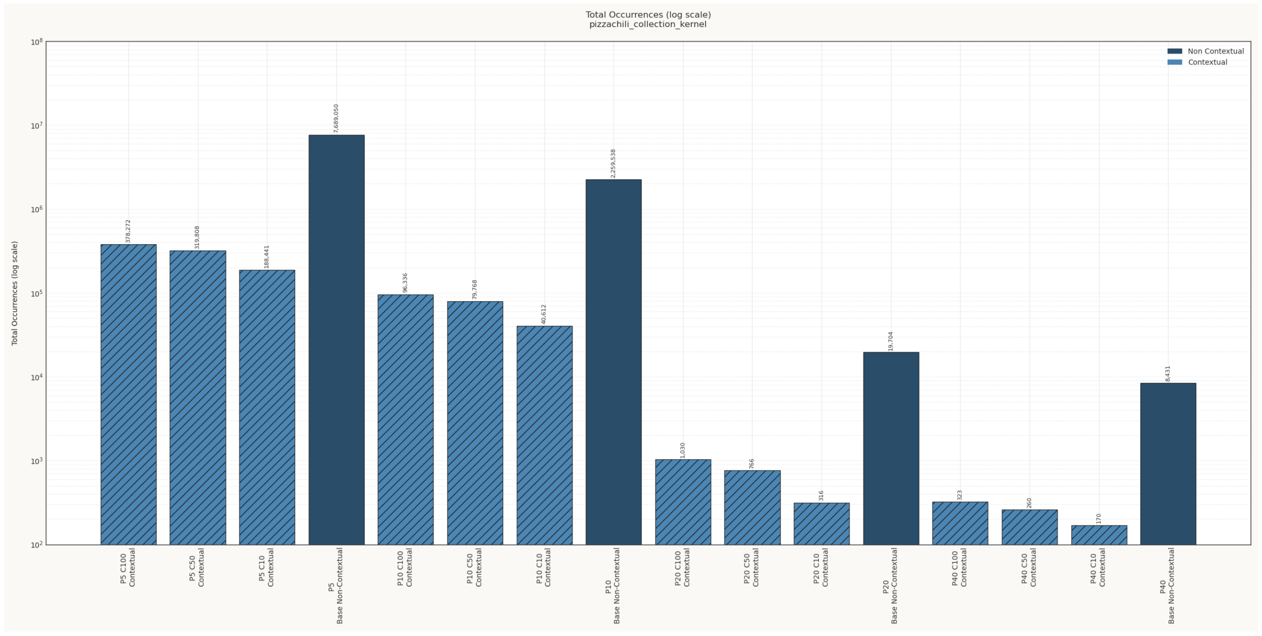
Figure B.20: Total reported occurrences on collection Kernel, for each pattern and context length including the non-contextual case. Aggregates pruned and non-pruned approaches. Logarithmically scaled.

B.3.1. LMS

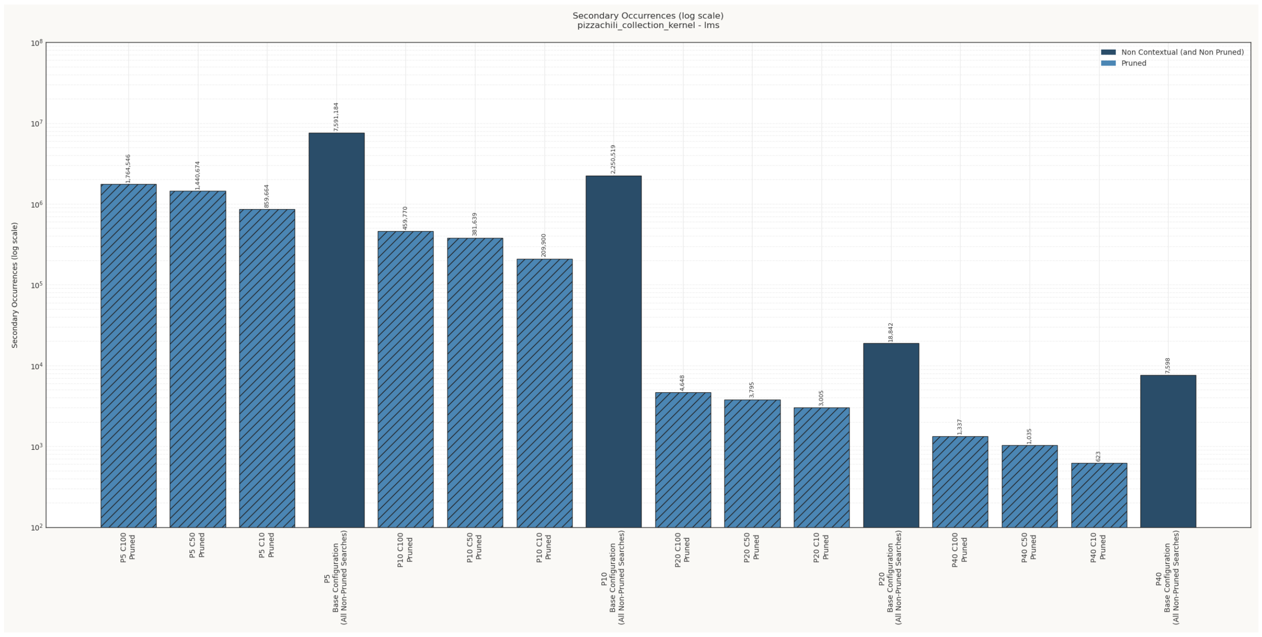

Figure B.21: Secondary occurrences found before filtering on LMS, for each pattern and context length including the non-contextual case. Aggregates non-pruned results to the non-contextual case. Logarithmically scaled.

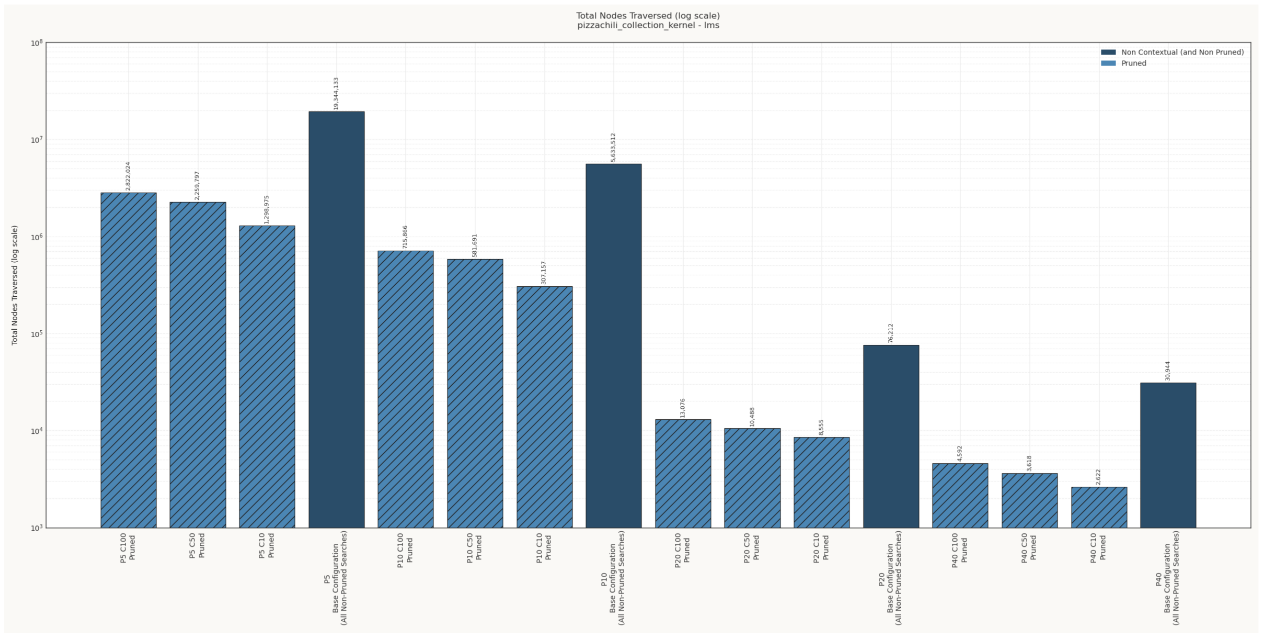

Figure B.22: Nodes traversed while computing secondary occurrences on LMS, for each pattern and context length including the non-contextual case. Aggregates non-pruned results to the non-contextual case. Logarithmically scaled.

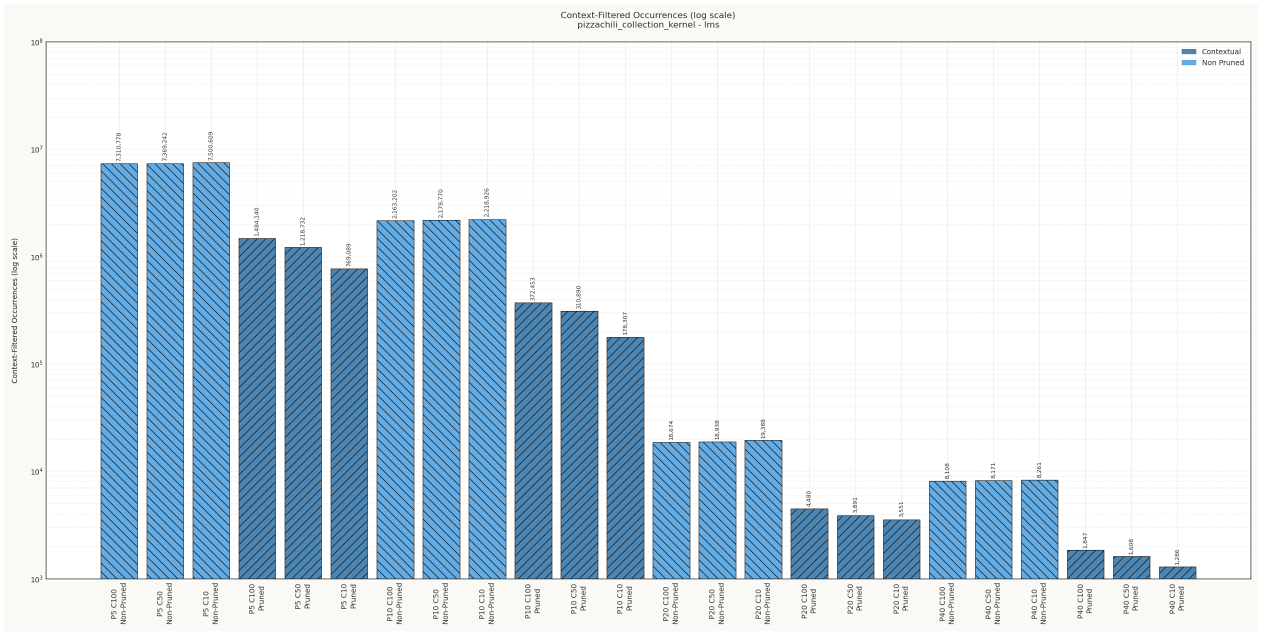
Figure B.23: Contextually filtered occurrences on LMS, for each pattern and context length and for both the pruned and non-pruned approaches. The non-contextual case is not shown since its value is 0. Logarithmically scaled.

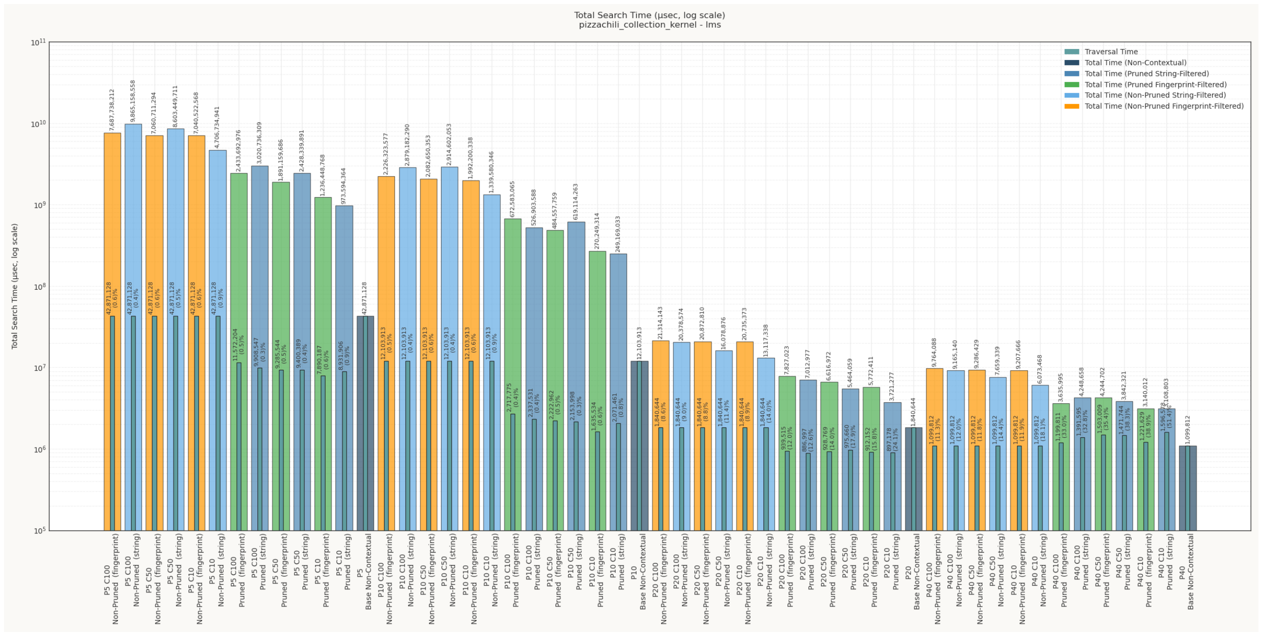
Figure B.24: Total search time, for each pattern and context length including the non-contextual case, for both the pruned and non-pruned approaches, on LMS. Additionally shows the traversal time on top of total search time for comparison, with a percentage annotation for the relationship between both values. Logarithmically scaled.

B.3.2. SLP

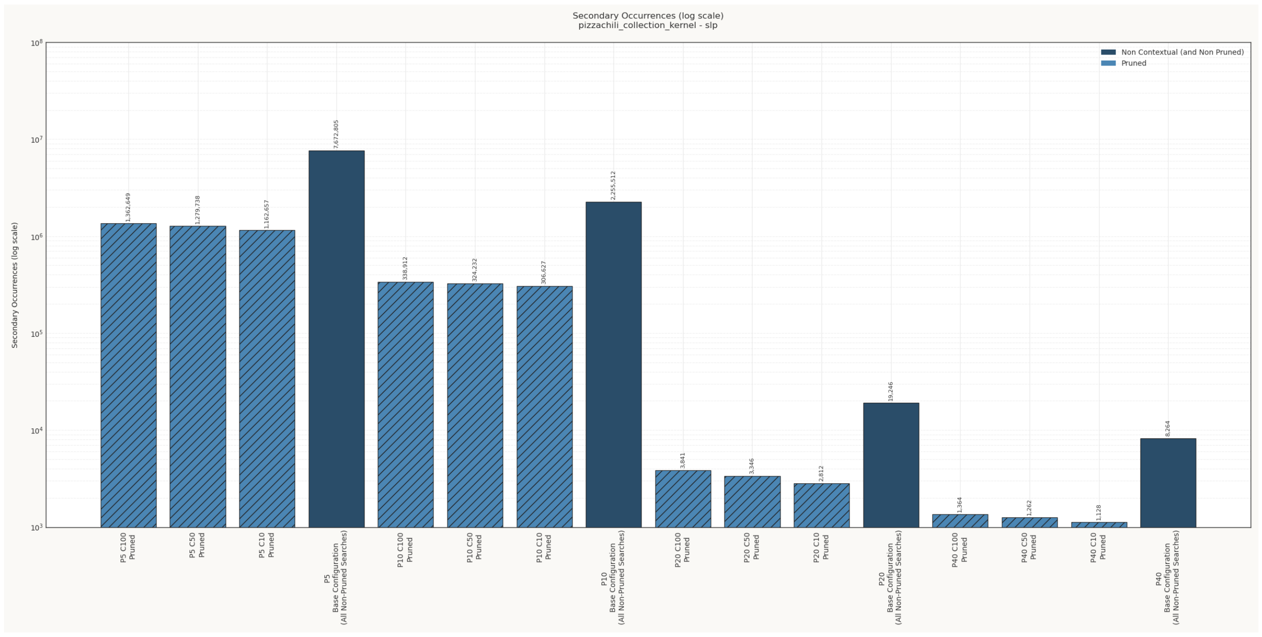

Figure B.25: Secondary occurrences found before filtering on SLP, for each pattern and context length including the non-contextual case. Aggregates non-pruned results to the non-contextual case. Logarithmically scaled.

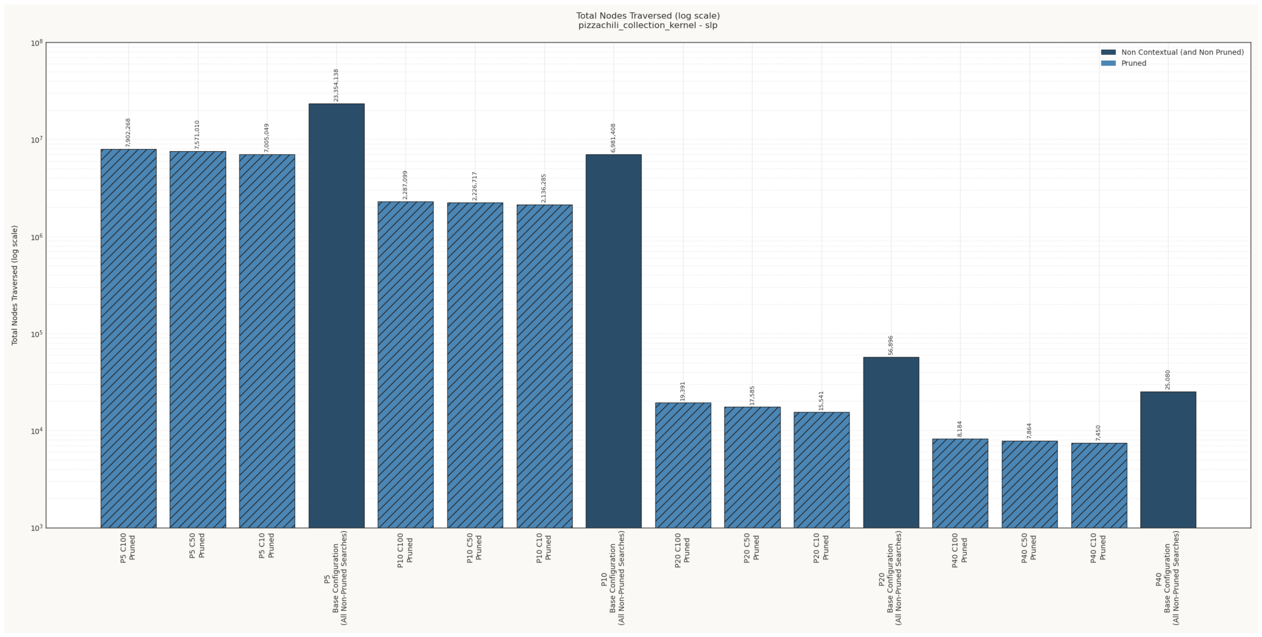

Figure B.26: Nodes traversed while computing secondary occurrences on SLP, for each pattern and context length including the non-contextual case. Aggregates non-pruned results to the non-contextual case. Logarithmically scaled.

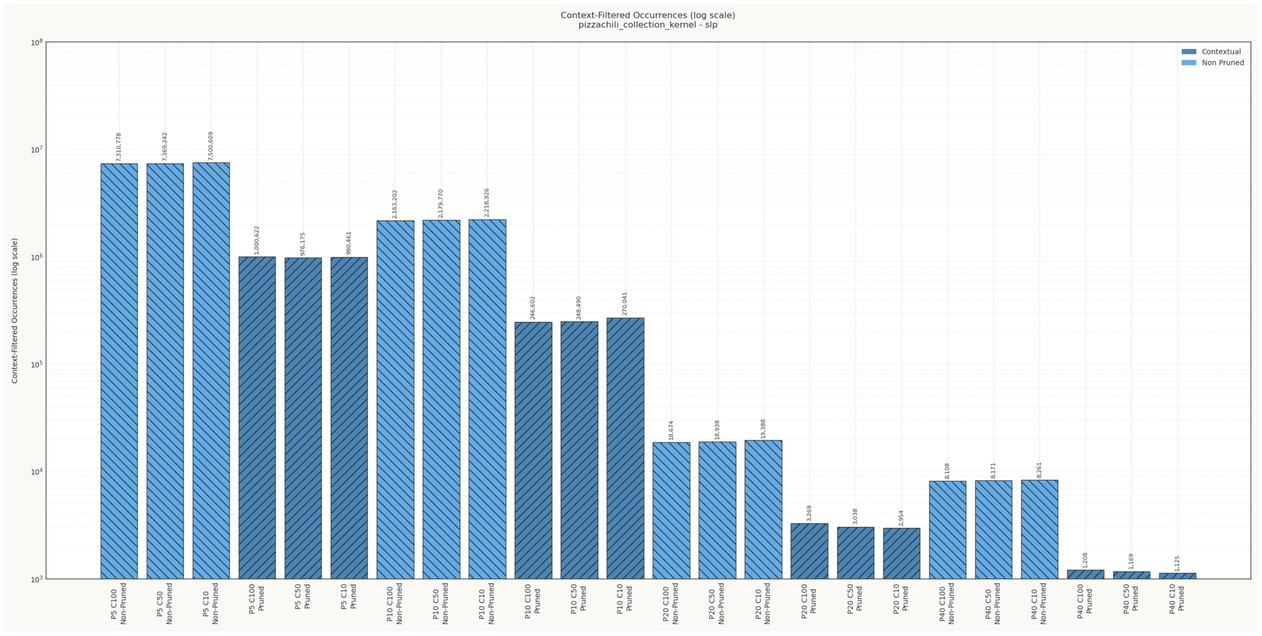

Figure B.27: Contextually filtered occurrences on SLP, for each pattern and context length and for both the pruned and non-pruned approaches. The non-contextual case is not shown since its value is 0. Logarithmically scaled.

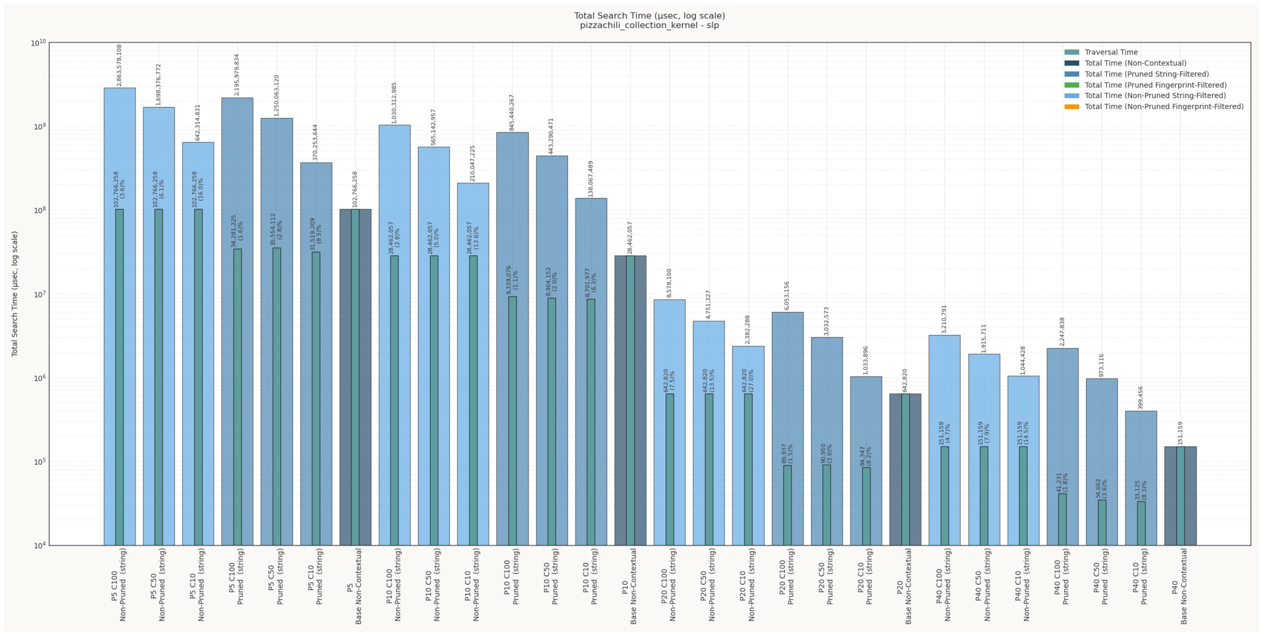

Figure B.28: Total search time, for each pattern and context length including the non-contextual case, for both the pruned and non-pruned approaches, on SLP. Additionally shows the traversal time on top of total search time for comparison, with a percentage annotation for the relationship between both values. Logarithmically scaled.

B.4. Pizzachili Collection Escherichia Coli

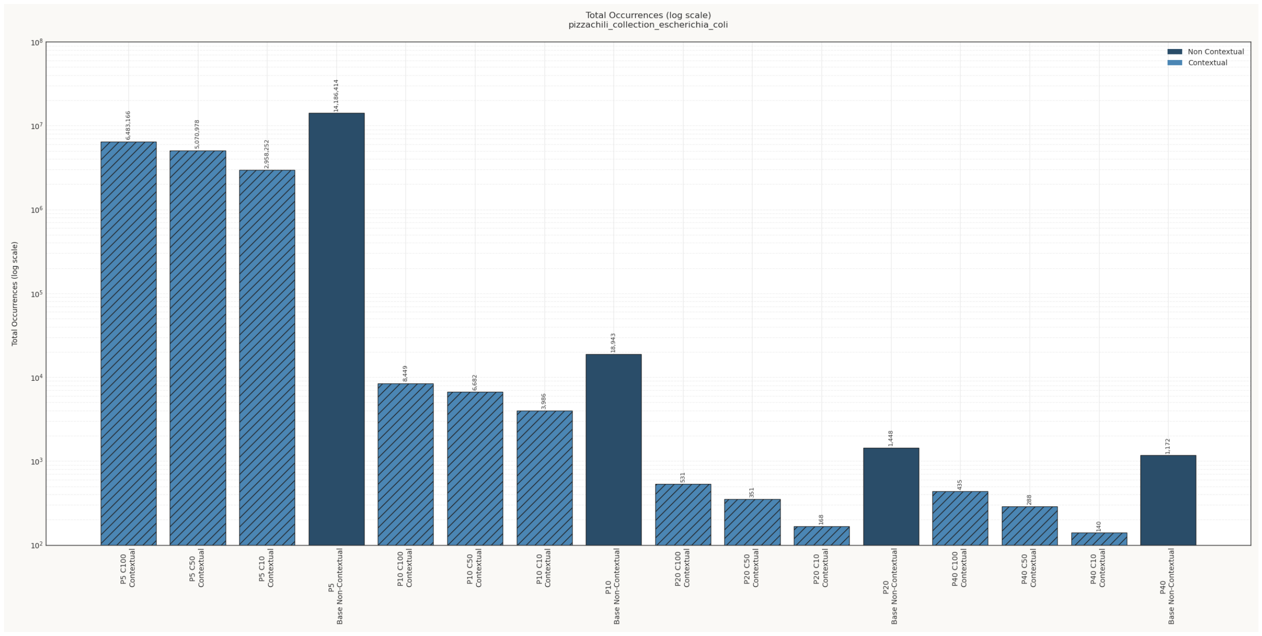

Figure B.29: Total reported occurrences on collection Escherichia Coli, for each pattern and context length including the non-contextual case. Aggregates pruned and non-pruned approaches. Logarithmically scaled.

B.4.1. LMS

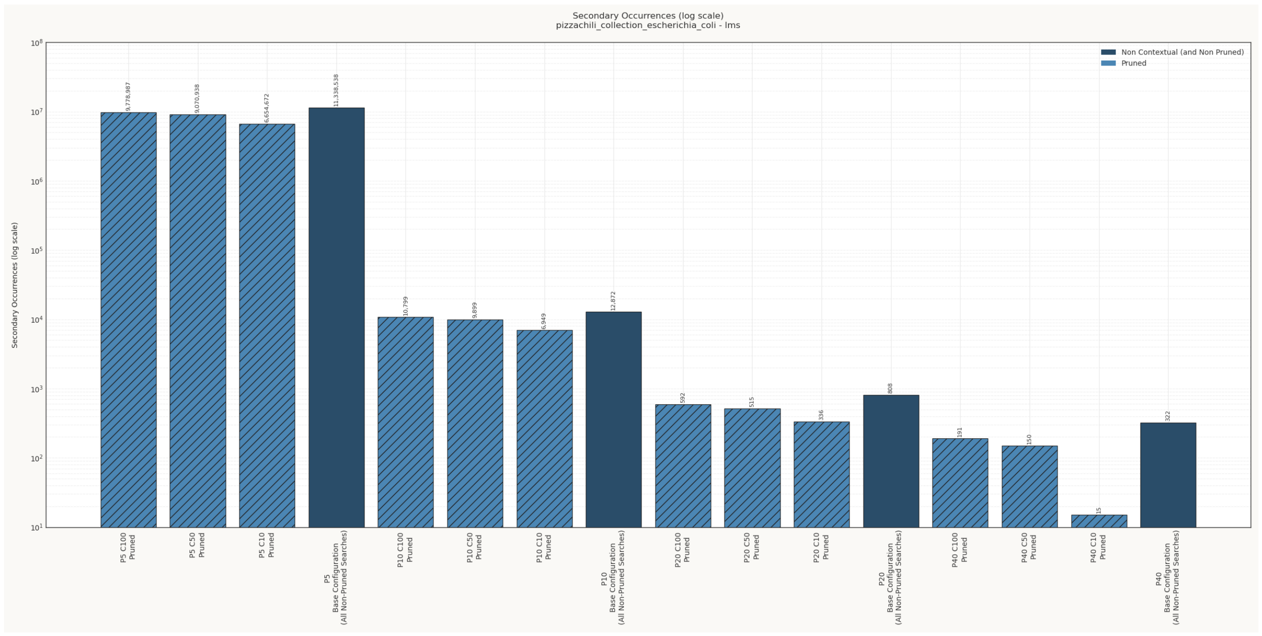

Figure B.30: Secondary occurrences found before filtering on LMS, for each pattern and context length including the non-contextual case. Aggregates non-pruned results to the non-contextual case. Logarithmically scaled.

