Online Grammar Compression for Frequent Pattern Discovery*

Shouhei Fukunaga

Yoshimasa Takabatake

Tomohiro I

Hiroshi Sakamoto

Department of Informatics, Kyushu Institute of Technology
680-4 Kawazu, Iizuka, Fukuoka 820-8502, Japan

Abstract

Various grammar compression algorithms have been proposed in the last decade. A grammar compression is a restricted CFG deriving the string deterministically. An efficient grammar compression develops a smaller CFG by finding duplicated patterns and removing them. This process is just a frequent pattern discovery by grammatical inference. While we can get any frequent pattern in linear time using a preprocessed string, a huge working space is required for longer patterns, and the whole string must be loaded into the memory preliminarily. We propose an online algorithm approximating this problem within a compressed space. The main contribution is an improvement of the previously best known approximation ratio \( \Omega\left(\frac{1}{\lg^2 m}\right) \) to \( \Omega\left(\frac{1}{\lg \star N \lg m}\right) \) where \( m \) is the length of an optimal pattern in a string of length \( N \) and \( \lg^\star \) is the iteration of the logarithm base 2. For a sufficiently large \( N, \lg^\star N \) is practically constant. The experimental results show that our algorithm extracts nearly optimal patterns and achieves a significant improvement in memory consumption compared to the offline algorithm.

Keywords: Grammar Compression, Online Algorithm, Approximation Algorithm, Frequent Pattern Discovery

1. Introduction

A grammar compression of a string is a context-free grammar (CFG) that derives only the string. In recent decades, various grammar compression algorithms have been proposed, showing good performance, especially for a repetitive string in which long identical patterns (substrings) can be observed many times. Such data are currently ubiquitous, for example, in genome sequences collected from similar species and in versioned documents maintained by Wikipedia and GitHub, etc. Because repetitive strings are growing rapidly, data processing methods on grammar compression have been extensively studied as a promising way to address repetitive strings (e.g., Larsson and Moffat (2000); Lehman and Shelat (2002); Rytter (2003); Sakamoto (2005); Charikar et al. (2005); Maruyama et al. (2012, 2013b); Tabei et al. (2013); Maruyama and Tabei (2014)).

* The full version is available from arXiv: CoRR abs/1607.04446. This work was supported by KAKENHI(26280088, 26540119, 16K16009, 15J05902).

Frequent pattern discovery is a classic problem in pattern mining for sequence data (e.g., Aggarwal and Han (2014)), where we focus on a string and say that a pattern (substring) is frequent if it occurs at least twice. Longer patterns are often the target of discovery, as they seem to characterize the input string better. Although we have linear time solutions using a full-text index such as suffix tree and suffix array by Sadakane (2000), it requires a huge working space for large-scale data. Even if we opt for space-efficient alternatives of these data structures, such as FM-index by Ferragina and Manzini (2000), we still have to load the whole string into memory, at least at the construction phase, because there are no known algorithms to construct them in a streaming fashion. Due to these drawbacks, it is difficult to apply these algorithms to stream data.

A reasonable approach to avoid this difficulty is to seek an approximate frequent pattern instead of the exact solution. In the framework of grammar compression, an approximate pattern is found as a frequent subtree. Then, a suitable parsing tree should preserve as many occurrences of a common substring as possible. Edit-sensitive parsing (ESP) by Cormode and Muthukrishnan (2007) matches the claim; ESP approximately solves the NP-hard problem of the generalized edit distance for measuring the similarity of two strings, and online algorithms and applications of ESP were widely proposed (e.g., Hach et al. (2012); Maruyama et al. (2013a); Takabatake et al. (2014, 2015, 2016); Nishimoto et al. (2015)).

As seen above, grammar compression is closely related to the approximate pattern discovery because a good compression ratio is achieved by finding frequent substrings and replacing them by a variable that derives the substrings. Nakahara et al. (2013) focused on a grammar compression algorithm (called ESP-comp) based on ESP and showed that it approximately solves the frequent pattern discovery problem. That is, they showed that for any frequent pattern $P$, there is a variable $X$ such that (1) $X$ derives a string of length $\Omega(\frac{|P|}{\lg^*|P|})$ that is a substring of $P$ and (2) $X$ accompanies any occurrence of $P$ in the string. They confirmed by computational experiments that the algorithm efficiently finds long frequent patterns from large repetitive data.

