Online Grammar-Based Self-Index and Its Applications

<table>
<thead>
<tr>
<th>著者</th>
<th>島 嘉将</th>
</tr>
</thead>
<tbody>
<tr>
<td>その他のタイトル</td>
<td>オンライン文法圧縮に基づく自己索引とそのアプリケーション</td>
</tr>
<tr>
<td>学位授与年度</td>
<td>平成33年度</td>
</tr>
<tr>
<td>学位授与番号</td>
<td>理工学甲情報工第345号</td>
</tr>
<tr>
<td>トピック</td>
<td>詳細な説明</td>
</tr>
</tbody>
</table>

URL: http://hdl.handle.net/10228/00006315

九州工業大学学術情報リポジトリ Kyutacar
Kyoju Institute of Technology Academic Repository
Online Grammar-Based Self-Index and Its Applications

Yoshimasa Takabatake
Acknowledgement

Numerous people helped me in my research and in the writing of my thesis during these past six years at the Kyushu Institute of Technology. I would like to thank them for all of their help and support.

First, I would like to thank Professor Hiroshi Sakamoto as the supervisor and mentor during my life as a student at the Kyushu Institute of Technology. He sparked in me an interest in grammar compression theories and practicalities.

Second, I would like to thank those individuals with whom I wrote several papers; Dr. Yasuo Tabei provided me with extensive knowledge regarding succinct data structures which I then applied to most of my research efforts. Professor Tetsuji Kuboyama introduced me to several interesting problems for my research. And Dr. Tomohiro I provided me with ideas and knowledge for processing compressed text.

Finally, I would like to thank Hiroshi Sakamoto’s laboratory members. They studied with me in the laboratory and encouraged me in both good and bad times.
Abstract

Text collections including many repetitions of substrings, called *highly repetitive text collections*, have been increasing in various fields like the version control system and the genome database. For the efficient use of such texts collections, the importance of the compression algorithm and compressed indexes is rapidly increasing more and more.

The grammar-based self-indexes are suitable for the problem of information retrieval on a compressed text because they support the random access on the compressed text. However, the constructing algorithms of existing grammar-based self-indexes are offline, that is, these algorithms require the whole input beforehand. Therefore, the memory consumption depends on the size of input text explicitly. In order to overcome this difficulty, we propose the first online algorithm for grammar-based self-index, called *Online Edit-Sensitive Parsing index* (OESP-index, for short). The proposed algorithm directly transforms the input text into the corresponding variable-length encoded string reading the input symbol one-by-one.

Compared to the existing self-indexes, the memory consumption of our online algorithm depends on the size of output, that is, the size of compressed text. Additionally, we also present three applications based on the grammar-based compression: (i) the online pattern matching problem for string edit-distance with moves called online ESP (OESP); (ii) the string index for edit-distance with moves called siEDM; (iii) the online grammar compression for frequent pattern discovery in smaller space. For these applications, we demonstrate the performance of our algorithm for large-scale data.
# Contents

Acknowledgements i

List of figures vii

List of tables ix

1 Introduction 1

2 Preliminaries 8

2.1 Basic notations ................................. 8
2.2 Straight-line program .............................. 9
2.3 Phrase dictionary and reverse dictionary ................... 9
2.4 Succinct data structures ............................ 10
2.5 Edit Sensitive Parsing ............................. 10
2.6 Online construction of an ESP-tree and its succinct encoding ........ 13

3 Structure of the OESP-index 16

3.1 Dynamic wavelet tree (DWT) ............................... 16
3.2 Complexity of building the OESP-index ........................ 17
3.3 Query search and substring extraction .......................... 18
3.4 Experiments .................................. 21

4 Online pattern matching for string edit distance with moves 25
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Context-free grammar</td>
<td>25</td>
</tr>
<tr>
<td>4.2</td>
<td>Phrase and reverse dictionaries</td>
<td>26</td>
</tr>
<tr>
<td>4.3</td>
<td>Problem definition</td>
<td>26</td>
</tr>
<tr>
<td>4.4</td>
<td>Online Algorithm</td>
<td>27</td>
</tr>
<tr>
<td>4.4.1</td>
<td>Post-order CFG</td>
<td>27</td>
</tr>
<tr>
<td>4.4.2</td>
<td>Online construction of a POCFG</td>
<td>28</td>
</tr>
<tr>
<td>4.4.3</td>
<td>Compressed phrase dictionary</td>
<td>30</td>
</tr>
<tr>
<td>4.4.4</td>
<td>Compressed reverse dictionary</td>
<td>31</td>
</tr>
<tr>
<td>4.4.5</td>
<td>Online pattern matching with EDM</td>
<td>32</td>
</tr>
<tr>
<td>4.5</td>
<td>An upper bound on our approximation</td>
<td>33</td>
</tr>
<tr>
<td>4.6</td>
<td>Experiments</td>
<td>34</td>
</tr>
<tr>
<td>5</td>
<td>siEDM: an efficient string index and search algorithm for edit distance with moves</td>
<td>37</td>
</tr>
<tr>
<td>5.1</td>
<td>Problem</td>
<td>37</td>
</tr>
<tr>
<td>5.2</td>
<td>Approximate computations of the EDM from ESP-trees</td>
<td>38</td>
</tr>
<tr>
<td>5.3</td>
<td>Index structure of ESP-trees</td>
<td>39</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Efficient encoding scheme</td>
<td>39</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Query processing on the ESP-tree</td>
<td>41</td>
</tr>
<tr>
<td>5.3.3</td>
<td>Other data structures</td>
<td>42</td>
</tr>
<tr>
<td>5.4</td>
<td>Search algorithm</td>
<td>43</td>
</tr>
<tr>
<td>5.4.1</td>
<td>Baseline search algorithm</td>
<td>43</td>
</tr>
<tr>
<td>5.4.2</td>
<td>Improved search algorithm</td>
<td>43</td>
</tr>
<tr>
<td>5.4.3</td>
<td>Finding candidates</td>
<td>45</td>
</tr>
<tr>
<td>5.4.4</td>
<td>Computing Positions</td>
<td>48</td>
</tr>
<tr>
<td>5.5</td>
<td>Experiments</td>
<td>50</td>
</tr>
<tr>
<td>6</td>
<td>Online grammar compression for frequent pattern discovery</td>
<td>57</td>
</tr>
<tr>
<td>Section</td>
<td></td>
<td></td>
</tr>
<tr>
<td>---------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6.1.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6.1.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 6.1 Approximate frequent pattern

- **Algorithm**: 58
- **Experimental Results**: 62

### 7 Conclusions and future work

- **Chapter summary**: 66
- **Future works**: 67

**References**: 69
List of Figures

2.1 Edit sensitive parsing. In (i), an underlined \( v[i] \) indicates a landmark. In (i) and (ii), a dashed node corresponds to an intermediate node in a 2-2-tree. 13

2.2 An example of a parse tree showing (I) for a post-order SLP, (II) the post order partial parse tree, and (III) the self-index structure. Note that the self-index structure consists of four data structures, each of which is directly constructed from the parse tree. .......................... 13

3.1 Example of dynamic wavelet tree. \( L \) is the leaf label of POPPT; Code is the integer representation of \( L \); \( A_i \) is the bit vector representing elements in code; Only \( A_i \) at each node is stored. .......................... 17

3.2 The required working memory for each method in MB for einstein (left) and cere (right). .......................... 24

3.3 The required working space of dictionary \( D \), length array \( R \), and hash table \( H \) for einstein (left) and cere (right). .......................... 24

3.4 Construction times for each method in seconds for einstein (left) and cere (right). 24

3.5 Locating times for each method in milliseconds for einstein (left) and cere (right). .......................... 24

4.1 Examples of (I) a POCFG, (II) the parse tree, (III) a post-order partial parse tree (POPPT), and (IV) the succinct representation of a POPPT. .......................... 28

4.2 Computation time in seconds versus the length of the text. .......................... 36
4.3 Working space of the dictionary and hash table versus the length of text. . 36
4.4 Working space of a POUDS (B), a label sequence (L) and a bit string (P)
which combine to organize a dictionary. .................... 36
4.5 The number of substrings whose EDM to a query is no more than each
threshold. ..................................... 36

5.1 Illustration of (I) an ESP-tree and (II) its corresponding characteristic vector. 39
5.2 Illustration of the encoding scheme for the ESP tree built from input string
S = babababaaba. .................................. 40
5.3 Illustration of candidate finding and $L_1$-distance computation. ........... 47
5.4 Comparison of search times for einstein (left) and cere (right). ........... 52
5.5 Details of search times for different $|Q|$ and $\tau$, including times for (a)-(b)
candidate findings (CF), (c)-(d) $L_1$-distance computations (DIST), and (e)-(f)
position computations (PC). (a) and (b) correspond to CF, (c) and (d)
correspond to DIST, and (e) and (f) correspond to PC of einstein and cere,
respectively. ........................................ 53
5.6 Statistical information of the query searches, showing (a)-(b) the num-
ber of traversed nodes (#TN), (c)-(d) the number of candidate $|Q|$-grams
(#CAND), (e)-(f) the number of true positives (#TP), (g)-(h) the number
of occurrences (#OCC). ................................ 54
5.7 Search time in seconds for repetitive texts, i.e., E.Coli (left) and influenza
(right). .................................................. 56

6.1 Memory consumption (MB) for each of the six data sets used .......... 64
6.2 Computation time in seconds for each of the six data sets used ........ 65
List of Tables

1.1 Comparing our OESP-index with offline methods. Construction time, search time and extraction time are presented using a big-$O$ notation that is omitted for space limitations. Here $N$ is the length of the given text, $m$ is the length of the query pattern, $n$ is the number of variables in the grammar, $\sigma$ is alphabet size, $h$ is the height of the parse tree of the straight-line program, $z$ is the number of phrases in LZ77, $d$ is the length of nesting in LZ77, $occ$ is the number of occurrences of the query pattern in the text, $occ_q$ is the number of candidate appearances of the query pattern, $\lg^*$ is the iterated logarithm, and $1/\alpha > 1$ is a load factor for the corresponding hash table. Finally, $\lg$ stands for $\log_2$. ........................................ 4

3.1 Index size in megabytes(MB). ........................................ 22
3.2 Working memory of dictionary $D$ consisting of bit string $B$ and dynamic wavelet tree $L$ for einstein and cere. ........................................ 22

4.1 Space requirements for POUDS $B$, label sequence $P$ and bit string $P$ organizing a dictionary on dna.200MB and english.200MB. .......................... 35

5.1 Summary of data sets. ........................................ 50
5.2 A comparison of the memory consumption for each query search. ........ 51
5.3 Comparing index sizes and construction times. .......................... 51
5.4 Summary of additional data sets. ........................................ 55
5.5 The memory consumption for the query searches. .................. 56
5.6 Comparing the index sizes and construction times for the additional data sets. .................................................. 56

6.1 Statistical information regarding benchmarking string $S$ .................. 63
6.2 Length of optimal $P$ extracted by the suffix array approach (SA) and approximate $X$ by our proposed algorithm (AFP(online)) with approximation ratio $\frac{|val(X)|}{|P|} (%)$ for the top-100 patterns. ............... 63
Chapter 1

Introduction

Text collections that include numerous repetitions of substrings are called *highly repetitive text collections* and have been increasing in frequency and size in version-controlled texts (e.g., Dropbox¹, Github², and Wikipedia³) and genome databases (e.g., the 1,000 genomes project [3]). To effectively and efficiently use these text collections, developing a fast and space-efficient compression algorithm and an index that supports fast queries on such compressed data is crucial.

*Self-indexes* are suitable for this problem in terms of space-efficiency. More specifically, although general indexes (e.g., *suffix array*, *suffix tree*, and *inverted index*) require input texts and an index structure for fast queries, self-indexes represent input texts using compressed texts that also supports fast queries. Self-indexes can support three functions on the compressed texts; these functions are *counting, location, and extraction*. Counting computes the frequency of a given query string appearing in the input texts, location finds the positions of the occurrences of a given query string in the input texts, and extraction obtains the substring of a given query interval in the input texts. Self-indexes have been proposed using various compressors (e.g., *Burrows-wheeler transform* (BWT) [6, 38], *lempel-ziv* (LZ) [20, 30, 36], and *straight-line program* (SLP) [1, 2, 8, 24, 45]).

---

¹https://www.dropbox.com
²https://github.com
³https://dumps.wikimedia.org
Of particular interest, grammar-based self-indexes (GSIs) [1, 2, 8, 24, 30, 45] are effective for highly repetitive texts because in GSIs, the texts is compressed and still supports fast query response times. The construction of GSIs requires the following two phases: (1) building a small context-free grammar (CFG) deriving a given input text deterministically, and (2) encoding the generated grammar into the self-index.

Here, phase (1) is called grammar compression. Although CFGs may take several forms, we focus on the CFG called a straight-line program (SLP) [19]. In [21], Lehman and Shelat [21] proved that finding the smallest grammar (i.e., the grammar with the fewest number of productions) is an NP-hard problem. Therefore, there are several approximation algorithms to solve the problem posed in phase (1) above. The best approximation ratio for the smallest grammar problem is $O(\lg(N/g_*)$) [16, 17, 21, 37, 39], where $N$ is the size of the input text and $g_*$ is the size of the smallest grammar. Unfortunately, the best approximation algorithms here are offline, i.e., they require the entire input text in advance. Furthermore, their space requirements depends on the input text size. Therefore, to add a new input text, they require additional time and space depending on the input text length since they require a full rebuild.

To save space, construction algorithms based on edit-sensitive parsing (ESP) techniques [4] have been developed (LCA [40], online LCA (OLCA) [25], fully-online LCA (FOLCA) [27], and FOLCA in constant space [26]). Although their approximation ratio is worse than the best approximation algorithms, their working space is theoretically small and thus practical for large-scale repetitive texts. LCA is an offline algorithm with an $O(\lg^* N \lg N)$ approximation ratio that requires $O(N)$ bits of space and $O(N)$ time, where $\lg^*$ is the iterated logarithm and $n$ is the grammar size. To compress increasingly large texts such as streaming texts, an online version of LCA called OLCA has also been proposed. OLCA builds a grammar in the working space of the grammar size while adding a new input character in $O(1)$ time. Theoretically, OLCA has an $O(\lg^2 N)$ approximation ratio and requires $O(n \lg n)$ bits of space and $O(N)$ time. For
even more space-efficiency of the given working space of OLCA, fully OLCA (FOLCA) has been developed. Although OLCA encodes a grammar into the succinct representation of SLP [42] after building the standard representation of the grammar using $2n \lg n$ bits of space, FOLCA directly encodes an input text into the succinct representation of SLP. The data structures are the succinct representation of SLP and a hash table for the grammar. To support updates, the succinct representation of SLP is indexed using a dynamic range min/max tree [33]. Theoretically, FOLCA has an $O(\lg^2 N)$ approximation ratio and requires $(1 + \alpha)n \lg(n + \sigma) + n(3 + \lg(\alpha n))$ bits of space and $O(\frac{N \lg n}{\alpha \lg \lg n})$ time where $\alpha$ is a load factor of the hash table and $\sigma$ is alphabet size. FOLCA in constant space [26] builds a grammar in constant space and linear time by using the direct encoding method of FOLCA and counting algorithms for stream data [5, 18, 23]. Although the approximation ratio here is worse than that of FOLCA, the application is practical for large-scale repetitive texts.

For phase (2) of the construction of GSIs, i.e., encoding the generated grammar into a self-index, several encodings for self-indexes have been proposed [1, 2, 8, 24, 45]. SLP-indexes [1, 2] represent a grammar via a wavelet tree [31] and support fast substring searches in the grammar-sized space. In [8], Gagie et al. provided a technique that supports substring searches faster than SLP-indexes. Furthermore, an ESP-index [24] encoded a grammar built by an LCA into a self-index that supports fast searches of long substrings. In [45], the search time of the ESP-index is improved using a rank/select dictionary for large alphabets [9]. In addition, the size of the grammar representation of ESP-indexes is asymptotically equal to the theoretic lower bound of SLP (i.e., $n + \lg n! + o(n)$), as proven by [42]. Encodings of grammars based on ESP are presented not only self-indexes but also in various applications [22, 25, 27, 42, 44]; many of these applications represent a grammar by its small-size data. Furthermore, [25],[42] and [44] presented offline encodings. Given the standard grammar representation of $2n \lg n$ bits of space, these offline approaches encode the given grammars in a succinct space. In addition,
Table 1.1: Comparing our OESP-index with offline methods. Construction time, search time and extraction time are presented using a big-$O$ notation that is omitted for space limitations. Here $N$ is the length of the given text, $m$ is the length of the query pattern, $n$ is the number of variables in the grammar, $\sigma$ is alphabet size, $h$ is the height of the parse tree of the straight-line program, $z$ is the number of phrases in LZ77, $d$ is the length of nesting in LZ77, $\text{occ}$ is the number of occurrences of the query pattern in the text, $\text{occ}_q$ is the number of candidate appearances of the query pattern, $\lg^*$ is the iterated logarithm, and $1/\alpha > 1$ is a load factor for the corresponding hash table. Finally, $\lg$ stands for $\log_2$.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Working space(bits)</th>
<th>Index size(bits)</th>
<th>Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>LZ-index [30]</td>
<td>$O(N)$</td>
<td>$z \lg N + 5z \lg N$</td>
<td>Offline</td>
</tr>
<tr>
<td>Gagie et al. [8]</td>
<td>$O(N)$</td>
<td>$2n \lg n + O(z \lg N)$</td>
<td>Offline</td>
</tr>
<tr>
<td>SLP-index [1, 2]</td>
<td>$O(N)$</td>
<td>$n \lg N + O(n \lg n)$</td>
<td>Offline</td>
</tr>
<tr>
<td>ESP-index [45]</td>
<td>$O(N)$</td>
<td>$n \lg N + n \lg n$</td>
<td>Offline</td>
</tr>
<tr>
<td>OESP-index</td>
<td>$n \lg N + O((n + \sigma) \lg(n + \sigma))$</td>
<td>$n \lg N + O((n + \sigma) \lg(n + \sigma))$</td>
<td>Online</td>
</tr>
</tbody>
</table>

[44] presented the first online encoding; it uses a rank/select dictionary [35] and directly encodes an input text into a grammar representation that uses $\frac{7}{4}n \lg n + 4n + o(n)$ bits of space. Using a dynamic range min/max tree [33], [27] required for online encoding to $2n + n \lg n + o(n)$ bits of space. Furthermore, [22] described the string indexes for the hamming distance. Encodings of general SLPs for various applications have also been proposed (e.g., pattern matching [48], pattern mining [10], and edit distance computation [14]).