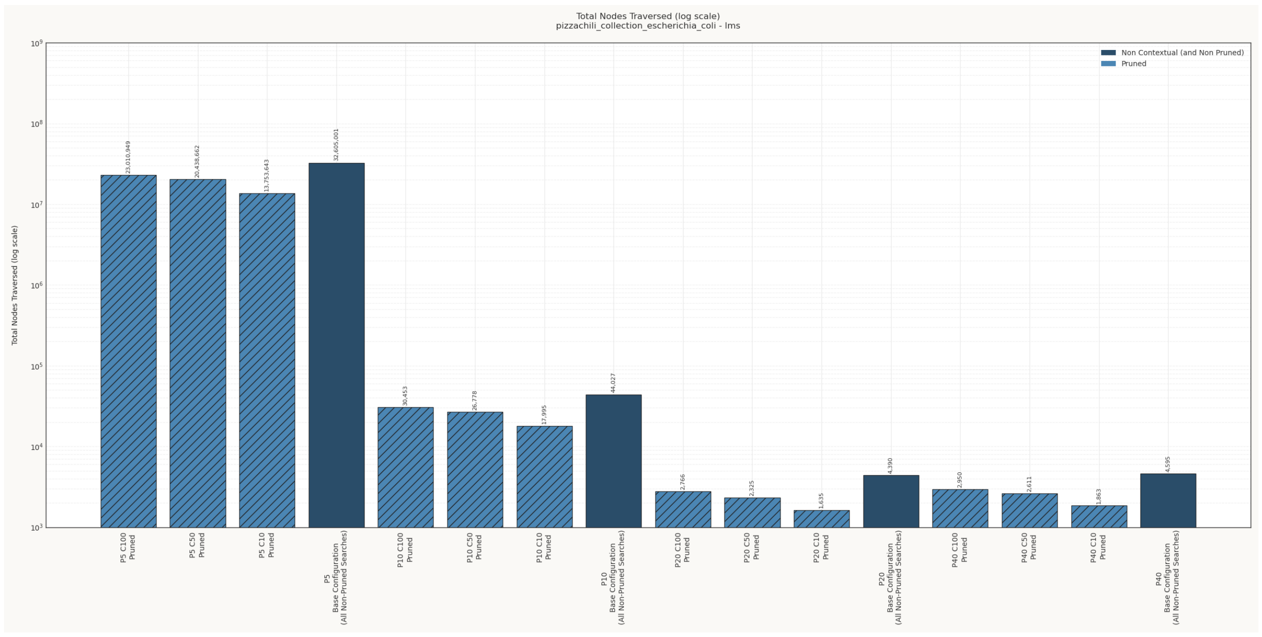

Figure B.31: Nodes traversed while computing secondary occurrences on LMS, for each pattern and context length including the non-contextual case. Aggregates non-pruned results to the non-contextual case. Logarithmically scaled.

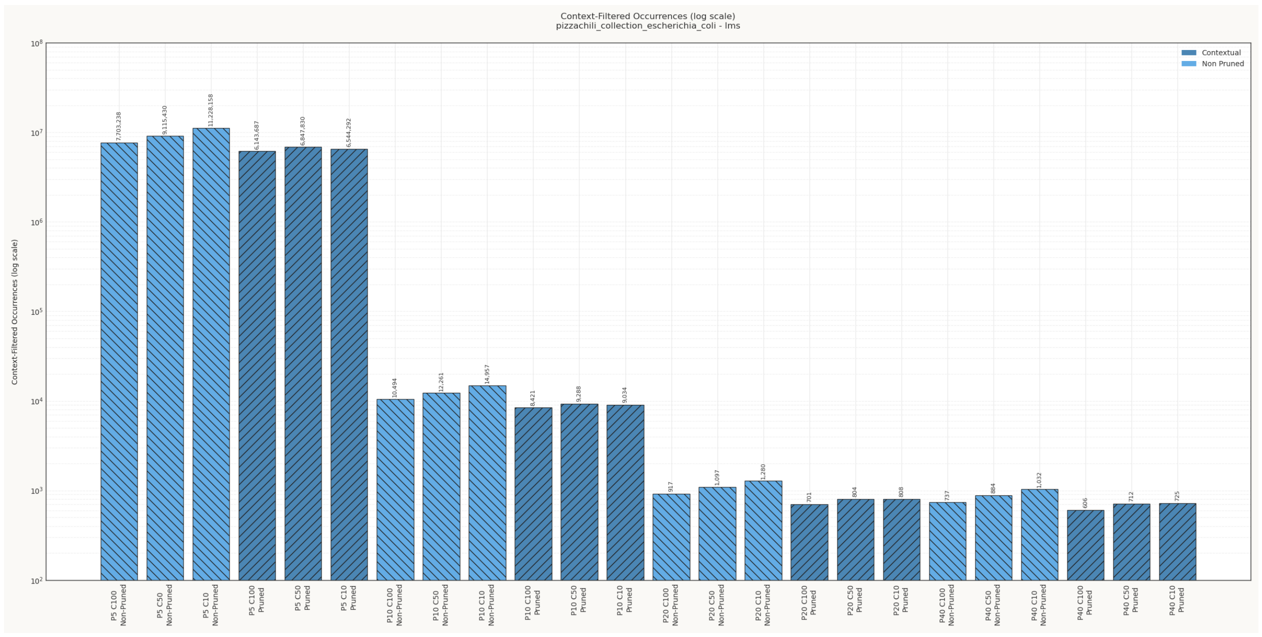

Figure B.32: Contextually filtered occurrences on LMS, for each pattern and context length and for both the pruned and non-pruned approaches. The non-contextual case is not shown since its value is 0. Logarithmically scaled.

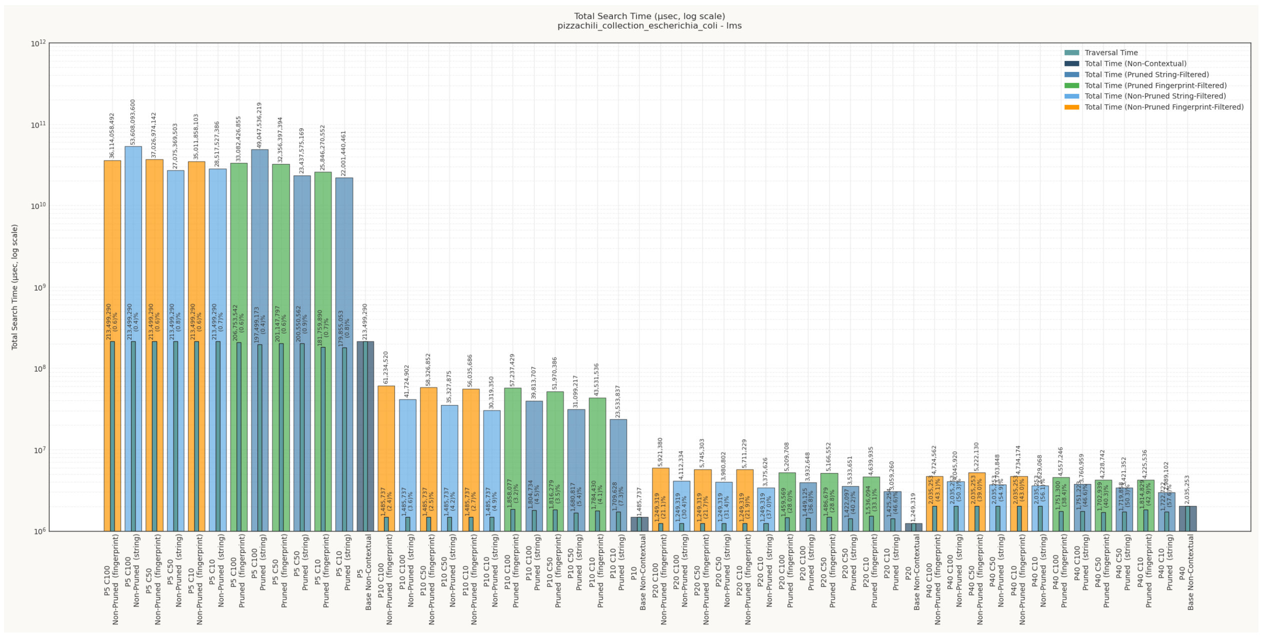
Figure B.33: Total search time, for each pattern and context length including the non-contextual case, for both the pruned and non-pruned approaches, on LMS. Additionally shows the traversal time on top of total search time for comparison, with a percentage annotation for the relationship between both values. Logarithmically scaled.

B.4.2. SLP

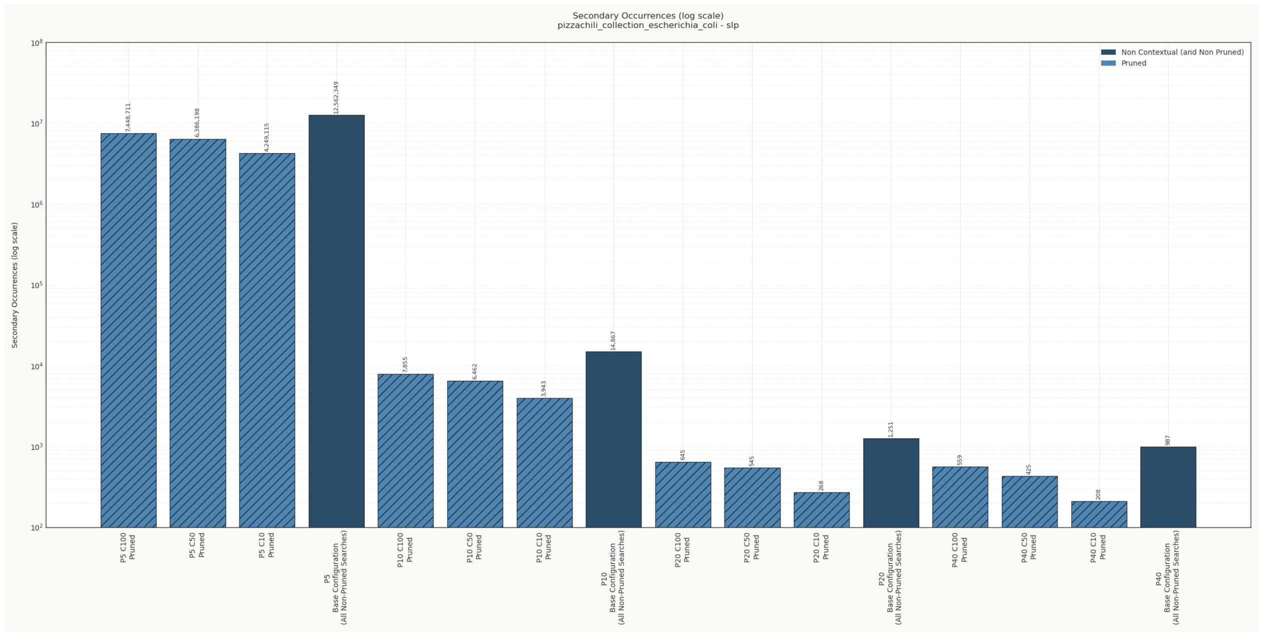

Figure B.34: Secondary occurrences found before filtering on SLP, for each pattern and context length including the non-contextual case. Aggregates non-pruned results to the non-contextual case. Logarithmically scaled.

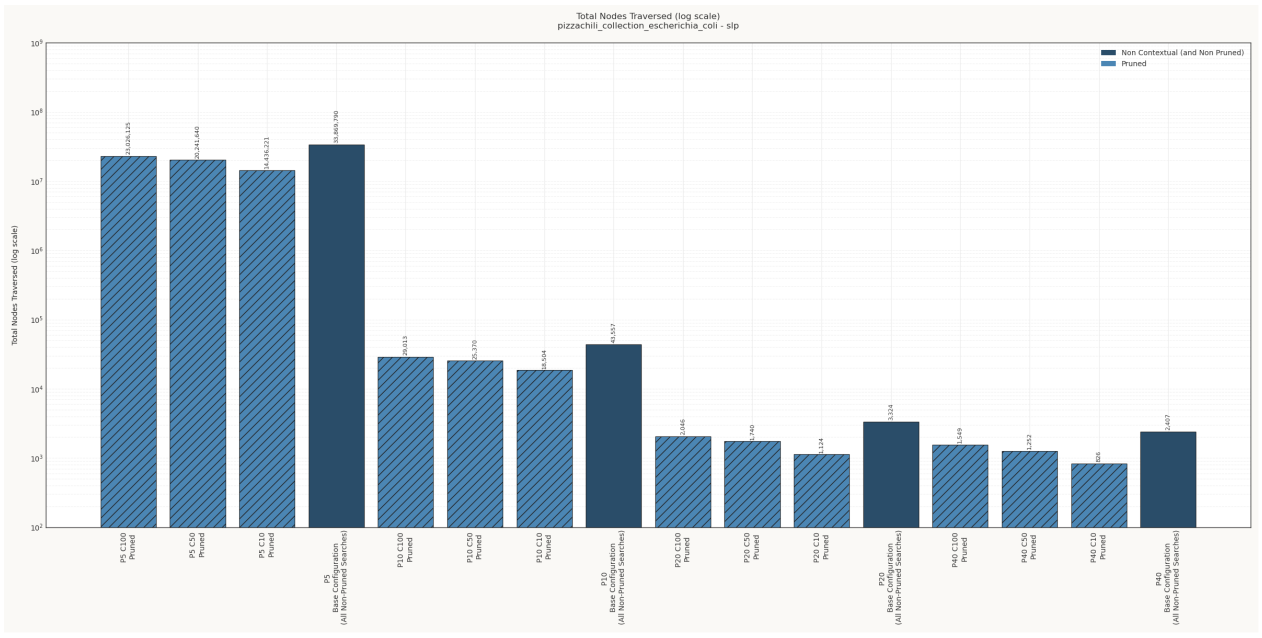

Figure B.35: Nodes traversed while computing secondary occurrences on SLP, for each pattern and context length including the non-contextual case. Aggregates non-pruned results to the non-contextual case. Logarithmically scaled.

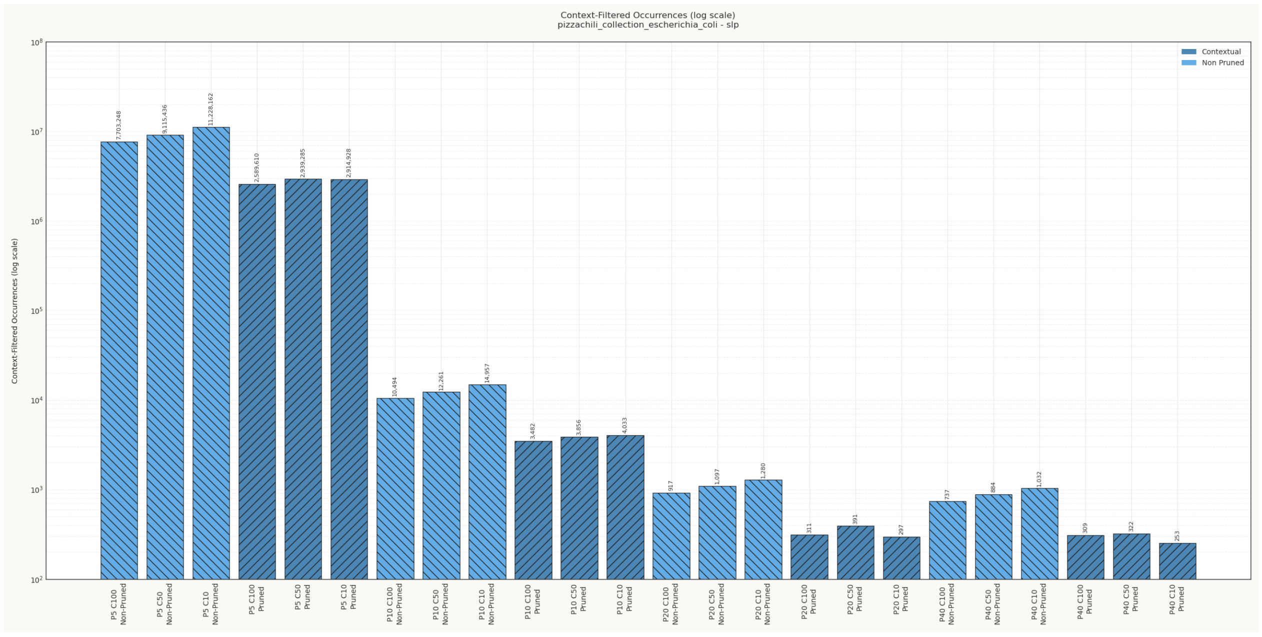

Figure B.36: Contextually filtered occurrences on SLP, for each pattern and context length and for both the pruned and non-pruned approaches. The non-contextual case is not shown since its value is 0. Logarithmically scaled.

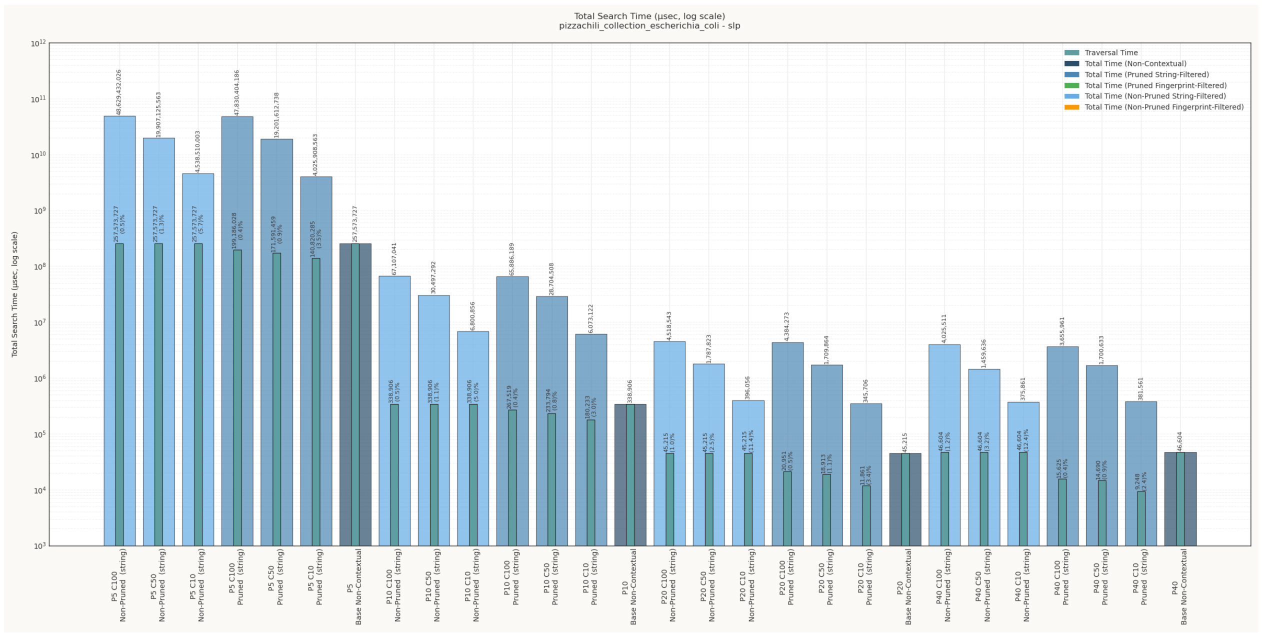

Figure B.37: Total search time, for each pattern and context length including the non-contextual case, for both the pruned and non-pruned approaches, on SLP. Additionally shows the traversal time on top of total search time for comparison, with a percentage annotation for the relationship between both values. Logarithmically scaled.

B.5. Pizzachili Collection Influenza

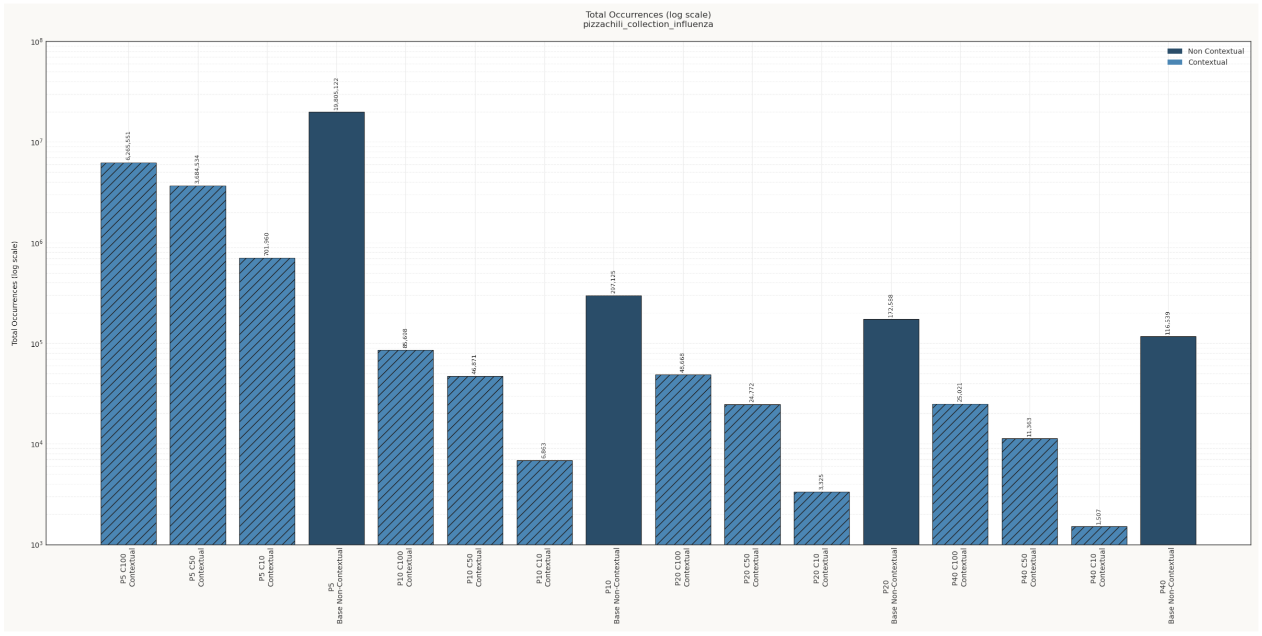

Figure B.38: Total reported occurrences on collection Influenza, for each pattern and context length including the non-contextual case. Aggregates pruned and non-pruned approaches. Logarithmically scaled.

B.5.1. LMS

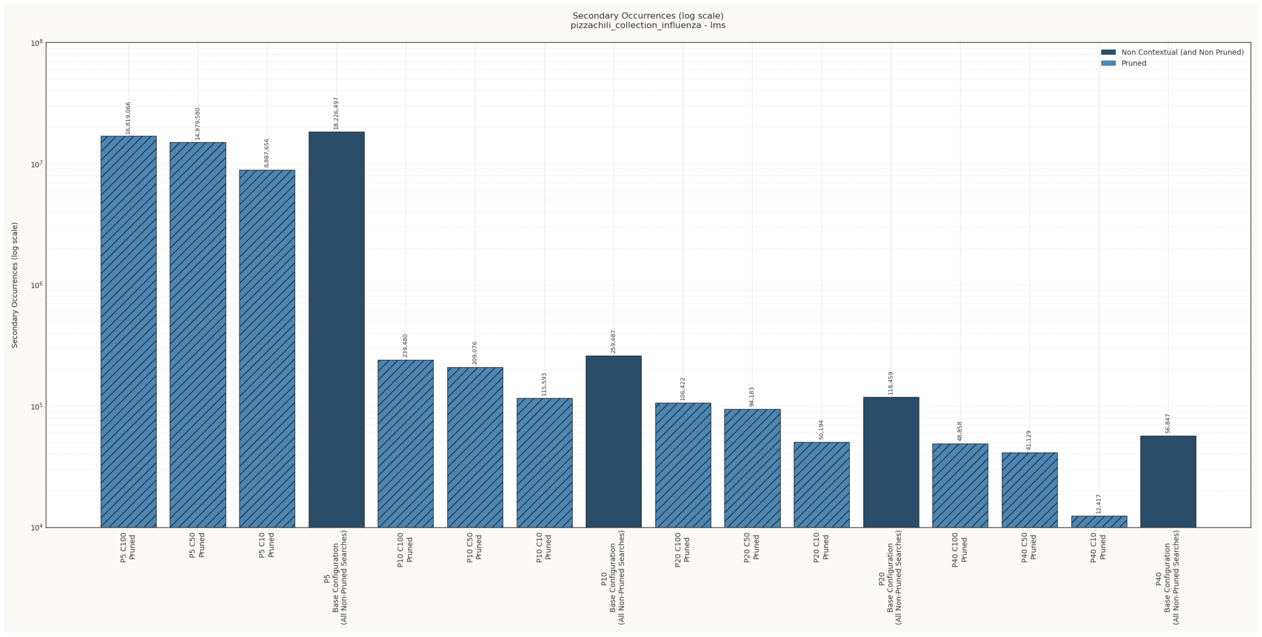

Figure B.39: Secondary occurrences found before filtering on LMS, for each pattern and context length including the non-contextual case. Aggregates non-pruned results to the non-contextual case. Logarithmically scaled.

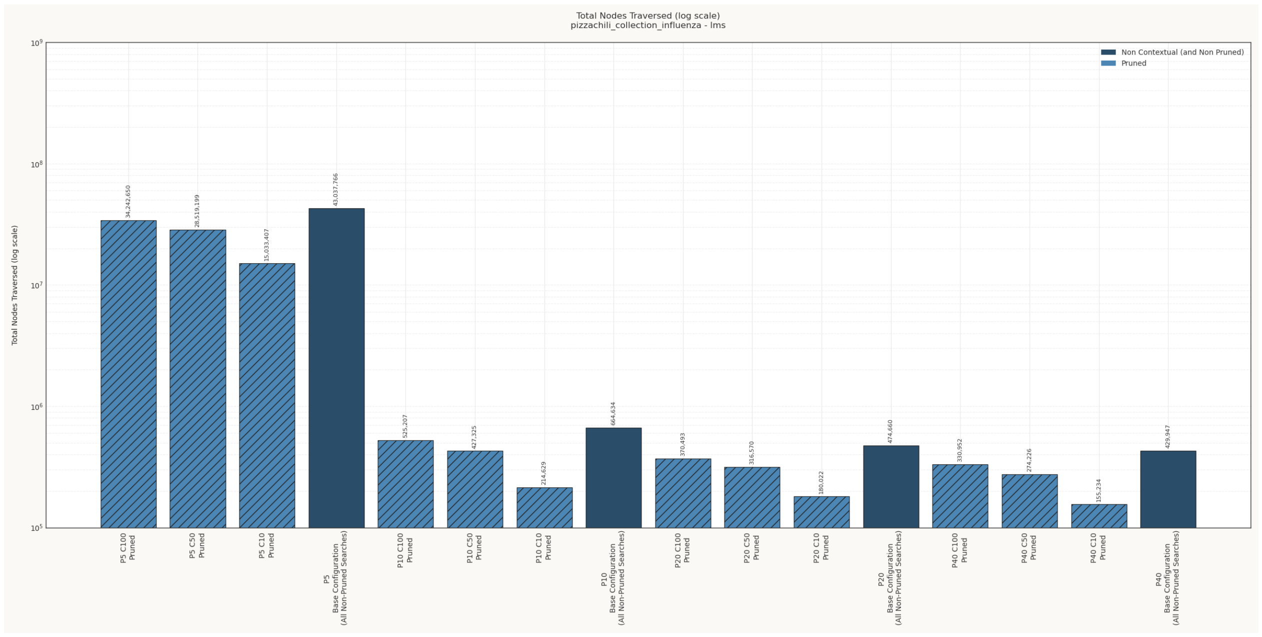

Figure B.40: Nodes traversed while computing secondary occurrences on LMS, for each pattern and context length including the non-contextual case. Aggregates non-pruned results to the non-contextual case. Logarithmically scaled.

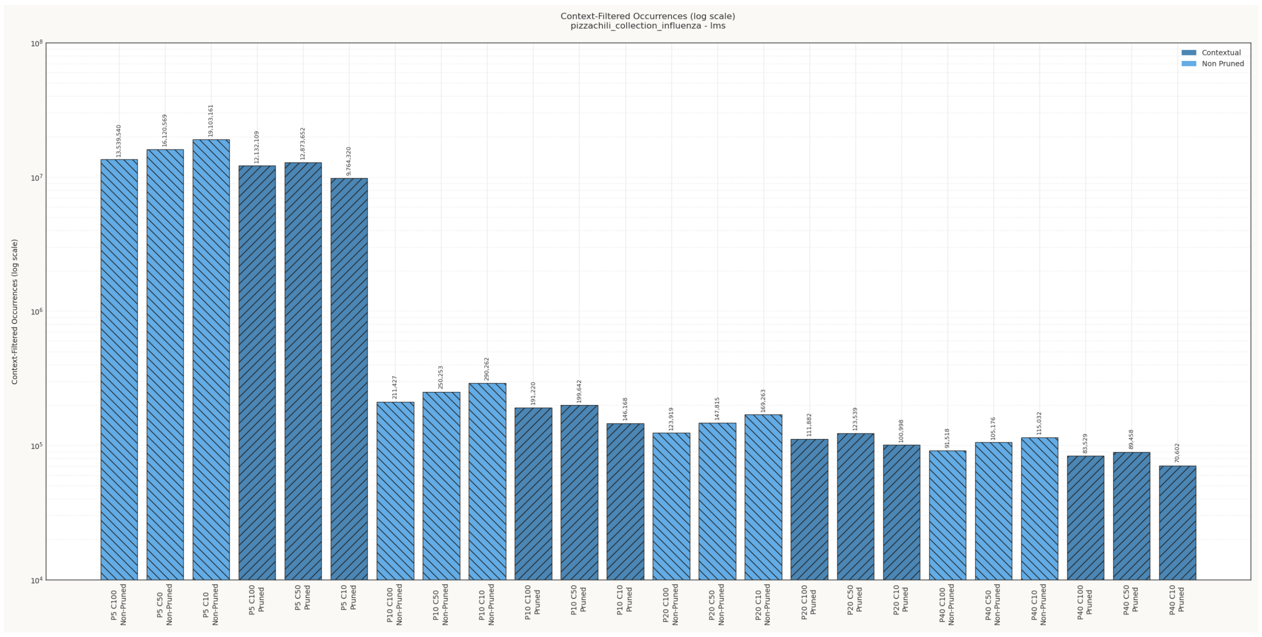
Figure B.41: Contextually filtered occurrences on LMS, for each pattern and context length and for both the pruned and non-pruned approaches. The non-contextual case is not shown since its value is 0. Logarithmically scaled.