In this paper, we follow the previous work and show a new lower bound $\Omega(\frac{1}{\lg^*N \lg|P|})$ for approximation, where $N$ is the length of the string and $\lg^*$ is the iteration of logarithm base 2. This improves the previous bound $\Omega(\frac{1}{\lg|P|})$ as $\lg^*N \leq \lg|P|$ in practice. In addition, we establish an online approximation algorithm within a compressed space using ESP-comp. Note that the previous algorithm was not online, i.e., the whole string must be loaded into memory, but recent progress by Maruyama et al. (2013b) has enabled the computation of ESP-comp in compressed space in a streaming fashion. We implement our algorithm and show experimentally that the approximation is nearly optimal and the improvement of memory consumption is significant for real data.

2. Definition

2.1. Notation

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., $\Sigma^q = \{w \in \Sigma^* : |w| = q\}$ and 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 0, namely $|\epsilon| = 0$. 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 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 integer $q \geq 0$, let $pre(S, q) = S[1, q]$ and $suf(S, q) = S[|S| - q + 1, |S|]$. For strings $S$ and $P$, let $freq(P)$ denote the number of occurrences of $P$ in $S$, i.e., $freq(P) = |\{i : S[i, i + |P| - 1] = P\}|$.

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

### 2.2. Grammar Compression

We consider a special type of context-free grammar (CFG) $G = (\Sigma, V, D, X_s)$ where $V$ is a finite subset of $X$, $D$ is a finite subset of $V \times (V \cup \Sigma)^*$, and $X_s \in V$ is the start symbol. A grammar compression of a string $S$ is a CFG deriving only $S$ deterministically, i.e., for any $X \in V$ there exists exactly one production rule in $D$ and there is no loop. Because each $G$ has its Chomsky normal form, we can assume that any grammar compression is in *Straight-line program (SLP)* by Karpinski et al. (1997): any production rule is in the form of $X_k \rightarrow X_iX_j$ where $X_i, X_j \in \Sigma \cup V$ and $1 \leq i, j < k \leq n + \sigma$.

The size of an SLP is the number of variables, i.e., $|V|$ and let $n = |V|$. $val(X_i)$ for variable $X_i \in V$ denotes the string derived from $X_i$. For $w \in (\Sigma \cup V)^*$, let $val(w) = val(w[1]) \cdots val(w[|w|])$.

The parse tree of $G$ is a rooted ordered binary tree such that (i) the internal nodes are labeled by variables and (ii) the leaves are labeled by alphabet symbols. In a parse tree, any internal node $Z$ corresponds to a production rule $Z \rightarrow XY$, and has the left child with label $X$ and the right child with label $Y$.

A phrase dictionary $D$ is a data structure for directly accessing the phrase $X_iX_j$ for any $X_k$ if $X_k \rightarrow X_iX_j$ exists. On the other hand, a reverse dictionary $D^{-1}$ is a data structure for directly accessing $X_k$ for $X_iX_j$ if $X_k \rightarrow X_iX_j$ exists.

### 2.3. Approximate Frequent Pattern

A substring $P = S[i, j]$ is said to be frequent if it appears at least twice, i.e., $freq(P) \geq 2$. We focus on an approximation of the problem to find all frequent patterns defined as follows.

**Problem 1** Let $T$ be a parsing tree of a grammar compression deriving $S \in \Sigma^*$. 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 $T$ deriving 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 approximated frequent pattern (AFP) is to compute $T$ 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 $\delta = \frac{1}{|P|}$. Nakahara et al. (2013) pro-
posed an offline algorithm with approximation $\Omega(\frac{1}{\log |P|})$. We aim to construct the parsing tree by an online algorithm in a compressed space with a larger $\delta$ improving the best known approximation ratio. In our algorithm, a grammar compression is represented by ESP (edit sensitive parsing) and succinctly encoded by POSLP (post-order SLP). We next review the related techniques.