These online SLP-based methods do not support fast queries and the corresponding construction algorithms of existing self-indexes, including GSIs, are performed offline. Therefore, to efficiently index changing texts such as streaming texts, the development of an online self-index that supports the fast addition of new input characters into the working space of the compressed text size is a key challenge.

In our studies, both in theory and practice, we aimed to develop an online GSI working
with a fast construction time, a compressed working space and fast search times. Our theoretical goal is \( O(1) \) addition time per input character, the working space (which equals the index size) that depends on the grammar size and a search time matching those of ESP-index \([45]\) The search times for ESP-indexes and other approaches are shown in Table 1.1. In practice, our first goal is to achieve additions at a rate of 1.0MB/s; furthermore, we aim to achieve a working space of the compressed text size and a search time of 1 (pattern of length 100)/ms. By attaining these goals, we can expect to index increasing and large-scale texts in real time and compressed space.

In this thesis, we first present a first online GSI called the \textit{online edit-sensitive parsing index} (OESP-index), which we present with comparisons to other approaches in Table 1.1. Furthermore, we present three applications that are based on ESP; these applications are (1) online pattern matching for determining string edit distances with moves called online ESP (OESP), (2) a string index for determining edit distance with moves (siEDM), and (3) online grammar compression for frequent pattern discovery.

In Chapter 3, we describe details behind our OESP-index. The algorithm uses the direct encoding method of FOLCA and represents a part of the data structure of FOLCA using a dynamic wavelet tree \([12]\). Although the construction and search times of OESP-index are slower than ESP-index because of the use of the dynamic wavelet tree, OESP-index is built in a working space that depends on the grammar size and is able to add an input character in \( O\left( \frac{k \cdot n \cdot \log^* N}{\alpha} \right) \) time. We compare our algorithm here with existing offline grammar-based self-indexes in Table 1.1. Furthermore, in our experiments, we show the space-efficiency of our OESP-index’s working space for two benchmark texts.

In Chapter 4, we expand the search algorithm based on ESP from exact matches to ambiguous pattern searches for edit distance with moves (EDM). The search algorithm is an online pattern matching for EDM called OESP. In the online pattern matching for EDM, given a query text \( Q \), a streaming text \( S(|S| \geq |Q|) \) and a distance threshold \( k \), we output all positions \( i \in [1, |S| - |Q|] \) such that the approximated EDM between \( Q \) and
$S[i, (i + |Q|)]$ is smaller than $k$. EDM between two strings is the minimum number of editing operations (i.e., character insertions, deletions, and replacements, and substring moves) required to transform from a given text to another given text. Note that EDM is efficient for the discovery of plagiarism and many other applications, but its computation is NP-complete. Therefore, several approximation algorithms have been proposed [28, 41]; however, these algorithms operate offline, i.e., they cannot quickly perform online pattern matching problem for EDM since they require two entire texts beforehand. ESP [4] is also one of the offline approximation algorithm. The approximation ratio for exact EDM is $O(\lg^* N \lg N)$; to compute this for the approximated EDM, ESP builds two grammars that generate a given text and another given text. OESP is based on ESP. By using online construction techniques of FOLCA, OESP computes the approximated EDM in $O\left(\frac{\lg N \lg n}{n \lg \lg n}\right)$ time per position while building the grammar for text $S$ within the working space of the grammar size. The approximation ratio of OESP is $O(\lg^2 N)$. Furthermore, our experiments show the practical working space for highly repetitive texts.

In Chapter 5, we describe siEDM, which is an offline string index (i.e., the “si” portion of siEMD) for use in the fast computation of the online pattern matching for EDM. Although OESP computes the EDM of $|S| - |Q|$ positions, siEDM reduces the computations to $n|Q|$ positions by a characteristic of SLP presented in [10]. Furthermore, siEDM uses an ESP-index [45] and an additional data structure. Based on this data structure, the index size is larger than that of OESP; however, siEDM can achieve faster searches than OESP. In our experiments, we show the resulting fast computation times for highly repetitive texts.

In Chapter 6, we explain online grammar compression for frequent pattern discovery. More specifically, we find approximated frequent patterns on FOLCA’s data structure. Here, frequent pattern discovery is finding all substrings that appear more than once in an input text. To find all frequent patterns exactly, general methods (e.g. suffix array) require space that depends on the input text size. To efficiently solve this problem in
terms of space, Nakahara et al. [29] proposed an approximation algorithm for grammar compression. This algorithm finds not a frequent pattern \( P \) but rather a substring \( R \) of \( P \); this \( R \) is guaranteed to have \( |R| \geq \Omega\left(\frac{|P|}{\log^2 N}\right) \). More specifically, the algorithm builds a grammar based on ESP and identifies the frequent production rules. Next, each substring generated from the frequent production rules represents the approximated frequent pattern; however, the algorithm is an offline algorithm, \textit{i.e.}, the algorithm cannot be applied to streaming texts. By using the online construction approach of FOLCA, our proposed method finds the frequent production rules as it builds a grammar based on ESP with a working space that depends on grammar size. In addition, we improve the length guarantee of the approximated frequent pattern to \( |R| \geq \Omega\left(\frac{|P|}{\log^2 N \log N}\right) \). Our experiments show the space-efficiency of our proposed method for highly repetitive texts.

Finally, in Chapter 7, we conclude our thesis and provide directions for future work.
Chapter 2

Preliminaries

In this chapter, in providing the preliminaries, please note that Section 2.1 and 2.5 refer to [7], while other sections refer to [47].

2.1 Basic notations

Let \( \Sigma \) be a finite alphabet, and \( \sigma \) be \( |\Sigma| \). All elements in \( \Sigma \) are totally ordered. Let us denote by \( \Sigma^* \) the set of all strings over \( \Sigma \), and by \( \Sigma^q \) the set of strings of length \( q \) over \( \Sigma \), i.e., \( \Sigma^q = \{ w \in \Sigma^* : |w| = q \} \); further an element in \( \Sigma^q \) is called a \( q \)-gram. The length of a string \( S \) is denoted by \( |S| \). The empty string \( \epsilon \) is a string of length zero, i.e., \( |\epsilon| = 0 \). For a string \( S = \alpha\beta\gamma \), \( \alpha \), \( \beta \) and \( \gamma \) are called the prefix, substring, and suffix of \( S \), respectively. The \( i \)-th character of a string \( S \) is denoted by \( S[i] \) for \( i \in [1, |S|] \). For a string \( S \) and an interval \( [i,j] \) (\( 1 \leq i \leq j \leq |S| \)), let \( S[i,j] \) denote the substring of \( S \) that begins at position \( i \) and ends at position \( j \), and let \( S[i,j] \) be \( \epsilon \) when \( i > j \). For a string \( S \) and an integer \( q \geq 0 \), let \( \text{pre}(S,q) = S[1,q] \) and \( \text{suf}(S,q) = S[|S| - q + 1, |S|] \). For strings \( S \) and \( P \), let \( \text{freq}_S(P) \) denote the number of occurrences of \( P \) in \( S \), i.e., \( \text{freq}_S(P) = |\{ i : S[i, i + |P| - 1] = P \}| \).

We assume a recursive enumerable set \( \mathcal{X} \) of variables with \( \Sigma \cap \mathcal{X} = \emptyset \). Here, all elements in \( \Sigma \cup \mathcal{X} \) are totally ordered, where all elements in \( \Sigma \) must be smaller than those in \( \mathcal{X} \). In this paper, we call a sequence of symbols from \( \Sigma \cup \mathcal{X} \) a string. Let us define \( \lg^{(1)} u = \lg u \),
and \( \lg^{(i+1)} u = \lg (\lg^i u) \) for \( i \geq 1 \). The iterated logarithm of \( u \) is denoted by \( \lg^* u \) and defined as the number of times the logarithmic function must be applied before the result is less than or equal to 1, i.e., \( \lg^* u = \min \{ i : \lg^i u \leq 1 \} \).

### 2.2 Straight-line program

A CFG in Chomsky normal form is represented by a quadruple \( G = (\Sigma, V, D, X_s) \) where \( V \) is a finite subset of \( \mathcal{X} \), \( D \) is a finite subset of \( V \times (V \cup \Sigma)^2 \) and \( X_s \in V \) is the start symbol. Furthermore, an element in \( D \) is called a production rule, and a variable in \( V \) is called a nonterminal symbol. Also, \( \text{val}(X_i) \) denotes the string derived from \( X_i \in V \). For \( X_1, X_2, \ldots, X_k \in V \), let \( \text{val}(X_1, X_2, \ldots, X_k) = \text{val}(X_1)\text{val}(X_2)\ldots\text{val}(X_k) \). A grammar compression of \( S \) is a CFG that derives only \( S \). The size of a CFG is the number of variables, i.e., \( |V| \); further, let \( n = |V| \).

The parse tree of \( G \) is a rooted ordered binary tree such that (1) internal nodes are labeled by variables in \( V \) and (2) leaf nodes are labeled by symbols in \( \Sigma \), i.e., the label sequence in leaf nodes is equal to input string \( S \). In a parse tree, any internal node \( Z \) corresponds to the production rule \( Z \to XY \) and has a left child with label \( X \) and a right child with label \( Y \). A partial parse tree \([37]\) is an ordered tree formed by traversing the parsing tree in a depth-first manner and pruning out all descendants under every node of variables that appear no less than twice.

Finally, an SLP) \([19]\) is defined as a grammar compression over \( \Sigma \cup V \) with production rules in the form \( X_k \to X_iX_j \) where \( X_k, X_i, X_j \in \Sigma \cup V, 1 \leq i \), and \( j < k \leq n + \sigma \).

### 2.3 Phrase dictionary and reverse dictionary

A phrase dictionary is a data structure for directly accessing a digram \( X_iX_j \) from a given \( X_k \) if \( X_k \to X_iX_j \in D \). Here a phrase dictionary is typically implemented by an array that requires \( 2n \log(n + \sigma) \) bits for storing \( n \) production rules. Additionally, a reverse
dictionary $D^{-1}$ is a mapping from a digram to an associated variable. Here, $D^{-1}(XY)$ returns the variable $Z$ if $Z \rightarrow XY \in D$; otherwise, it creates a new variable $Z' \not\in V$ and returns $Z'$.

### 2.4 Succinct data structures

As one of our data structures, we use the fully indexable dictionary (FID) for indexing bit strings. Our method represents CFGs that use a rank/select dictionary, which is a succinct data structure for representing a bit string $B$ [15] that supports the following queries: (1) $\text{rank}_c(B, i)$ returns the number of occurrences of $c \in \{0, 1\}$ in $B[0, i]$; (2) $\text{select}_c(B, i)$ returns the position of the $i$-th occurrence of $c \in \{0, 1\}$ in $B$; and (3) $\text{access}(B, i)$ returns the $i$-th bit of $B$. In terms of implementation, data structures with $|B| + o(|B|)$ bit storage to achieve $O(1)$ time rank and select queries [35] have been presented.

For online grammar compression, we adopt the dynamic range min/max tree (DR-MMT) [33] for the online construction of the parse tree. We obtain $\text{parent}(B, i)$, i.e., the parent of node $i$ of DRMMT $B$ in $O\left(\frac{\log n}{\log \log n}\right)$ time where $n$ is the tree’s number of nodes.

We consider the wavelet tree (WT) [11], as an extension of FID for general alphabets. A WT is a data structure used to represent strings over finite alphabets and can compute the rank and select queries on a string $S$ over $\Sigma^*$ in $O(\log \sigma)$ time using $|S| \log \sigma (1 + o(1))$ bits.

### 2.5 Edit Sensitive Parsing

Originally, edit-sensitive parsing (ESP) was introduced by [4] and widely applied to data compression and information retrieval (e.g., [13, 46, 47, 43, 34]). ESP is a parsing technique intended to efficiently construct a consistent parsing of the same substrings, whose further description hereafter follows.

In this subsection, we review the algorithm for ESP as presented by [43]. This algo-
algorithm, referred to as ESP-comp, computes an SLP from an input string $S$. The tasks of ESP-comp are as follows: (1) partition $S$ into $s_1 s_2 \cdots s_\ell$ such that $2 \leq |s_i| \leq 3$ for each $1 \leq i \leq \ell$; (2) if $|s_i| = 2$, generate the production rule $X \to s_i$ and replace $s_i$ by $X$ (with this subtree referred to as a 2-tree), and if $|s_i| = 3$, generate the production rule $Y \to AX$ and $X \to BC$ for $s_i = ABC$, and replace $s_i$ by $Y$ (referred to as a 2-2-tree); (3) iterate this process until $S$ becomes a symbol. Finally, the ESP-comp builds an SLP that represents the string $S$.

Next, we focus on how to determine the partition $S = s_1 s_2 \cdots s_\ell$. A string of the form $a^r$ with $a \in \Sigma \cup V$ and $r \geq 2$ is called a repetition. Furthermore, a repetition $S[i, j]$ is said to be maximal if $S[i] \neq S[i - 1], S[j + 1]$. First, $S$ is uniquely partitioned into the form $w_1 x_1 w_2 x_2 \cdots w_k x_k w_{k+1}$ by its maximal repetitions, where each $x_i$ is a maximal repetition of a symbol in $\Sigma \cup V$, and each $w_i \in (\Sigma \cup V)^*$ contains no repetition. Then, each $x_i$ is called type1, each $w_i$ of a length of at least $2 \lg^* |S|$ is type2, and any remaining $w_i$ is type3. If $|w_i| = 1$, this symbol is attached to $x_{i-1}$ or $x_i$ with preference $x_{i-1}$ when both cases are possible. Thus, if $|S| > 2$, each $x_i$ and $w_i$ is longer than or equal to two.

Next, ESP-comp parses each substring $v$ depending on the type. For type1 and type3 substrings, the algorithm performs the left aligned parsing as follows. If $|v|$ is even, the algorithm builds a 2-tree from $v[2j - 1, 2j]$ for each $j \in \{1, 2, \ldots, \lfloor |v|/2 \rfloor\}$; otherwise, the algorithm builds a 2-tree from $v[2j - 1, 2j]$ for each $j \in \{1, 2, \ldots, \lfloor (|v| - 3)/2 \rfloor\}$ and builds a 2-2-tree from the last trigram $v[|v| - 2, |v|]$. If $v$ is type2, the algorithm further partitions it into short substrings of length two or three by the following alphabet reduction algorithm. Given a type2 string $v$, consider $v[i]$ and $v[i - 1]$ as binary integers. Let $p$ be the position of the least significant bit of $v[i] \oplus v[i - 1]$ and let $\text{bit}(p, v[i])$ be the bit of $v[i]$ at the $p$-th position. Then, $L(v)[i] = 2p + \text{bit}(p, v[i])$ is defined for any $i \geq 2$. Because $v$ is repetition-free (i.e., type2), the label string $L(v)[2, |v|]$ is also type2. Suppose that any symbol in $v$ is an integer in $\{0, \ldots, N\}$; then $L(v)[2, |v|]$ is a sequence of integers in $\{0, \ldots, 2 \lg N + 1\}^{|v| - 1}$. If we apply this procedure $\lg^* N$ times, we obtain
$L^*(v)[\lg^* N + 1, |v|]$ a sequence of integers in $\{0, \ldots, 5\}^{\lg^* N}$, where $L^*(v)[1, \lg^* N]$ is not defined. When $L^*(v)[i - 1], L^*(v)[i], L^*(v)[i + 1]$ are defined, $v[i]$ is called the landmark if $L^*(v)[i] > \max\{L^*(v)[i - 1], L^*(v)[i + 1]\}$.

An iteration of alphabet reduction transforms $v$ into $L^*(v)$ such that any substring of $L^*(v)[\lg^* N + 1, |v|]$ of length at least 10 contains at least one landmark because $L^*(v)[\lg^* N + 1, |v|] \in \{0, \ldots 5\}$ is also type2. Using this characteristic, the ESP-comp algorithm determines the bigrams $v[i, i + 1]$ to be replaced for any landmark $v[i]$, where any two landmarks are not adjacent, thus the replacement is deterministic. After replacing all landmarks, any remaining maximal substring $s$ is replaced by the left aligned parsing, where, if $|s| = 1$, it is attached to its left or right block.