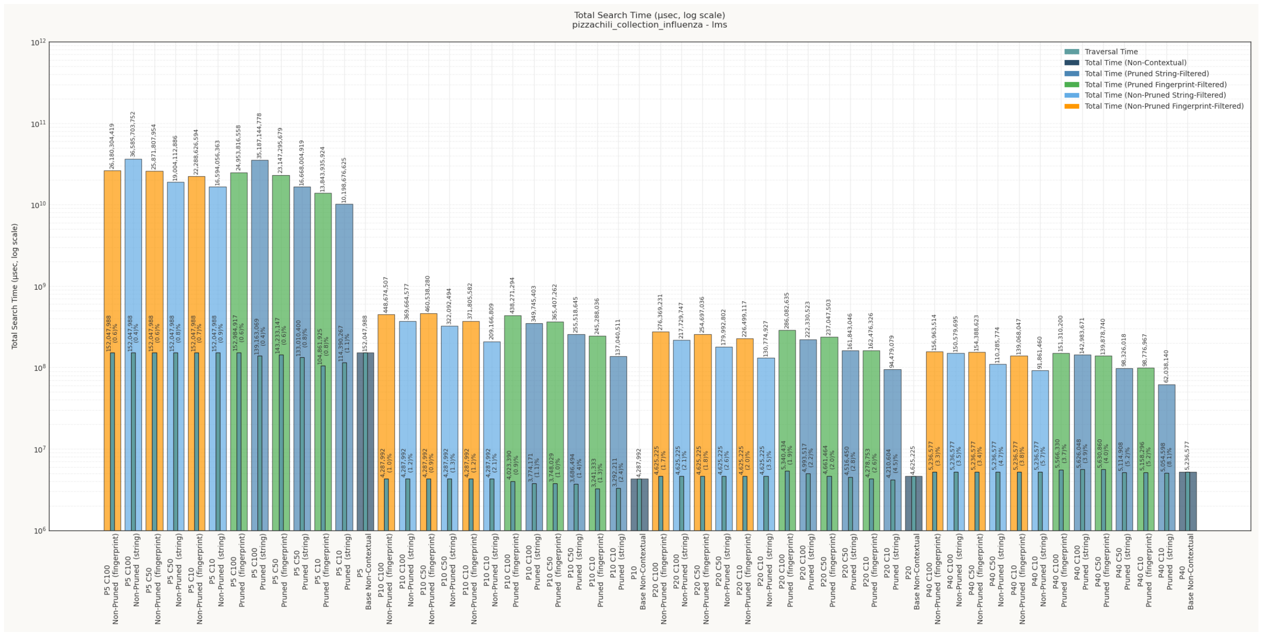
Figure B.42: Total search time, for each pattern and context length including the non-contextual case, for both the pruned and non-pruned approaches, on LMS. Additionally shows the traversal time on top of total search time for comparison, with a percentage annotation for the relationship between both values. Logarithmically scaled.

B.5.2. SLP

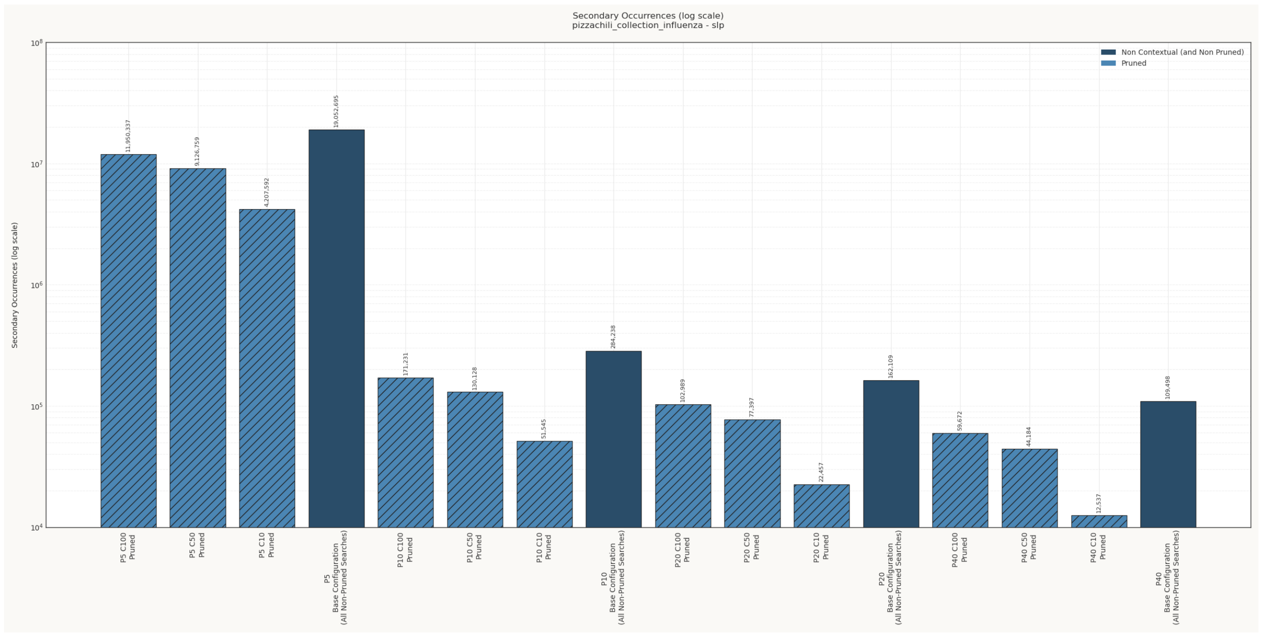

Figure B.43: Secondary occurrences found before filtering on SLP, for each pattern and context length including the non-contextual case. Aggregates non-pruned results to the non-contextual case. Logarithmically scaled.

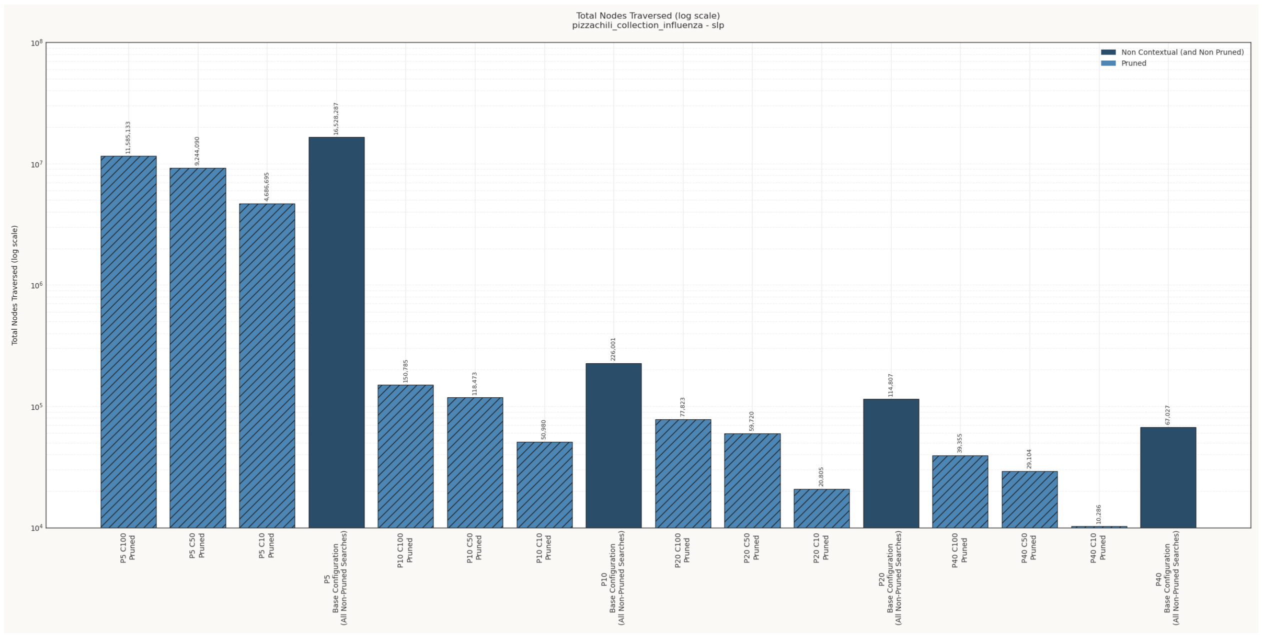

Figure B.44: Nodes traversed while computing secondary occurrences on SLP, for each pattern and context length including the non-contextual case. Aggregates non-pruned results to the non-contextual case. Logarithmically scaled.

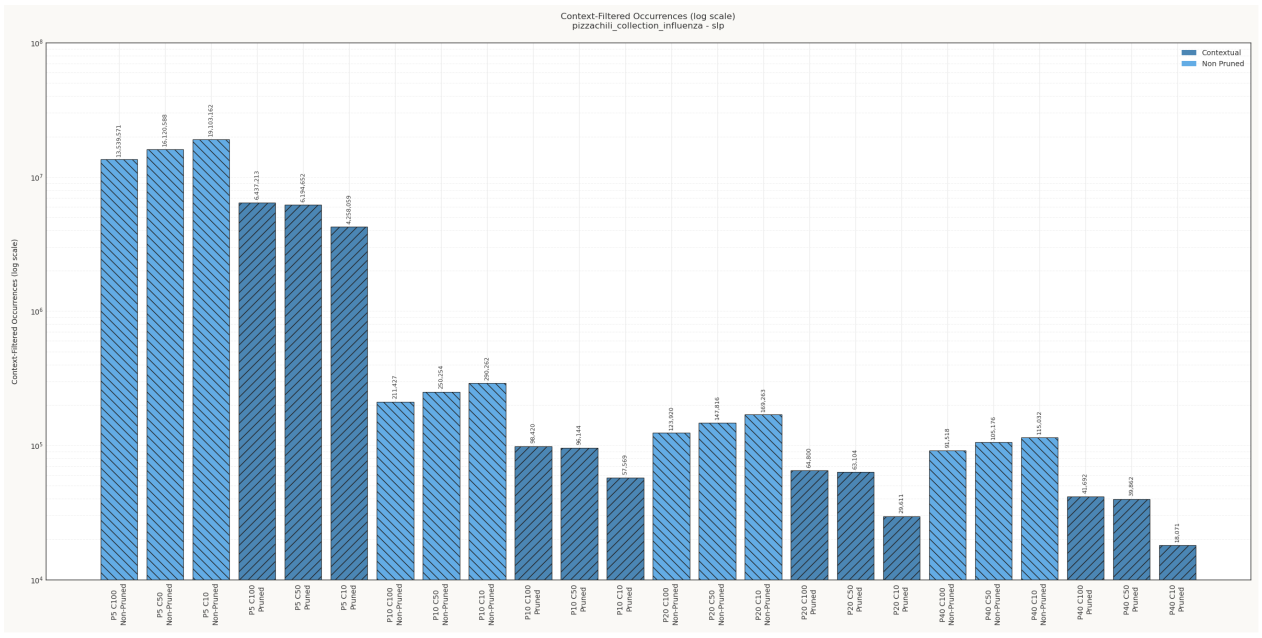

Figure B.45: Contextually filtered occurrences on SLP, for each pattern and context length and for both the pruned and non-pruned approaches. The non-contextual case is not shown since its value is 0. Logarithmically scaled.

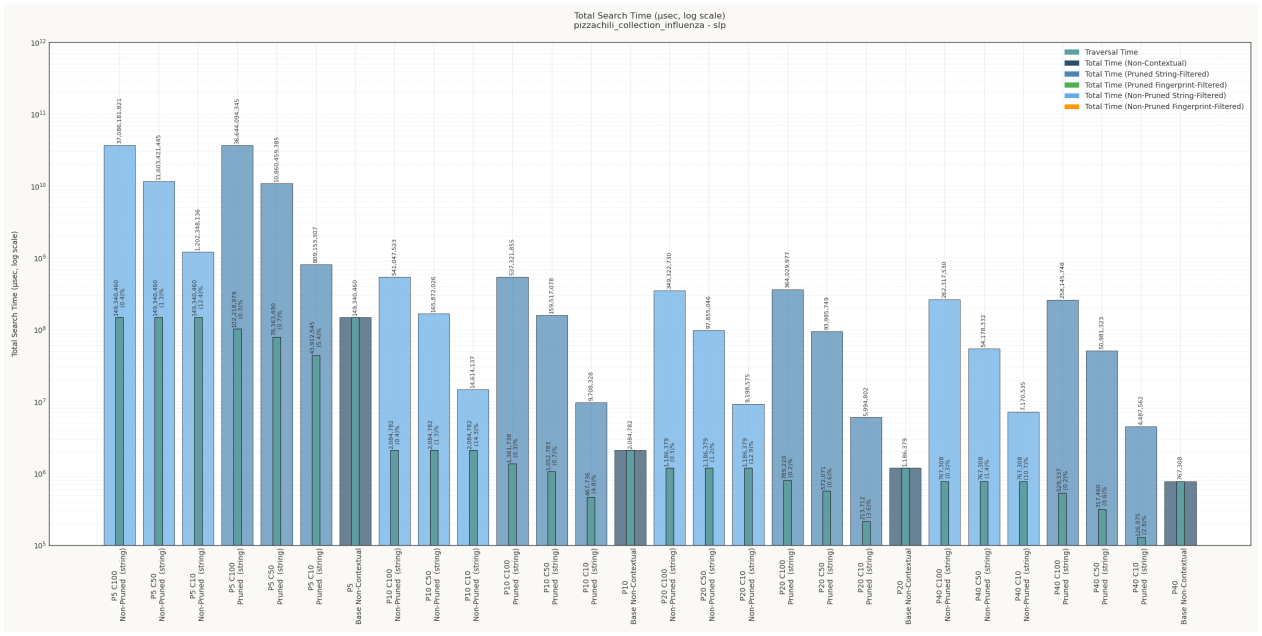

Figure B.46: Total search time, for each pattern and context length including the non-contextual case, for both the pruned and non-pruned approaches, on SLP. Additionally shows the traversal time on top of total search time for comparison, with a percentage annotation for the relationship between both values. Logarithmically scaled.

B.6. Pizzachili Collection Para

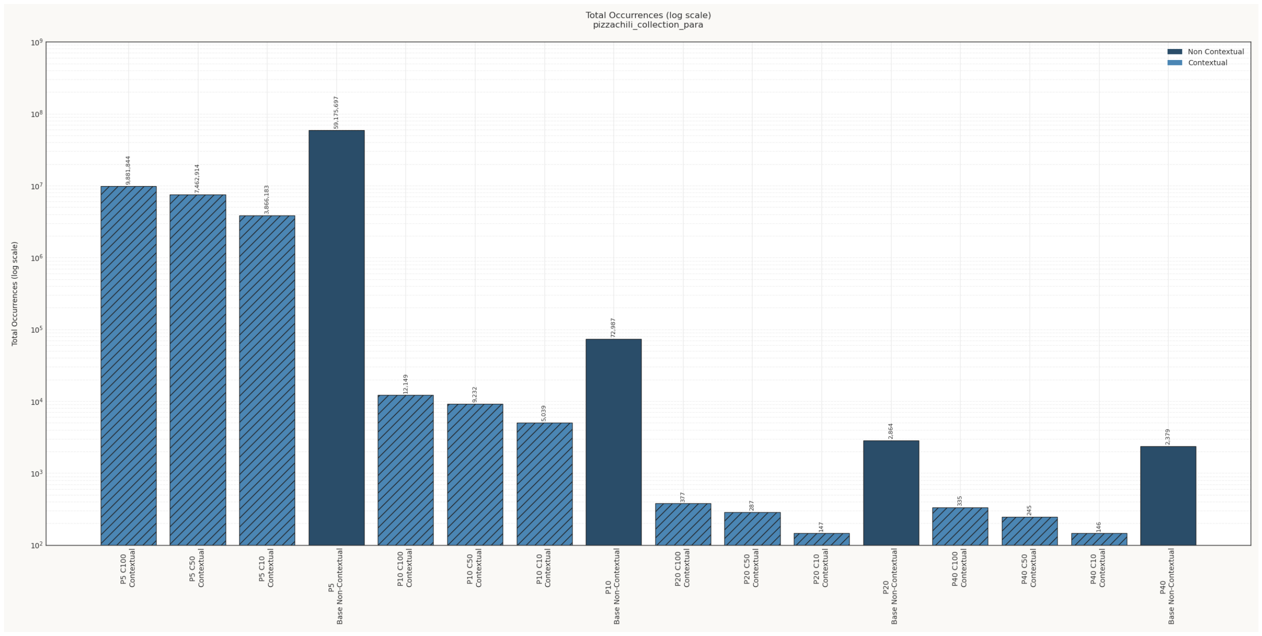

Figure B.47: Total reported occurrences on collection Para, for each pattern and context length including the non-contextual case. Aggregates pruned and non-pruned approaches. Logarithmically scaled.

B.6.1. LMS

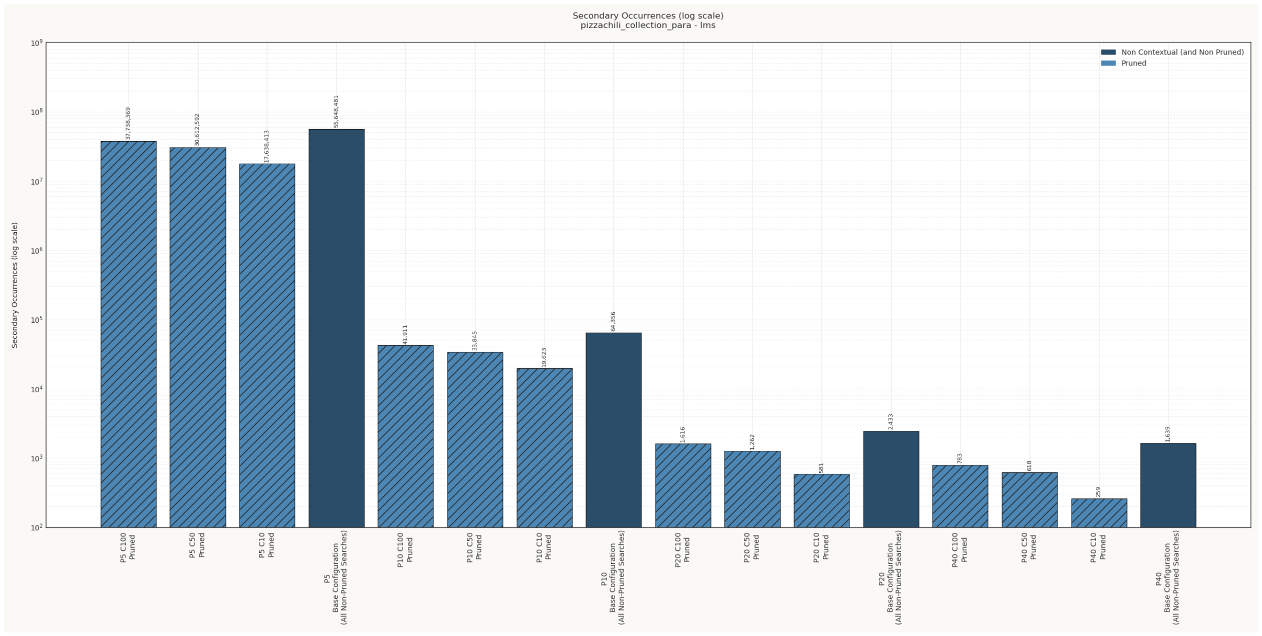

Figure B.48: Secondary occurrences found before filtering on LMS, for each pattern and context length including the non-contextual case. Aggregates non-pruned results to the non-contextual case. Logarithmically scaled.

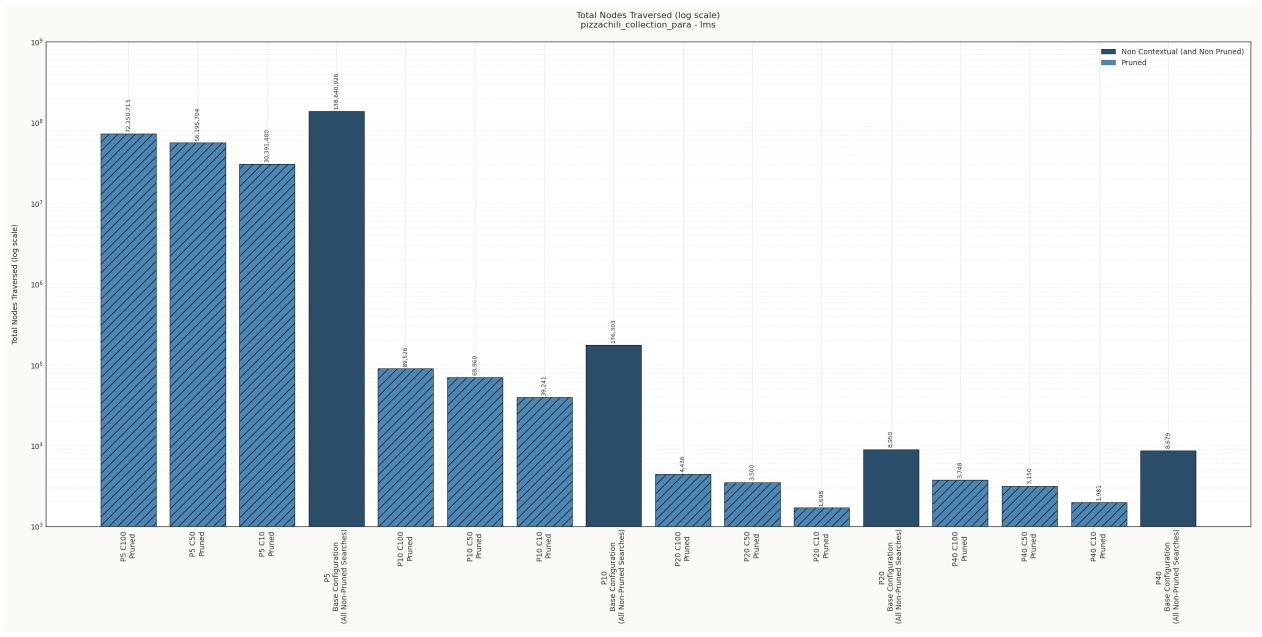

Figure B.49: Nodes traversed while computing secondary occurrences on LMS, for each pattern and context length including the non-contextual case. Aggregates non-pruned results to the non-contextual case. Logarithmically scaled.

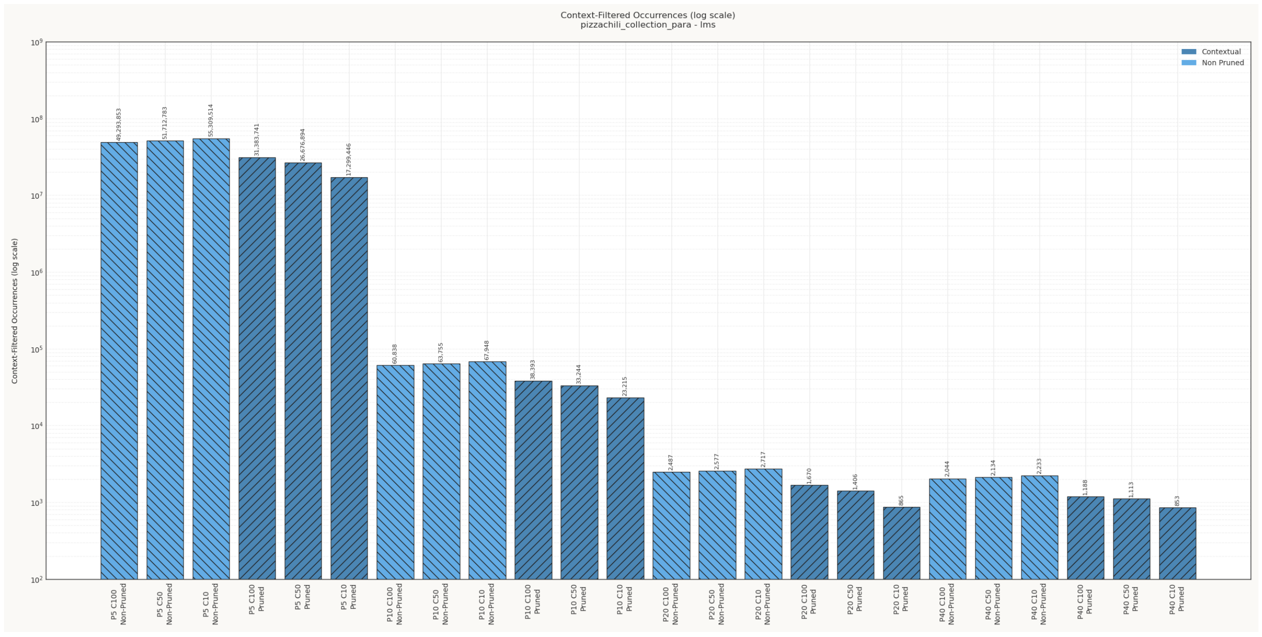

Figure B.50: Contextually filtered occurrences on LMS, for each pattern and context length and for both the pruned and non-pruned approaches. The non-contextual case is not shown since its value is 0. Logarithmically scaled.

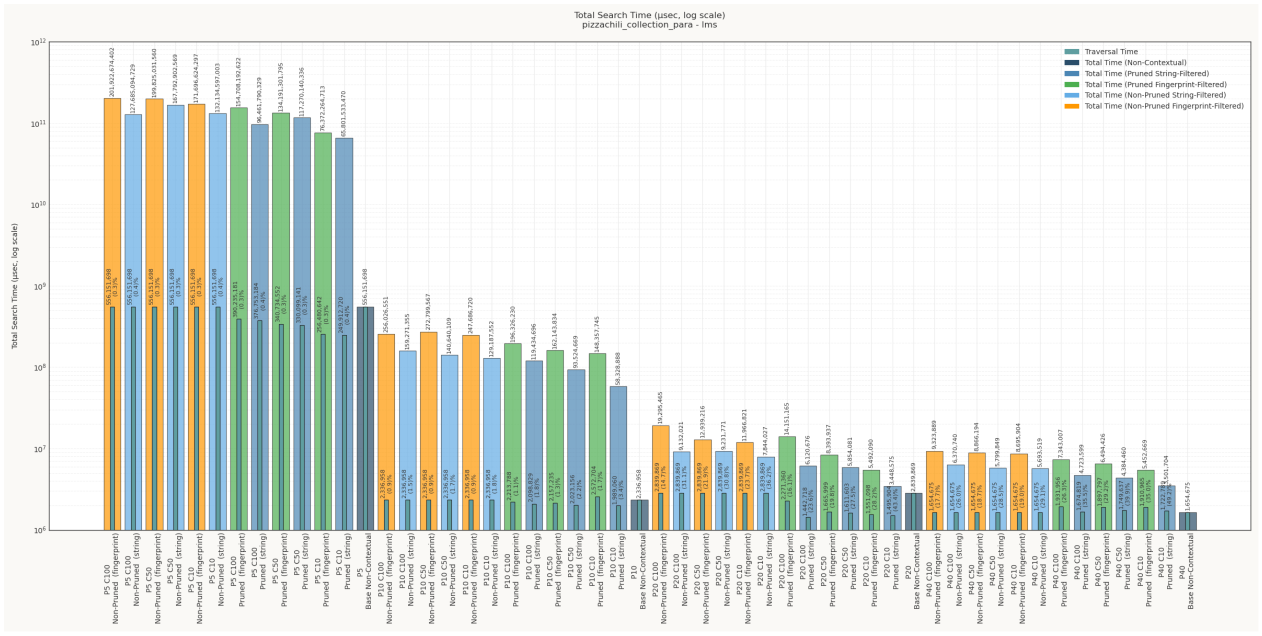

Figure B.51: Total search time, for each pattern and context length including the non-contextual case, for both the pruned and non-pruned approaches, on LMS. Additionally shows the traversal time on top of total search time for comparison, with a percentage annotation for the relationship between both values. Logarithmically scaled.

B.6.2. SLP

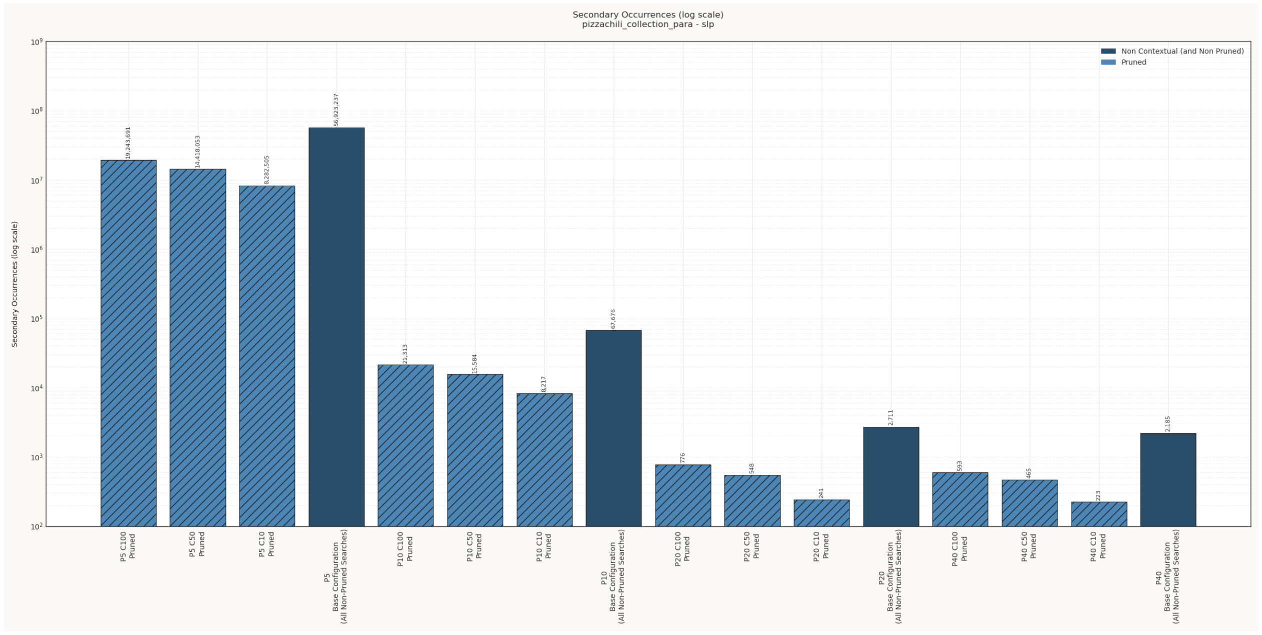

Figure B.52: Secondary occurrences found before filtering on SLP, for each pattern and context length including the non-contextual case. Aggregates non-pruned results to the non-contextual case. Logarithmically scaled.