2.4. Edit Sensitive Parsing

Originally, ESP (Edit Sensitive Parsing) was introduced by Cormode and Muthukrishnan (2007) and widely applied in data compression and information retrieval (e.g., Hach et al. (2012); Takabatake et al. (2014, 2015, 2016); Nishimoto et al. (2015)). ESP is a parsing technique intended to efficiently construct a consistent parsing for same substrings as follows.

For each substring $S[i, j]$, we can decompose it into a sequence of subtrees rooted by symbols $X_1, X_2, \ldots, X_q$. For each frequent $P$ (e.g., $S[i, j] = S[k, \ell] = P$), we can find a consistent decomposition for the occurrences by the trivial decomposition of $X_1 = S[i], X_2 = S[i + 1], \ldots, X_q = S[j]$. For this problem, ESP tree guarantees a better decomposition: $(X_1, X_2, \ldots, X_q)$ is embedded into any occurrence of $P$ with a small $q$ e.g., Nakahara et al. (2013) showed that $q = \Omega(|P|/\log^2 |P|)$ and we improve it to $\Omega(|P|/\log^2 |P| \log |P|)$ in this paper. Any symbol in the decomposition is regarded to a necessary condition of an occurrence of $P$. Using this fact, we can find an approximate pattern from ESP tree. Using this result, we can efficiently compute a smaller grammar compression closely related to AFP.

We review the algorithm for ESP presented in Takabatake et al. (2016). This algorithm, referred to as ESP-comp, computes an SLP from an input string $S$. The tasks of ESP-comp are to (i) partition $S$ into $s_1 s_2 \cdots s_\ell$ such that $2 \leq |s_i| \leq 3$ for each $1 \leq i \leq \ell$, (ii) if $|s_i| = 2$, generate the production rule $X \rightarrow s_i$ and replace $s_i$ by $X$ (this subtree is referred to as a 2-tree), and if $|s_i| = 3$, generate the production rule $Y \rightarrow AX$ and $X \rightarrow BC$ for $s_i = ABC$, and replace $s_i$ by $Y$ (referred to as a 2-2-tree), (iii) iterate this process until $S$ becomes a symbol. Finally, the ESP-comp builds an SLP representing the string $S$.

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. A repetition $S[i, j]$ is called to be maximal if $S[i] \neq S[i-1], S[j+1]$. First, $S$ is uniquely partitioned into the form $u_1 x_1 u_2 x_2 \cdots u_k x_k u_{k+1}$ by its maximal repetitions, where each $x_i$ is a maximal repetition of a symbol in $\Sigma \cup V$, and each $u_i \in (\Sigma \cup V)^*$ contains no repetition. Then, each $x_i$ is called type1, each $u_i$ of length at least $2\log^* |S|$ is type2, and any remaining $u_i$ is type3. If $|u_i| = 1$, 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 $u_i$ 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 2-tree from $v[2j - 1, 2j]$ for each $j \in \{1, 2, \ldots, |v|/2\}$; otherwise, the algorithm builds a 2-tree from $v[2j - 1, 2j]$ for each $j \in \{1, 2, \ldots, (|v| - 3)/2\}$ 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.

**Alphabet reduction:** 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 ≥ 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 \( \{0, \ldots, N\} \), \( L(v)[2, |v|] \) is a sequence of integers in \( \{0, \ldots, 2\log N + 1\}^{|v|−1} \). If we apply this procedure \( \log^i N \) times, then we get \( L^*(v)[\log^i N + 1, |v|] \) a sequence of integers in \( \{0, \ldots, 5\}^{|v|−\log^i N} \), where \( L^*(v)[1, \log^i N] \) is not defined 1. When \( L^*(v)[i − 1], L^*(v)[i], L^*(v)[i + 1] \) are defined, \( v[i] \) is called the landmark if \( L^*(v)[i] > \max\{L^*(v)[i − 1], L^*(v)[i + 1]\} \).