We provide an example of the ESP of an input string in Figure 2.1-(i) and (ii). For a type2 substring $v$ (i.e., Figure 2.1-(i)), $v$ is parsed according to landmarks; in this case, landmarks are determined by conducting an alphabet reduction twice for the sake of simplicity. Any other remaining substrings, including type1 and type3 substrings, are parsed by the left aligned parsing, as shown in Figure 2.1-(ii). In this figure, a dashed node indicates an intermediate node in a 2-2-tree. Originally, an ESP tree is a ternary tree in which each node has at most three children. The intermediate node is introduced to represent the ESP tree as a binary tree.

The following characteristics are well-known for ESPs. By Theorem 1, we can obtain the locally consistent parsing for $S$, i.e., an iteration of ESP for $S$. For any substring $P$ of $S$, there exists an interval $[i, j]$ of length at least $|P| - O(\lg^* |S|)$ such that the substring $P[i, j]$ with each occurrence of $P$ is transformed into the same string. Iterating through this, the resulting ESP tree contains a large subtree for $P$ regardless of its occurrences; this tree expresses an approximation of $P$. Theorem 2 is clear by the definition of ESP.

**Theorem 1.** ([4]) For a type2 substring $v$, whether $v[i]$ is a landmark or not is determined by only $v[i - O(\lg^* |S|), i + O(1)]$.

---

1The number of iterations an alphabet reduction is performed should not be changed arbitrarily according to each $v$; therefore $N$ is set in advance to be a sufficiently large integer, e.g., $N = O(|S|)$. 

---
CHAPTER 2. PRELIMINARIES

(i) Parsing for a type 2 substring and the alphabet reduction

Binary of v: 0000 0001 1001 0000
First labels: 1 7 0 3 2 1 0 1 7 0 3 2 1 0
Final labels: 1 3 0 1 0 1 0 1 3 0 1 0 1 0

Figure 2.1: Edit sensitive parsing. In (i), an underlined v[i] indicates a landmark. In (i) and (ii), a dashed node corresponds to an intermediate node in a 2-2-tree.

(ii) Left aligned parsing for a type 1 substring of odd length

Figure 2.2: An example of a parse tree showing (I) for a post-order SLP, (II) the post order partial parse tree, and (III) the self-index structure. Note that the self-index structure consists of four data structures, each of which is directly constructed from the parse tree.

Theorem 2. \([4]\) The height of the ESP tree of \(S\) is \(O(\lg |S|)\).

2.6 Online construction of an ESP-tree and its succinct encoding

FOLCA [27] is an online algorithm used to builds an ESP-tree as a post-order partial parse tree (POPPT) based on the parsing rule in the ESP algorithm from a given string. As illustrated in Figure 2.2, the post-order SLP (POSLP) and corresponding post-order partial parse tree (POPPT) are defined as follows;
**Definition 1** (POS LP and POP PT [27]). A **POS LP** is an SLP whose parse tree’s internal nodes have post-order variables. A **POP PT** is a partial parse tree whose internal nodes have post-order variables.

Here, Figure 2.2-(I) and -(II) show an example of a parse tree and a POP PT.

Additionally, FOLCA represents a POP PT by its succinct data structure, which includes the succinct tree of the POP PT $B$ and a non-negative integer array $L$. Here, $B$ is a bit string made by traversing the POP PT in post-order, and putting '0' if a leaf node and '1' otherwise. The last bit '1' in $B$ represents a virtual node, and $B$ is indexed by the DRMMT [33]. The succinct tree supports the following three operations: (1) $parent(B, i)$ returns the parent node of node $i$; (2) $left\_child(B, i)$ returns the left child of node $i$; and (3) $right\_child(B, i)$ returns the right child of node $i$. These are each computed in $O(\lg n / (\lg \lg n))$ time. The space for our succinct tree is at most $2n + o(n)$ bits. Figure 2.2-(III)-(i) shows an example of $B$.

Finally, a non-negative integer array $L$ stores symbols at the leaf nodes from the leftmost leaf to the rightmost leaf in the POP PT. The space required for $L$ is $(n + 1) \lg n$ bits. Figure 2.2-(III)-(iii) shows an example of $L$.

As an additional data structure for constructing a POP PT, FOLCA uses a hash table $H$. More specifically, a reverse dictionary $H : (V \cup \Sigma) \times (V \cup \Sigma) \rightarrow V$ is implemented using a chaining hash table. Here, let $1/\alpha$ be a constant number larger than 1 and called the load factor. The hash table has $\alpha n$ entries and each entry stores a list of integers $i$ that represents the left hand side of rule $X_i \rightarrow X_jX_k$. The size of the data structure is $\alpha n \lg (n + \sigma)$ bits for the hash table and $n \lg (n + \sigma)$ bits for the lists. Therefore, the total size is $n(1 + \alpha) \lg (n + \sigma)$ bits. The access time is expected to be $O(1/\alpha)$ time. Figure 2.2-(III)-(ii) shows an example of $H$.

By using the aforementioned data structures, i.e., $B$, $L$ and $H$, FOLCA constructs a POP PT based on ESP from an input string in an online manner. From this, we obtain the following theorem.
Theorem 3 ([27]). The POSLP of \( n = O(g_* \lg^* |S| \lg |S|) \) variables and \( \sigma \) alphabet symbols that support the phrase and reverse dictionaries can be constructed in \( O\left(\frac{|S| \lg n \lg^* |S|}{\alpha \lg \lg n}\right) \) expected time using \((1+\alpha)n \lg(n+\sigma)+n(3+\lg(\alpha n))\) bits memory where \( g_* \) is the minimum grammar size, \( S \) is an input string, and \( 1/\alpha \in (0,1) \) is the load factor of a hash table.

The OESP-index uses FOLCA’s direct encoding method for a POPPT and constructs an index structure in an online manner for fast query searches and substring extractions, which have been explained in this chapter and are further explained in the next chapter.
Chapter 3

Structure of the OESP-index

In this chapter, we describe our online grammar-based self-index, which performs the addition of an input character in $O\left(\frac{\log n \log^* N}{\alpha}\right)$ time. Here, the working space depends on the grammar size and has $O\left(\log(n + \sigma)(\frac{m}{\alpha} + \text{occ}_q(\log N + \log m \log^* N))\right)$ search time. Note that this chapter refer to [47].

OESP-index’s succinct representation consists of four data structures: (i) $B$ : succinct tree of POPPT, (ii) $H$ : hash table (iii) $L$ : non-negative integer array indexed by wavelet tree and (iv) $R$ : non-negative integer array.

$B$, $H$ and $L$ are FOLCA’s data structures. $B$ and $H$ are completely same as themselves of FOLCA’s data structures. $L$ is only represented by dynamic wavelet tree (DWT) that is presented in the next subsection. Each element of non-negative integer array $R$ is the length of the string derived from a variable, i.e., $|\text{val}(X_i)|$ for $X_i \in V$. The size of $R$ is $n \log |S|$ bits. Figure 2.2-(III)-(iv) shows an example of $R$.

3.1 Dynamic wavelet tree (DWT)

Our DWT is a wavelet tree that supports the operation of adding an element to the tail of a sequence. Such an operation is called a pushback operation, which is necessary for implementing DWT. Further, a wavelet tree for a sequence $L$ over a range of alphabets
CHAPTER 3. STRUCTURE OF THE OESP-INDEX

Figure 3.1: Example of dynamic wavelet tree. $L$ is the leaf label of POPPT; Code is the integer representation of $L$; $A_i$ is the bit vector representing elements in code; Only $A_i$ at each node is stored.

and variables $[1..(n+\sigma)]$ can be recursively described over a sub-range $[a..b] \subseteq [1..(n+\sigma)]$.

A wavelet tree (WT) over a range $[a..b]$ is a binary balanced tree with $b-a+1$ leaf nodes. Here, if $a = b$, the tree is just a leaf labeled $a$; otherwise it has an internal root node that represents $L$. The root has a bit string $A_{\text{root}}[1,|S|]$ defined as follows: if $L[i] \leq (a+b)/2$ then $A_{\text{root}}[i] = 0$, else $A_{\text{root}}[i] = 1$. We define $L_0[1,\ell_0]$ as the subsequence of $L$ formed by symbols $c \leq (a+b)/2$ and $L_1[1,\ell_1]$ as the subsequence of $L$ formed by symbols $c > (a+b)/2$.

Then, the left child of the root is a WT for $L_0[1,\ell_0]$ over a range $[a..[(a+b)/2]]$ and the right child of the root is a WT for $L_1[1,\ell_1]$ over a range $[1+[(a+b)/2],..b]$.

Implementing WTs without pointers uses a small space of $n \lg (n+\sigma) + o(n \lg (n+\sigma))$ bits, but supporting the pushback operation is difficult. Thus, we implement DWTs using pointers by which the binary tree is explicitly represented. When a new symbol exceeding the representation ability of the current binary tree in the DWT is added to the DWT, the DWT adds new nodes to the binary tree, thus resulting in an increase in the height of the tree. The space used by a DWT is $(3n+2\sigma) \lg (n+\sigma) + o(n \lg (n+\sigma))$ bits. Figure 3.1 shows an example of a DWT.

3.2 Complexity of building the OESP-index

**Theorem 4.** The size of the OESP-index is $n \lg |S| + O((n+\sigma) \lg (n+\sigma))$ bits. Further, the construction time is $O\left(\frac{1}{\alpha}|S| \lg (n+\sigma) \lg^* |S|\right)$ and the memory consumption is the same.
as the index size, where \( S \) is an input string, \( n \) is the number of variables, and \( 1/\alpha \) is a load factor of the hash table; we assume the size of the alphabet is constant. The update time for the next input symbol is \( O\left(\frac{1}{\alpha} \log(n + \sigma) \log^* |S|\right) \).

Proof. The size required to represent the length array \( R \) is \( n \log |S| \) bits for \( n \) variables. The size of \( B \) is \( 2n + o(n) \) bits. And the size of \( L \) and \( H \) are \( O((n + \sigma) \log(n + \sigma)) \) bits each. We can access \( Z = H(XY) \) in \( O(1/\alpha) \) time with a load factor \( 1/\alpha > 1 \). The alphabet reduction is iterated through at most \( \log^* |S| \) times for each symbol. The time required to obtain the parent and left/right children of a node in the partial parse tree is \( O(\log(n + \sigma)) \) using the rank/select queries over the DWT for \( L \). Therefore, the construction time of the parse tree is \( O\left(\frac{1}{\alpha} |S| \log(n + \sigma) \log^* |S|\right) \). Analogously, the update time is clear. \( \square \)

3.3 Query search and substring extraction

Given a node \( v \) of the parse tree of string \( S \in \Sigma^* \), and \( \text{yield}(v_1 \cdots v_k) = \text{yield}(v_1) \cdots \text{yield}(v_k) \). \( \text{Label}(v) \) denotes the label of \( v \) and \( \text{Label}(v_1 \cdots v_k) = \text{Label}(v_1) \cdots \text{Label}(v_k) \). If \( \text{Label}(v) = X \), then \( \text{yield}(X) \) is identical to \( \text{yield}(v) \). \( \text{lca}(u, v) \) is the lowest common ancestor of nodes \( u \) and \( v \). For a pattern \( P \in \Sigma^* \), nodes \( \{v_1, \ldots, v_k\} \) such that \( \text{yield}(v_1 \cdots v_k) = P \) are called
embedding nodes of \( P \). For embedding nodes \( \{v_1, \ldots, v_k\} \), string \( Q = \text{Label}(v_1 \cdots v_k) \) is called an evidence of pattern \( P \). Since the trivial evidence \( Q \) identical to \( P \) always exists, the notion of evidence is well-defined. In addition, for embedding nodes \( \{v_1, \ldots, v_k\} \), a node \( z \) such that \( z = \text{lca}(v_1, v_k) \) is called an occurrence node of \( P \).

The following theorem states that we can find shorter evidence depending on \( |P| \).

**Theorem 5.** ([24]) There exists an evidence \( Q = Q_1 \cdots Q_t \) of \( P \) such that each \( Q_i \) is a maximal repetition or a symbol and \( t = O(\lg |P| \lg^* |S|) \).

The time required to find the evidence \( Q \) of pattern \( P \) is bounded by the construction time of the parsing tree of \( P \). In our data structure for OESP-index, the time required to find the evidence \( Q \) is estimated as described in the following theorem.

**Theorem 6.** The time required to find \( Q \) is \( O(\frac{1}{\alpha} |P| \lg(n + \sigma) \lg^* |S|) \).

*Proof.* This bound is clear by Theorem 4.

Let us consider the simple case that \( |Q_i| = 1 \) for any \( i \). In this case, \( Q \) contains no repetition such that \( Q = q_1 \cdots q_t \in \Sigma^t \). A symbol \( q_k \) is called a maximal core if \( |\text{yield}(q_k)| \geq |\text{yield}(q_i)| \) for any \( i \). For an internal node \( v \) of the parse tree \( T \) of \( S \) with \( \text{Label}(v) = q_k \), an ancestor \( z \) of \( v \) is the occurrence node of \( P \) iff all \( q_1, \ldots, q_{k-1} \) and \( q_{k+1}, \ldots, q_t \) can be embedded around \( v \). Moreover, any occurrence node of \( P \) is restricted by the case in which \( \text{Label}(v) = q_k \). For a general case \( Q_i = a^\ell (\ell \geq 2) \), i.e., \( Q_i \) is a repetition, we can reduce the embedding of \( a^\ell \) to the embedding of string \( AB \cdots C \) of length at most \( O(\lg \ell) \) such that \( \text{yield}(AB \cdots C) = a^\ell \). Thus, the embedding of a type1 string is easier than the others, and, without loss of generality, we can assume \( |Q_i| = 1 \) for any \( i \).

The remaining task of the search problem is the random access to all occurrences of maximal core \( q_k \) over the POPPT, i.e., the pruned parse tree. By the definition of POPPT, the internal node with rank \( k \) is the leftmost occurrence of symbol \( q_k \) itself. In previous indexes [24, 45], a next occurrence of \( q_k \) is obtained using a data structure
based on renaming variables in a lexicographical order; however, this data structure is not dynamically constructable. Therefore, we develop the search algorithm NextCore, presented as Algorithm 1 for our OESP-index.

The NextCore algorithm visits all occurrences of the maximal core on the parse tree $T$ using its implicit POPPT $T'$. When NextCore receives a candidate node $v$ that contains a maximal core $q$ as its descendant, it computes the pair $(u, p)$ where $u$ is the next occurrence of $v$ in $T'$ and $p$ is the path from $u$ to $q$. Therefore, $(u, p)$ indicates the occurrence of $q$ in the explicit parse tree. Below, we show the correctness of this algorithm and its complexity.

**Lemma 1.** The NextCore algorithm finds any occurrence of the maximal core exactly once. Furthermore, the amortized time to find a next occurrence is $O\left(\frac{\lg n \lg |S|}{\lg \lg n}\right)$.

**Proof.** Let $T$ be the parse tree and $T'$ be the POPPT $(B, L)$. By the definition of $T'$, any internal node $x$ of $B$ is the variable itself, i.e., $\text{Label}(x) = x$. For the maximal core $q$, let $v_1 > v_2 > \cdots > v_k$ be the post-order of its occurrences in $T$. We show that the algorithm finds any $v_i$ as $(u, p)$ by induction on $i$. Given $q$, an internal node $q$ of $B$ represents the leftmost occurrence of $q$ itself. Then, for the base case $i = 1$, the occurrence is obtained $v_1$ as $(q, p)$ with $|p| = 0$. Next, assume the induction hypothesis on some $i$. Since the node $v_{i+1}$ was pruned in $T'$, let $u$ be the leaf node of $T'$ that corresponds to the root node of the pruned maximal subtree containing $v_{i+1}$. For $\text{Label}(u) = u'$, there is the leftmost occurrence of $u'$ as an internal node of $B$. The subtree on the node $u'$ contains an occurrence of $q$ because the two subtrees on $u$ and $u'$ in $T$ are identical to one another. Let $p$ be the path from $u'$ to $v'$ for some $v' \in \{v_1, \ldots, v_i\}$. By the induction hypothesis, the algorithm finds $v'$ as $(u', p)$. Then, $v_{i+1}$ can also be found as $(u, p)$. On the other hand, if any $(u, p)$ is unique, then the algorithm finds any occurrence of $q$ exactly once. For the time complexity, the number of executed select operations is bounded by the number of different $(u, p)$, i.e., $O(\text{occ}_q \log |S|)$ where $\text{occ}_q$ is the number of occurrences of $q$. Each select operation on $L$ and parent operation on $B$ require $O(\lg(n + \sigma))$ and $O\left(\frac{\lg n}{\lg \lg n}\right)$ time,
respectively. Therefore, the total time is \( O(occ_q(\log(n + \sigma) + \frac{\log n \log |S|}{\log \log n})) = O(occ_q(\frac{\log n \log |S|}{\log \log n})) \) and the amortized time to find a next occurrence of \( q \) is \( O(\frac{\log n \log |S|}{\log \log n}) \).