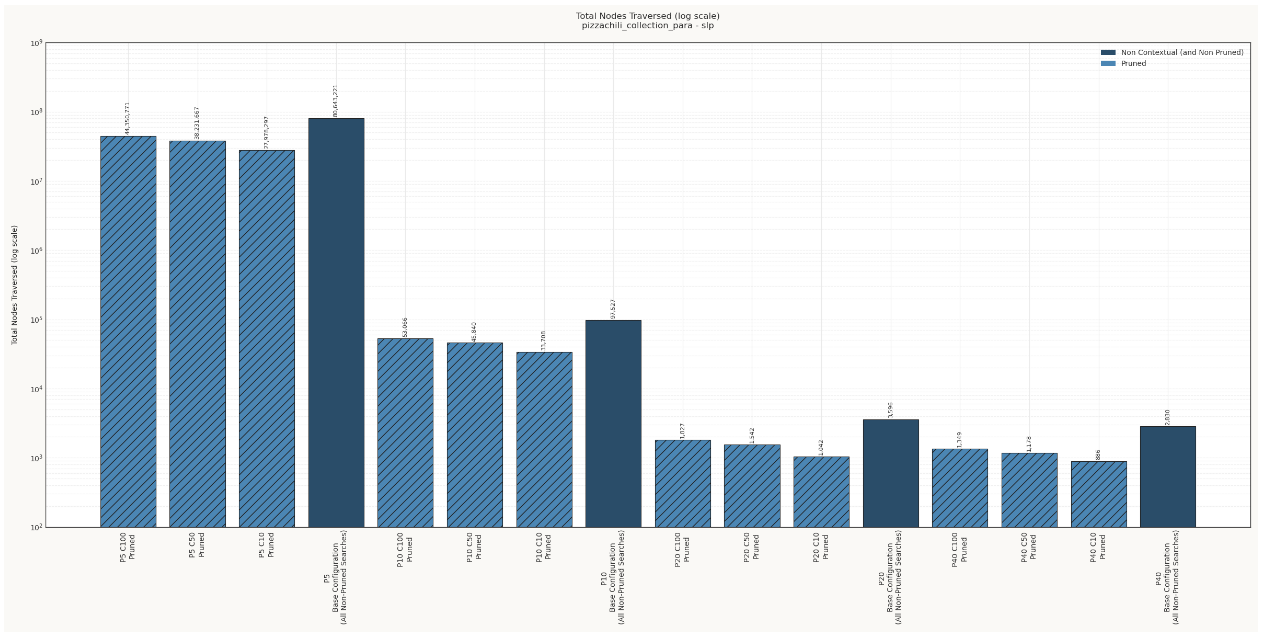

Figure B.53: Nodes traversed while computing secondary occurrences on SLP, for each pattern and context length including the non-contextual case. Aggregates non-pruned results to the non-contextual case. Logarithmically scaled.

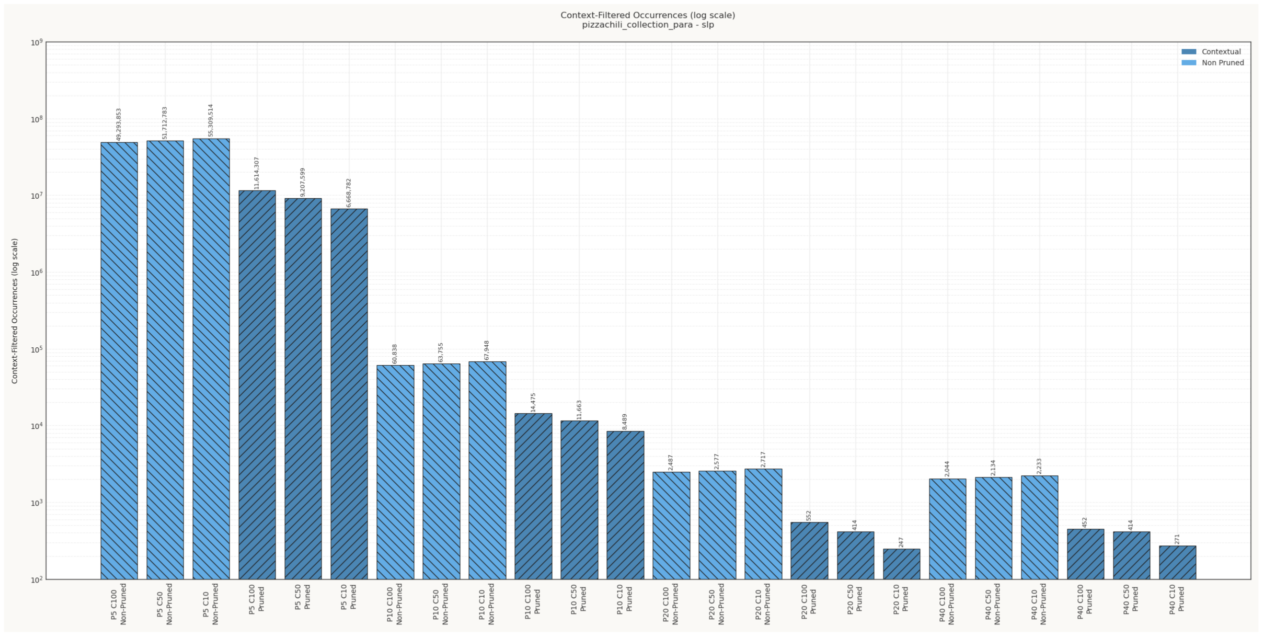

Figure B.54: Contextually filtered occurrences on SLP, for each pattern and context length and for both the pruned and non-pruned approaches. The non-contextual case is not shown since its value is 0. Logarithmically scaled.

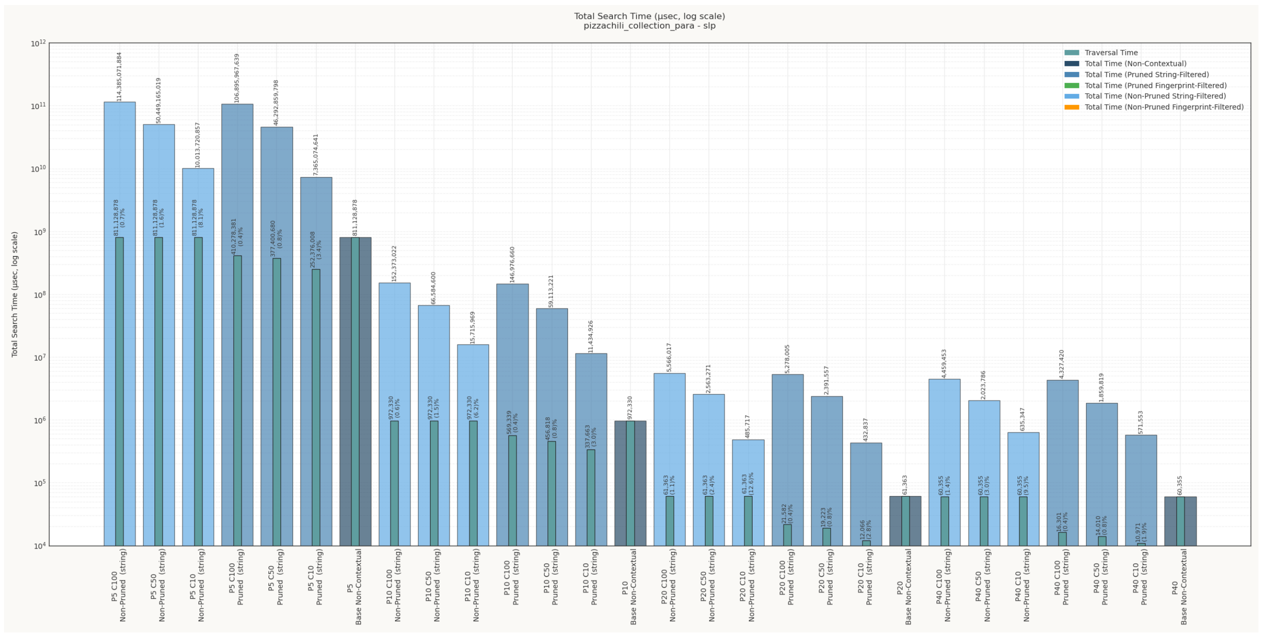
Figure B.55: Total search time, for each pattern and context length including the non-contextual case, for both the pruned and non-pruned approaches, on SLP. Additionally shows the traversal time on top of total search time for comparison, with a percentage annotation for the relationship between both values. Logarithmically scaled.

B.7. Pizzachili Collection Cere

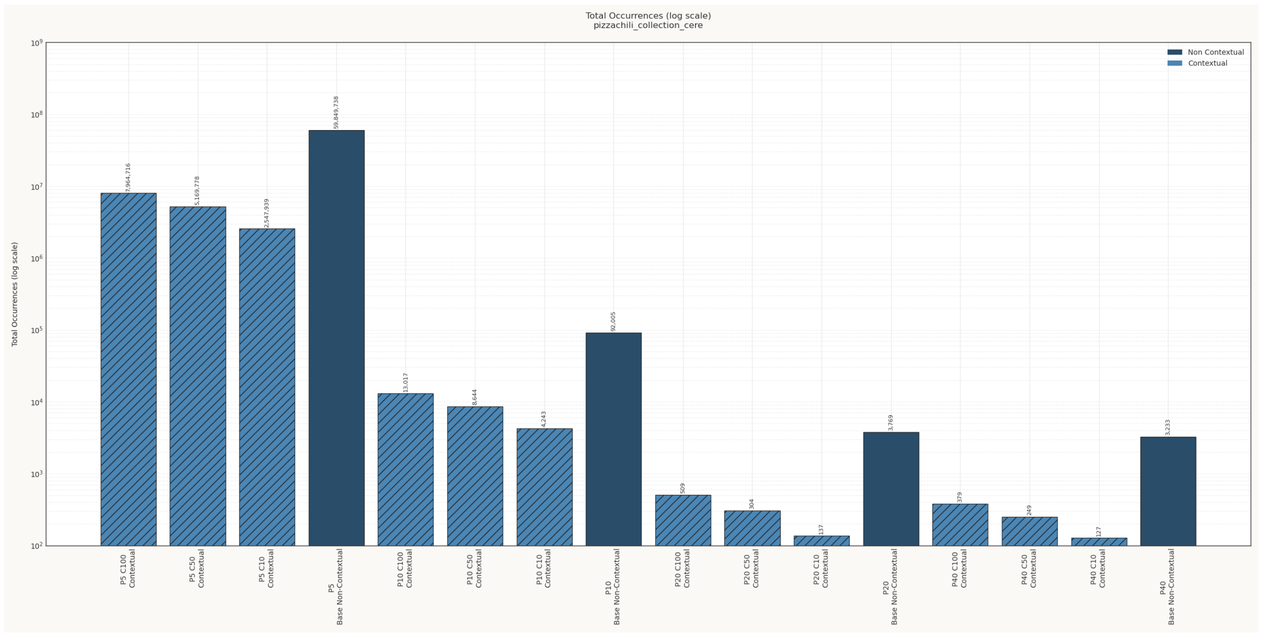
Figure B.56: Total reported occurrences on collection Cere, for each pattern and context length including the non-contextual case. Aggregates pruned and non-pruned approaches. Logarithmically scaled.

B.7.1. LMS

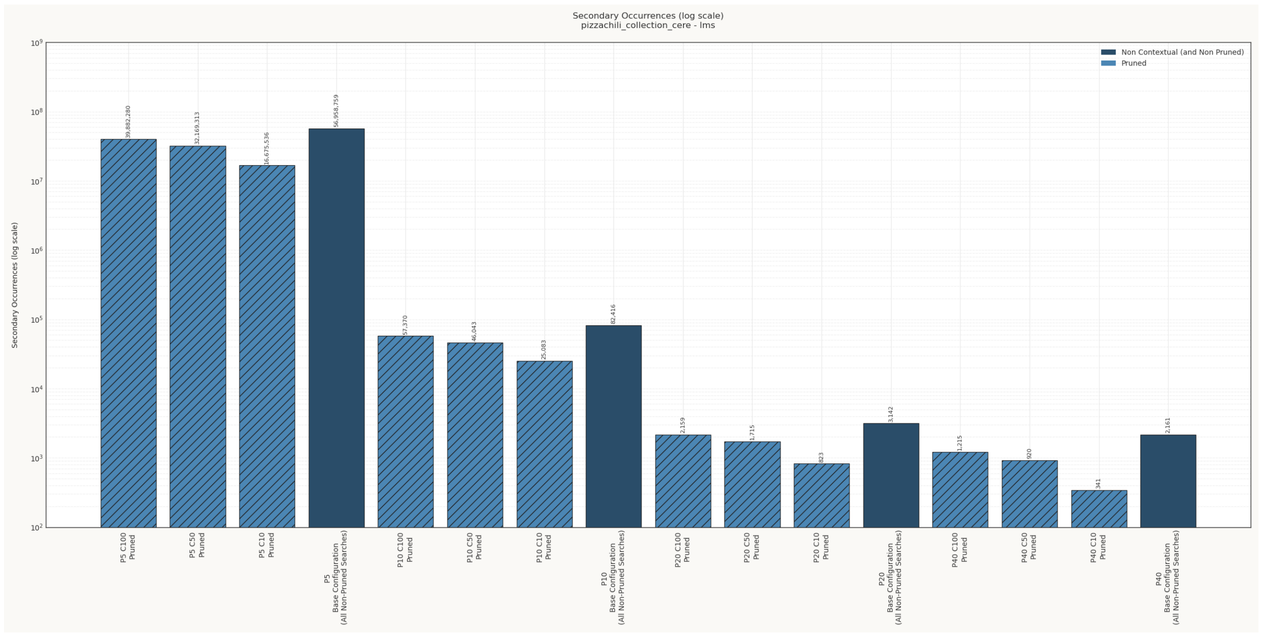

Figure B.57: Secondary occurrences found before filtering on LMS, for each pattern and context length including the non-contextual case. Aggregates non-pruned results to the non-contextual case. Logarithmically scaled.

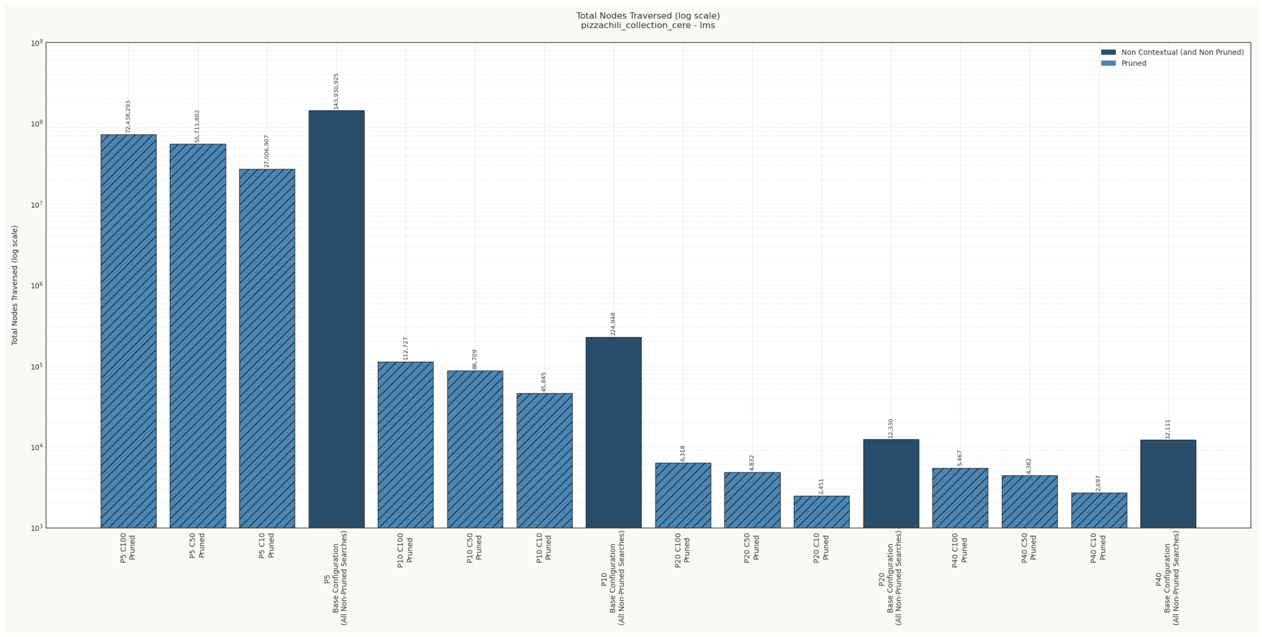

Figure B.58: Nodes traversed while computing secondary occurrences on LMS, for each pattern and context length including the non-contextual case. Aggregates non-pruned results to the non-contextual case. Logarithmically scaled.

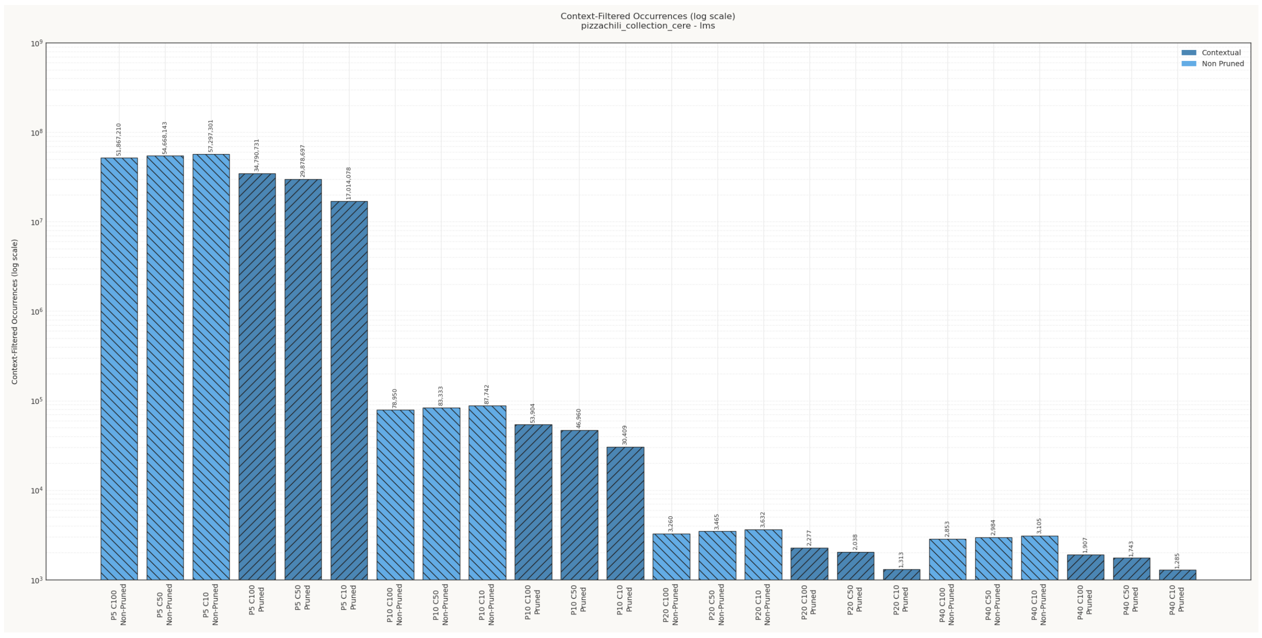

Figure B.59: Contextually filtered occurrences on LMS, for each pattern and context length and for both the pruned and non-pruned approaches. The non-contextual case is not shown since its value is 0. Logarithmically scaled.

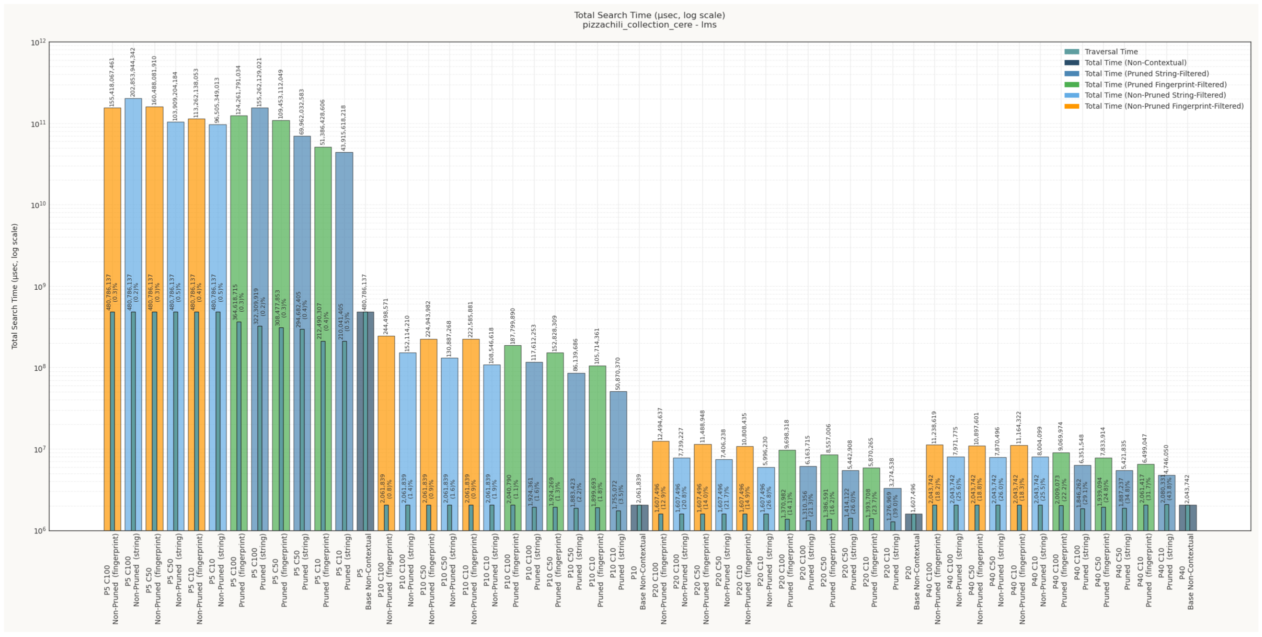

Figure B.60: Total search time, for each pattern and context length including the non-contextual case, for both the pruned and non-pruned approaches, on LMS. Additionally shows the traversal time on top of total search time for comparison, with a percentage annotation for the relationship between both values. Logarithmically scaled.

B.7.2. SLP

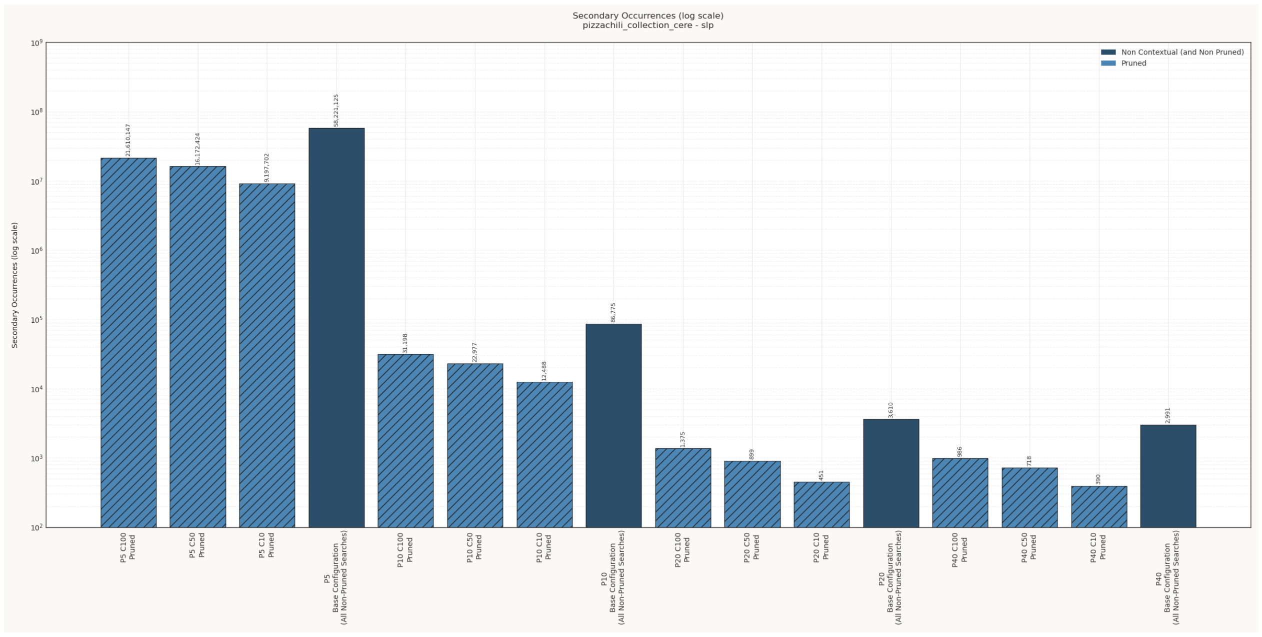

Figure B.61: Secondary occurrences found before filtering on SLP, for each pattern and context length including the non-contextual case. Aggregates non-pruned results to the non-contextual case. Logarithmically scaled.

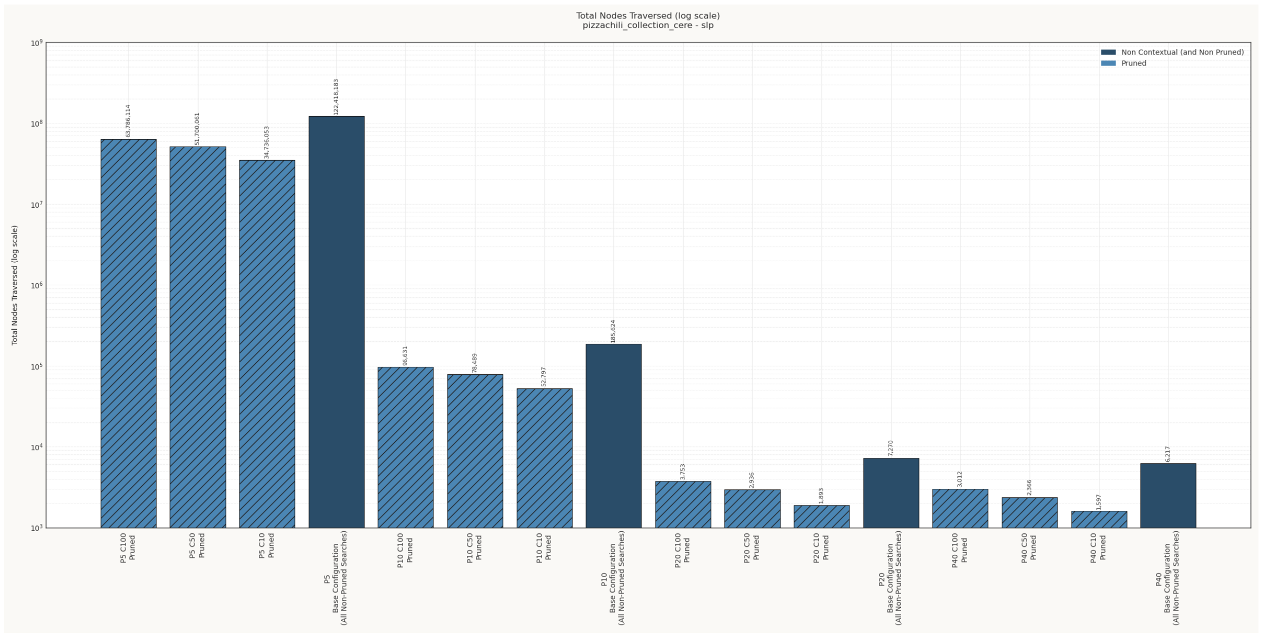

Figure B.62: Nodes traversed while computing secondary occurrences on SLP, for each pattern and context length including the non-contextual case. Aggregates non-pruned results to the non-contextual case. Logarithmically scaled.

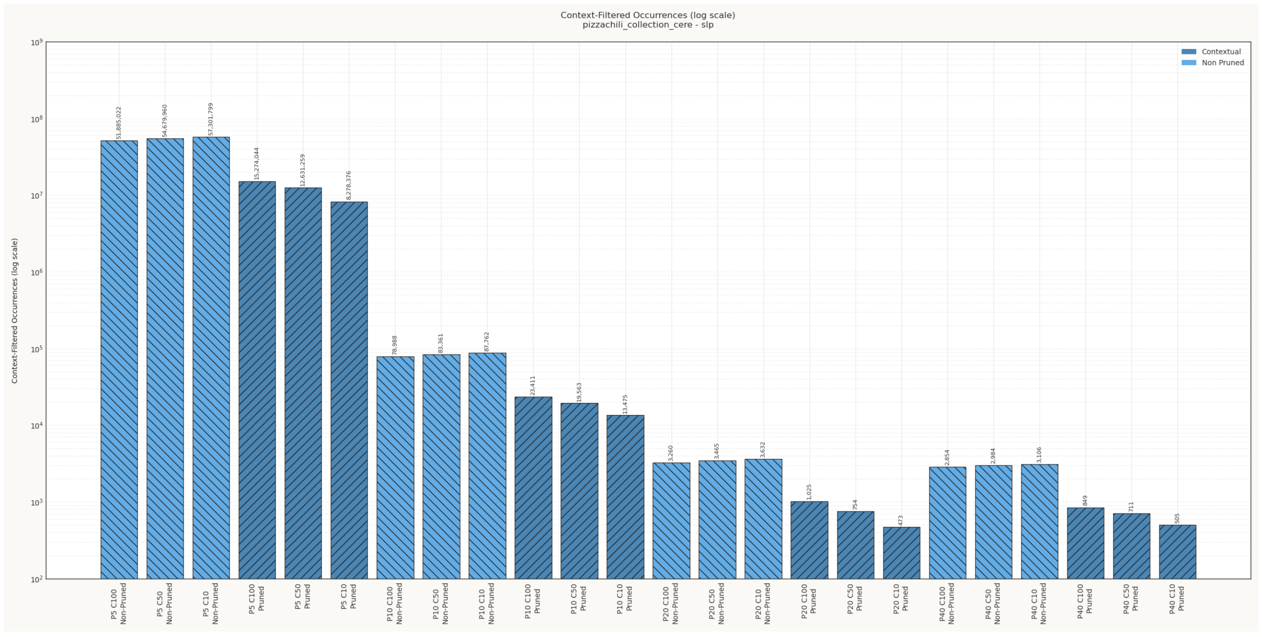
Figure B.63: Contextually filtered occurrences on SLP, for each pattern and context length and for both the pruned and non-pruned approaches. The non-contextual case is not shown since its value is 0. Logarithmically scaled.