The iteration of alphabet reduction transforms v into \( L^*(v) \) such that any substring of \( L^*(v)[\log^i N + 1, |v|] \) of length at least 12 contains at least one landmark because \( L^*(v)[\log^i N + 1, |v|] \) is also type2. Using this characteristic, the algorithm ESP-comp determines the landmarks \( v[i], i + 1 \) to be replaced for any landmark \( v[i] \), where any two landmarks are not adjacent, and then 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.

The following theorems are well-known for ESP. By Theorem 1, we can obtain the locally consistent parsing for \( S \): An iteration of ESP for \( S \), for any substring \( P \) of \( S \) there exists an interval \([i, j]\) of length at least \(|P| = \text{O}(\log^i |S|)\) such that the substring \( P[i, j] \) with each occurrence of \( P \) is transformed into a same string. Iterating this, the resulting ESP tree contains a common subtree for \( P \) regardless of its occurrence. Theorem 2 is clear by the definition of ESP. Adopting such theorems, we derive our results in the following section.

**Theorem 1** (Cormode and Muthukrishnan (2007)) For type2 substring \( v \), whether \( v[i] \) is a landmark or not is determined by only \( v[i − \text{O}(\log^i |S|)], i + \text{O}(1) \).

**Theorem 2** (Cormode and Muthukrishnan (2007)) The height of ESP tree of \( S \) is \( \text{O}(\log |S|) \).

### 2.5. Succinct Encoding

Rytter (2003) defined a partial parse tree as a binary tree built by traversing a parse tree in a depth-first manner and pruning out all the descendants under every node of a nonterminal symbol appearing before. Maruyama et al. (2012) introduced the post-order SLP (POSLP) and post-order partial parse tree (POPPT) as follows.

---

1. The number of iteration of alphabet reduction should not be changed arbitrarily according to each \( v \), and so \( N \) is set in advance to be a sufficiently large integer, e.g. \( N = \text{O}(|S|) \).
Definition 3 (POSLP and POPPT) A post-order partial parse tree is a partial parse tree whose internal nodes have post-order variables. A post-order SLP is an SLP whose partial parse tree is a post-order partial parse tree.

For a POSLP of $n$ variables, the number of nodes in the POPPT is $2n+1$ because the numbers of internal nodes and leaves are $n$ and $n+1$, respectively. Figure 1-(i)(iii) shows an example of POSLP and POPPT, respectively. The resulting POPPT (iii) has internal nodes consisting of post-order variables.

Maruyama et al. (2013b) proposed FOLCA, the fully online algorithm for computing succinct POSLP $(B, L)$. $B$ is the bit string obtained by traversing POPPT in post-order, and putting ‘0’ if a node is a leaf and ‘1’ otherwise. The last bit ‘1’ in $B$ represents the super root. $L$ is the sequence of leaves of the POPPT. The dynamic sequences $B$ and $L$ are encoded using the succinct data structure by Navarro and Sadakane (2014). Then, the following result was shown.

Theorem 4 (Maruyama et al. (2013b)) The POSLP of $n$ variables and $\sigma$ alphabet symbols supporting the phrase and reverse dictionaries can be constructed in $O\left(\frac{|S|\log\alpha}{n\log|S|}\right)$ expected time using $(1+\alpha)n\log(n+\sigma)+n(3+\log(\alpha n))$ bits memory where $\alpha \in (0,1)$ is the load factor of a hash table.

3. Algorithm

In this section, we propose a modified FOLCA for AFP with saving-space. We show the improved lower bound of the size of extracted core as well as time and space complexities. We first summarize the proposed algorithm. Let $S_i (i = 0, 1, \ldots, \lceil \log|S| \rceil)$ 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 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, and a production rule such as $A \rightarrow aa$ is generated. $a^+$ is dequeued from $q_0$, and the resulting sequence of $A$s 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. By 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 done in 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 the leaves of $T$. The pseudo code is shown in Algorithm 1.