**Theorem 7.** The counting/locating time of pattern and extraction time are \( O(\log(n + \sigma)(\frac{|P|}{\alpha} + occ_q(\log |S| + \log |P| \log^* |S|))) \) and \( O(\log(n + \sigma)(|P| + \log |S|)) \), respectively, where \( P \) is a query pattern and \( occ_q \) is the number of occurrences of the maximal core of \( P \) in the parse tree.

**Proof.** Since we can obtain the length of the substring encoded by any variable in \( O(1) \) time, the locating time is the same as the counting time. Given the pattern \( P \), as previously shown, the evidence \( Q \) of \( P \) is found in \( O(\frac{1}{\alpha}|P| \log(n + \sigma) \log^* |S|) \) time. For each occurrence of a maximal core, we can check if the sequence of symbols of length \( O(\log |P| \log^* |S|) \) is embedded around the core in \( O(\log(n + \sigma)(\log |S| + \log |P| \log^* |S|)) \) time. Therefore, by Lemma 1, the total counting time of the pattern is

\[
\begin{align*}
O \left( \frac{|P|}{\alpha} \log(n + \sigma) \log^* |S| + occ_q \log(n + \sigma)(\log |S| + \log |P| \log^* |S|) + occ_q \frac{\log n \log |S|}{\log \log n} \right) \\
= O(\log(n + \sigma)(\frac{|P|}{\alpha} + occ_q(\log |S| + \log |P| \log^* |S|))).
\end{align*}
\]

Conversely, for any \( S[i,j] \) of length \( m \), we can find \( S[i] \) in \( O(\log |S|) \) time and visit all leaf nodes in \( S[i,j] \) in \( O(|P|) \) time because the parsing tree is balanced. This follows the extraction time. \( \Box \)

### 3.4 Experiments

We evaluated the actual performance of our OESP-index using real data\(^1\). The environment we used was an Intel(R) Core(TM)i7-2620M CPU(2.7GHz) machine with 16GB memory. We used einstein.en.txt (einstein, 446 MB) and cere (cere, 440 MB), where einstein was highly repetitive.

\(^1\)http://pizzachili.dcc.uchile.cl/repcorpus/real/
Table 3.1: Index size in megabytes (MB).

<table>
<thead>
<tr>
<th></th>
<th>OESP-index</th>
<th>ESP-index</th>
<th>SLP-index</th>
<th>LZ-index</th>
<th>FM-index</th>
</tr>
</thead>
<tbody>
<tr>
<td>einstein</td>
<td>22.84</td>
<td>1.76</td>
<td>2.28</td>
<td>177.02</td>
<td>942.85</td>
</tr>
<tr>
<td>cere</td>
<td>364.92</td>
<td>27.40</td>
<td>45.74</td>
<td>438.05</td>
<td>806.52</td>
</tr>
</tbody>
</table>

Table 3.2: Working memory of dictionary \( D \) consisting of bit string \( B \) and dynamic wavelet tree \( L \) for einstein and cere.

<table>
<thead>
<tr>
<th>Size of text (MB)</th>
<th>50</th>
<th>100</th>
<th>150</th>
<th>200</th>
<th>250</th>
<th>300</th>
<th>350</th>
<th>400</th>
<th>all</th>
</tr>
</thead>
<tbody>
<tr>
<td>( B ) (MB)</td>
<td>0.04</td>
<td>0.07</td>
<td>0.07</td>
<td>0.07</td>
<td>0.14</td>
<td>0.14</td>
<td>0.14</td>
<td>0.14</td>
<td>0.14</td>
</tr>
<tr>
<td>( L ) (MB)</td>
<td>5.06</td>
<td>6.63</td>
<td>7.84</td>
<td>8.99</td>
<td>10.88</td>
<td>12.23</td>
<td>13.38</td>
<td>14.37</td>
<td>15.14</td>
</tr>
<tr>
<td>( B ) (MB)</td>
<td>1.10</td>
<td>1.10</td>
<td>1.10</td>
<td>2.20</td>
<td>2.20</td>
<td>2.20</td>
<td>2.20</td>
<td>2.20</td>
<td>2.20</td>
</tr>
<tr>
<td>( L ) (MB)</td>
<td>102.34</td>
<td>131.58</td>
<td>164.05</td>
<td>179.90</td>
<td>199.08</td>
<td>216.87</td>
<td>225.70</td>
<td>235.62</td>
<td>245.81</td>
</tr>
</tbody>
</table>

Comparative self-indexes we used were the offline version of ESP-index [45], other grammar-based self-index (i.e., SLP-index [1, 2]), LZ-based index (i.e., LZ-index)\(^2\), and BWT-based self-index (i.e., FM-index)\(^3\).

Figure 3.2 shows the required working memory (in MB) in response to increases in the size of the input string. For the offline algorithms, we evaluated the working memory for each static set of data with the indicated size. Figure 3.3 shows a breakdown of the required memory by the data structures of the OESP-index, including dictionary \( D \), length array \( R \), and hash table \( H \). Furthermore, Table 3.2 presents a breakdown of \( D \) by bit string \( B \) and wavelet tree \( L \).

Table 3.1 shows the size of the indexes for all methods. The size of the OESP-index is smaller than that of LZ-index and FM-index but larger than ESP and SLP indexes. We note here that the increase of index size arose from the DWT. Reducing this data size is important for future work.

The memory consumption of our OESP-index was smallest for both types of data. The required memory of the OESP-index, measured as a percentage of the offline ESP-index, was 2.5% for einstein and 40% for cere. The space efficiency of the OESP-index decreased

\(^2\)http://pizzachili.dcc.uchile.cl/indexes/LZ-index/LZ-index1
\(^3\)https://code.google.com/p/fmindex-plus-plus/
when the data set was not large and not highly repetitive, as illustrated in the right-hand portion of Figure 3.3. In particular, $L$, which is represented by the DWT, consumed a large space (as indicated by Table 3.2) due to the pointers and reservation spaces of the bit string of the DWT.

Figure 3.4 shows the construction times for the various indexes. From the figure, we observe that the OESP-index was slowest for both data sets across all methods. The OESP-index was 57.1 times (einstein) and 58.1 times (cere) slower than the ESP-index because the original ESP-index uses GMR [9], a faster wavelet tree algorithm that is not available in the online version.

Finally, Figure 3.5 shows the search times for each of the algorithms. Here, the search time represents the locating time since the counting time is almost the same as the locating time. We note that results for SLP-index is not shown because it could not work for this data set. The range of the length of the query pattern is $[10, 1000]$. The locating time of the OESP-index was slowest in both data sets for all query lengths. OESP-index was 163.2 times (cere) and 24.9 times (einstein) slower than the ESP-index, respectively.
Chapter 3. Structure of the OESP-Index

Figure 3.2: The required working memory for each method in MB for einstein (left) and cere (right).

Figure 3.3: The required working space of dictionary $D$, length array $R$, and hash table $H$ for einstein (left) and cere (right).

Figure 3.4: Construction times for each method in seconds for einstein (left) and cere (right).

Figure 3.5: Locating times for each method in milliseconds for einstein (left) and cere (right).
Chapter 4

Online pattern matching for string edit distance with moves

In this chapter, we expand the search algorithm based on ESP from exact matches to ambiguous pattern searches for edit distance with moves (EDM). Note that we refer to [46]. Furthermore, ESP differs from our other chapters as follow: (1) the number of alphabet reduction iterations of ESP is restricted to one iteration and (2) we use 3-trees ($X \rightarrow ABC$) instead of 2-2-trees ($X \rightarrow AY, Y \rightarrow BC$). Based on these differences, we use the special CFG and phrase/reverse dictionaries described in the subsections that follows.

4.1 Context-free grammar

A context-free grammar (CFG) is represented by a quadruple $G = (\Sigma, V, D, Z_s)$ where $V$ is a finite subset of $\mathcal{X}$, $D$ is a finite subset of $V \times (V \cup \Sigma)^*$ of production rules, and $Z_s \in V$ represents the start variable. Note that $D$ is also called a phrase dictionary, and variables in $V$ are called nonterminal symbols. The set of strings in $\Sigma^*$ derived from $Z_s$ by $G$ is denoted as $L(G)$. Furthermore, a CFG $G$ is called admissible if for any $Z \in \mathcal{X}$, there is exactly one production rule $Z \rightarrow \gamma \in D$. We assume $|\gamma| = 2$ or 3 for any production rule $Z \rightarrow \gamma$. 
The parse tree of $G$ is represented as a rooted ordered tree with internal nodes labeled by variables in $V$ and leaf nodes labeled by elements in $\Sigma$; here, the label sequence of leaf nodes of the parse tree is equal to an input string. Any internal node $Z \in V$ in a parse tree corresponds to a production rule in the form of $Z \rightarrow \gamma$ in $D$. Finally, the height of $Z$ is the height of the subtree whose root is $Z$.

### 4.2 Phrase and reverse dictionaries

For a set $V$ of production rules, a **phrase dictionary** $D$ is a data structure used to directly access phrases $S \in (\Sigma \cup V)^*$ for any given $Z \in V$ if $Z \rightarrow S \in D$. Furthermore, a **reverse dictionary** $D^{-1} : (\Sigma \cup V)^* \rightarrow V$ is a mapping from a given sequence of symbols to a variable. $D^{-1}$ returns a variable $Z$ associated with a string $S$ if $Z \rightarrow S \in D$; otherwise, it creates a new variable $Z' \notin V$ and returns $Z'$. As an example, if $D = \{Z_1 \rightarrow abc, Z_2 \rightarrow cd\}$, $D^{-1}(a, b, c)$ returns $Z_1$, whereas $D^{-1}(b, c)$ creates $Z_3$ and returns it.

### 4.3 Problem definition

To describe our method, we first review the notion of the EDM. The EDM $d(S, Q)$ between two strings $S$ and $Q$ is the minimum number of edit operations required to transform $S$ into $Q$. The edit operations are as follows:

1. **Insertion**: A character $a$ at position $i$ in $S$ is inserted, which generates $S[1, i - 1]aS[i]S[i + 1, |S|]$,

2. **Deletion**: A character $a$ at position $i$ in $S$ is deleted, which generates $S[1, i - 1]S[i + 1, |S|]$,

3. **Replacement**: A character at position $i$ is replaced by $a$, which generates $S[1, i - 1]aS[i + 1, |S|]$,

**Problem 1** (Online pattern matching for EDM). *For a streaming text $S \in \Sigma^*$, a query $Q \in \Sigma^*$, and a distance threshold $k \geq 0$, find all $i \in [1, |S|]$ such that the EDM between a substring $S[i,i + |Q|]$ and $Q$ is at most $k$, i.e., $d(S[i,i + |Q|], Q) \leq k$.*

Cormode and Muthukrishnan [4] presented an offline algorithm for computing the EDM. In their algorithm, a special derivation tree called an ESP tree was constructed to approximately compute the EDM. We present an online variant of the ESP. Our algorithm approximately solves Problem 1 and achieves the following: (1) the space-efficient online construction of a parse tree and (2) the approximate computation of the EDM from the parse tree. Although our method is an approximation algorithm, it guarantees an upper bound for the exact EDM. These two parts of our algorithm are described in the next subsection.

### 4.4 Online Algorithm

OESP builds a special form of the CFG and directly encodes it into a succinct representation in an online manner. Such a representation can be used as space-efficient phrase/reverse dictionaries, which thereby reduce the working space. In this section, we first present a simple variant of ESP to introduce the notion of *alphabet reduction* and *landmark*. We then detail our OESP and its approximate online computations of the EDM. In the next section, we present an upper bound of the approximate EDM for the exact EDM.

#### 4.4.1 Post-order CFG

OESP builds a POPPT and directly encodes it into a succinct representation.
CHAPTER 4. ONLINE PATTERN MATCHING FOR STRING EDIT DISTANCE WITH MOVES

Figure 4.1: Examples of (I) a POCFG, (II) the parse tree, (III) a post-order partial parse tree (POPPT), and (IV) the succinct representation of a POPPT.

Definition 2 (POCFG [27]). A post-order CFG (POCFG) is a CFG whose partial parse tree is a POPPT.

Note that the number of nodes in the POPPT is at most $3n$ for a POCFG of $n$ variables because the right-hand sides consist of digrams or trigrams in the production rules, and the numbers of internal nodes and leaves are $n$ and at most $2n$, respectively.

Examples of a POCFG and POPPT are shown in Figures 4.1-i) and iii), respectively. The POPPT is built by traversing the parse tree in Figure 4.1-ii) in a depth-first manner and pruning out all descendants under the node with the second $X_3$. The resulting POPPT in Figure 4.1-iii) consists of internal nodes with post-order variables.

A major advantage of the POPPT is that we can directly encode it into a succinct representation that can then be used as a phrase dictionary. Such a representation enables us to reduce the working space of our OESP by using it in a combination with a reverse dictionary.

4.4.2 Online construction of a POCFG

From a given input string, OESP constructs a POCFG that guarantees upper bounds on parsing discrepancies between the same substrings in the string. The basic idea of our OESP is to (1) start from symbols in an input text, (2) replace as many identical digrams or trigrams as possible in common substrings by the same nonterminal symbols, and (3)
Algorithm 2 Online construction of ESP. $D$ is phrase dictionary, $D^{-1}$ is reverse dictionary, and $q_k$ is queue at level $k$.

```plaintext
1: function OESP
2:   $D := \emptyset$; initialize queues $q_k$
3: while reading a new character $c$ from an input text do
4:   ProcessSymbol($q_1$, $c$)
5: end while
6: end function
7: function ProcessSymbol($q_k$, $X$)
8:   $q_k$.enqueue($X$)
9: if $q_k$.size() = 4 then
10:   if $L(q_k, 2) = 0$ then  \(\triangleright\) Build a 2-tree
11:     $Z := D^{-1}(q_k[3], q_k[4])$; $D := D \cup \{Z \rightarrow q_k[3]q_k[4]\}$
12:     ProcessSymbol($q_{k+1}$, $Z$)
13:     $q_k$.dequeue(); $q_k$.dequeue()
14: end if
15: else if $q_k$.size() = 5 then
16:   \(\triangleright\) Build a 3-tree
17:     $Z := D^{-1}(q_k[3], q_k[4], q_k[5])$; $D := D \cup \{Z \rightarrow q_k[3]q_k[4]q_k[5]\}$
18:     ProcessSymbol($q_{k+1}$, $Z$)
19:     $q_k$.dequeue(); $q_k$.dequeue(); $q_k$.dequeue()
20: end if
21: end function
```

iterate this process in a bottom-up manner until it generates a complete POCFG. Note that the POCFG is constructed online, and the POPPT corresponding to it consists of nodes with two or three children.

From strings $XY$ and $XYZ$, OESP builds two types of subtrees in a POPPT. The first type is a 2-tree corresponding to a production rule in the form $Z \rightarrow XY$. The second type is a 3-tree corresponding to a production rule in the form $Z \rightarrow WXY$.

Here, OESP constructs a 2-tree or 3-tree subtree from a substring of limited length. Let $u$ be a string of length $m$. A function $L : (\Sigma \cup V)^m \times [m] \rightarrow \{0, 1\}$ classifies whether or not the $i$-th position of $u$ has a landmark, i.e., the $i$-th position of $u$ has a landmark if $L(u, i) = 1$. Here, $L(u, i)$ is computed from a substring $u[i-1, i+2]$ of length 4. Similarly, OESP builds a 3-tree from a substring $u[i+1, i+3]$ of length 3 if the $i$-th position of $u$ does not have a landmark; otherwise, it builds a 2-tree from a substring $u[i+2, i+3]$ of length 2. The landmarks on a string are decided such that they are synchronized in long common subsequences to ensure the parsing discrepancies as small as possible.

Our algorithm uses a set of queues, $q_k(k = 1, \ldots, m)$, where $q_k$ processes the string at the $k$-th level of a parse tree of a POCFG and builds 2-trees and 3-trees at each $k$. Since OESP builds a balanced parse tree, the number of these queues $m$ is bounded by $\lg |S|$. 

In addition, landmarks are decided on strings of length at most 4, and the length of each queue is also fixed at 5. Algorithm 2 shows the OESP and ProcessSymbol functions.

The main function here is OESP, which reads new characters from an input text and passes them to the ProcessSymbol function one by one. The ProcessSymbol function builds a POCFG in a bottom-up manner. There are two cases according to whether or not a queue $q_k$ has a landmark. For the first case in which $\mathcal{L}(q_k, 2) = 0$, i.e., $q_k$ does not have a landmark, the 2-tree corresponding to a production rule $Z \rightarrow q_k[3]q_k[4]$ in a POCFG is built for the third and fourth elements $q_k[3]$ and $q_k[4]$ of the $k$-th queue $q_k$, respectively. For the other case, the 3-tree corresponding to a production rule $Z \rightarrow q_k[3]q_k[4]q_k[5]$ is built for the third, fourth and fifth elements $q_k[3]$, $q_k[4]$ and $q_k[5]$, respectively, of the $k$-th queue $q_k$. In both cases, the reverse dictionary $D^{-1}$ returns a nonterminal symbol that replaces a sequence of symbols. The generated symbol $Z$ is given to the higher $q_{k+1}$, which enables the bottom-up construction of a POCFG in an online manner.