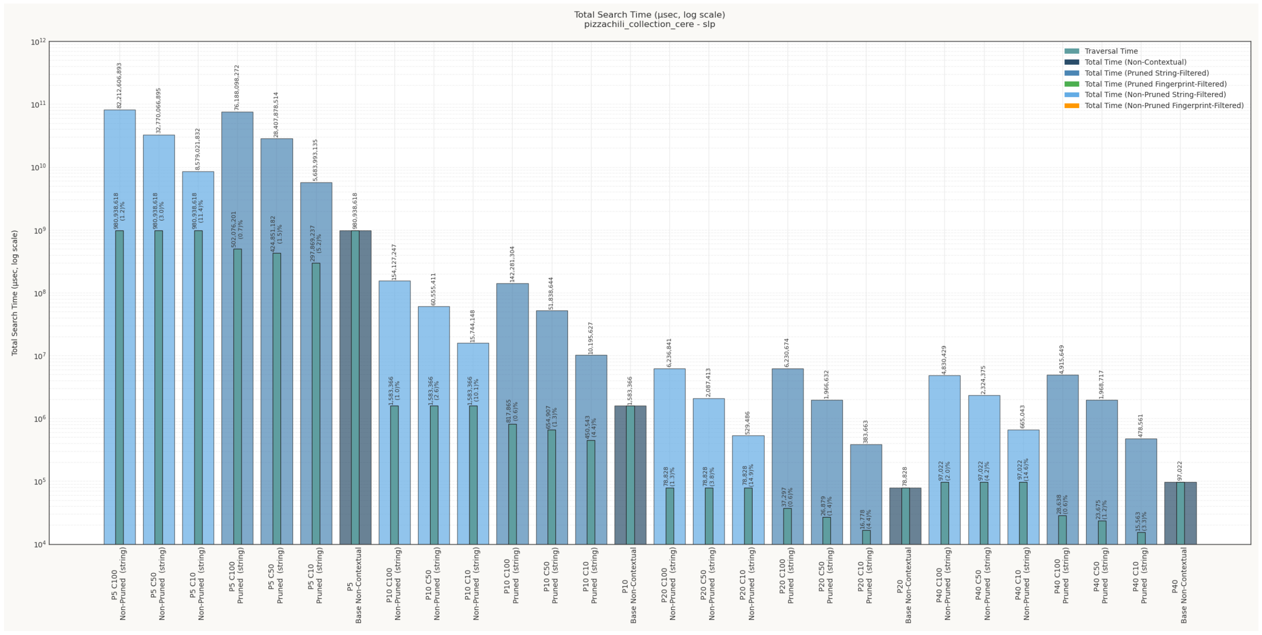
Figure B.64: Total search time, for each pattern and context length including the non-contextual case, for both the pruned and non-pruned approaches, on SLP. Additionally shows the traversal time on top of total search time for comparison, with a percentage annotation for the relationship between both values. Logarithmically scaled.

B.8. Total Time Comparisons

B.8.1. Pattern Length 5, Context Length 10

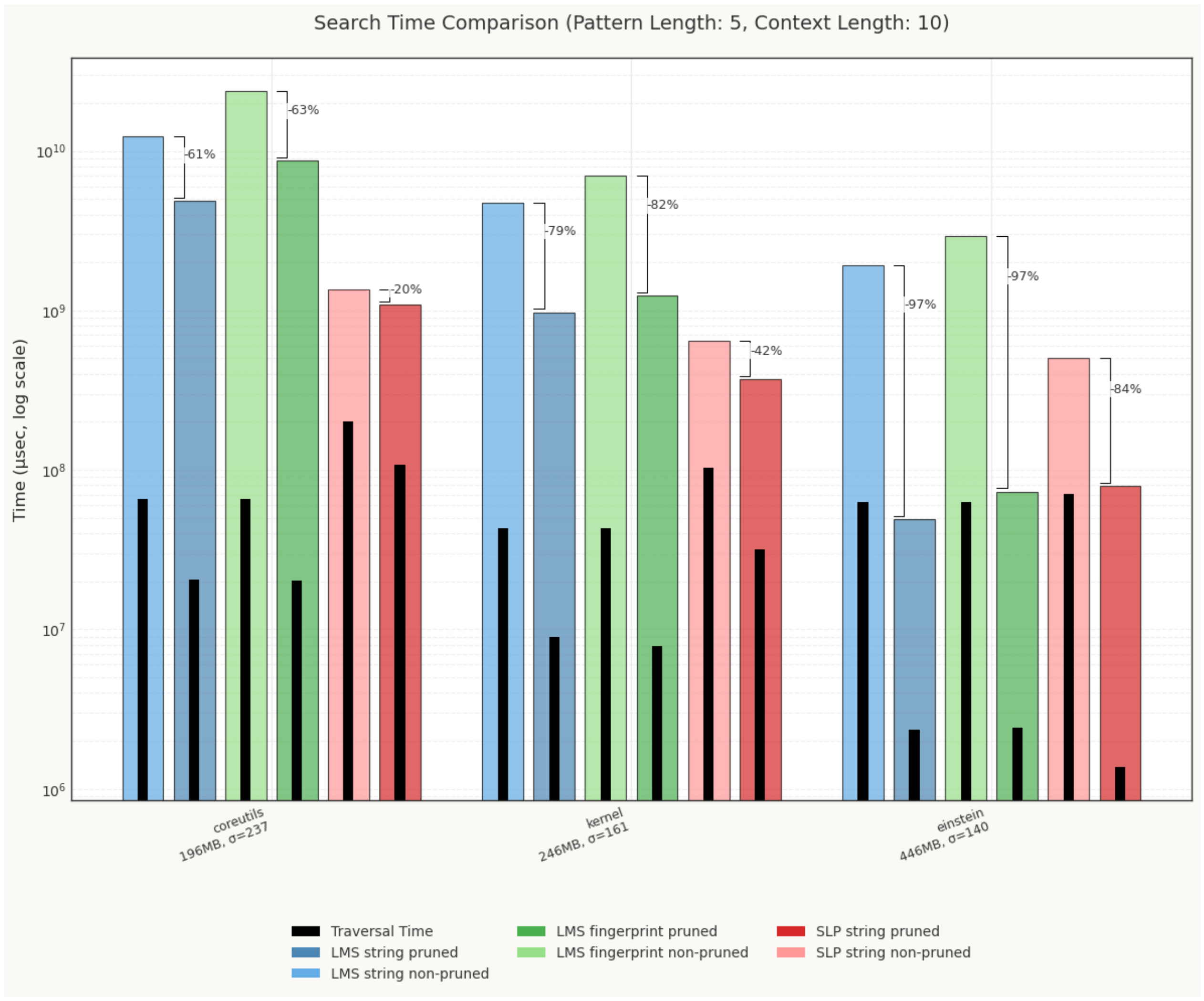

Figure B.65: Total search time, for each indexing program and filtering method, comparing both the pruned and non-pruned approaches on collections with large (>=100) alphabets. Additionally shows the traversal time on top of total search time for comparison. Displays the percentage of search time saved by pruning if applicable. Logarithmically scaled.

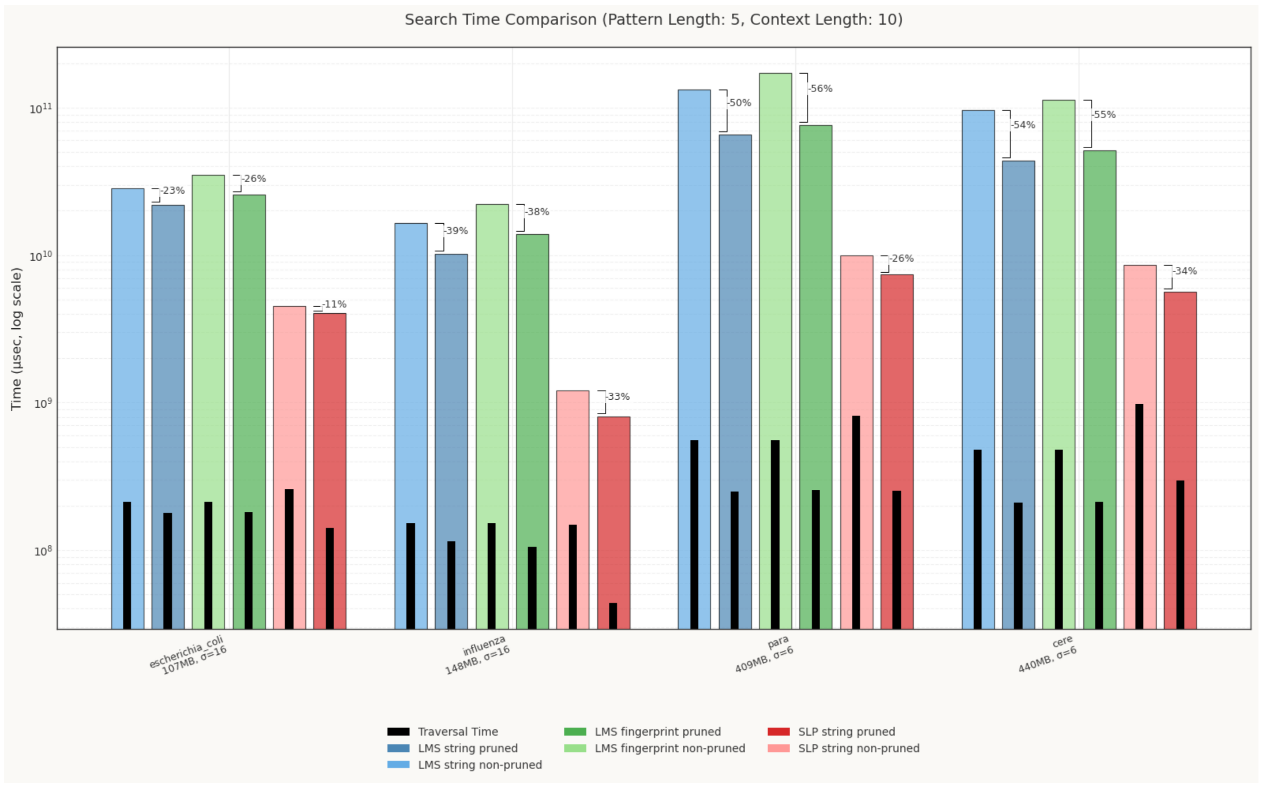

Figure B.66: Total search time, for each indexing program and filtering method, comparing both the pruned and non-pruned approaches on collections with small (<100) alphabets. Additionally shows the traversal time on top of total search time for comparison. Displays the percentage of search time saved by pruning if applicable. Logarithmically scaled.

B.8.2. Pattern Length 5, Context Length 50

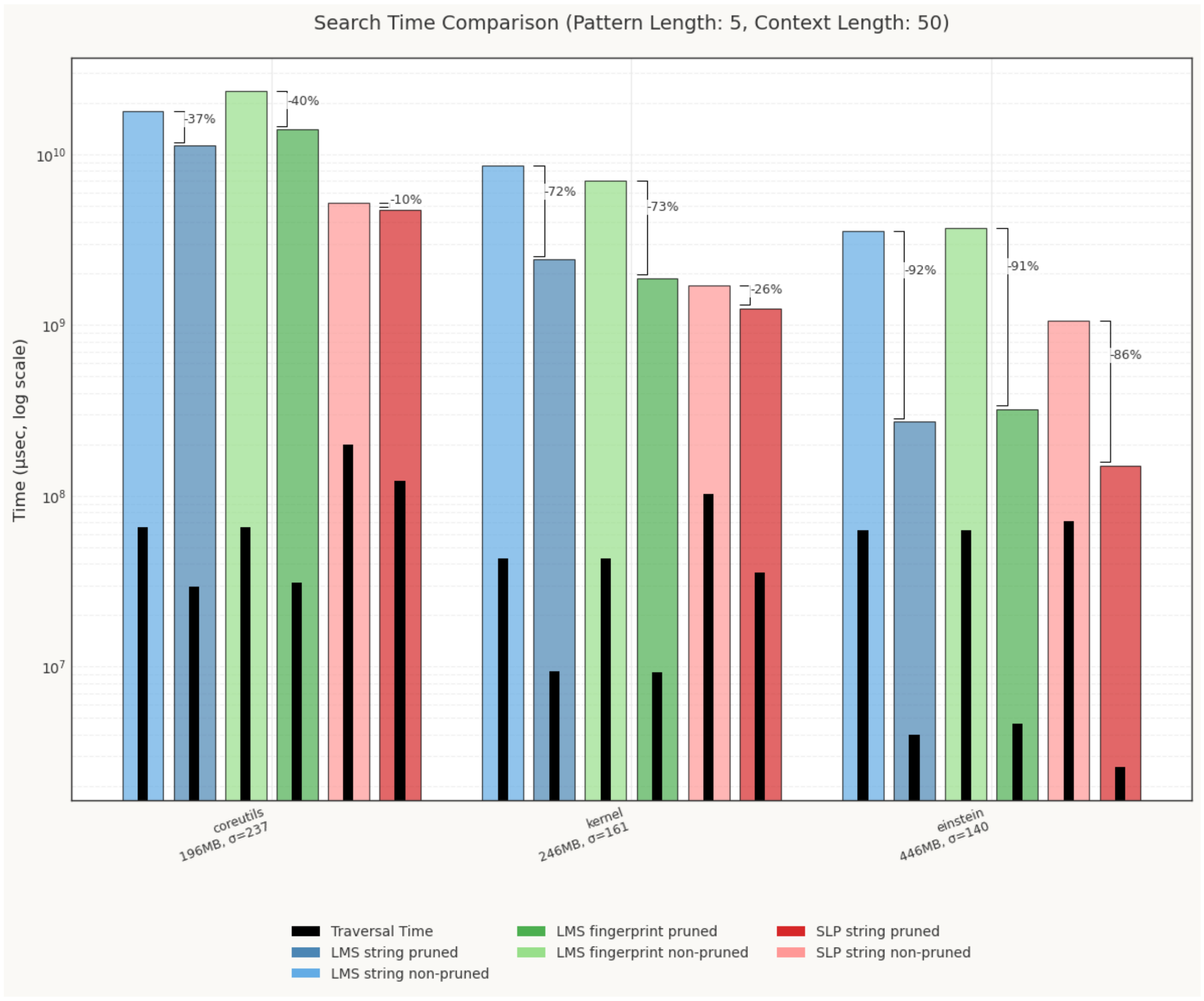

Figure B.67: Total search time, for each indexing program and filtering method, comparing both the pruned and non-pruned approaches on collections with large (>=100) alphabets. Additionally shows the traversal time on top of total search time for comparison. Displays the percentage of search time saved by pruning if applicable. Logarithmically scaled.

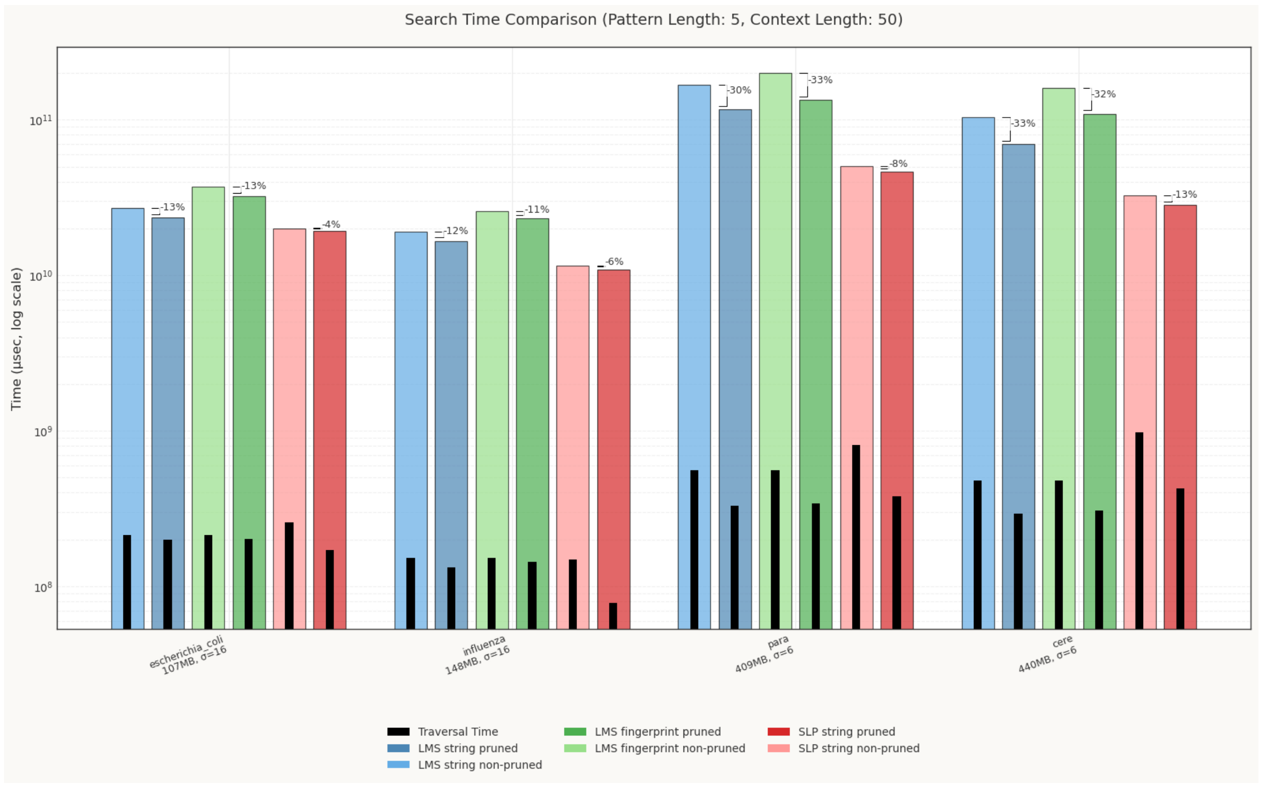

Figure B.68: Total search time, for each indexing program and filtering method, comparing both the pruned and non-pruned approaches on collections with small (<100) alphabets. Additionally shows the traversal time on top of total search time for comparison. Displays the percentage of search time saved by pruning if applicable. Logarithmically scaled.

B.8.3. Pattern Length 5, Context Length 100

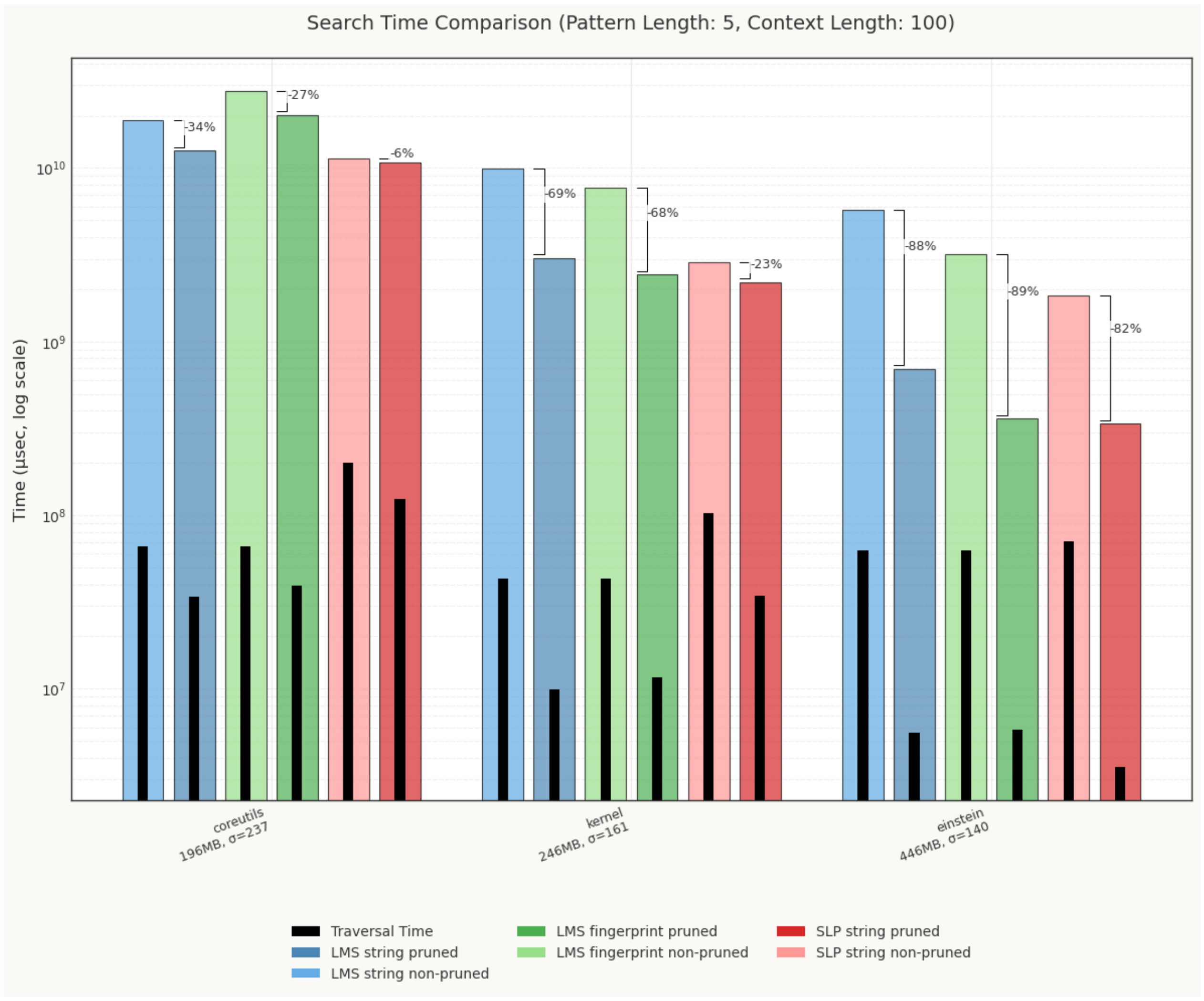

Figure B.69: Total search time, for each indexing program and filtering method, comparing both the pruned and non-pruned approaches on collections with large (>=100) alphabets. Additionally shows the traversal time on top of total search time for comparison. Displays the percentage of search time saved by pruning if applicable. Logarithmically scaled.

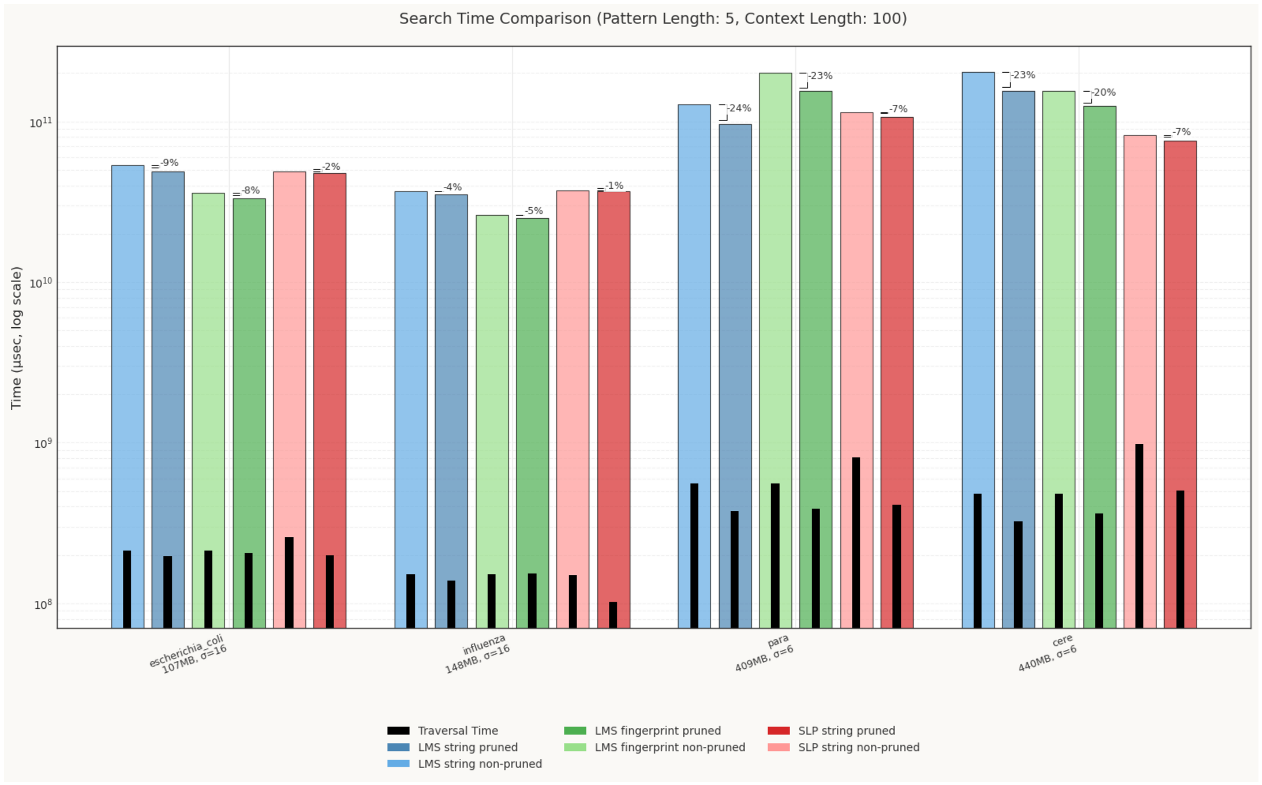

Figure B.70: Total search time, for each indexing program and filtering method, comparing both the pruned and non-pruned approaches on collections with small (<100) alphabets. Additionally shows the traversal time on top of total search time for comparison. Displays the percentage of search time saved by pruning if applicable. Logarithmically scaled.

B.8.4. Pattern Length 5, Non-Contextual

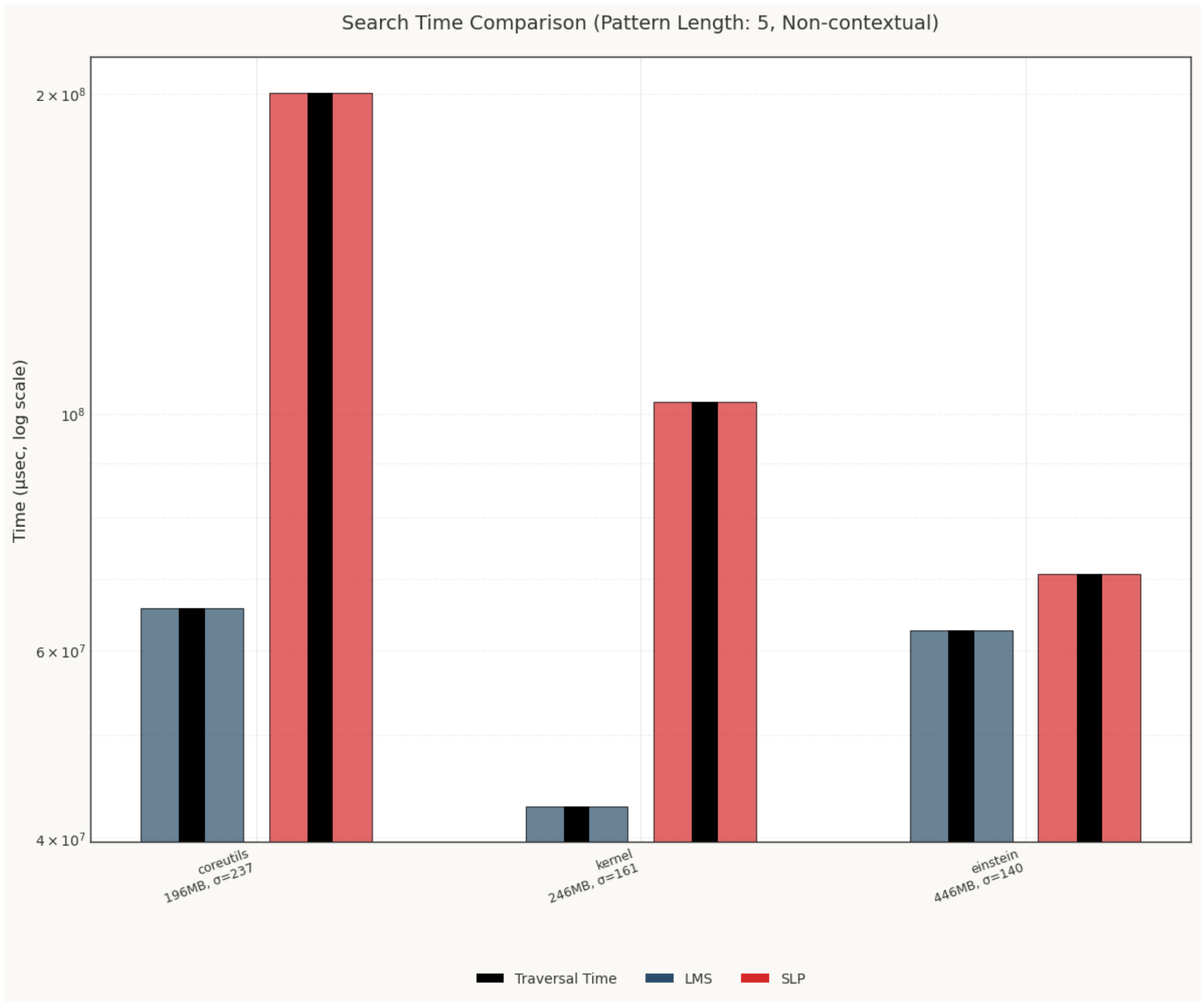

Figure B.71: Total search time, for each indexing program and filtering method, comparing both the pruned and non-pruned approaches on collections with large (>=100) alphabets. Additionally shows the traversal time on top of total search time for comparison. Displays the percentage of search time saved by pruning if applicable. Logarithmically scaled.

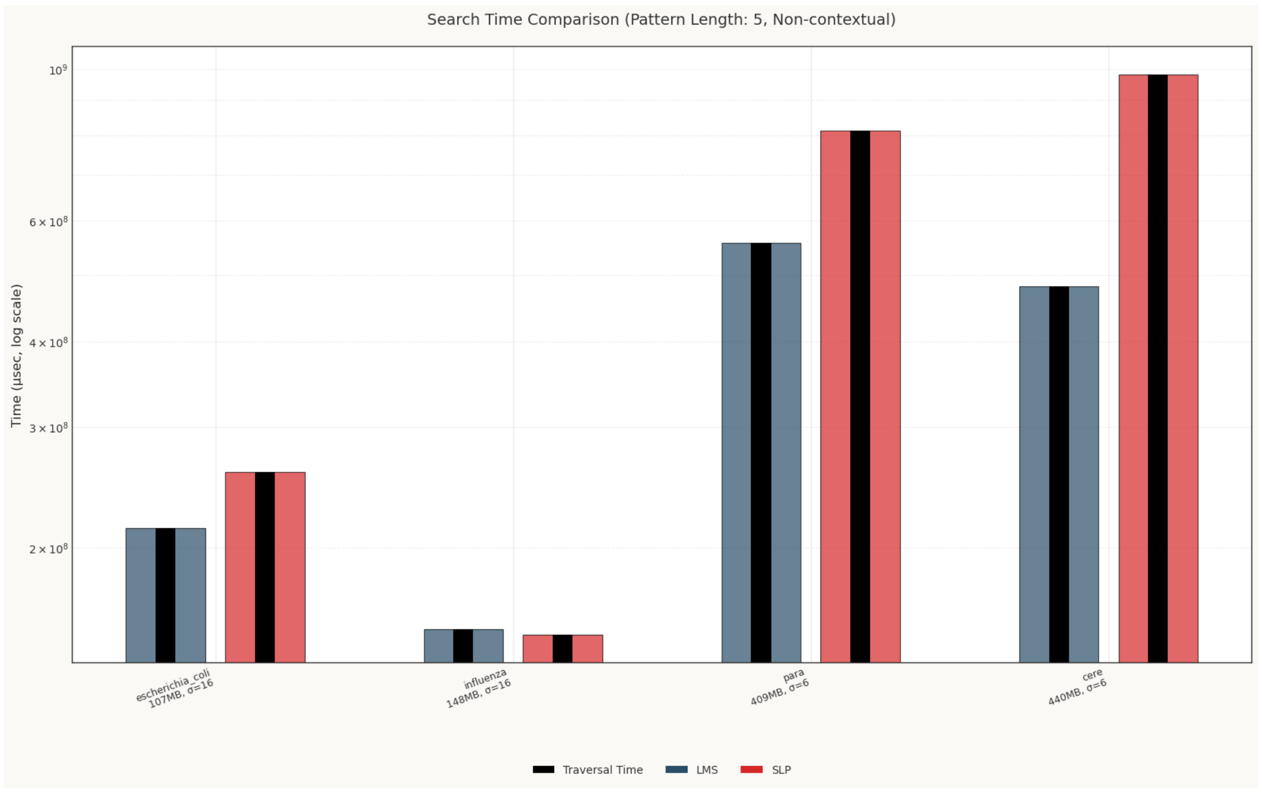

Figure B.72: Total search time, for each indexing program and filtering method, comparing both the pruned and non-pruned approaches on collections with small (<100) alphabets. Additionally shows the traversal time on top of total search time for comparison. Displays the percentage of search time saved by pruning if applicable. Logarithmically scaled.