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 of the lower bound shown by Nakahara et al. (2013).

Theorem 5 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 $\Omega\left(\frac{|P|}{\log^*|S|\log^*|P|}\right)$. 

98
Algorithm 1 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 $FB[i] = 1$ if $freq_T(X_i) \geq 2$, $D^{-1}$: 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 $\{s, ib, \ell_1, \ell_2, \ell_{\lg^* |S|}\}$ 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 $q_k[q_k.length()].\ell_i (i \in \{1, 2, \lg^* |S|\})$
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 ($q_k[u - 3].s \& (q_k[u - 2].s > q_k[u - 1].s)$ then
28:     return 0
29: else
30:     return 1
31: end if
32: end function
33: function Update($X, Y$)
34:   $z := D^{-1}(X, s, Y)$
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
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 then the parsing of $Q'_1$ inside $P$ is identical regardless of any occurrence of $P$. Otherwise, by Theorem 1, we can partition $P = Q_1Q'_1$ such that $|Q_1| = O(\log^* |S|)$, and $Q'_1$ is also identically parsed inside $P$. Let $P_1$ be the common substring in $S_1$ deriving $Q'_1$. Then, for each case, $Q_1P_1$ is a sequence of cores of $P$. Iterating this process for $P_1$ at most $k(\leq \lfloor \log |P| \rfloor)$ times, we can get a sequence $Q_1Q_2 \cdots Q_k$ of cores such that $Q_i$ is either a repetition of the form $Q_i = c_i^+$ ($c_i \in \Sigma \cup V$) or a string of length $O(\log^* |S|)$.

We show that for any $1 \leq i \leq k$, there exists a core $X_i$ in $Q_i$ with $|val(X_i)| = \Omega\left(\frac{|val(Q_i)|}{\log |S|}\right)$. If the length of $Q_i$ is $O(\log^* |S|)$, 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 $c_i^2$ is created over $Q_i$ (except for the last one, which may be a 2-2-tree deriving $c_i^3$). Iterating the parsing on the type2 sequence, we will get a large complete balanced binary tree of $c_i$. Assuming that the largest one covers $2^h c_i$’s in $Q_i$, we can see that the number of $c_i$’s in $Q_i$ is less than $5 \cdot 2^h$, namely, there is a node covering 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 $c \neq c_i$ (that must be the first character in the whole string) compose a node having $2^h$ 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), and 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 $|val(X_i)| = \Omega\left(\frac{|val(Q_i)|}{\log |S|}\right)$.

Because there is at least one $Q_j$ such that $|val(Q_j)| \geq |P|/k \geq |P|/\log |P|$, there exists a core of $P$ that derives a string of length $\Omega\left(\frac{|val(Q_j)|}{\log |S|}\right) = \Omega\left(\frac{|P|}{\log |S| \log |P|}\right)$. \hfill $\blacksquare$

Theorem 6 Algorithm 1 approximates the problem of AFP with the ratio $\Omega\left(\frac{|S| \log n}{\alpha \log \log n}\right)$ in $O\left(\frac{|S| \log n}{\alpha \log \log n}\right)$ time and $O(n + \lg |S|)$ space.

Proof The algorithm simulates the ESP of $S$ using queues $q_i$ ($i = 0, 1, \ldots, |S|$); $q_i$ stores a substring of $S_i$ to determine whether $S_i[j]$ is a landmark or not. By Theorem 1, the space for each $q_i$ is $O(\log^* |S|)$. We can reduce this space to $O(1)$ using a table of size at most $\log^* |S| \log \log |S|$ bits as follows. Applying two iterations of alphabet reduction, each symbol $A$ is transformed into a label $L_A$ of size at most $\log \log \log |S|$ bits. Whether the $A$ is a landmark or not depends on its consecutive $O(\log^* |S|)$ neighbours. Thus, the size of a table storing a 1-bit answer is at most $\log^* |S| \log \log \log |S|$ bits. It follows that the space for parsing $S$ is $O(\log |S|)$. On the other hand, by Theorem 4, the POSLP $T$ of $S$ is computable in $O\left(\frac{|S| \log n}{\alpha \log \log n}\right)$ time. By Theorem 5, for each frequent $P$, $T$ contains at least one core $X$ of $P$ satisfying $|val(X)| = \Omega\left(\frac{|P|}{\log |S| \log |P|}\right)$. Thus, finding all variables $X$ appearing 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.