The computation time and working space here depend on the implementations of phrase and reverse dictionaries. The phrase dictionary for a POCFG of $n$ variables can be implemented using a standard array of at most $3n \lg(n + \sigma)$ bits of space and $O(1)$ access time. In addition, the reverse dictionary can be implemented using a chaining hash table and a phrase dictionary implemented as an array. Therefore, the working space of OESP using these data structures is at most $n(4 + \alpha) \lg(n + \sigma)$ bits. In the following subsections, we present space-efficient representations of phrase/reverse dictionaries.

### 4.4.3 Compressed phrase dictionary

OESP directly encodes a POCFG into a succinct representation that consists of bit strings $B$, $P$ and a label sequence $L$. A bit string $B$ is built by traversing a POPPT and generating $c$ zeros and one for a node with $c$ children in the post-order. The final zero in $B$ represents the super node. We call the bit string representation of a POPPT the posterior order.
unary degree sequence (POUDS). To dynamically build a tree and access any node in the POPPT, we index $B$ using the dynamic range min/max tree [33]. Our POUDS supports the following two tree operations: (1) $\text{child}(B, i, j)$ returns the $j$-th child of a node $i$ and (2) $\text{num}_\text{child}(B, i)$ returns the number of children associated with a node $i$. These are both computed in $O(\lg m/\lg \lg m)$ time and use $2m + o(1)$ bits of space for a tree with $m$ nodes.

Next, a bit string $P$ is constructed by traversing a POPPT and generating one for a leaf node and zero for an internal node in post-order. Here, $P$ is indexed by the rank/select dictionary [15, 32]. The label sequence $L$ stores symbols corresponding to the leaf nodes in a POPPT.

We can access any element in $L$ as a child of node $i$ using the following approach. First, we compute $c = \text{num}_\text{child}(B, i)$ and children nodes $p = \text{child}(B, i, j)$ for $j \in [1, c]$. Then, we compute the positions in $L$ corresponding to the positions of these children as $q = \text{rank}_1(P, p)$ that return the number of occurrences of one in $P[0, p]$ in $O(1)$ time. We obtain leaf labels as $L[q]$. For a POCFG of $n$ nonterminal symbols, we can access the right-hand side of the symbols from the left-hand side of a symbol of a production rule in $O(\lg n/\lg \lg n)$ time while using at most $n \lg (n + \sigma) + 5n + o(n)$ bits of space.

### 4.4.4 Compressed reverse dictionary

We implement a reverse dictionary using a chaining hash table with a load factor $1/\alpha > 1$ in combination with the phrase dictionary. The hash table has $\alpha n$ entries and each entry stores a list of integers $i$ that represent the left-hand side $X_i$ of a rule. For the rule $X_i \rightarrow S$, the hash value is computed from the right-hand side $S$. Then, the list corresponding to the hash value is scanned to search for $X_i$ while checking elements referred to as $S$ in a phrase dictionary. Therefore, the expected access time is $O(1/\alpha)$. The space for a POCFG with $n$ nonterminal symbols is $\alpha n \lg(n + \sigma)$ bits for the hash table and $n \lg(n + \sigma)$ bits for the lists, which result in $n(\alpha + 1) \lg(n + \sigma)$ bits in total.
A crucial observation regarding our OESP is that indexes $i$ for nonterminal symbols $X_i$ are created in a strictly increasing order. Therefore, we can organize each list in a hash table as a strictly increasing sequence of indexes of nonterminal symbols. We insert a new index $i$ into a list in the hash table, then append it at the end of the list. Each list in the hash table consists of a strictly increasing sequence of indexes. To make each index smaller, we compute the difference between an index $i$ and the previous one $j$, and we encode it by the delta code, which results in the difference $i - j$ being encoded in $1 + \lfloor \lg (i - j) \rfloor + 2 \lfloor 1 + \lg (i - j) \rfloor$ bits. For all $n$ nonterminal symbols, the space required for the lists has an upper bounded by $n(1 + \lg (\alpha n) + 2 \lg \lg (\alpha n))$ bits. The space required for the hash table is $\alpha n \lg (n + \sigma + n(1 + \lg (\alpha n) + 2 \lg \lg (\alpha n)))$ bits in total, resulting in $\alpha n \lg (n + \sigma) + n(1 + \lg (\alpha n))$ bits by multiplying the original $\alpha$ by a constant.

Since the reverse dictionary is implemented using the chaining hash and the phrase dictionary, its total required space is at most $n(\alpha + 1) \lg (n + \sigma) + n(5 + \lg (\alpha n)) + o(n)$ bits. We can obtain the following result.

**Lemma 2.** For a string of length $N$, OESP constructs a POCFG of $n$ nonterminal symbols and its corresponding phrase/reverse dictionaries in $O(\frac{N \lg n}{\alpha \lg \lg n})$ expected time using at most $n(\alpha + 1) \lg (n + \sigma) + n \lg (\alpha n) + 5n + o(n)$ bits of space.

### 4.4.5 Online pattern matching with EDM

We approximately solve Problem 1 by using our OESP. First, the parse tree is computed from a query $Q$ using OESP. Let $T(Q)$ be a set of node labels in the parse tree for $Q$. We then compute a vector $V(Q)$, each dimension $V(Q)(e)$ of which represents the frequency of corresponding node label $e$ in $T(Q)$.

OESP constructs another parse tree for a streaming text $S$ in an online manner. Here, $T(S)[i, i + |Q|]$ is a set of node labels included in the subtree corresponding to a substring $S[i, i + |Q|]$ from $i$ to $i + |Q|$ in $T(S)$. $V(S)[i, i + |Q|]$ can be constructed for each $i \in [1, |S| - |Q|]$ by adding the node labels that correspond to $S[i, i + |Q|]$ and
substituting the node labels not included in $T(S)[i, i + |Q|]$ from $V(S)[i, i + |Q|]$, which can be performed in $\lg |S|$ time.

The $L_1$-distance approximates the EDM between $V(S)[i, i + |Q|]$ and $V(Q)$, and it is computed as $$||V(S)[i, i + |Q|] - V(Q)|| = \sum_{e \in (T(S)[i, i + |Q|] \cup T(Q))} |V(S)[i, i + |Q|](e) - V(Q)(e)|.$$ We thereby obtain the theorem below with respect to computational time and space for computing the $L_1$ distance from Lemma 2.

**Theorem 8.** For a streaming text $S$ of length $N$, OESP approximately solves Problem 1 in $O(N \frac{\sigma \lg n}{n \lg \sigma n})$ expected time using at most $n(\alpha + 1) \lg (n + \sigma) + n \lg (\alpha n) + 5n + o(n)$ bits of space.

### 4.5 An upper bound on our approximation

In this section, we present an upper bound on the approximation of the EDM.

**Theorem 9.** $||V(S) - V(Q)|| = O(\lg^2 m)d(S, Q)$ for any $S, Q \in \Sigma^*$ and $m = \max\{|S|, |Q|\}$.

**Proof.** Let $e_1, e_2, \ldots, e_d$ be the shortest series of editing operations such that $S_{k+1} = S_k(e_k)$ where $S_1 = S$, $S_d(e_d) = Q$, and $d = d(S, Q)$. It is sufficient to prove the assumption that there exists a constant $c$ such that $||V(S) - V(Q)|| \leq c \lg^2 m$ for $R(e) = S$. Here, $S(i)$ denotes the string generated by the $i$-th iteration of ESP, where $S(0) = S$. Let $p_i, q_i$ be the smallest integers satisfying $S(i)[p_i] \neq Q(i)[p_i]$ and $S(i)[|S(i)| - q_i] \neq Q(i)[|Q(i)| - q_i]$, respectively. We show that $q_i - p_i \leq \lg m + 1$ for each height $i$, which derives $||V(S) - V(Q)|| \leq 2 \lg m(\lg m + 1)$ because $i \leq \lg m$.

We begin with a case in which $e$ is an insertion of a symbol. Clearly, it is true for $i = 0$ since $q_0 - p_0 \leq 1$. We next assume the hypothesis on some height $i$. Let $S(i)[p]$ be the closest landmark from $S(i)[p_i]$ with $p < p_i$ and $S(i)[q']$ be the closest landmark from $S(i)[q_i]$ with $q_i < q'$. For the next height, let $S(i + 1) = S_1S_2S_3$ such that the tail of $S_1$ derives $S(i)[p_i]$ and the tail of $S_2$ derives $S(i)[q_i]$, and let $Q(i + 1) = Q_1Q_2Q_3$ such that $|Q_1| = |S_1|$ and $|Q_3| = |S_3|$. On any iteration of ESP, the left aligned parsing is performed
from a landmark to its closest landmark. It then follows that for $S_1$, $S_1[j] = Q_1[j]$, except for their tails, i.e., $S_2, |S_2| \leq \lceil \frac{1}{2} (q_i - p_i) \rceil \leq \lfloor \frac{1}{2} (\log m + 1) \rfloor$, and for $S_3$, we can estimate $S_3[j] = Q_3[j]$ for any $j > \lfloor \frac{1}{2} \log m \rfloor$. Therefore, $q_{i+1} - p_{i+1} \leq \lfloor \frac{1}{2} (\log m + 1) \rfloor + \lfloor \frac{1}{2} \log m \rfloor \leq \log m + 1$. Since $d(S, Q) = d(Q, S)$, this bound is true for the deletion of any symbol. The case in which $e$ is a replacement operation is similar.

The bound also holds for the case of insertion or deletion of any string of length at most $\log m$. Using this, we reduce the case of a move operation of a substring $u$ as follows. Without loss of generality, we assume that $u$ is a type 2 substring and let $u = xyz$ such that $x/z$ is the shortest prefix/suffix of $u$ that contains a landmark, respectively. We then note that the $y$ inside of $u$ is transformed into the same string for any occurrence of $u$. Therefore, the case of moving $u$ from $S$ to obtain $Q$ is reduced to the case of deleting $x/z$ at some positions and inserting them into other positions. Since $|x|, |z| \leq \log m$, the case of moving $u$ is identical to the case of inserting two symbols and deleting two symbols, i.e., $\|V(S) - V(Q)\| \leq 8 \log m (\log m + 1)$.

From Theorems 8 and 9 above, we obtain the following main theorem.

**Theorem 10.** EDM is $O(\log^2 N)$-approximable by our proposed online algorithm with $O(\frac{N \log N}{\alpha \log \log n})$ expected time and $n(\alpha + 1) \log (n + \sigma) + n(5 + \log (\alpha n)) + o(n)$ bits of space.

**Proof.** By the theorem 9, we obtain the bound $\|V(S[i, i+|Q|] - V(Q))\| = O(\log^2 |Q|)d(S[i, i+|Q|], Q)$ for any $i \in [1, |S| - |Q|]$. The time complexity is proved by Theorem 8. Thus, for strings $S$ and $Q$ with $N = |S| \geq |Q|$, the result is concluded.

### 4.6 Experiments

We evaluated our OESP method on one core of an eight-core Intel Xeon CPU E7-8837 (2.67GHz) machine with 1024GB memory. We used two standard benchmark texts, i.e., dna.200MB and english.200MB, which are downloadable from [http://pizzachili.dcc](http://pizzachili.dcc).
Table 4.1: Space requirements for POUDS $B$, label sequence $P$ and bit string $P$ organizing a dictionary on dna.200MB and english.200MB.

\begin{tabular}{l|c|c|c}
\hline
\hline
dna.200MB & 89.95 & 17.62 & 7.73 \\
english.200MB & 95.72 & 14.99 & 8.22 \\
\hline
\end{tabular}

uchile.cl/texts.html. We sampled texts of length 100 from these texts as queries. We also used computation time and working space as evaluation measures.

Figure 4.2 shows computation times as we increased the length of the text. As shown in the figure, the computation time increased linearly with the length of text.

Similarly, Figure 4.3 shows the working space for as we increased the length of the text. The space of the dictionary was much smaller than that of the hash table. More specifically, the dictionary used 115MB for dna.200MB and 121MB for english.200MB, whereas the hash table used 368MB for dna.200MB and 382MB for english.200MB.

Figure 4.4 shows the space required for the POUDS, a label sequence, and a bit string organizing a dictionary as we increased the length of the text. The space required for the bit string of the dictionary was much smaller than that of the label sequence for both dna.200MB and english.200MB. Table 4.1 provides details of those space requirements.

Finally, Figure 4.5 shows the number of substrings for which the EDMs for a query is at most a distance threshold. There were distance thresholds where the number of substrings dramatically increased.
Figure 4.2: Computation time in seconds versus the length of the text.

Figure 4.3: Working space of the dictionary and hash table versus the length of text.

Figure 4.4: Working space of a POUDS (B), a label sequence (L) and a bit string (P) which combine to organize a dictionary.

Figure 4.5: The number of substrings whose EDM to a query is no more than each threshold.
Chapter 5

siEDM: an efficient string index and search algorithm for edit distance with moves

In this chapter, we present an offline string index for the fast computation of OESP-based problem. Note that this chapter refers to [43].

5.1 Problem

Problem 2 (Query search for EDM). For a string $S \in \Sigma^*$, a query $Q \in \Sigma^*$ and a distance threshold $\tau \geq 0$, find all $i \in [1, |S|]$ that satisfy $d(S[i, i + |Q| - 1], Q) \leq \tau$.

In [41], Shapira and Storer proved the NP-completeness of the EDM problem and proposed a polynomial-time algorithm for a restricted EDM. In [4], Cormode and Muthukrishnan presented an approximation algorithm called ESP for computing the EDM. Based on these works, we present a string index and search algorithm that leverage the idea behind ESP to solve Problem 2. Our method consists of the following two-key parts: (1) an efficient index structure for a given string $S$ and (2) a fast algorithm to search for query $Q$ in the index structure of $S$ with respect to the EDM. Although our method is
also an approximation algorithm, it guarantees upper and lower bounds in relation to the exact EDM.

5.2 Approximate computations of the EDM from ESP-trees

ESP-trees enable us to approximately compute the EDM for two given strings. After constructing ESP-trees for two strings, their characteristic vectors are defined as follows. Let $T(S)$ be the ESP-tree for string $S$. We define an integer vector $F(S)$ as the characteristic vector if $F(S)(X)$ represents the number of times the variable $X$ appears in $T(S)$ as the root of a 2-tree. For a string $S$, $T(S)$ and its characteristic vector are illustrated in Figure 5.1. The EDM between two strings $S$ and $Q$ can then be approximated by the $L_1$-distance between two characteristic vectors $F(S)$ and $F(Q)$ as follows:

\[ \|F(S) - F(Q)\|_1 = \sum_{e \in V(S) \cup V(Q)} |F(S)(e) - F(Q)(e)| \]

In [4], Cormode and Muthukrishnan showed the upper and lower bounds on the $L_1$-distance between characteristic vectors for the exact EDM, which we repeat in the following theorem.

**Theorem 11** (Upper and lower bounds of the approximated EDM [4]). For $N = \max(|S|, |Q|)$,

\[ d(S, Q) \leq 2\|F(S) - F(Q)\|_1 = O(\log N \log^* N)d(S, Q) \]
CHAPTER 5. SIEDM: AN EFFICIENT STRING INDEX AND SEARCH ALGORITHM FOR EDIT DISTANCE

5.3 Index structure of ESP-trees

5.3.1 Efficient encoding scheme

The siEDM encodes an ESP-tree built from a string for fast query searches. This encoding scheme sorts production rules of an ESP-tree such that the left symbols on the right hand side of the production rules are in a monotonically increasing order, thus enabling the efficient encoding of these production rules and supporting fast operations for ESP-trees. Note that the encoding scheme is performed from the first (i.e., leaf node) and second levels to the top level (i.e., root node) in an ESP-tree.

Here, the set of production rules at the first and second levels in the ESP-tree are sorted in an increasing order of the left symbols on the right hand side of the production rules, i.e., $X_{l(i)}$ in the form $X_i \rightarrow X_{l(i)}X_{r(i)}$, which results in a sorted sequence of these production rules. The variables in the left hand side in the sorted production rules are renamed in the sorted order, thus generating a set of new production rules assigned to the corresponding nodes in the ESP-tree. We apply the same scheme to the next level of the ESP-tree and iterate until we reach the root node.

Figure 5.2 shows an example of the encoding scheme used for the ESP-tree built from an input string $S = babababaaba$. At the first and second levels in the ESP-tree, a set of production rules, $\{X_1 \rightarrow ab, X_2 \rightarrow bX_1, X_3 \rightarrow aa, X_4 \rightarrow ba\}$, is sorted in lexicographical order.
order of the left symbols on the right hand side of the production rules, thus resulting in the sequence of production rules \((X_1 \rightarrow ab, X_3 \rightarrow aa, X_2 \rightarrow bX_1, X_4 \rightarrow ba)\). As noted above, the variables on the right hand side of the production rules are renamed in sorted order and thus result in a new sequence \((X_1 \rightarrow ab, X_2 \rightarrow aa, X_3 \rightarrow bX_1, X_4 \rightarrow ba)\), whose production rules are assigned to the corresponding nodes in the ESP-tree. In this example, this scheme is repeated until it reaches level 4.