B.8.5. Pattern Length 10, Context Length 10

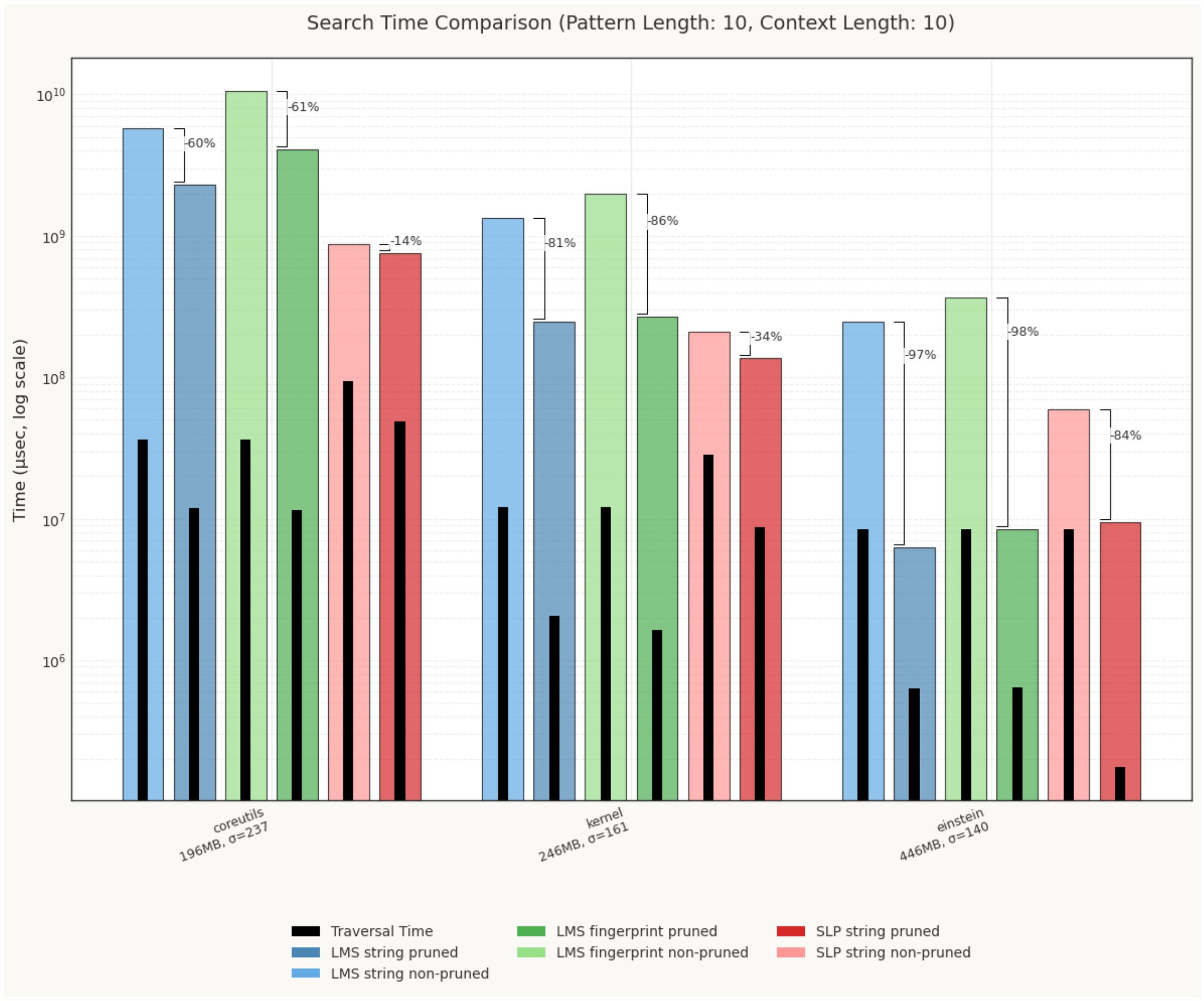

Figure B.73: Total search time, for each indexing program and filtering method, comparing both the pruned and non-pruned approaches on collections with large (>=100) alphabets. Additionally shows the traversal time on top of total search time for comparison. Displays the percentage of search time saved by pruning if applicable. Logarithmically scaled.

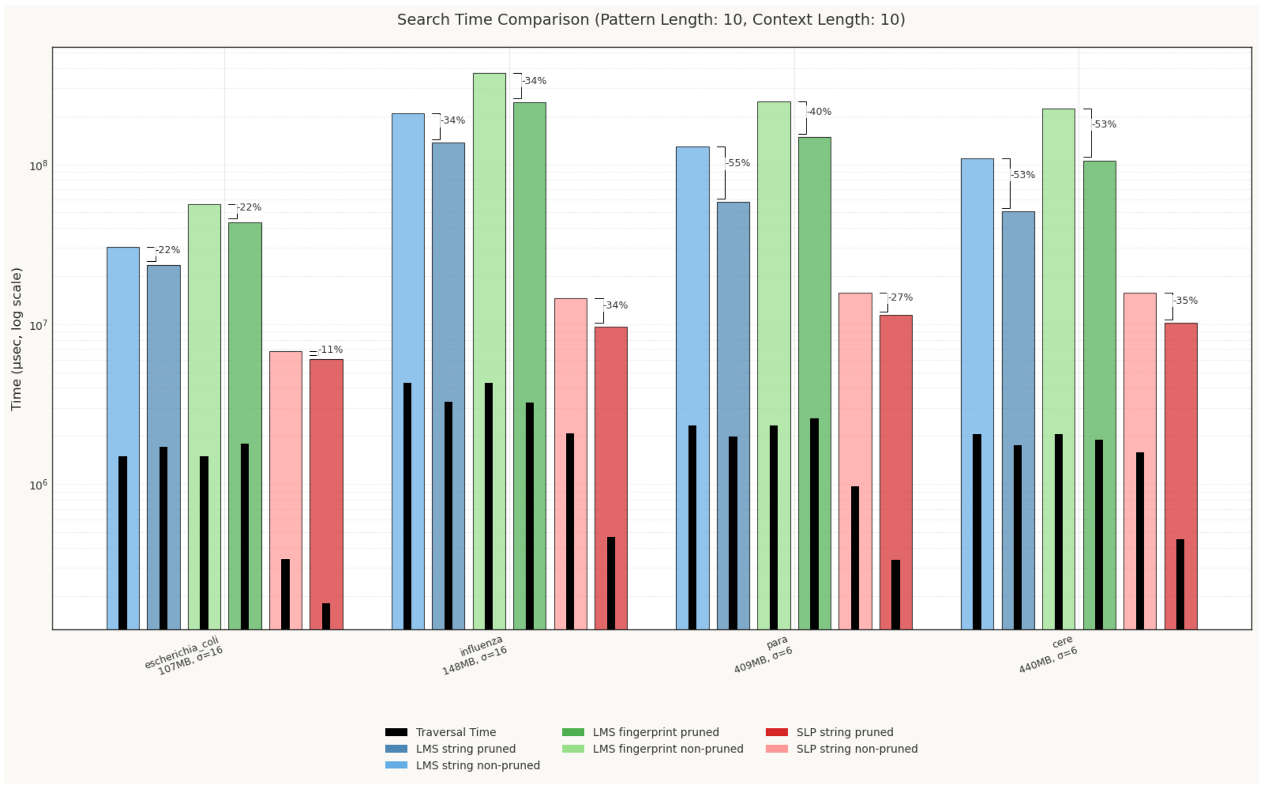

Figure B.74: Total search time, for each indexing program and filtering method, comparing both the pruned and non-pruned approaches on collections with small (<100) alphabets. Additionally shows the traversal time on top of total search time for comparison. Displays the percentage of search time saved by pruning if applicable. Logarithmically scaled.

B.8.6. Pattern Length 10, Context Length 50

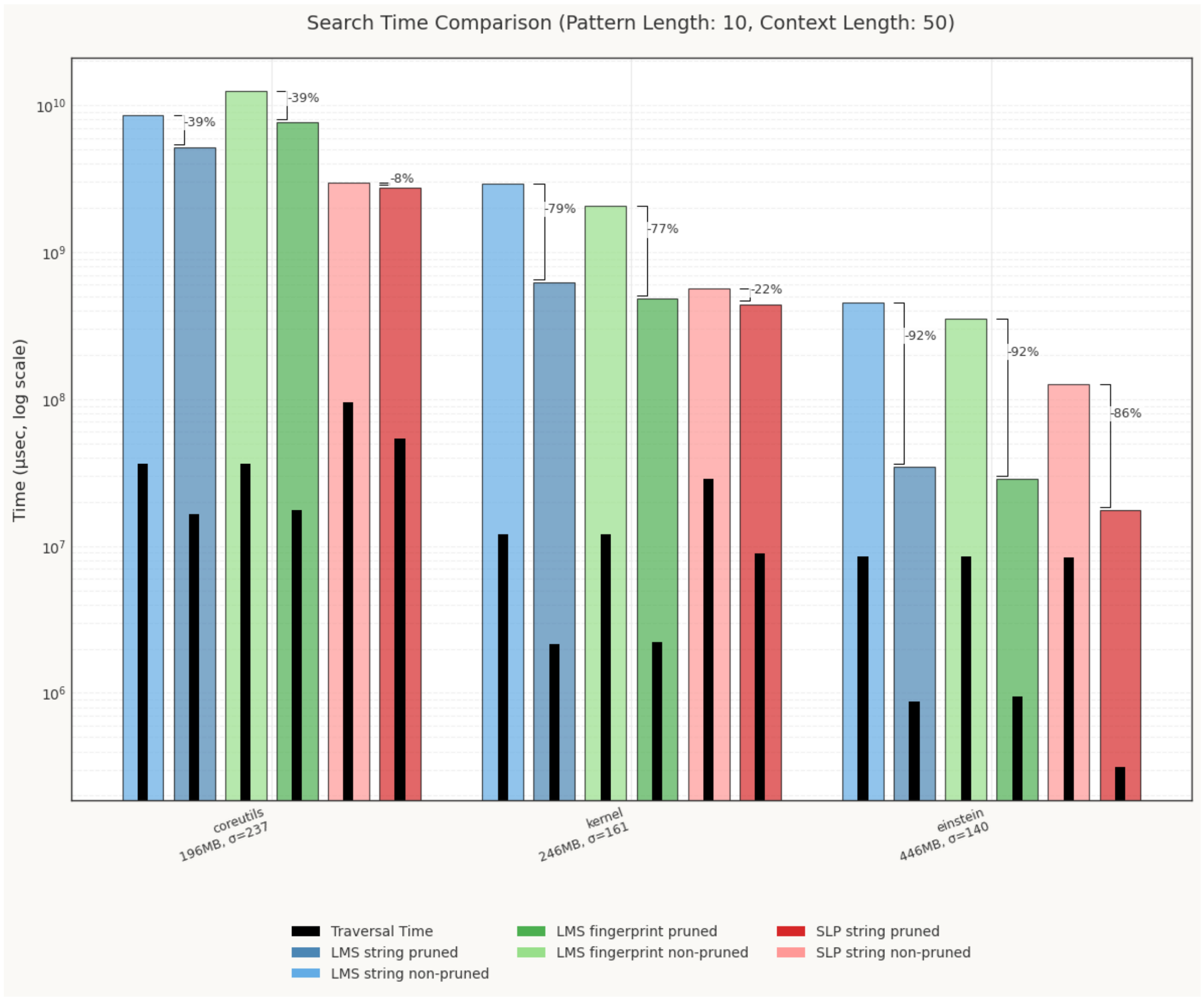

Figure B.75: Total search time, for each indexing program and filtering method, comparing both the pruned and non-pruned approaches on collections with large (>=100) alphabets. Additionally shows the traversal time on top of total search time for comparison. Displays the percentage of search time saved by pruning if applicable. Logarithmically scaled.

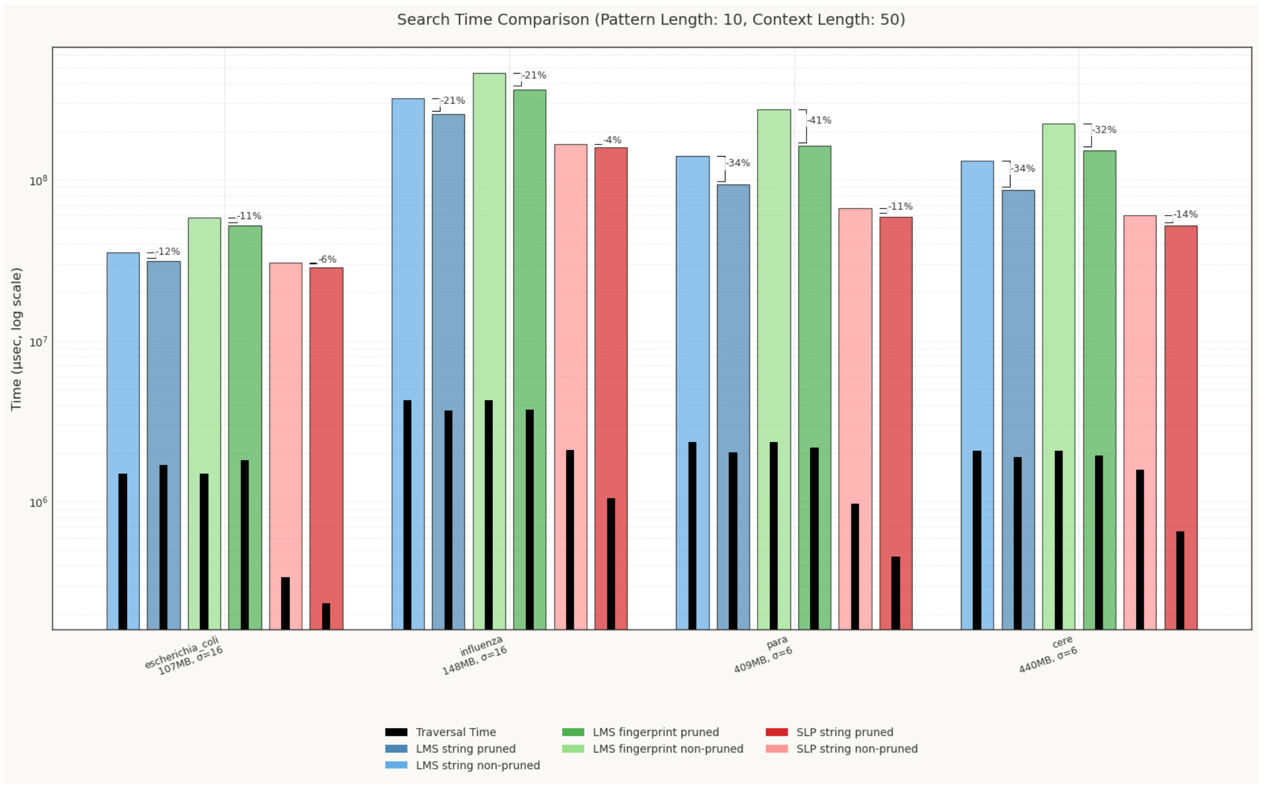

Figure B.76: Total search time, for each indexing program and filtering method, comparing both the pruned and non-pruned approaches on collections with small (<100) alphabets. Additionally shows the traversal time on top of total search time for comparison. Displays the percentage of search time saved by pruning if applicable. Logarithmically scaled.

B.8.7. Pattern Length 10, Context Length 100

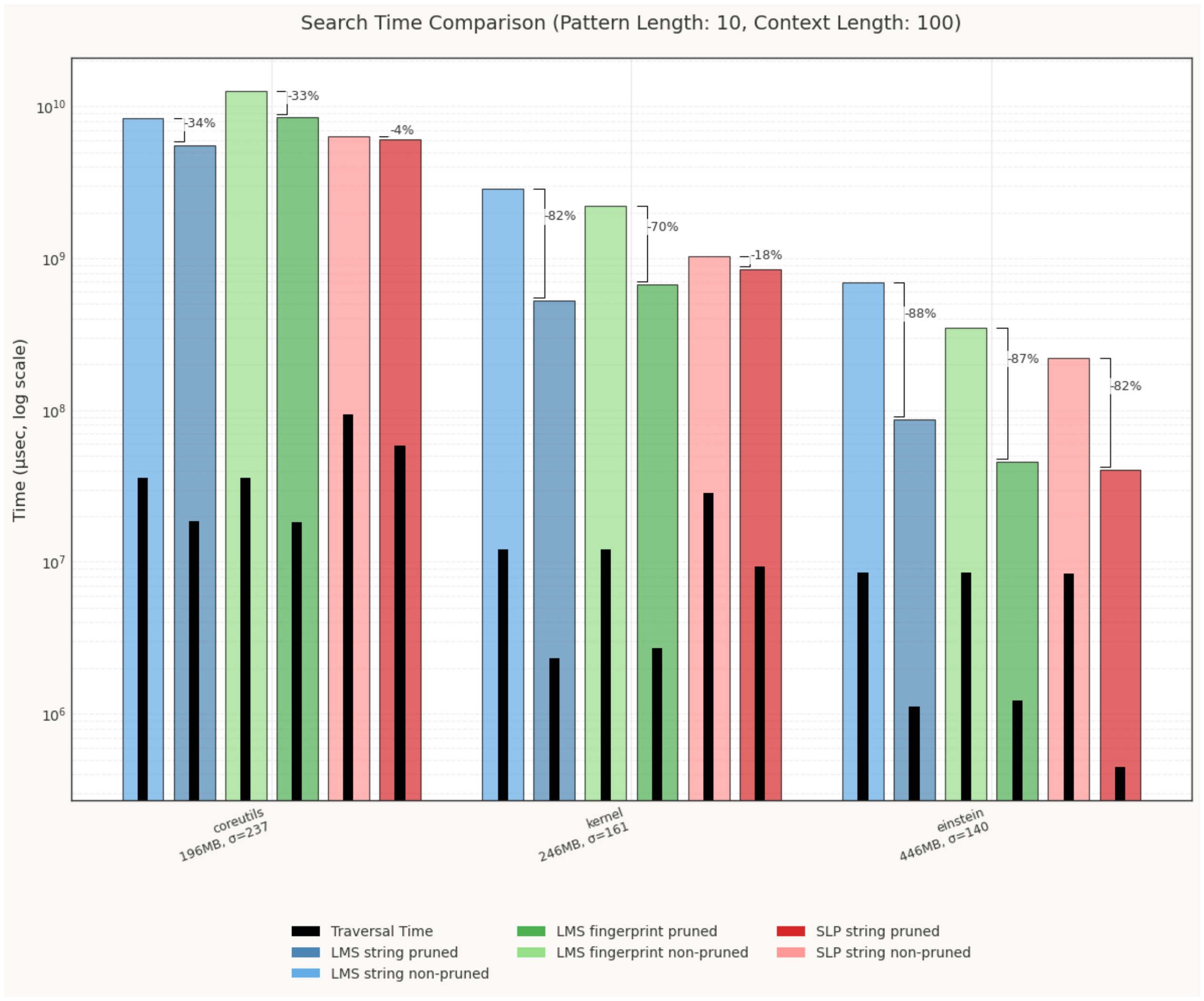

Figure B.77: Total search time, for each indexing program and filtering method, comparing both the pruned and non-pruned approaches on collections with large (>=100) alphabets. Additionally shows the traversal time on top of total search time for comparison. Displays the percentage of search time saved by pruning if applicable. Logarithmically scaled.

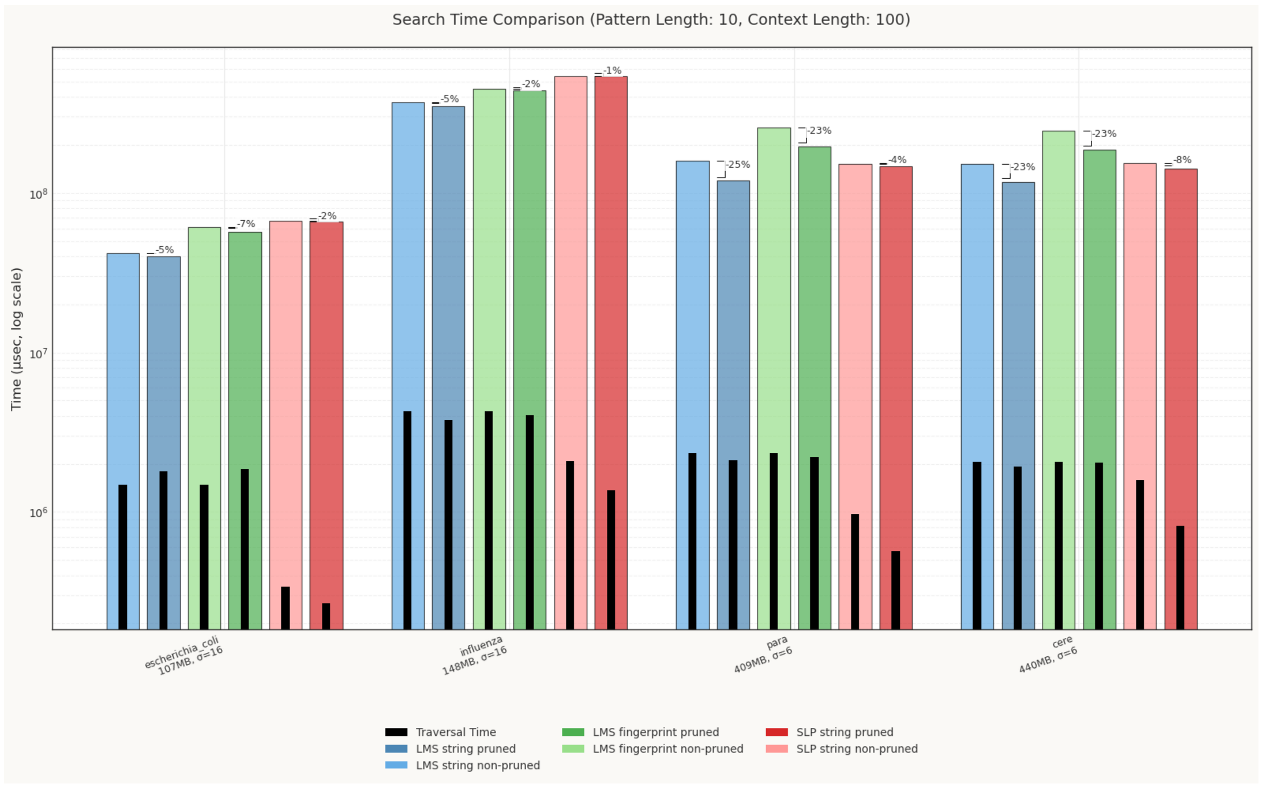

Figure B.78: Total search time, for each indexing program and filtering method, comparing both the pruned and non-pruned approaches on collections with small (<100) alphabets. Additionally shows the traversal time on top of total search time for comparison. Displays the percentage of search time saved by pruning if applicable. Logarithmically scaled.

B.8.8. Pattern Length 10, Non-Contextual

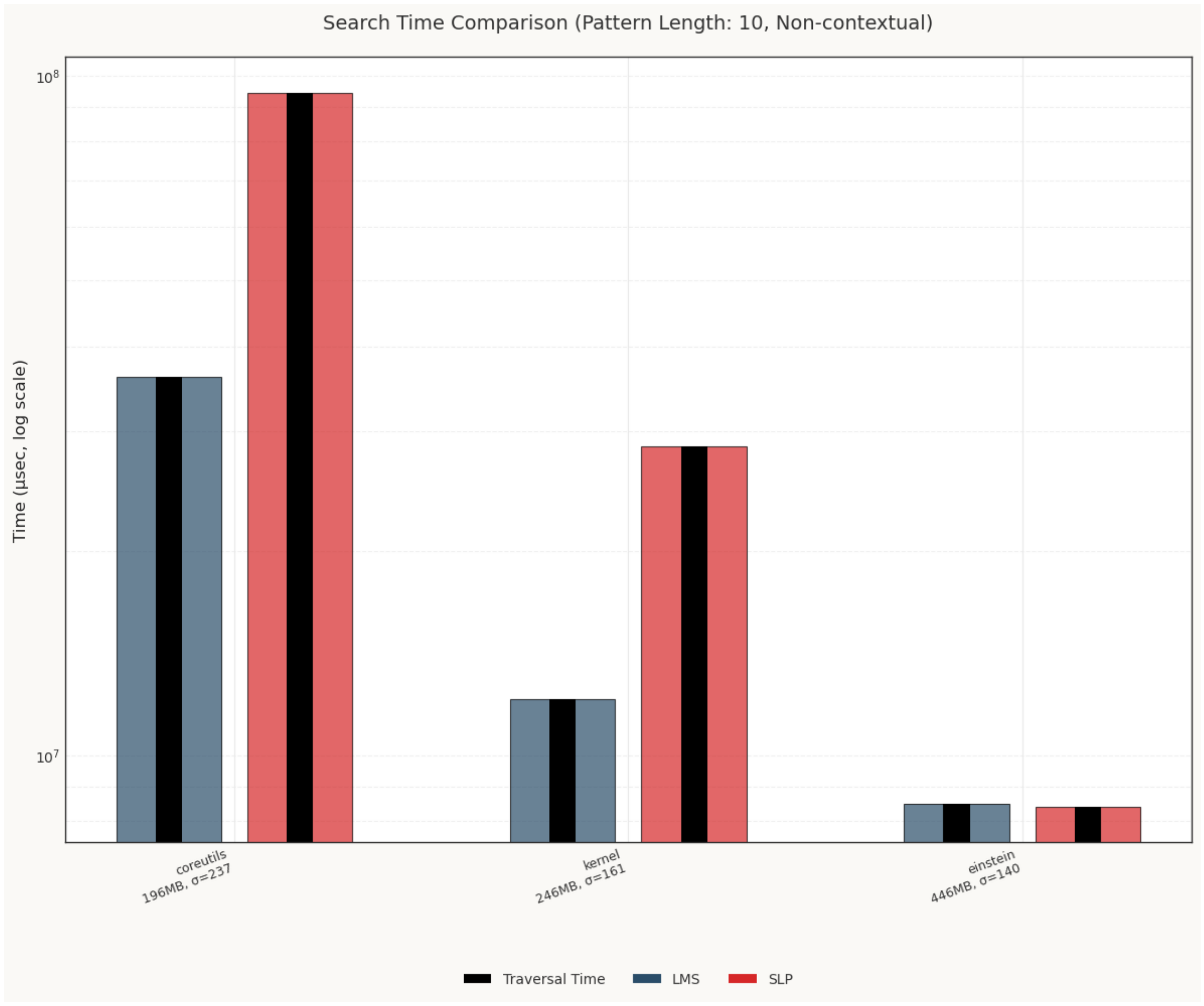

Figure B.79: Total search time, for each indexing program and filtering method, comparing both the pruned and non-pruned approaches on collections with large (>=100) alphabets. Additionally shows the traversal time on top of total search time for comparison. Displays the percentage of search time saved by pruning if applicable. Logarithmically scaled.

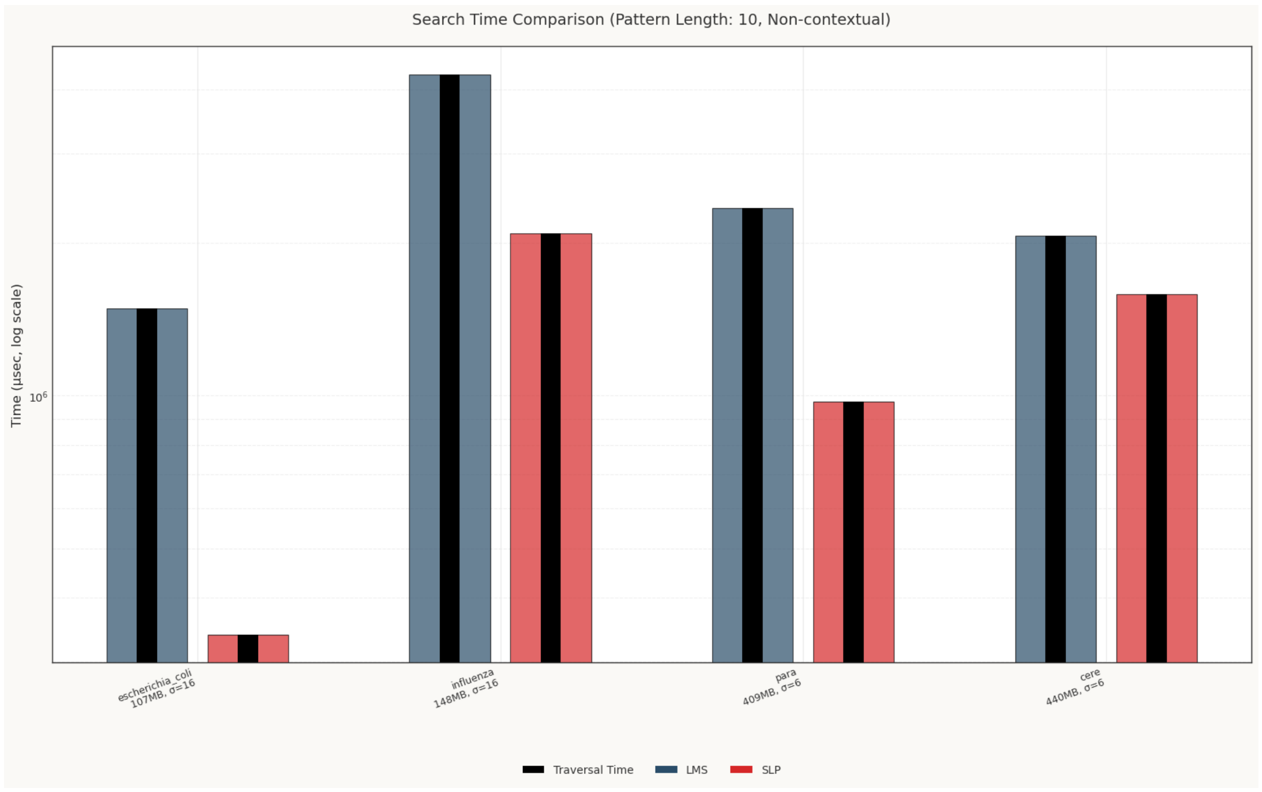

Figure B.80: Total search time, for each indexing program and filtering method, comparing both the pruned and non-pruned approaches on collections with small (<100) alphabets. Additionally shows the traversal time on top of total search time for comparison. Displays the percentage of search time saved by pruning if applicable. Logarithmically scaled.