4. Experimental Results

We evaluate the performance of the proposed approximation algorithm on one core of a quad-core Intel Xeon Processor E5540 (2.53GHz) machine with 144GB memory. We adopt a lightweight version of the fully-online ESP, called FOLCA Maruyama et al. (2013b), as a subroutine for the grammar compression.

We use several standard benchmarks from a text collection\(^2\), which is detailed in Table 1. We choose texts with a high and small amount of repetitions. For these texts, we examine the practical approximation ratio of the algorithm as follows. For each text \( S \), we obtained the set of frequent substrings by the compressed suffix array (SA) by Sadakane (2000), and we selected the top-100 longest patterns so 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 candidates. For each frequent substring \( P \) and a variable \( X \) reported by the algorithm, we estimate the cover ratio \( \frac{\text{val}(X)}{|P|} \) and show the average for all \( P \). However, as shown in the result below (Figure 2), the suffix array cannot be executed for a larger \( S \) due to memory consumption. Additionally, we examined the time and memory consumption of the offline algorithm by Nakahara et al. (2013).

Table 2 shows the length of optimum frequent patterns extracted by suffix array and the length of the corresponding cores extracted by our algorithm as well as the approximation ratio to the optimal one, where min./max. denote the shortest/longest pattern in the candidates, respectively. Our algorithm extracted sufficiently long cores for each benchmark.

Figure 2 shows the memory consumption for repetitive strings (Figure 2a-2c) and normal strings (Figure 2d-2f). The working space was significantly saved by our online strategy, where offline and SA were executed for each static size of data noted in the figures.

Figure 3 shows the computation time for each benchmark. Due to the time-space tradeoff of a succinct data structure, our algorithm was a few times slower than the offline and SA. The increase in computation time is acceptable for each case.

5. Conclusion

For the problem of finding frequent patterns, we proposed an online approximation algorithm with a compressed space. We improved the theoretical lower bound of the approximation ratio and presented experimental results exhibiting the efficiency for highly repetitive

\(^2\) http://pizzachili.dcc.uchile.cl/repcorpus.html
Table 2: Length of optimal $P$ extracted by suffix array (SA) and approximate $X$ by proposed algorithm (PA) with approximation ratio $\frac{|\text{val}(X)|}{|P|}$ (%) for 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 198,606</td>
<td>4,562</td>
<td>442,124</td>
<td>43,985</td>
<td>3,271</td>
<td>4,776</td>
</tr>
<tr>
<td></td>
<td>PA 18,625</td>
<td>4,096</td>
<td>37,205</td>
<td>3,382</td>
<td>268</td>
<td>477</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 935,920</td>
<td>303,204</td>
<td>2,755,550</td>
<td>98,7770</td>
<td>97,979</td>
<td>307,871</td>
</tr>
<tr>
<td></td>
<td>PA 342,136</td>
<td>58,906</td>
<td>662,630</td>
<td>16,1320</td>
<td>24,834</td>
<td>57,508</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 259,451</td>
<td>111,284</td>
<td>727,443</td>
<td>116,920</td>
<td>8,241</td>
<td>14,498</td>
</tr>
<tr>
<td></td>
<td>PA 56,584</td>
<td>12,723</td>
<td>152,903</td>
<td>24,703</td>
<td>1,926</td>
<td>3,279</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 2: Memory consumption (MB)

texts. There is still a large gap of approximation ratio between theory and practical result. An improvement of the lower bound is an important future work.
Online Grammar Compression for Frequent Pattern Discovery

References

C.C. Aggarwal and J. Han, editors. *Frequent Pattern Mining*. Springer, 2014.


Figure 3: Computation time (sec)