![Figure 5.2](image-url)

**Figure 5.2**: Illustration of the encoding scheme for the ESP tree built from input string \(S = \text{babababaaba}\).

Using the above encoding scheme, we obtain a monotonically increasing sequence of left symbols on the right hand side of the production rules, *i.e.*, \(X_{l(i)}\) in the form of \(X_i \rightarrow X_{l(i)}X_{r(i)}\). Let \(A_t\) be the increasing sequence; \(A_t\) can be efficiently encoded into a bit string by using gap encoding and unary coding. For example, gap encoding represents a sequence \((1, 1, 3, 5, 8)\) by \((1, 0, 2, 2, 3)\), which is further transformed into the bit
CHAPTER 5. SIEDM: AN EFFICIENT STRING INDEX AND SEARCH ALGORITHM FOR EDIT DISTANCE

string $0^110^010^210^210^31 = 0110010010001$ using unary coding. In general, for a sequence $(x_1, x_2, \ldots, x_n)$, its unary code $U$ represents $x_i$ by $\text{rank}_0(U, \text{select}_1(U, i))$. Because the number of zeros and ones is $n + \sigma$ and $n$, respectively, the size of $U$ is $2n + \sigma$ bits. Note that the bit string is indexed by the rank/select dictionary.

Note, let $A_r$ be a sequence consisting of the right symbols on the right hand side of the production rules, i.e., $X_r(i)$ in the form $X_i \rightarrow X_{l(i)}X_{r(i)}$. $A_r$ is represented using $(n + \sigma) \log (n + \sigma)$ bits and is indexed by GMR [9].

The space required to store $A_l$ and $A_r$ is $(n + \sigma) \log (n + \sigma) + 2n + \sigma + o((n + \sigma) \log (n + \sigma))$ bits in total. Here, $A_l$ and $A_r$ enable us to simulate fast queries on encoded ESP-trees, which we present further in the next subsection.

5.3.2 Query processing on the ESP-tree

The encoded ESP-trees support four tree operations, i.e., $\text{LeftChild}$, $\text{RightChild}$, $\text{LeftParents}$ and $\text{RightParents}$, which are used in our search algorithm. $\text{LeftChild}(X_k)$ returns the left child $X_{l(k)}$ of $X_k$ and can be implemented on bit string $A_l$ in $O(1)$ time as $m = \text{select}_1(A_l, X_k)$ and $\text{LeftChild}(X_k) = m - X_k$. $\text{RightChild}(X_k)$ returns the right child $X_{r(k)}$ of $X_k$ and can be implemented on array $A_r$ in $O(\log \log (n + \sigma))$ time as $X_j = \text{access}(A_r, X_k)$.

Furthermore, $\text{LeftParents}(X_k)$ and $\text{RightParents}(X_k)$ return sets of parents of $X_k$ as left and right children, respectively, i.e., $\text{LeftParents}(X_k) = \{X_i \in V : X_i \rightarrow X_kX_j, \forall X_j \in (\Sigma \cup V)\}$ and $\text{RightParents}(X_k) = \{X_i \in V : X_i \rightarrow X_jX_k, \forall X_j \in (\Sigma \cup V)\}$.

Because $A_l$ is a monotonic sequence, any $X_k$ appears consecutively in $A_l$. Using the unary encoding of $A_l$, $\text{LeftParents}(X_k)$ is computed by $\{p+i : p = \text{select}_1(A_l, X_k), \text{rank}_0(A_l, p+ i) = \text{rank}_0(A_l, p)\}$ in $O(|\text{LeftParents}(X_k)|)$ time. $\text{RightParents}(X_k)$ can be computed by repeatedly applying select operations for $X_k$ on $A_r$ until no more $X_k$ appears, i.e., $\text{select}_{X_k}(A_r, p)$ for $1 \leq p \leq n$. Thus, $\text{RightParents}(X_k)$ for $X_k \in V$ can be computed in $O(|\text{RightParents}(X_k)|)$ time.
5.3.3 Other data structures

As a supplemental data structure, siEDM computes the node characteristic vector, denoted by $F(X_i)$, for each variable $X_i$ as the characteristic vector that consists of the frequency of any variable derived from $X_i$. The space for storing all node characteristic vectors of $n$ variables is at most $n^2 \log |S|$ bits. Figure 5.2-(V) shows an example of the node characteristic vectors for the ESP-tree in Figure 5.2-(III). In addition, let $V(X_i)$ be a set of $X_i$ and variables that appear in all descendant nodes under $X_i$, i.e., $V(X_i) = \{ e \in (V \cup \Sigma) : F(X_i)(e) \neq 0 \}$. Practically, $F(X_i)$ is represented by a sequence of a pair of $X_j \in V(X_i)$ and $F(X_i)(X_j)$. In addition, because $F(X_i) = F(\text{LeftChild}(X_i)) + F(\text{RightChild}(X_i)) + (X_i, 1)$ (+$(X_i, 1)$ represents adding one to dimension $X_i$), the characteristic vectors can be stored per level 2 of the ESP-tree. The data structure here is represented by a bit array $FB$ indexed by a rank/select dictionary and the characteristic vectors reduced per level 2 of ESP-tree. $FB$ is set at one for the $i$-th bit if $F(X_i)$ is stored; otherwise it is zero. Then, $F(X_i)$ can be computed by the $\text{rank}_1(FB, i)$-th characteristic vector if the $i$-th bit of $FB$ is one; otherwise, $F(\text{LeftChild}(X_i)) + F(\text{RightChild}(X_i)) + (X_i, 1)$.

Another data structure that siEDM uses is a non-negative integer vector called length vector, each dimension of which is the length of the substring derived from the corresponding variable (See Figure 5.2-(VI)). The space for storing length vectors of $n$ variables is $n \log |S|$ bits.

From the above argument, the required space for the siEDM’s index structure for $n$ variables

$$n(n + 1) \log |S| + (n + \sigma) \log (n + \sigma) + 2n + \sigma + o((n + \sigma) \log (n + \sigma)) \text{ bits in total.}$$
5.4 Search algorithm

5.4.1 Baseline search algorithm

Given a \( T(S) \), the maximal subtree decomposition of \( S[i,j] \) is a sequence \( (X_1, X_2, \ldots, X_m) \) of variables in \( T(S) \) defined recursively as follows. \( X_1 \) is the variable of the root of the maximal subtree that satisfies that \( S[i] \) is its leftmost leaf node and \( |val(X_1)| \leq j - i \). If \( val(X_1) = S[i,j] \), then \( (X_1) \) is the maximal subtree decomposition of \( S[i,j] \). Otherwise, let \( X_1, X_2, \ldots, X_m \) be already determined and \( |val(X_1)val(X_2) \cdots val(X_m)| = k < j - i \). Then, let \( X_{m+1} \) be the variable of the root of the maximal subtree that satisfies that \( S[i + k + 1] \) is its leftmost leaf node and \( |val(X_{m+1})| \leq j - i - k \). By repeating this process until \( val(X_1)val(X_2) \cdots val(X_m) = S[i,j] \), the maximal subtree decomposition is determined.

Based on the maximal subtree decomposition, we explain the outline of the baseline algorithm, which is called the online ESP [46], to compute an approximation of the EDM between two strings. First, \( T(S) \) is constructed beforehand. Given a pattern \( Q \), the online ESP computes \( T(Q) \), and for each substring \( S[i,j] \) of length \( |Q| \), it computes the approximate EDM by computing the maximal subtree decomposition \( (X_1, X_2, \ldots, X_m) \) of \( S[i,j] \). Then, the distance \( \|F(Q) - F(S[i,j])\|_1 \) is approximated by \( \|F(Q) - \sum_{k=1}^{m} F(X_k)\|_1 \) because ESP-tree is balanced and then \( \|F(S[i,j]) - \sum_{k=1}^{m} F(X_k)\|_1 = O(\lg m) \). This baseline algorithm is, however, required to compute the characteristic vector of \( S[i,j] \) at each position \( i \). In the next subsection, we improve the time and space requirements of the online ESP by finding \( |Q| \)-grams for each variable \( X \) in \( V(S) \) instead of each position \( i \).

5.4.2 Improved search algorithm

The siEDM approximately solves Problem 2 with the same guarantees presented in Theorem 11. Let \( X_i \in V(S) \) be such that \( |val(X_i)| > |Q| \). There are \( |Q| \)-grams formed
by the string $suf(val(X_{i(i)}),|Q| - k)pre(val(X_{r(i)}), k)$ with $k = 1, 2, \ldots , (|Q| - 1)$. Then, the variable $X_i$ is said to stab the $|Q|$-grams, and the set of $|Q|$-grams stabbed by $X_i$ is denoted as $itv(X_i)$. Let $itv(S)$ be the set of $itv(X_i)$ for all $X_i$ that appear in $T(S)$. Note that $itv(S)$ includes any $|Q|$-gram in $S$. Using this characteristic, we can reduce the search space as follows.

If a $|Q|$-gram $R$ is in $itv(X_i)$, then there exists a maximal subtree decomposition $X_{i_1}, X_{i_2}, \ldots , X_{i_m}$. Then, the $L_1$-distance of $F(Q)$ and $\sum_{j=1}^{m} F(X_{i_j})$ guarantees the same upper bounds as the original ESP, as described in the theorem below.

**Theorem 12.** Let $R \in itv(X_i)$ be a $|Q|$-gram on $S$ and $X_{i_1}, X_{i_2}, \ldots , X_{i_m}$ be its maximal subtree decomposition in the tree $T(X_i)$. Then, it holds that

$$\|F(Q) - \sum_{j=1}^{m} F(X_{i_j})\|_1 = O(\lg |Q| \lg^* |S|)d(Q, R)$$

**Proof.** By Theorem 11, $\|F(Q) - F(R)\|_1 = O(\lg |Q| \lg^* |S|)d(Q, R)$. Conversely, for an occurrence of $R$ in $S$, let $T(X_i)$ be the smallest subtree in $T(S)$ that contains the occurrence of $R$, i.e., $R \in itv(X_i)$. For $T(R)$ and $T(X_i)$, let $s(R)$ and $s(X_i)$ be the sequences of the level 2 symbols in $T(R)$ and $T(X_i)$, respectively. By the definition of the ESP, it holds that $s(R) = \alpha\beta\gamma$ and $s(X_i) = \alpha'\beta'\gamma'$ for some strings that satisfy $|\alpha\alpha'\gamma\gamma'| = O(\lg^* |S|)$, and this is true for the remaining string $\beta$ iteratively. Thus, $\|F(R) - F(X_i)\|_1 = O(\lg |R| \lg^* |S|)$ since the trees are balanced. Hence, by the equation

$$\|F(Q) - \sum_{j=1}^{m} F(X_{i_j})\|_1 = O(\lg |Q| \lg^* |S|)d(Q, R) + O(\lg |Q| \lg^* |S|)$$

we obtain the approximation ratio. \qed

To further enhance the search efficiency, we also present a lower bound on the $L_1$-distance between characteristic vectors, which can be used to reduce the search space.
Theorem 13 (A lower bound $\mu$). For any $X_i \in V(S) \cup V(Q)$, the inequality $\|F(S) - F(Q)\|_1 \geq \mu(X_i)$ holds where

$$\mu(X_i) = \sum_{e \in V(S) \cup V(Q)} F(X_i)(e)$$

Proof. The $L_1$ distance between $F(S)$ and $F(Q)$ is divided into the following four classes of terms: (1) both members in $F(S)$ and $F(Q)$ are non-zero; (2) both members in $F(S)$ and $F(Q)$ are zero; (3) members in $F(S)$ and $F(Q)$ are zero and non-zero, respectively; and (4) members in $F(S)$ and $F(Q)$ are non-zero and zero, respectively. Terms that consist of class (3) and (4) can be written as $\sum_{e \in V(S) \cup V(Q)} F(S)(e)$, which is a lower bound on the $L_1$-distance. Therefore, $\|F(S) - F(Q)\|_1 \geq \sum_{e \in V(S) \cup V(Q)} F(S)(e)$. \qed

Theorem 14 (Monotonicity of $\mu$). If a variable $X_i$ derives $X_k$, the inequality $\mu(X_i) \geq \mu(X_k)$ holds.

Proof. Every entry in $F(X_k)$ is less than or equal to the corresponding entry in $F(X_i)$. Thus, the inequality holds. \qed

5.4.3 Finding candidates

By Theorems 12, 13 and 14 above, the task of the algorithm is reduced to finding a maximal subtree decomposition $(X_{i_1}, X_{i_2}, \ldots, X_{i_m})$ within $X_i$. Given a threshold $\tau \geq 0$, for each $|Q|$-gram in $itv(S)$, the algorithm finds a candidate, i.e., a maximal subtree decomposition $(X_{i_1}, X_{i_2}, \ldots, X_{i_m})$ that satisfies $\mu(X_{i_1}) + \mu(X_{i_2}) + \cdots + \mu(X_{i_m}) \leq \tau$.

For an $X_i$ and an occurrence of some $|Q|$-gram in $itv(X_i)$, the $|Q|$-gram is formed by $suf(val(X_{i(i)}), |Q| - k)pre(val(X_{r(i)}), k)$ for some $k$ $(1 \leq k \leq |Q| - 1)$. The algorithm computes the maximal subtree decompositions $(x_1, x_2, \ldots, x_p)$ that covers $suf(val(X_{i(i)}), |Q| - k)$ and $(y_1, y_2, \ldots, y_q)$ that covers $pre(val(X_{r(i)}), k)$, and outputs $(x_1, \ldots, x_p, y_1, \ldots, y_q)$ that covers the $|Q|$-gram when $\sum_{1 \leq i \leq p} \mu(x_i) + \sum_{1 \leq i \leq q} \mu(y_i) \leq \tau$. We illustrate the computation of candidates that satisfies $\mu(X_{i_1}) + \mu(X_{i_2}) + \cdots + \mu(X_{i_m}) \leq \tau$ in Figure 5.3 and
present the pseudocode as Algorithm 3.

Applying all variables to Algorithm 3 enables us to find the candidates that covers all solutions. There are no possibilities for missing any $|Q|$-grams in $\text{itv}(S)$ such that the $L_1$-distances between their characteristic vectors and $F(Q)$ are at most $\tau$, i.e., false negatives. The set here may include a false positive, i.e., the solution set encodes a $|Q|$-gram such that the $L_1$-distance between its characteristic vector and $F(Q)$ is more than $\tau$; however, false positives are efficiently removed by computing the $L_1$-distance $\|F(Q) - \sum_{j=1}^{m} F(X_{i_j})\|_1$ as a post-processing step.

**Theorem 15.** The computation time of $\text{FindCandidates}$ is $O(n|Q| \lg \lg (n + \sigma)(\lg |S| + \lg |Q|))$.

**Proof.** Because the height of the ESP-tree is $O(\lg |S|)$, for each variable $X$, the number of visited nodes is $O(\lg |Q| + \lg |S|)$. The computation times for $\text{LeftChild}(X)$ and $\text{RightChild}(X)$ are both $O(\lg \lg (n + \sigma))$, and the times for $\text{FindLeft}$ and $\text{FindRight}$ are both $O(|Q| \lg \lg (n + \sigma)(\lg |S| + \lg |Q|))$. Thus, for $n$ iterations of the functions, the total computation time is $O(n|Q| \lg \lg (n + \sigma)(\lg |S| + \lg |Q|))$. \qed
(I) ESP-tree built from a query string $Q$, a characteristic vector $F(Q)$ and a distance threshold $\tau$.

$$F(Q) = (3, 2, 1, 0, 0, 0, 0, 0, 1, 1)$$

$$\tau = 2$$

(II) Find a maximal subtree decomposition $\{X_{i1}, X_{i2}, ..., X_{im}\}$ for each $|Q|$-gram in $\text{itv}(S)$. If $\mu(X_{i0}) + \mu(X_{i1}) + \mu(X_{i2}) + ... + \mu(X_{im}) \leq \tau$, the L1-distance between $F(Q)$ and $F(X_{i1}) + F(X_{i2}) + ... + F(X_{im})$ is computed.

(i) The computation for $\text{suf}(\text{val}(X_3), 3)$ and $\text{pre}(\text{val}(X_3), 2)$ in $\text{itv}(X_i)$.

The maximal subtree decomposition:

$$\{X_3, X_1\}$$

$$F(X_3) + F(X_1)$$

$$= (2, 3, 1, 0, 1, 0, 0, 0, 0, 0)$$

$$\mu(X_{10}) + \mu(X_3) + \mu(X_1)$$

$$= 2 + 1 + 0 = 3 > \tau$$

(ii) The computation for $\text{suf}(\text{val}(X_3), 2)$ and $\text{pre}(\text{val}(X_3), 3)$ in $\text{itv}(X_i)$.

The maximal subtree decomposition:

$$\{a, b, X_1, a\}$$

$$F(a) + F(b) + F(X_1) + F(a)$$

$$= (3, 2, 1, 0, 0, 0, 0, 0, 0, 0)$$

$$\mu(X_10) + \mu(a) + \mu(b) + \mu(X_1) + \mu(a)$$

$$= 2 + 0 + 0 + 0 = 2 \leq \tau$$

$L_1$-distance: 2

(iii) The computation for $\text{suf}(\text{val}(X_3), 1)$ and $\text{pre}(\text{val}(X_3), 4)$ in $\text{itv}(X_i)$.