B.8.9. Pattern Length 20, Context Length 10

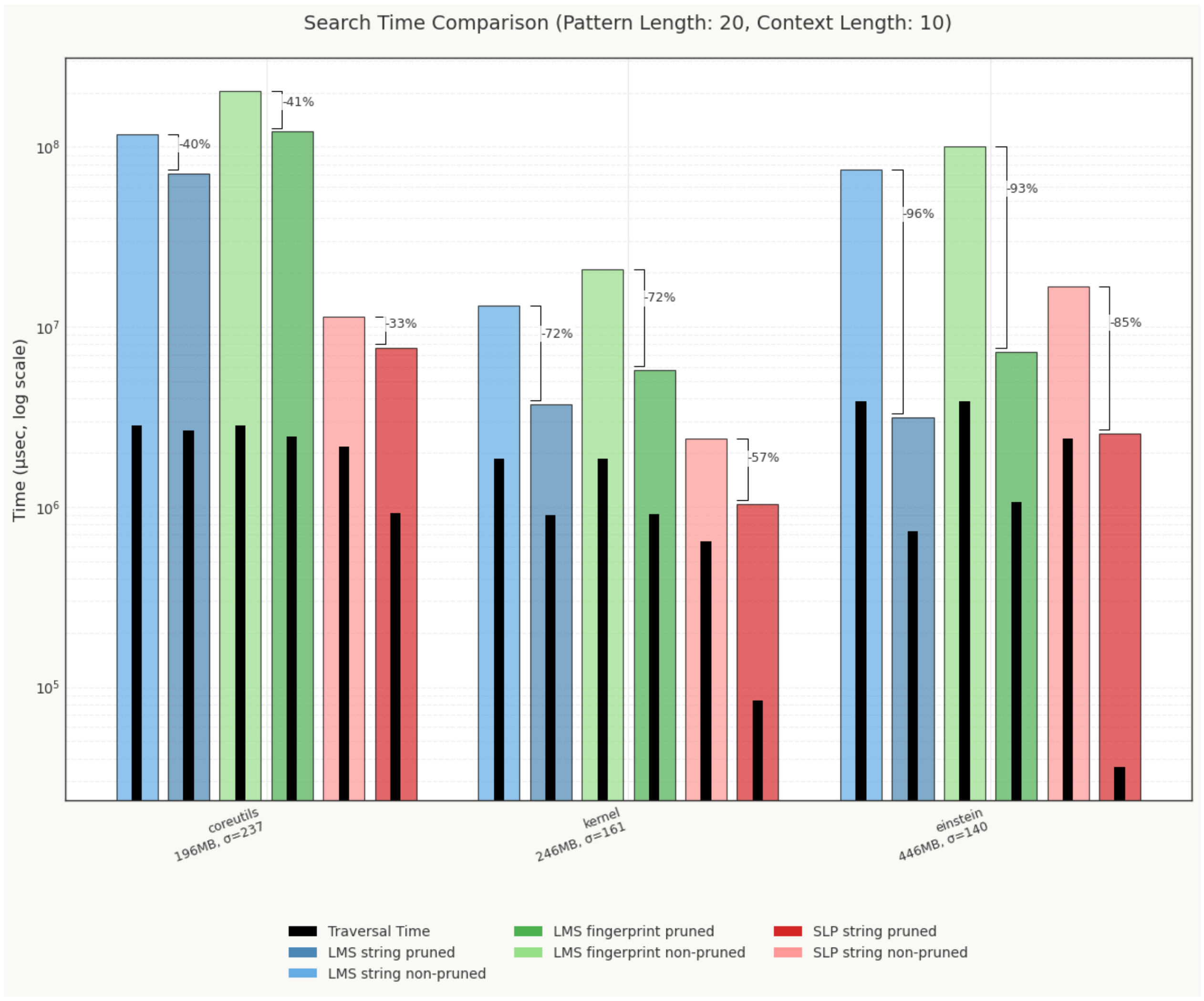

Figure B.81: Total search time, for each indexing program and filtering method, comparing both the pruned and non-pruned approaches on collections with large (>=100) alphabets. Additionally shows the traversal time on top of total search time for comparison. Displays the percentage of search time saved by pruning if applicable. Logarithmically scaled.

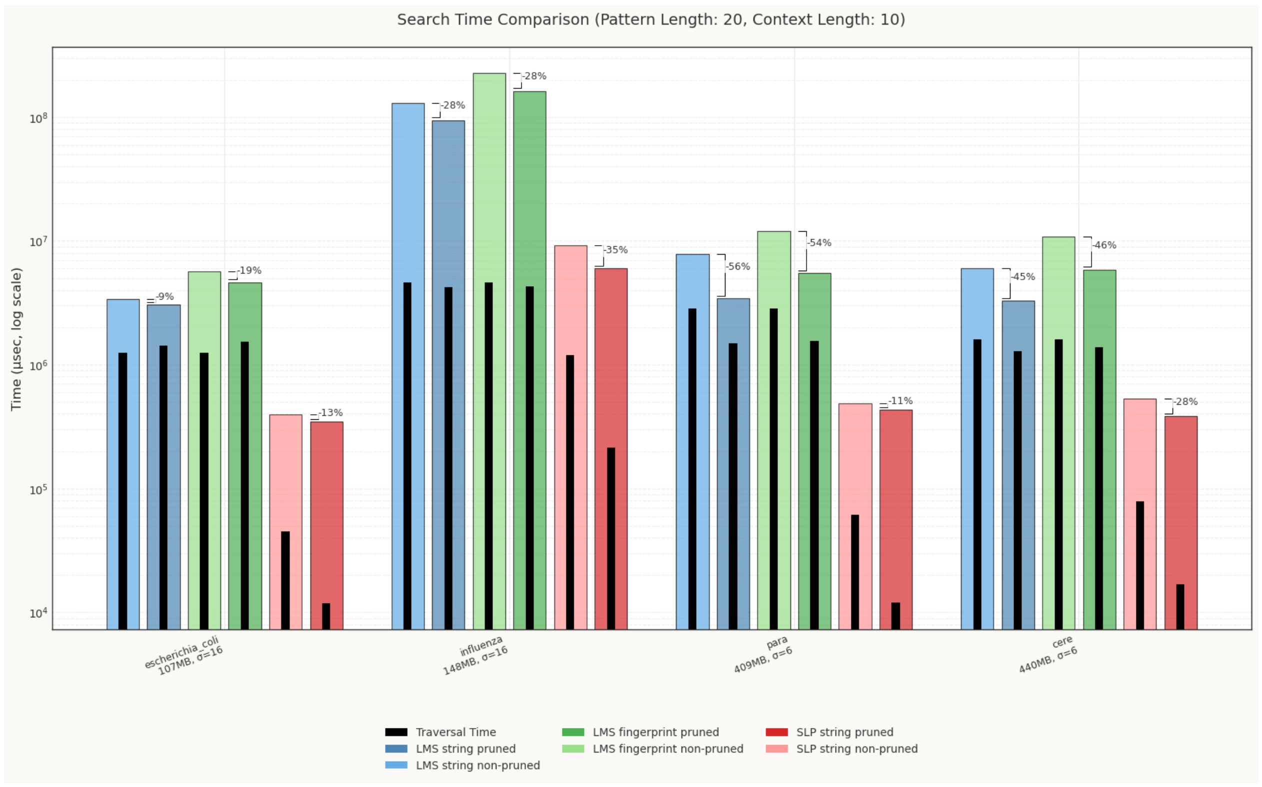

Figure B.82: Total search time, for each indexing program and filtering method, comparing both the pruned and non-pruned approaches on collections with small (<100) alphabets. Additionally shows the traversal time on top of total search time for comparison. Displays the percentage of search time saved by pruning if applicable. Logarithmically scaled.

B.8.10. Pattern Length 20, Context Length 50

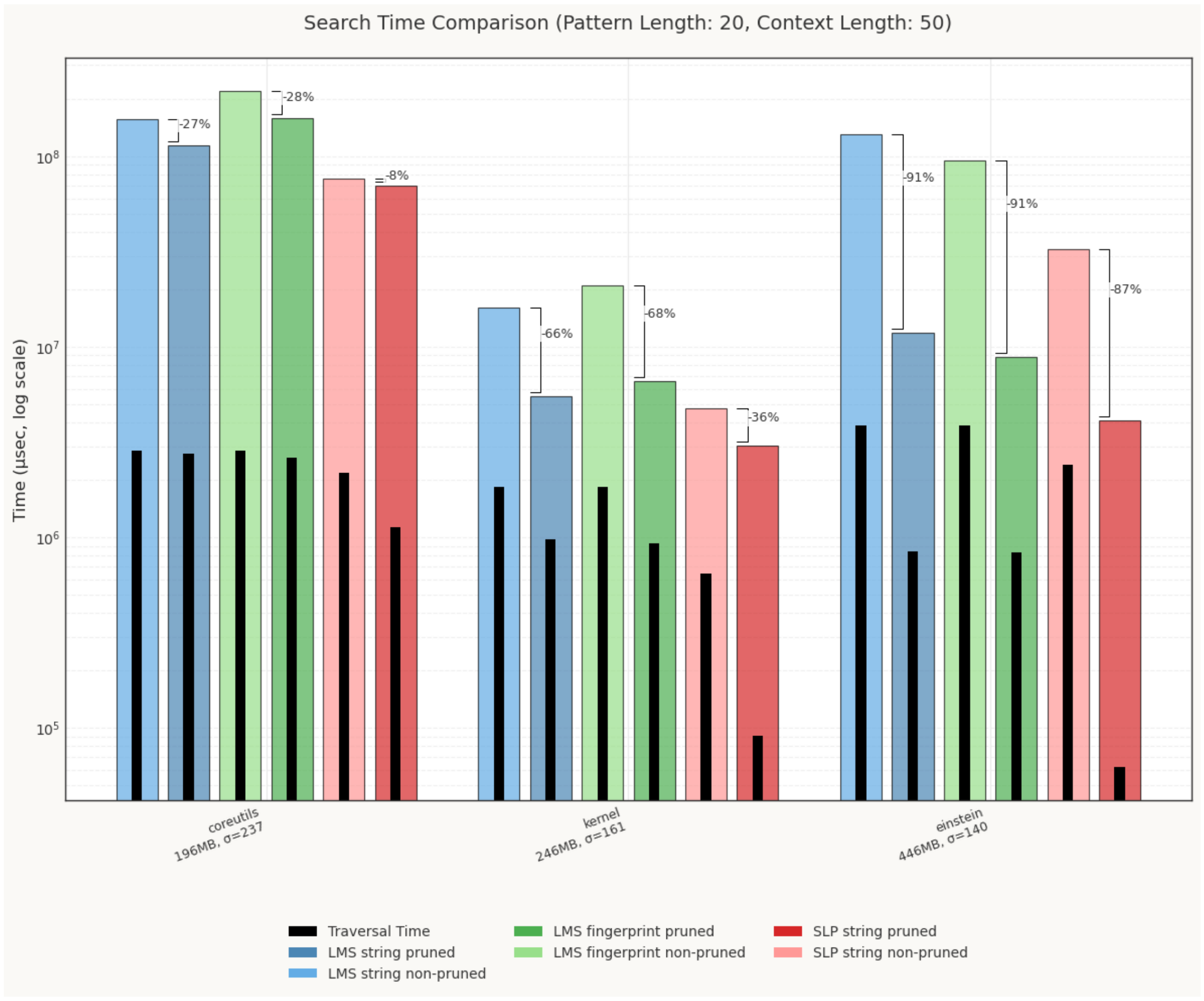

Figure B.83: Total search time, for each indexing program and filtering method, comparing both the pruned and non-pruned approaches on collections with large (>=100) alphabets. Additionally shows the traversal time on top of total search time for comparison. Displays the percentage of search time saved by pruning if applicable. Logarithmically scaled.

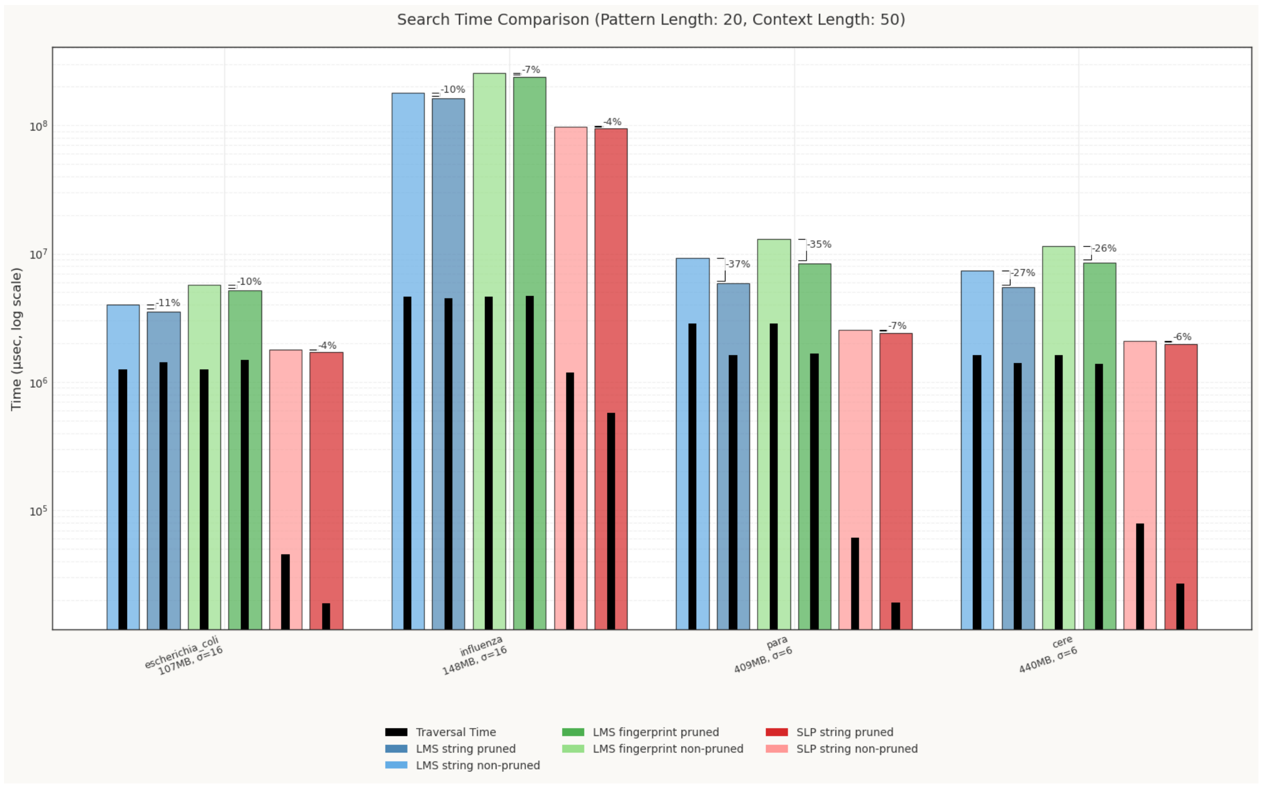

Figure B.84: Total search time, for each indexing program and filtering method, comparing both the pruned and non-pruned approaches on collections with small (<100) alphabets. Additionally shows the traversal time on top of total search time for comparison. Displays the percentage of search time saved by pruning if applicable. Logarithmically scaled.

B.8.11. Pattern Length 20, Context Length 100

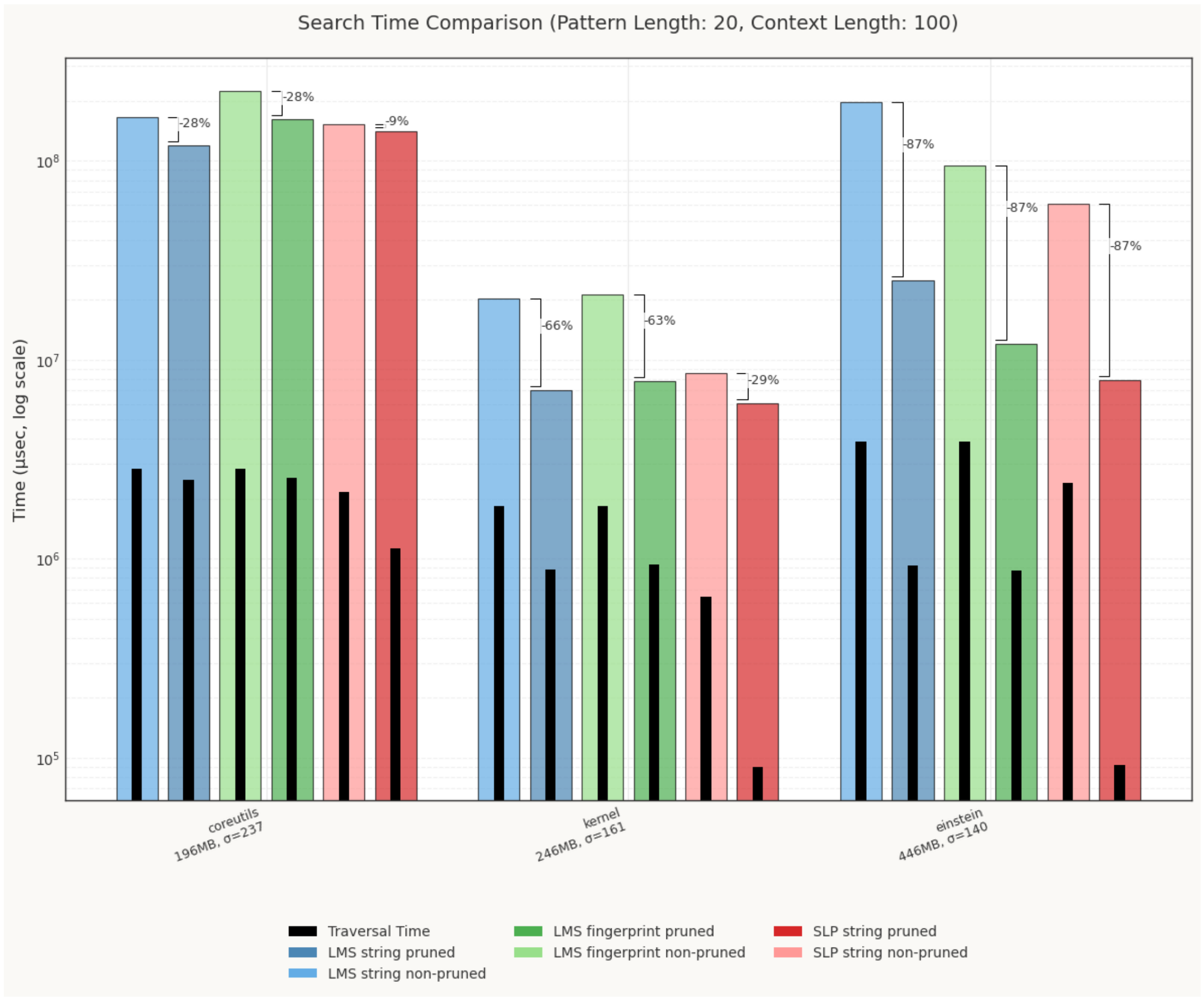

Figure B.85: Total search time, for each indexing program and filtering method, comparing both the pruned and non-pruned approaches on collections with large (>=100) alphabets. Additionally shows the traversal time on top of total search time for comparison. Displays the percentage of search time saved by pruning if applicable. Logarithmically scaled.

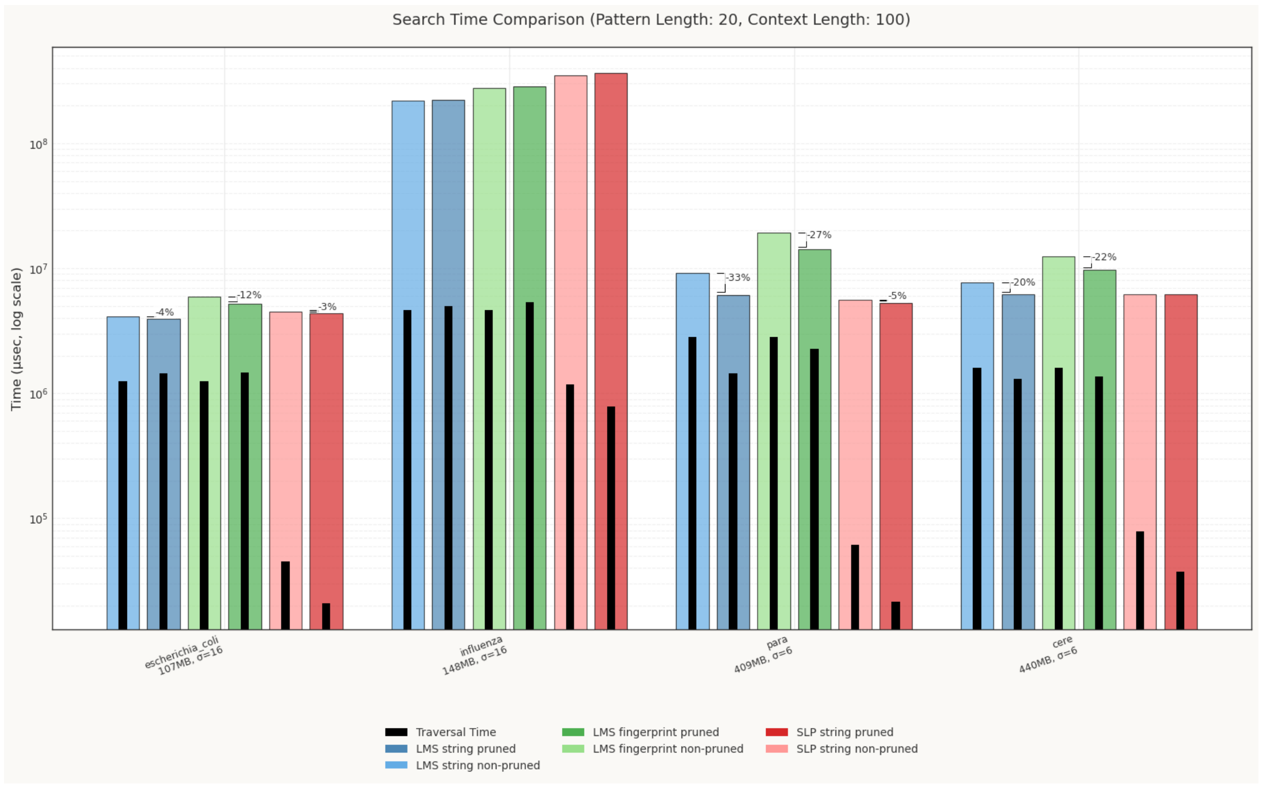

Figure B.86: Total search time, for each indexing program and filtering method, comparing both the pruned and non-pruned approaches on collections with small (<100) alphabets. Additionally shows the traversal time on top of total search time for comparison. Displays the percentage of search time saved by pruning if applicable. Logarithmically scaled.

B.8.12. Pattern Length 20, Non-Contextual

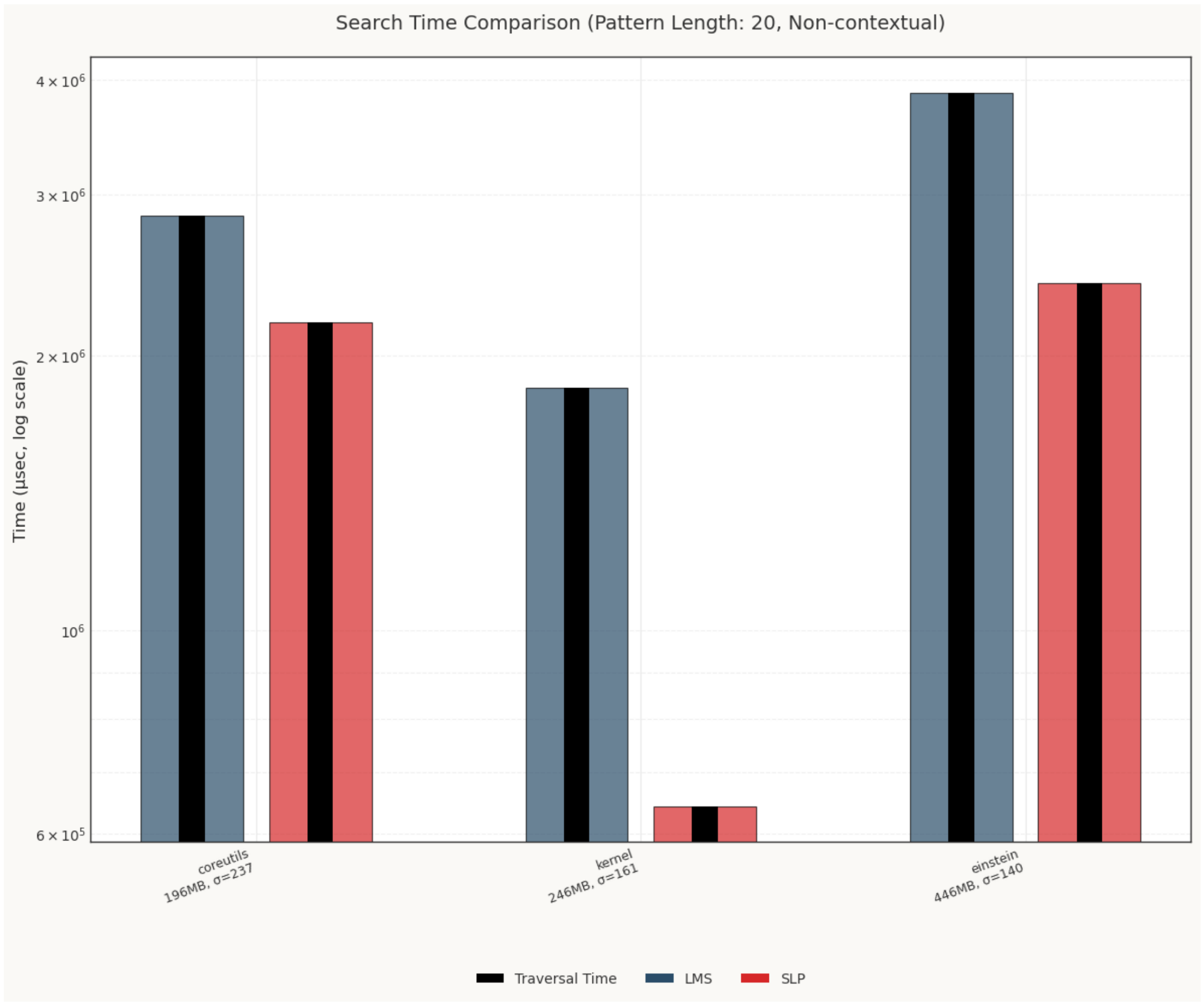

Figure B.87: Total search time, for each indexing program and filtering method, comparing both the pruned and non-pruned approaches on collections with large (>=100) alphabets. Additionally shows the traversal time on top of total search time for comparison. Displays the percentage of search time saved by pruning if applicable. Logarithmically scaled.

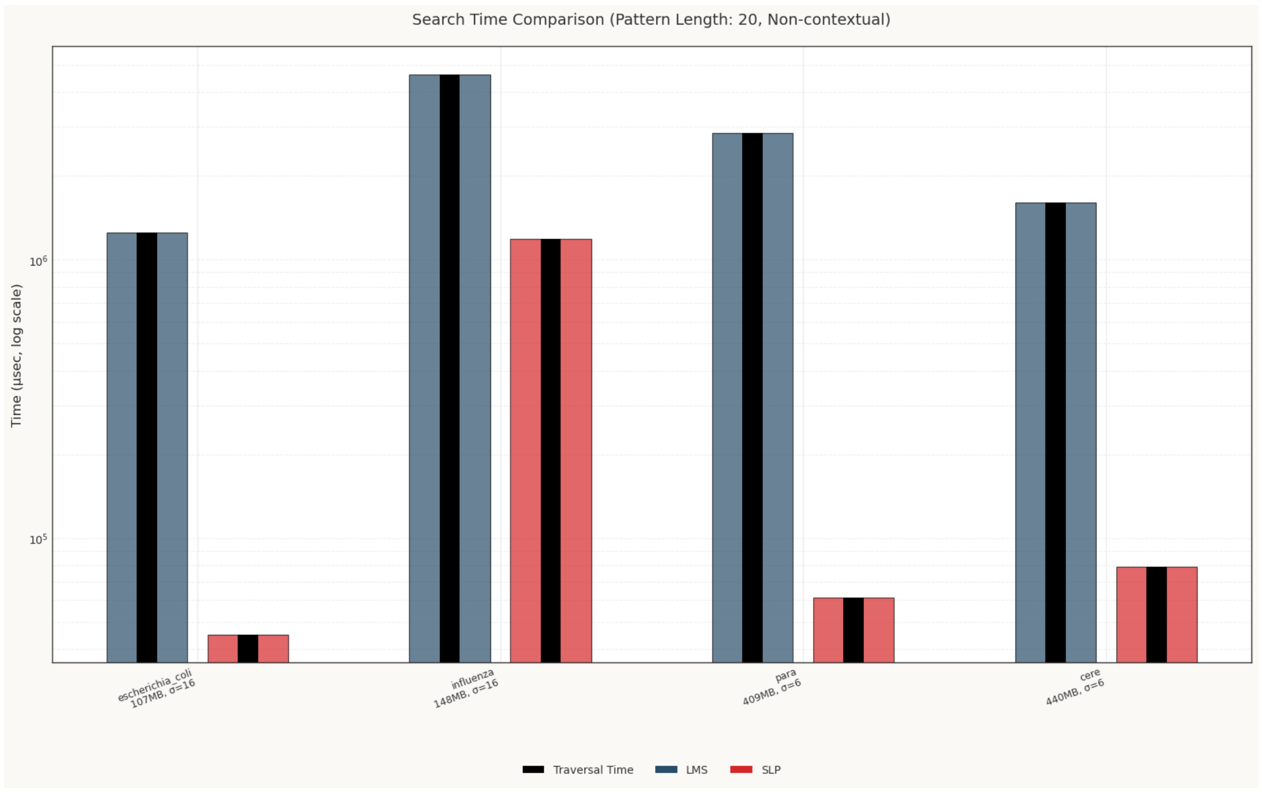

Figure B.88: Total search time, for each indexing program and filtering method, comparing both the pruned and non-pruned approaches on collections with small (<100) alphabets. Additionally shows the traversal time on top of total search time for comparison. Displays the percentage of search time saved by pruning if applicable. Logarithmically scaled.