The maximal subtree decomposition:

$$\{b, X_1, X_3\}$$

$$F(b) + F(X_1) + F(X_3)$$

$$= (2, 3, 2, 0, 0, 0, 0, 0, 0, 0)$$

$$\mu(X_{10}) + \mu(b) + \mu(X_3) + \mu(X_1)$$

$$= 2 + 0 + 0 + 0 = 2 \leq \tau$$

$L_1$-distance: 5

Figure 5.3: Illustration of candidate finding and $L_1$-distance computation.
Algorithm 3 to output the candidate $R \subseteq V(S)$ for $X \in V(S)$, a query pattern $Q$ and a distance threshold $\tau$.

```
function FindCandidates(X, Q, \tau)
    for $j = 1, 2, \ldots, |Q|$ do
        $R \leftarrow \emptyset$ \hspace{1cm} (Initialize solution set)
        $q_1 \leftarrow$ FindLeft(LeftChild(X), $|Q| - j$, 0, R) \hspace{1cm} (for left child)
        $q_2 \leftarrow$ FindRight(RightChild(X), $j$, 0, R) \hspace{1cm} (for right child)
        if $q_1 = 0$ and $q_2 = 0$ then
            Output $R$
        end if
    end for
end function
```

```
function FindLeft(X, q, d, R)
    if $d > \tau$ then
        return $\infty$
    else if $q = 0$ then
        return 0
    else if $X \in \Sigma$ then
        $d \leftarrow d + 1$ if $X \notin V(Q)$
        $R \leftarrow R \cup \{X\}$
        return $q - 1$
    else if $|\text{val}(X)| \leq q$ then
        $d \leftarrow d + \mu(X)$
        $R \leftarrow R \cup \{X\}$
        return $q - |\text{val}(X)|$
    end if
    $q' \leftarrow$ FindLeft(RightChild(X), $q$, $d$, $R$)
    if $q' \neq 0$ then
        return FindLeft(LeftChild(X), $q'$, $d$, $R$)
    end if
end function
```

```
function FindRight(X, q, d, R)
    if $d > \tau$ then
        return $\infty$
    else if $q = 0$ then
        return 0
    else if $X \in \Sigma$ then
        $d \leftarrow d + 1$ if $X \notin V(Q)$
        $R \leftarrow R \cup \{X\}$
        return $q - 1$
    else if $|\text{val}(X)| \leq q$ then
        $d \leftarrow d + \mu(X)$
        $R \leftarrow R \cup \{X\}$
        return $q - |\text{val}(X)|$
    end if
    $q' \leftarrow$ FindRight(LeftChild(X), $q$, $d$, $R$)
    if $q' \neq 0$ then
        return FindRight(RightChild(X), $q'$, $d$, $R$)
    end if
end function
```

5.4.4 Computing Positions

Our algorithm also computes all positions of $\text{val}(X_i)$, which we denote $P(X_i) = \{p \in \{1, 2, \ldots, |S|\} : S[p, p + |\text{val}(X_i)| - 1] = (X_i)\}$. Starting from $X_i$, the algorithm traverses up to the root of the ESP-tree built from $S$. Note that $p$ is initialized to zero at $X_i$. If $X_k$ in the traversal from $X_i$ to the root is the parent with a right child $X_{r(k)}$, a non-
Algorithm 4 to compute the set $P$ of all occurrence of $val(X)$ on $S$ for $X \in V(S)$.

1: function ComputePosition($X$)
2: \hspace{1cm} $P \leftarrow \{\emptyset\}$ \hspace{1cm} $\triangleright$ Initialize solution set
3: \hspace{1cm} Recursion($X, 1$)
4: end function
5: function Recursion($X, p$)
6: \hspace{1cm} if $X$ is the root node then
7: \hspace{2cm} $P \leftarrow P \cup \{p\}$
8: \hspace{2cm} return
9: \hspace{1cm} end if
10: \hspace{1cm} for each $X_p \in RightParents(X)$ do
11: \hspace{2cm} Recursion($X_p, p + |val(X_p)| - |val(X)|$) $\triangleright$ $X$ is the right child of $X_p$
12: \hspace{1cm} end for
13: \hspace{1cm} for each $X_p \in LeftParents(X)$ do
14: \hspace{2cm} Recursion($X_p, p$) $\triangleright$ $X$ is the left child of $X_p$
15: \hspace{1cm} end for
16: end function

negative integer ($|val(X_k)| - |val(X_r(k))|$) is added to $p$. Otherwise, nothing is added to $p$.

When the algorithm reaches the root, $p$ represents a start position of $val(X_i)$ on $S$, i.e., $val(X_i) = S[p, p + |val(X_i)| - 1]$. To compute the set $P(X_i)$, the algorithm starts from $X_i$ and traverses up to the root for each parent in $RightParents(X_i)$ and $LeftParents(X_i)$, which return sets of parents for $X_i$. Algorithm 4 shows the pseudocode of this algorithm.

**Theorem 16.** The computation time of $P(X)$ is $O(occ \lg |S|)$, where $occ$ is the number of occurrences of $X$ in $T(S)$.

**Proof.** Using the index structures of $RightParents(X)$ and $LeftParents(X)$, we can traverse the path from any node with label $(X)$ to the root of $T(S)$ by counting the position. The length of the path is $O(\lg |S|)$. \hfill $\Box$

**Theorem 17.** The search time is $O(n|Q| \lg \lg (n + \sigma)(\lg |S| + \lg |Q|) + occ \lg |S|)$ using the data structure of size $n(n + 1) \lg |S| + (n + \sigma) \lg (n + \sigma) + 2n + \sigma + o((n + \sigma) \lg (n + \sigma))$ bits.

**Proof.** The time required to compute $T(Q)$ and $F(Q)$ is $t_1 = O(|Q| \lg^* |S|)$. The time required to find candidates and compute $\|F(Q) - \sum_{j=1}^{m} X_{i_j}\|_1$ is $t_2 = O(n|Q| \lg \lg (n + \sigma)(\lg |S| + \lg |Q|))$ using Theorem 15. The time to compute positions is $O(occ \lg |S|)$, based on Theorem 16. Therefore, the total time for a query search is $t_1 + t_2 + t_3 = O(n|Q| \lg \lg (n + \sigma)(\lg |S| + \lg |Q|) + occ \lg |S|)$. The size of the data structure is derived using the results we presented in Section 5.3. \hfill $\Box$
Finally, note that in Theorem 17, \( n \) and \( \text{occ} \) are incomparable because \( \text{occ} > n \) is possible for a highly repetitive string.

## 5.5 Experiments

We evaluated the performance of the siEDM on one core of a quad-core Intel Xeon Processor E5540 (2.53GHz) machine with 144GB memory. We implemented the siEDM using the rank/select dictionary and GMR in libcds (https://github.com/fclaude/libcds). We used the two standard benchmark datasets of einstein and cere from repetitive text collections in the pizza and chili corpus (http://pizzachili.dcc.uchile.cl/repcorpus.html), which we further present in Table 5.1. As a comparative method, we used the online pattern matching for the EDM called online ESP (baseline) \cite{46} that approximates the EDM between a query \( Q \) and substrings of the length of \( |Q| \) at each position of an input text. We randomly selected \( S[i, j] \) as the query pattern \( Q \) for each \( |Q| = 50, 100, 500, \) and \( 1,000 \), and examined the performance of each.

| Dataset | Length        | \( |\Sigma| \) | Size (MB) |
|---------|---------------|-------------|-----------|
| einstein| 467,626,544   | 139         | 446       |
| cere    | 461,286,644   | 5           | 440       |

Table 5.2 shows the memory consumption during the search of the siEDM and the baseline. The memory consumption of the siEDM was larger than that of the baseline for both texts because the baseline does not have characteristic vectors of each node and length vector.

Table 5.3 shows the size of each component of the index structure, as well as the time required to construct the index structure for einstein and cere datasets. Note that most of the size of the index structure was consumed by the characteristic vector \( F \). Furthermore, the index size of cere was much larger than that of einstein. The index sizes
of cere and einstein were approximately 16 megabytes and 256 megabytes, respectively, because the number of variables generated from cere was much larger than that generated from einstein. The number of variables generated from einstein was 305,098, whereas the number of variables generated from cere was 4,512,406. Construction times for the index structures were 118 s for einstein and 472 s for cere. The results here for constructing the necessary index structures demonstrate the applicability of siEDM to moderately large repetitive texts.

Table 5.3: Comparing index sizes and construction times.

<table>
<thead>
<tr>
<th>Data set</th>
<th>Einstein (MB)</th>
<th>Cere (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Encoded ESP-tree</td>
<td>1.18</td>
<td>19.92</td>
</tr>
<tr>
<td>Index Size</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Characteristic vector $F$ (MB)</td>
<td>15.35</td>
<td>227.34</td>
</tr>
<tr>
<td>Length vector $L$ (MB)</td>
<td>0.59</td>
<td>7.49</td>
</tr>
<tr>
<td>Construction time (sec)</td>
<td>117.65</td>
<td>472.21</td>
</tr>
</tbody>
</table>

Figure 5.4 shows the total search time in seconds of the siEDM and the baseline for einstein and cere in distance thresholds $\tau$ from 10 to 60. Note that the results here do not contain the case $\tau < 10$ because siEDM found no candidates given such conditions. Furthermore, the query length was one of $\{50, 100, 500, 1000\}$. Because the search time of the baseline is linear in $|S| + |Q|$, we show only the fastest case, i.e., $q = |Q| = 50$. The search time of the siEDM was shorter than that of the baseline in most cases.
Figure 5.4: Comparison of search times for einstein (left) and cere (right).
Figure 5.5: Details of search times for different $|Q|$ and $\tau$, including times for (a)-(b) candidate findings (CF), (c)-(d) $L_1$-distance computations (DIST), and (e)-(f) position computations (PC). (a) and (b) correspond to CF, (c) and (d) correspond to DIST, and (e) and (f) correspond to PC of einstein and cere, respectively.

Figure 5.5 shows the detailed search times in seconds, including finding candidates (CF) of $Q$ in $T(S)$, computing approximated $L_1$ distances by characteristic vectors (DIST), and determining the positions of all $|Q|$-grams within the threshold $\tau$ (PC).
Figure 5.6: Statistical information of the query searches, showing (a)-(b) the number of traversed nodes (#TN), (c)-(d) the number of candidate $|Q|$-grams (#CAND), (e)-(f) the number of true positives (#TP), (g)-(h) the number of occurrences (#OCC).

Figure 5.6 shows the number of nodes $T(S)$ visited by the algorithm (#TN), the num-
ber of candidate $|Q|$-grams computed by \textit{FindCandidates} (#CAND), the number of true positives among candidate $|Q|$-grams (#TP), and the number of occurrences (#OCC). We observe here that the most time-consuming task was the candidate finding.

By the monotonicity of the characteristic vectors, pruning the search space for small distance thresholds and long query lengths is more efficient. Therefore, we expect that siEDM is faster for smaller distance thresholds and longer query lengths; our experimental results support this. The search time on cere was much longer than that of einstein because the number of generated production rules from cere is much larger than that from einstein, and a large number of iterations of \texttt{FindCandidates} is executed. In addition, comparing of #CAND and #TP validates the efficiency of the siEDM for candidate finding using our proposed pruning method.

As shown in Figure 5.6, the algorithm failed to find a candidate. Such a phenomenon often appears when the required threshold $\tau$ is too small, because the ESP-tree $T(Q)$ is not necessarily identical to $T(S[i,j])$ even if $Q = S[i,j]$. In General, the parsing of $T(S[i,j])$ is affected by a suffix of $S[1,i-1]$ and a prefix of $S[j+1,|S|]$ of length of at most $\lg^*|S|$.

As shown in Table 5.3 and Figure 5.4, the search time of the siEDM depends on the size of the encoded ESP-tree given as input. We confirmed this feature by an additional experiment on other repetitive texts. Tables 5.4, 5.5, and 5.6 describe several data sets from the aforementioned pizza & chili corpus. Figure 5.7 shows the search time of the siEDM and the baseline, with our results supporting our claim that the siEDM is suitable for computing the EDM of repetitive texts.

Table 5.4: Summary of additional data sets.

| Dataset         | Length             | $|\Sigma|$ | Size (MB) |
|-----------------|--------------------|-----------|-----------|
| influenza       | 154,808,555        | 15        | 147.64    |
| Escherichia_Coli| 112,689,515        | 15        | 107.47    |
CHAPTER 5. SIEDM: AN EFFICIENT STRING INDEX AND SEARCH ALGORITHM FOR EDIT DISTANCE

Table 5.5: The memory consumption for the query searches.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Influenza</th>
<th>Escherichia_Coli</th>
</tr>
</thead>
<tbody>
<tr>
<td>siEDM (MB)</td>
<td>164.87</td>
<td>262.01</td>
</tr>
<tr>
<td>baseline (MB)</td>
<td>53.01</td>
<td>100.81</td>
</tr>
</tbody>
</table>

Table 5.6: Comparing the index sizes and construction times for the additional data sets.

<table>
<thead>
<tr>
<th>Data set</th>
<th>Influenza</th>
<th>Escherichia_Coli</th>
</tr>
</thead>
<tbody>
<tr>
<td>Encoded ESP-tree (MB)</td>
<td>9.92</td>
<td>20.21</td>
</tr>
<tr>
<td>Index Size Characteristic vector F (MB)</td>
<td>150.87</td>
<td>234.91</td>
</tr>
<tr>
<td>Length vector L (MB)</td>
<td>4.08</td>
<td>6.88</td>
</tr>
<tr>
<td>Construction time (sec)</td>
<td>290.33</td>
<td>420.43</td>
</tr>
</tbody>
</table>

Figure 5.7: Search time in seconds for repetitive texts, *i.e.*, E.Coli (left) and influenza (right).
Chapter 6

Online grammar compression for frequent pattern discovery

In this chapter, we present our algorithm to find approximated frequent patterns from FOLCA’s data structure. Note this chapter refer to [7].

6.1 Approximate frequent pattern

A substring $P = S[i,j]$ is said to be frequent if it appears at least twice, i.e., $freq_s(P) \geq 2$.

We focus on an approximation of the problem to find all frequent patterns defined as follows.

**Problem 3.** Let $T$ be a parsing tree of a grammar compression that derives $S \in \Sigma^*$. A variable $X$ in $T$ is called a core of $P$ if for each occurrence $S[i,j] = P$, there exists an occurrence of $X$ in $T$ that derives a substring $S[\ell, r]$ for a subinterval $[\ell, r]$ of $[i, j]$. Then, $P$ is said to be approximated by $X$ with $\delta$ if $\frac{val(X)}{|P|} \geq \delta$. The problem of the approximated frequent pattern (AFP) is computing $T$ that guarantees a core $X$ of any frequent pattern $P$ in $S$ with an approximation ratio $\delta > 0$.

AFP is well-defined with a small $\delta$ because for any $S$ and its frequent substring $P$
any alphabet symbol forming $P$ satisfies the condition with $\delta = \frac{1}{|F|}$. In [29], Nakahara et al. proposed an offline algorithm with approximation $\Omega\left(\frac{1}{\log^2 |F|}\right)$. We aim to construct the parsing tree using an online algorithm in a compressed space with a larger $\delta$, thus improving the best known approximation ratio. In our algorithm, a grammar compression is represented by ESP and succinctly encoded by POSLP. In the subsections that follow, we next review the related techniques.

6.1.1 Algorithm

In this subsection, we propose a modified FOLCA for an AFP that requires less space. We describe the improved lower bound of the size of the extracted core as well as the time and space complexities. We first summarize our proposed algorithm. Let $S_i$ ($i = 0, 1, \ldots, \lfloor \log |S| \rfloor$) be the resulting string of the $i$-th iteration of ESP, where $S_0 = S$. The algorithm simulates the parsing of ESP using a queue $q_i$ for each level $i$. The queue $q_i$ stores a substring $S_i$ of length of at most $O(\log^* |S|)$ in a FIFO manner. At the beginning, input symbols are enqueued to $q_0$. If a prefix of $S$ is a repetition $a^+$, it is parsed in a left-aligned manner, thereby generating a production rule, e.g., $A \rightarrow aa$. Next, $a^+$ is dequeued from $q_0$, and the resulting sequence $As$ is enqueued to $q_1$. Otherwise, at most $O(\log^* |S|)$ symbols are enqueued to $q_0$, and $q_0[0, i-1]$ is parsed in a left-aligned manner, where $q_0[i]$ is the leftmost landmark. Based on Theorem 1, there is at least one landmark in $q_0$ of length $O(\log^* |S|)$. Then, the symbols in $q_0[0, i-1]$ are dequeued from $q_0$, and the generated symbols are enqueued to $q_1$. These computations are performed at each level. When a prefix of $S$ is enqueued, a sequence of production rules is generated such that it is encoded by a POSLP $T$ encoded by $(B, L)$, where $B$ is a bit sequence that represents the skeleton of $T$, and $L$ is the sequence of leaf nodes of $T$. The pseudocode is presented as Algorithm 5.

We next show that the ESP tree of $S$ contains a sufficiently large core for any substring $P$ that guarantees the approximation ratio of our algorithm. This result is an improvement
over the lower bound presented by [29].

**Theorem 18.** Let $T$ be the ESP tree of a string $S$ and $P$ be a substring of $S$. There exists a core of $P$ that derives a string of length $\Omega(\frac{|P|}{\lg^* |S| \lg |P|})$.

**Proof.** If a prefix of $P$ is a repetition, let $Q_1$ be the maximal one and $Q'_1$ be the remaining suffix of $P$. The parsing of $Q'_1$ is not affected by the string preceding $Q'_1$, and the parsing of $Q'_1$ inside $P$ is identical regardless of any occurrence of $P$. Otherwise, using Theorem 1, we can partition $P = Q_1Q'_1$ such that $|Q_1| = O(\lg^* |S|)$, and $Q'_1$ is also identically parsed inside $P$. Let $P_1$ be the common substring in $S_1$ that derives $Q'_1$. Then, for each case, $Q_1P_1$ is a sequence of cores of $P$. Iterating through this process for $P_1$ at most $k(\leq \lfloor \lg |P| \rfloor)$ times, we can obtain a sequence $Q_1Q_2 \cdots Q_k$ of cores such that $Q_i$ is either a repetition of the form $Q_i = c^+ \atop i$ (where $c_i \in \Sigma \cup V$) or a string of length $O(\lg^* |S|)$.

We show that for any $1 \leq i \leq k$, there exists a core $X_i$ in $Q_i$ with $|val(X_i)| = \Omega\left(\frac{|val(Q_i)|}{\lg^* |S|}\right)$. If the length of $Q_i$ is $O(\lg^* |S|)$, the claim is immediate from the pigeonhole principle. Otherwise $Q_i = c^+_i$. Because any maximal repetition is parsed in a left-aligned manner, a type2 sequence of bigrams $c^+_i$ is created over $Q_i$ (except for the last one, which may be a 2-2-tree that derives $c^+_i$). Iterating through the parsing of the type2 sequence, we obtain a large complete balanced binary tree of $c_i$. Assuming that the largest covers $2^h c_i$’s in $Q_i$, we observe that the number of $c_i$’s in $Q_i$ is less than $5 \cdot 2^h$, i.e., there is a node that covers at least one-fifth of the $c_i$’s in $Q_i$. The maximum length of $Q_i$ is achieved when $Q_i$ is parsed into $ABC_{h-1} \cdots C_0$, where $A$ contains $2^h - 1$ $c_i$’s, $B$ contains $2^h c_i$’s, and for any $0 \leq h' < h$, $C_{h'}$ contains $3 \cdot 2^{h'} c_i$’s. $A$ and its preceding character $c \neq c_i$, which must be the first character in the entire string, compose a node with $2^h$ characters, $B$ composes the largest complete binary tree with $2^h c_i$’s, and for any $0 \leq h' < h$, $C_{h'}$ composes a 2-2-tree over three complete binary trees with $2^{h'} c_i$’s. Note that adding even a single $c_i$ to the $Q_i$ results in creating a complete binary tree with $2^{h+1} c_i$’s, which may appear in a 2-2-tree over three complete binary trees with $2^h c_i$’s. Thus, the maximum number of $c_i$’s in $Q_i$ is $2^h - 1 + 2^h + \sum_{h' = 0}^{h-1} 3 \cdot 2^{h'} < 5 \cdot 2^h$. Therefore, there exists a variable
$X_i$ in $Q_i$ with $|\text{val}(X_i)| = \Omega\left(\frac{|\text{val}(Q_i)|}{\lg^* |S|}\right)$.

Because there is at least one $Q_j$ such that $|\text{val}(Q_j)| \geq |P|/k \geq |P|/\lg |P|$, there exists a core of $P$ that derives a string of length $\Omega\left(\frac{|\text{val}(Q_j)|}{\lg^* |S|\lg |P|}\right)$. □

**Theorem 19.** Algorithm 5 approximates the AFP problem with the ratio $\Omega\left(\frac{1}{\lg^* |S|\lg |P|}\right)$ in $O\left(\frac{|S|\lg n}{\alpha \lg \lg n}\right)$ time and $O(n + \lg |S|)$ space.

**Proof.** The algorithm simulates the ESP of $S$ using queues $q_i$ ($i = 0, 1, \ldots, |S|$); here, $q_i$ stores a substring of $S_i$ to determine whether $S_i[j]$ is a landmark or not. By Theorem 1, the space required for each $q_i$ is $O(\lg^* |S|)$. We can reduce this space to $O(1)$ using a table of a size of at most $\lg^* |S|\lg \lg \lg |S|$ bits as follows. Applying two iterations of alphabet reduction, each symbol $A$ is transformed into a label $L_A$ of a size of at most $\lg \lg \lg |S|$ bits. Whether the $A$ is a landmark depends on its consecutive $O(\lg^* |S|)$ neighbors. Thus, the size of a table that stores a one-bit answer is at most $\lg^* |S|\lg \lg \lg |S|$ bits. It follows that the space required to parse $S$ is $O(\lg |S|)$. Conversely, based on Theorem 3, the POSLP $T$ of $S$ is computable in $O\left(\frac{|S|\lg n}{\alpha \lg \lg n}\right)$ time. By Theorem 18, for each frequent $P$, $T$ contains at least one core $X$ of $P$ that satisfies $|\text{val}(X)| = \Omega\left(\frac{|P|}{\lg^* |S|\lg |P|}\right)$. Thus, finding all variables $X$ that appear at least twice in $T$ approximates this problem with the lower bound. Whether $freq_T(X_i) \geq 2$ can be stored in $n$ bits for all $i$ because an internal node $i$ of $T$ denotes the position of the first occurrence of $X_i$. Therefore, we obtain the complexities and approximation ratio. □
Algorithm 5 to compute a core $X$ of any frequent $P$ in $S$. $T$: POSLP representing the ESP tree of $S$, $B$: a succinct representation of skeleton of $T$, $L$: a sequence of leaves of $T$, $FB$: a bit vector storing $FB[i] = 1$ iff $freq_T(X_i) \geq 2$, $D^{-1}$: the reverse dictionary for production rules, $q_k$: a queue in $k$-th level, and let $u \in \max\{5, lg^* |S|\}$.

1: function ComputeAFP(S)
2: $B := \emptyset$, $L := \emptyset$; $FB := \emptyset$; initialize queues $q_k$
3: for $i := 1, 2, \ldots, |S|$ do
4:  BuildESPTree($S[i], 0, 0, 0, q_1$)
5: end for
6: end function
7: function BuildESPTree($X, q_k$) $\triangleright$ $X$ is a set $\{s, ib, k_1, k_2, k_3, k_4, k_5\}$ where $s$ is a symbol, $ib$ is 1 if $s$ is an internal node otherwise 0 and $\ell_i(i \in \{1, 2, lg^* |S|\})$ is a label applied $i$-th alphabet reduction for $s$.
8: $q_k.enqueue(X)$
9: compute $q_k[k_1.length()].\ell_i(i \in \{1, 2, lg^* |S|\})$
10: if $q_k.length() = u$ then
11:  if Is2Tree($q_k$) then
12:  $Y := Update(q_k[u - 1], q_k[u])$
13:  $q_k.dequeue(); q_k.dequeue()$
14:  BuildESPTree($Y, q_k + 1$)
15: end if
16: else if $q_k.length() = u + 1$ then
17:  $Y := Update(q_k[u], q_k[u + 1]); Z := Update(q_k[u - 1], Y)$
18:  $q_k.dequeue(); q_k.dequeue(); q_k.dequeue()$
19:  BuildESPTree($Z, q_k + 1$)
20: end if
21: end function
22: function Is2Tree($q_k$)
23:  if $(q_k[u - 4].s = q_k[u - 3].s) \& (q_k[u - 3].s \neq q_k[u - 2].s)$ then
24:    return 0
25:  else if $(q_k[u - 3].s \neq q_k[u - 2].s) \& (q_k[u - 2].s = q_k[u - 1].s)$ then
26:    return 0
27:  else if $(q_k[u - 3].\ell_i[k^{*} |S|] < q_k[u - 2].\ell_i[k^{*} |S|]) \& (q_k[u - 2].\ell_i[k^{*} |S|] > q_k[u - 1].\ell_i[k^{*} |S|])$ then
28:    return 0
29:  else
30:    return 1
31: end if
32: end function
33: function Update($X, Y$)
34:  $z := D^{-1}(X.s, Y.s)$
35:  if $z$ is a new symbol then
36:    UpdateLeaf($X$); UpdateLeaf($Y$)
37:    B.push_back(1); FB.push_back(0)
38:    return $\{z, 1, 0, 0, 0\}$
39:  else
40:    GetAFPNode($z$)
41:    return $\{z, 0, 0, 0, 0\}$
42: end if
43: end function
44: function UpdateLeaf($X$)
45:  if $X.ib = 0$ then
46:    L.push_back($X.s$); B.push_back(0)
47: end if
48: end function
49: function GetAFPNode($X_i$)
50:  if $FB[i] = 0$ then
51:    $FB[i] := 1$
52:    Output $X_i$
53: end if
54: end function
6.1.2 Experimental Results

We evaluated the performance of our proposed approximation algorithm (AFP(online)) on one core of a quad-core Intel Xeon Processor E5540 (2.53GHz) machine with 144GB memory. We adopted a lightweight version of the fully-online ESP, called FOLCA [27], as a subroutine for the grammar compression.

Furthermore, we used several standard benchmarks from the text collection\(^1\) detailed in Table 6.1; we selected texts with both high and low numbers of repetitions. From these texts, we examined the practical approximation ratio of the algorithm as follows. For each text \(S\), we obtained the set of frequent substrings using a compressed suffix array (SA) as introduced by [38]. We then selected the top-100 longest patterns such that any two \(P\) and \(Q\) are not inclusive of each other, where \(P\) is inclusive of \(Q\) if any occurrence of \(Q\) is included in an occurrence of \(P\). We removed such \(Q\) from the given candidates. For each frequent substring \(P\) and a variable \(X\) reported by the algorithm, we estimated the cover ratio \(|\text{val}(X)|/|P|\) and determined the average for all \(P\); however, as shown in the results below (and in Figure 6.1), the suffix array cannot be executed for larger \(S\) due to memory constraints. In addition, we examined the time and memory consumption of the offline algorithm using the approach presented by [29] (AFP(offline)).

Table 6.2 shows the length of optimum frequent patterns extracted by the suffix array approach and the length of the corresponding cores extracted by our algorithm, as well as the approximation ratio to the optimal one, where min. and max. denote the shortest and longest patterns in the candidates, respectively. Our algorithm extracted sufficiently long cores for each benchmark.

Figure 6.1 shows the memory consumption for repetitive strings (\(i.e.,\) Figure 6.1a-6.1c) and normal strings (\(i.e.,\) Figure 6.1d-6.1f). The amount of required space was significantly reduced by our online strategy when offline and SA were executed for each static size of data noted in the figures.

\(^1\)http://pizzachili.dcc.uchile.cl/repcorpus.html
Table 6.1: Statistical information regarding benchmarking string $S$

<table>
<thead>
<tr>
<th></th>
<th>einstein</th>
<th>cere</th>
<th>kernel</th>
<th>english</th>
<th>dna</th>
<th>sources</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>S</td>
<td>$ (MB)</td>
<td>446</td>
<td>446</td>
<td>246</td>
<td>200</td>
</tr>
<tr>
<td>$</td>
<td>\Sigma</td>
<td>$</td>
<td>139</td>
<td>5</td>
<td>160</td>
<td>239</td>
</tr>
</tbody>
</table>

Table 6.2: Length of optimal $P$ extracted by the suffix array approach (SA) and approximate $X$ by our proposed algorithm (AFP(online)) with approximation ratio $\frac{|val(X)|}{|P|}$ (%) for the top-100 patterns.

<table>
<thead>
<tr>
<th></th>
<th>einstein</th>
<th>cere</th>
<th>kernel</th>
<th>english</th>
<th>dna</th>
<th>sources</th>
</tr>
</thead>
<tbody>
<tr>
<td>min.</td>
<td>SA</td>
<td>198,606</td>
<td>4,562</td>
<td>442,124</td>
<td>43,985</td>
<td>3,271</td>
</tr>
<tr>
<td>%</td>
<td>7.6</td>
<td>2.3</td>
<td>6.9</td>
<td>7.3</td>
<td>7.1</td>
<td>7.3</td>
</tr>
<tr>
<td>max.</td>
<td>SA</td>
<td>935,920</td>
<td>303,204</td>
<td>2,755,550</td>
<td>98,7770</td>
<td>97,979</td>
</tr>
<tr>
<td>%</td>
<td>50.0</td>
<td>62.1</td>
<td>52.8</td>
<td>50.8</td>
<td>63.9</td>
<td>51.7</td>
</tr>
<tr>
<td>mean</td>
<td>SA</td>
<td>259,451</td>
<td>111,284</td>
<td>727,443</td>
<td>116,920</td>
<td>8,241</td>
</tr>
<tr>
<td>%</td>
<td>21.6</td>
<td>11.0</td>
<td>20.0</td>
<td>23.0</td>
<td>22.9</td>
<td>22.0</td>
</tr>
</tbody>
</table>

Figure 6.2 shows the computation time for each benchmark. Due to the time-space tradeoff of our succinct data structure, our algorithm was 2.5-6.8 times slower than the offline and SA.
Figure 6.1: Memory consumption (MB) for each of the six data sets used
Figure 6.2: Computation time in seconds for each of the six data sets used.
Chapter 7

Conclusions and future work

In this thesis, we presented OESP-index, which is, to our knowledge, the first online grammar-based self-index. Further, we proposed three applications based on ESP; these applications are (1) an online pattern matching algorithm for string edit distance with moves (EDM) called OESP, (2) a string index for EDM (siEDM), and (3) an online grammar compression algorithm for frequent pattern discovery. Through our experiments, we showed that these applications were very much applicable to highly repetitive texts.

7.1 Chapter summary

In Chapter 3, we have presented our OESP-index and shown it to work in $O\left(\frac{\log n \log^* N}{\alpha}\right)$ addition time per input character; we have further shown that the required working space depends on the grammar size and the search time is $O(\log(n + \sigma)(\frac{m}{\alpha} + \text{occ}_q(\log N + \log m \log^* N)))$. In our experiments, we have shown additions at a rate of 0.2MB/sec, and working space of the compressed text size and search times of 1 (pattern of length 100)/s.

In Chapter 4, we have presented OESP that works in $O\left(\frac{N \log N \log n}{\alpha \log \log n}\right)$ expected time, requires $n(\alpha + 1) \log(n + \sigma) + n \log(n \alpha n) + 5n + o(n)$ bits of space, and has an $O(\log^2 N)$ approximation ratio. In our experiments, we have shown our algorithms space-efficiency for highly repetitive texts; however, we have also observed that search time is slow for
computing each position’s EDM.

Next in Chapter 5, we have presented siEDM and shown it to work in $O(N)$ construction time and require $O(N)$ working space for construction, an $n(n + 1) \log |S| + (n + \sigma) \log (n + \sigma) + 2n + \sigma + o((n + \sigma) \log (n + \sigma))$ index size and an $O(n|Q| \log (n + \sigma)(\log |S| + \log |Q|) + occ \log |S|)$ search time. In our experiments, we have shown our faster search times than OESP and a larger index size than those of OESP. In particular, siEDM was fast for smaller distance thresholds and longer query texts.

In Chapter 6, we have presented the discovery algorithm for approximated frequent patterns using FOLCA. Our algorithm works in $\Omega(\frac{1}{\log |S| \log |P|})$ approximated ratio and $O(\frac{|S| \log n}{\alpha \log \log n})$ computation time, and requires $O(n + \log |S|)$ working space. In our experiments, we have shown the space-efficiency of our proposed method, as well as longer computation times than offline methods and $2.3 - 63.9\%$ approximation ratios.

In both theory and practice, we have achieved our goal of the working space for construction; however, we have not fully achieved our goals for the construction and search times.

### 7.2 Future works

To more efficiently compress and index streaming texts that are more than GB-scale, our future work will include the development of the online grammar-based self-index supporting $O(1)$ addition time and faster searches in the working space of the compressed space. More specifically, for the DWT of our OESP-index, we must improve its query and update times to $O(\log \log n)$ time same as the query time of a static rank/select dictionary for large alphabets [9].

Moreover, the cache misses of our OESP-index increase for larger texts since our OESP-index’s data structure also increases in size. Therefore, we must develop a constant-space construction for the OESP-index similar to the grammar compression technique presented by [26]. We need the reverse dictionary for the construction of the OESP-index; therefore,
we plan to store the reverse dictionary of frequent production rules in the main memory and the reverse dictionary of other production rules on disk. Then, although we will need to access the hard disk, we should be able to achieve the constant space construction. To confirm the slowness of disk access, we must test our constant-space construction ideas. Since we can apply this constant-space construction to OESP and the approximated frequent discovery algorithm, we expect to achieve fast computation times in practice.

For the siEDM, to efficiently index streaming texts, we must improve the algorithm to an online algorithm. The data structure of the siEDM require to support the following three functions for the ESP-tree: (1) traversing nodes; (2) computing the position of each node; and (3) computing the characteristic vector of each node. Our OESP-index supports (1) and (2). Furthermore, we can construct the characteristic vectors of each node in our OESP-index construction algorithm. Therefore, using our OESP-index, we can construct our siEDM in online. However, the search time of this online siEDM is slower than the offline siEDM because of the DWT of our OESP-index. To solve the slowness of the search time of the online siEDM, we also must develop a dynamic rank/select dictionary that computes queries in $O(\log \log n)$ time.

By completely achieving our goals stated here, we aim to index streaming and large-scale texts (e.g., Twitter's data and genome sequence reads) in real time within the compressed working space.

\footnote{http://twitter.com}
References