B.8.13. Pattern Length 40, Context Length 10

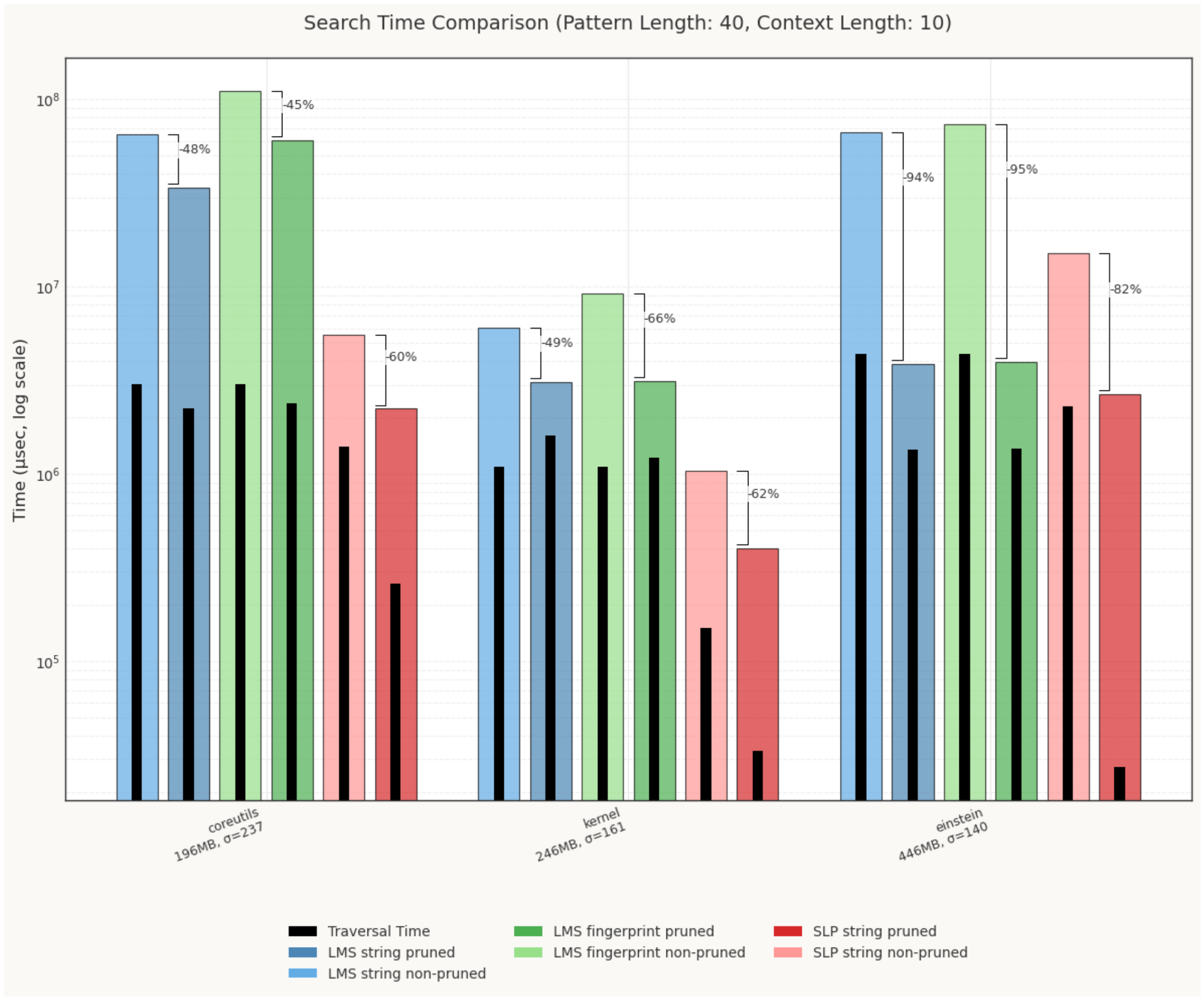

Figure B.89: Total search time, for each indexing program and filtering method, comparing both the pruned and non-pruned approaches on collections with large (>=100) alphabets. Additionally shows the traversal time on top of total search time for comparison. Displays the percentage of search time saved by pruning if applicable. Logarithmically scaled.

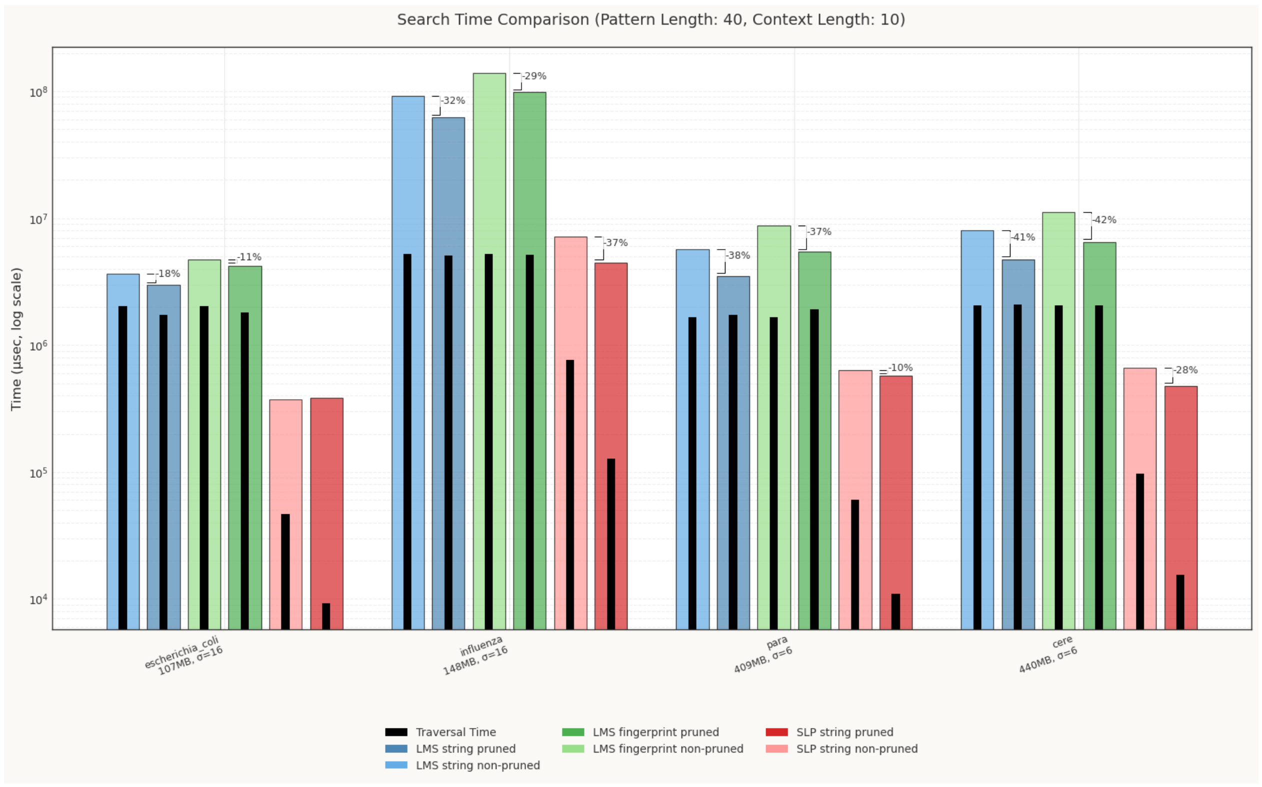

Figure B.90: Total search time, for each indexing program and filtering method, comparing both the pruned and non-pruned approaches on collections with small (<100) alphabets. Additionally shows the traversal time on top of total search time for comparison. Displays the percentage of search time saved by pruning if applicable. Logarithmically scaled.

B.8.14. Pattern Length 40, Context Length 50

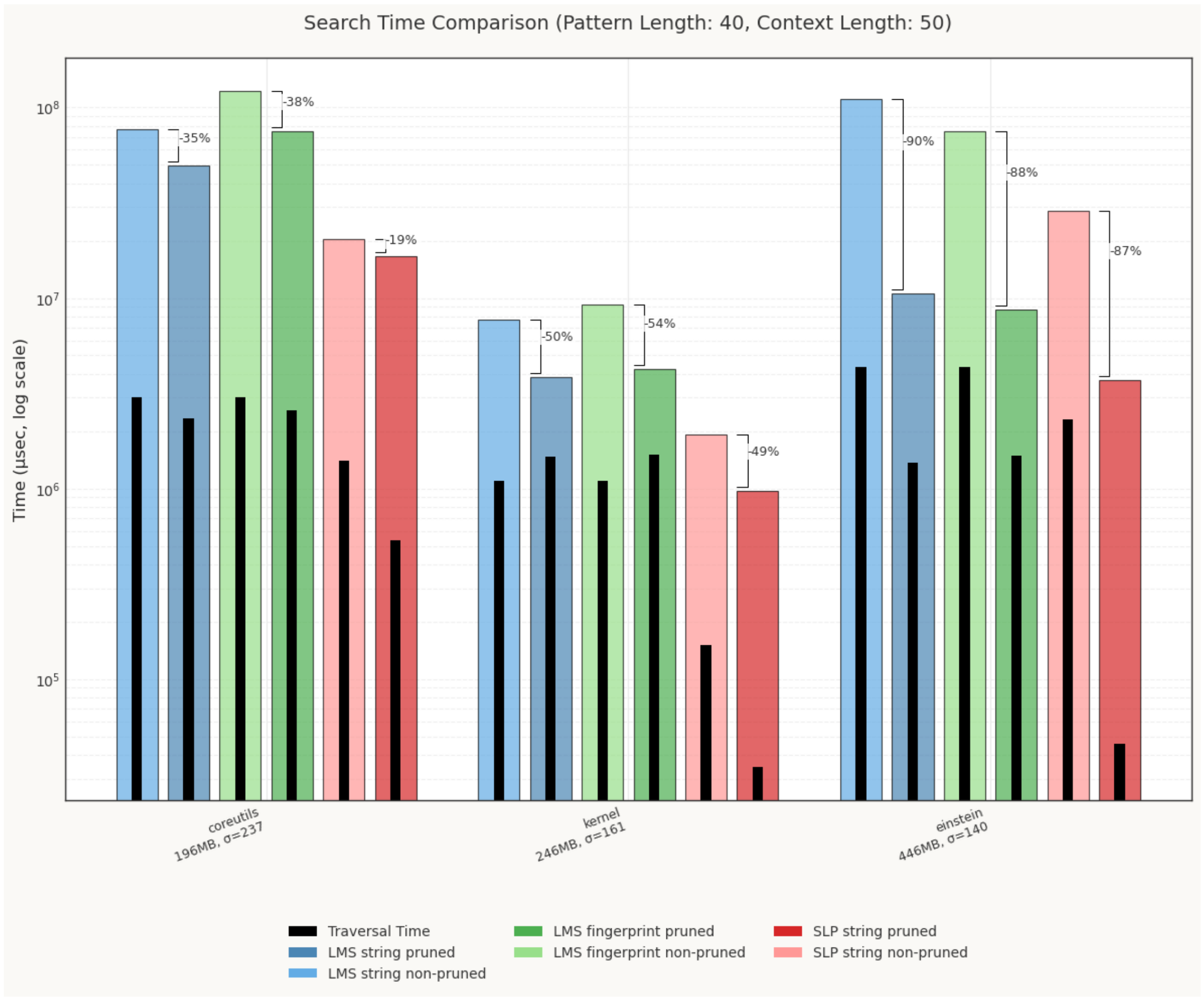

Figure B.91: Total search time, for each indexing program and filtering method, comparing both the pruned and non-pruned approaches on collections with large (>=100) alphabets. Additionally shows the traversal time on top of total search time for comparison. Displays the percentage of search time saved by pruning if applicable. Logarithmically scaled.

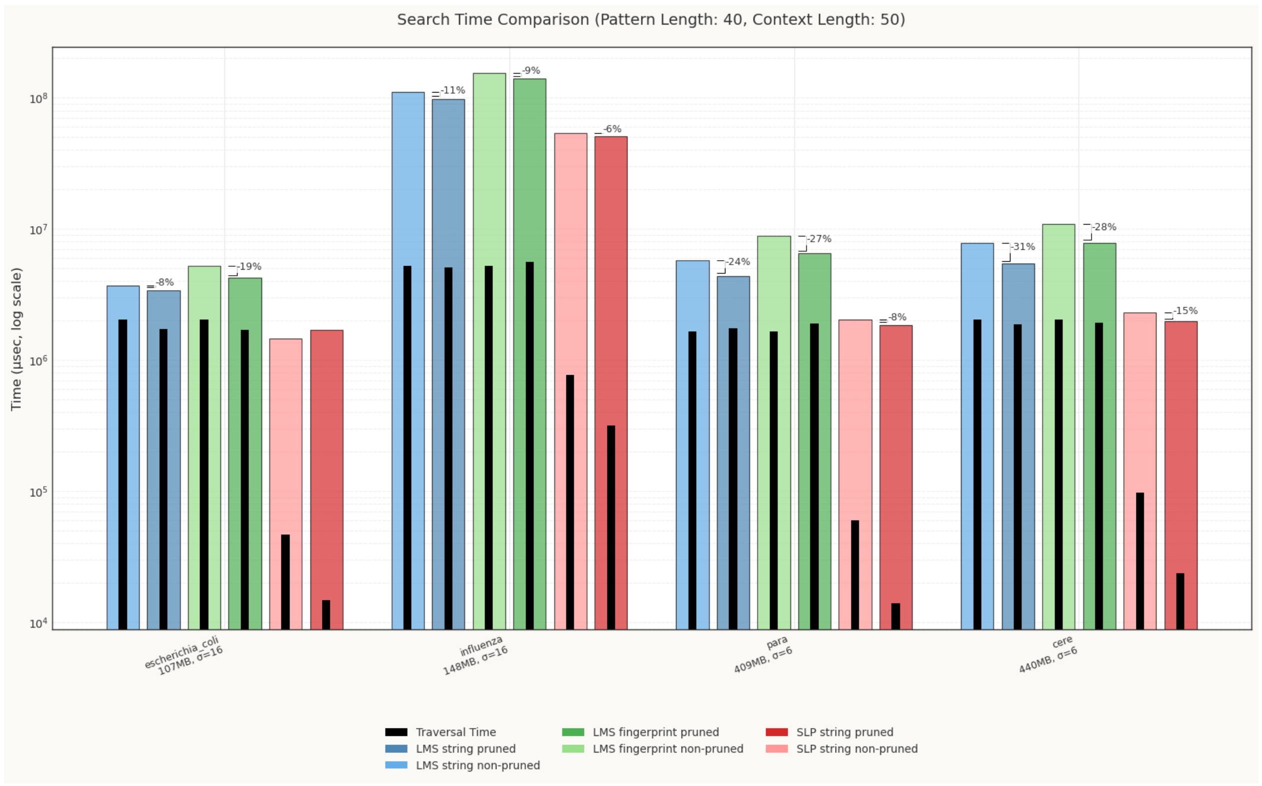

Figure B.92: Total search time, for each indexing program and filtering method, comparing both the pruned and non-pruned approaches on collections with small (<100) alphabets. Additionally shows the traversal time on top of total search time for comparison. Displays the percentage of search time saved by pruning if applicable. Logarithmically scaled.

B.8.15. Pattern Length 40, Context Length 100

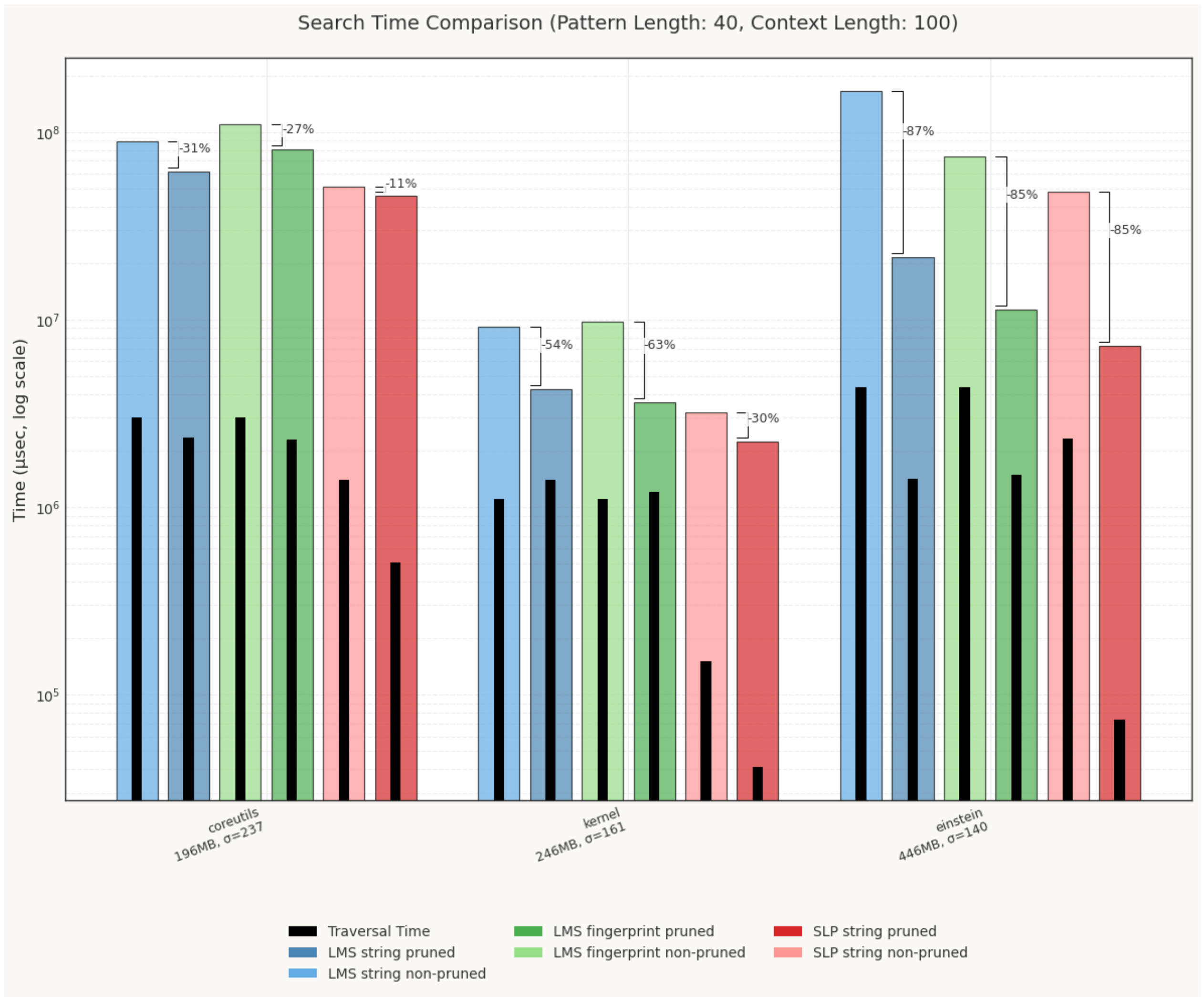

Figure B.93: Total search time, for each indexing program and filtering method, comparing both the pruned and non-pruned approaches on collections with large (>=100) alphabets. Additionally shows the traversal time on top of total search time for comparison. Displays the percentage of search time saved by pruning if applicable. Logarithmically scaled.

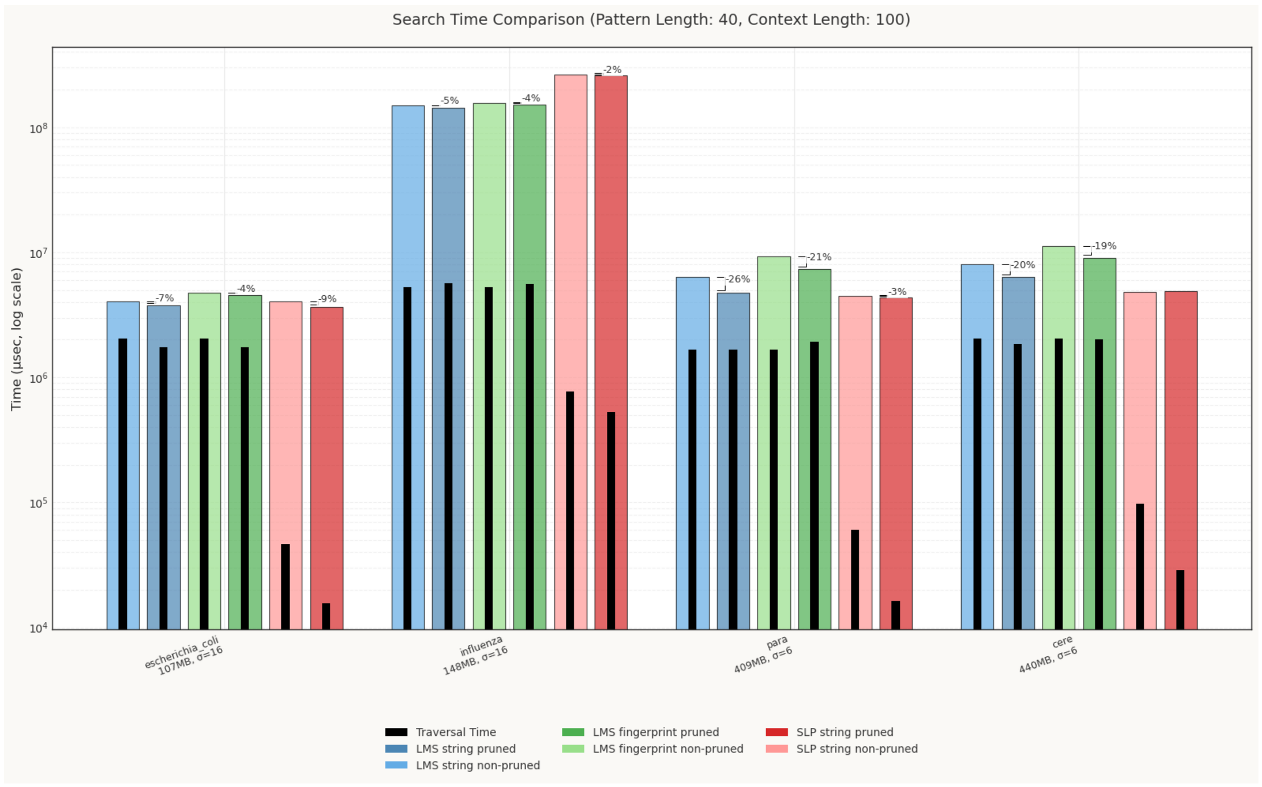

Figure B.94: Total search time, for each indexing program and filtering method, comparing both the pruned and non-pruned approaches on collections with small (<100) alphabets. Additionally shows the traversal time on top of total search time for comparison. Displays the percentage of search time saved by pruning if applicable. Logarithmically scaled.

B.8.16. Pattern Length 40, Non-Contextual

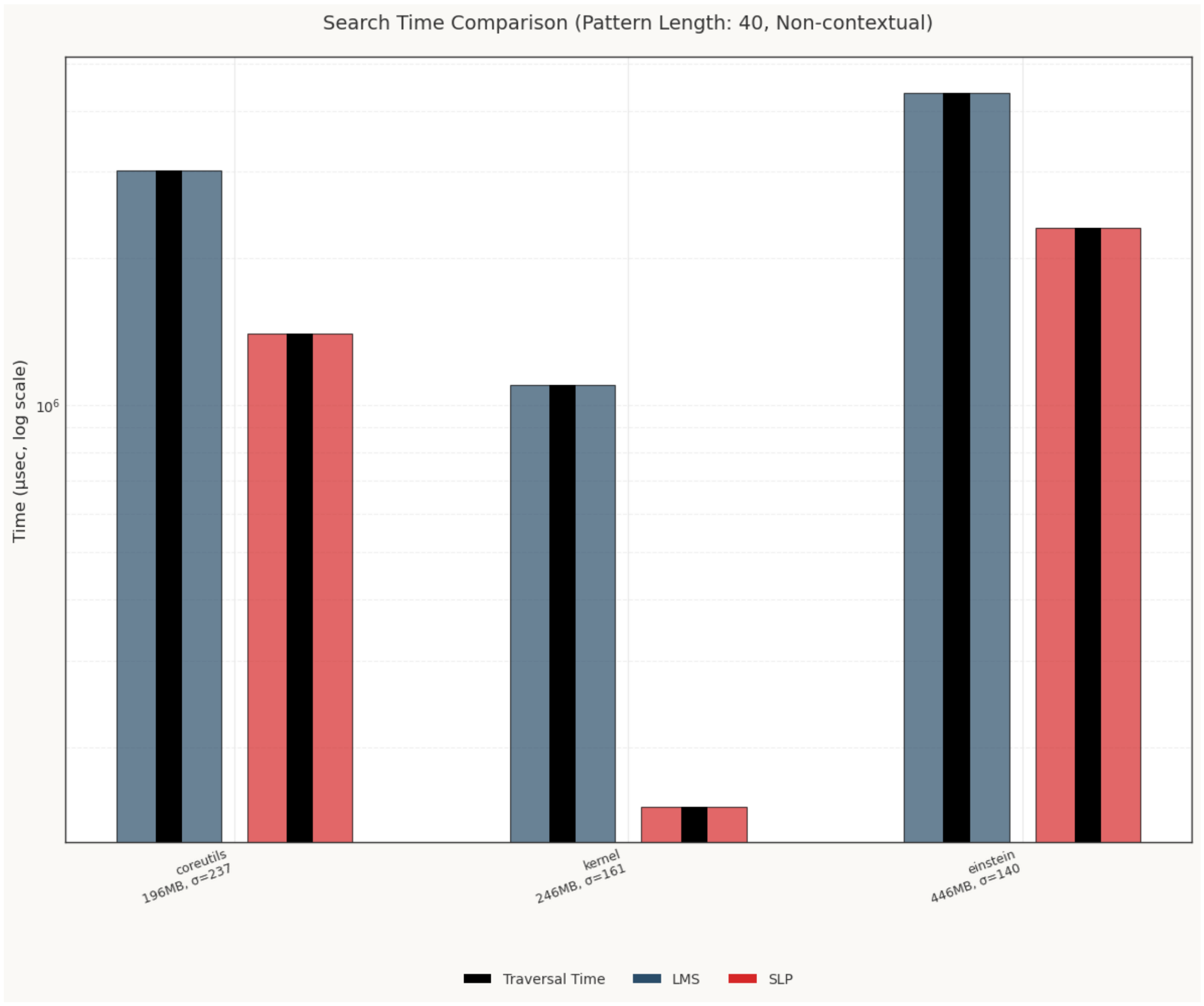

Figure B.95: Total search time, for each indexing program and filtering method, comparing both the pruned and non-pruned approaches on collections with large (>=100) alphabets. Additionally shows the traversal time on top of total search time for comparison. Displays the percentage of search time saved by pruning if applicable. Logarithmically scaled.

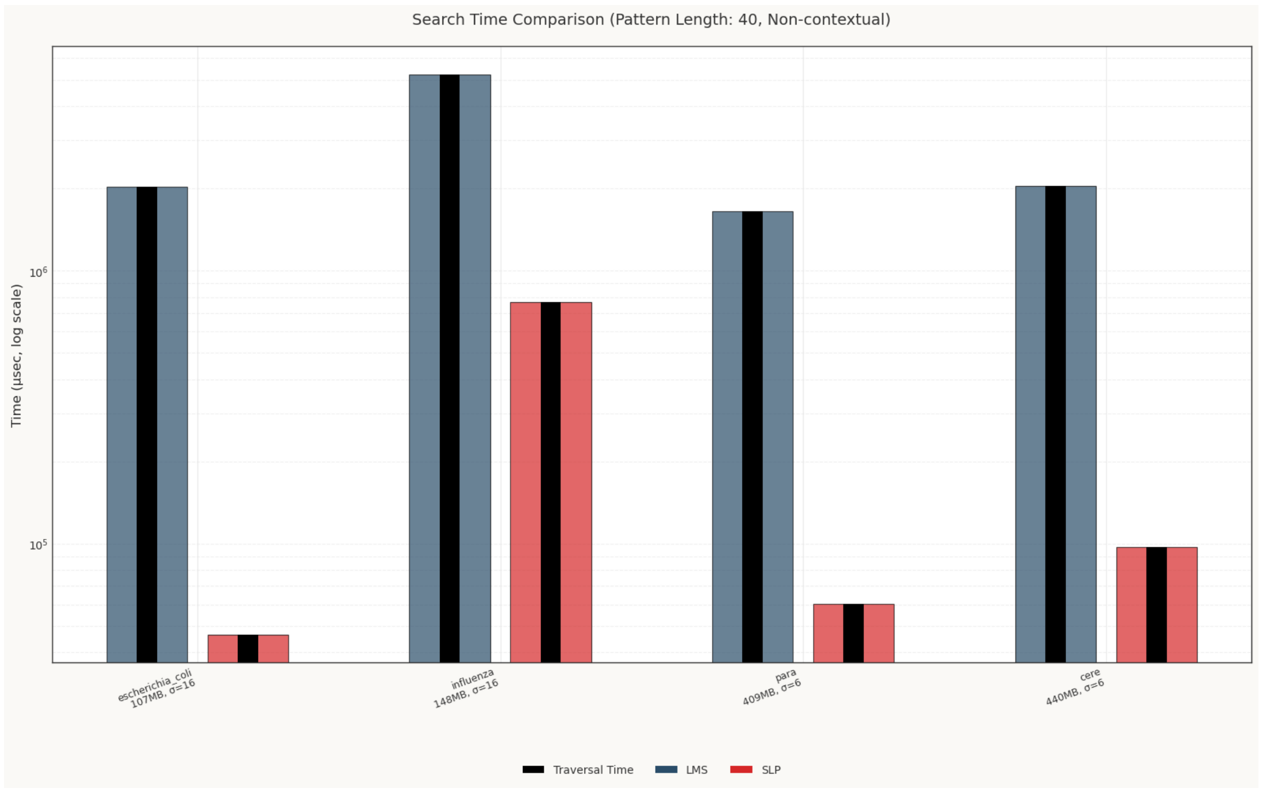

Figure B.96: Total search time, for each indexing program and filtering method, comparing both the pruned and non-pruned approaches on collections with small (<100) alphabets. Additionally shows the traversal time on top of total search time for comparison. Displays the percentage of search time saved by pruning if applicable. Logarithmically scaled.