Detecting regularities on grammar-compressed strings

Tomohiro I\textsuperscript{1,2}, Wataru Matsubara\textsuperscript{3}, Kouji Shimohira\textsuperscript{1}, Shunsuke Inenaga\textsuperscript{1}, Hideo Bannai\textsuperscript{1}, Masayuki Takeda\textsuperscript{1}, Kazuyuki Narisawa\textsuperscript{3}, and Ayumi Shinohara\textsuperscript{3}

\textsuperscript{1} Department of Informatics, Kyushu University, Japan
\{tomohiro.i, inenaga, bannai, takeda\}@inf.kyushu-u.ac.jp
\textsuperscript{2} Japan Society for the Promotion of Science (JSPS)
\textsuperscript{3} Graduate School of Information Sciences, Tohoku University, Japan
\{narisawa,ayumi\}@ecei.tohoku.ac.jp

Abstract. We solve the problems of detecting and counting various forms of regularities in a string represented as a Straight Line Program (SLP). Given an SLP of size $n$ that represents a string $s$ of length $N$, our algorithm computes all runs and squares in $s$ in $O(n^3 h)$ time and $O(n^2)$ space, where $h$ is the height of the derivation tree of the SLP. We also show an algorithm to compute all gapped-palindromes in $O(n^3 h + gnh \log N)$ time and $O(n^2)$ space, where $g$ is the length of the gap. The key technique of the above solution also allows us to compute the periods and covers of the string in $O(n^2 h)$ time and $O(nh(n + \log^2 N))$ time, respectively.

1 Introduction

Finding regularities such as squares, runs, and palindromes in strings, is a fundamental and important problem in stringology with various applications, and many efficient algorithms have been proposed (e.g., \cite{12,16,17,13,15,21}). See also \cite{5} for a survey.

In this paper, we consider the problem of detecting regularities in a string $s$ of length $N$ that is given in a compressed form, namely, as a straight line program (SLP), which is essentially a context free grammar in the Chomsky normal form that derives only $s$. Our model of computation is the word RAM: We shall assume that the computer word size is at least $\lceil \log_2 N \rceil$, and hence, standard operations on values representing lengths and positions of string $s$ can be manipulated in constant time. Space complexities will be determined by the number of computer words (not bits).

Given an SLP whose size is $n$ and the height of its derivation tree is $h$, Bannai et al. \cite{3} showed how to test whether the string $s$ is square-free or not, in $O(n^3 h \log N)$ time and $O(n^2)$ space. Independently, Khvorost \cite{8} presented an algorithm for computing a compact representation of all squares in $s$ in $O(n^3 h \log^2 N)$ time and $O(n^2)$ space. Matsubara et al. \cite{14} showed that a compact representation of all maximal palindromes occurring in the string $s$ can
be computed in $O(n^3 h)$ time and $O(n^2)$ space. Note that the length $N$ of the decompressed string $s$ can be as large as $O(2^n)$ in the worst case. Therefore, in such cases these algorithms are more efficient than any algorithm that work on uncompressed strings.

In this paper we present the following extension and improvements to the above work, namely,

1. an $O(n^3 h)$-time $O(n^2)$-space algorithm for computing a compact representation of squares and runs;
2. an $O(n^3 h + gnh \log N)$-time $O(n^2)$-space algorithm for computing a compact representation of palindromes with a gap (spacer) of length $g$.

We remark that our algorithms can easily be extended to count the number of squares, runs, and gapped palindromes in the same time and space complexities.

Note that Result 1 improves on the work by Khvorost [8] which requires $O(n^3 h \log^2 N)$ time and $O(n^2)$ space. The key to the improvement is our new technique of Section 3.3 called approximate doubling, which we believe is of independent interest. In fact, using the approximate doubling technique, one can improve the time complexity of the algorithms of Lifshits [10] to compute the periods and covers of a string given as an SLP, in $O(n^3 h)$ time and $O(nh(n + \log^2 N))$ time, respectively.

If we allow no gaps in palindromes (i.e., if we set $g = 0$), then Result 2 implies that we can compute a compact representation of all maximal palindromes in $O(n^3 h)$ time and $O(n^2)$ space. Hence, Result 2 can be seen as a generalization of the work by Matsubara et al. [14] with the same efficiency.

2 Preliminaries

2.1 Strings

Let $\Sigma$ be the alphabet, so an element of $\Sigma^*$ is called a string. For string $s = xyz$, $x$ is called a prefix, $y$ is called a substring, and $z$ is called a suffix of $s$, respectively. The length of string $s$ is denoted by $|s|$. The empty string $\epsilon$ is a string of length 0, that is, $|\epsilon| = 0$. For $1 \leq i \leq |s|$, $s[i]$ denotes the $i$-th character of $s$. For $1 \leq i \leq j \leq |s|$, $s[i..j]$ denotes the substring of $s$ that begins at position $i$ and ends at position $j$.

For any string $s$, let $s^R$ denote the reversed string of $s$, that is, $s^R = s[|s|] \cdots s[2]s[1]$. For any strings $s$ and $u$, let $lcp(s, u)$ (resp. $lcs(s, u)$) denote the length of the longest common prefix (resp. suffix) of $s$ and $u$.

We say that string $s$ has a period $c$ ($0 < c \leq |s|$) if $s[i] = s[i + c]$ for any $1 \leq i \leq |s| - c$. For a period $c$ of $s$, we denote $s = u^q$, where $u$ is the prefix of $s$ of length $c$ and $q = \frac{|s|}{c}$. For convenience, let $u^0 = \epsilon$. If $q \geq 2$, $s = u^q$ is called a repetition with root $u$ and period $|u|$. Also, we say that $s$ is primitive if there is no string $u$ and integer $k > 1$ such that $s = u^k$. If $s$ is primitive, then $s^2$ is called a square.

We denote a repetition in a string $s$ by a triple $⟨b, e, c⟩$ such that $s[b..e]$ is a repetition with period $c$. A repetition $⟨b, e, c⟩$ in $s$ is called a run (or maximal
periodicity in ([11]) if $c$ is the smallest period of $s[b..e]$ and the substring cannot be extended to the left nor to the right with the same period, namely neither $s[b-1..e]$ nor $s[b..e+1]$ has period $c$. Note that for any run $(b, e, c)$ in $s$, every substring of length $2c$ in $s[b..e]$ is a square. Let $\text{Run}(s)$ denote the set of all runs in $s$.

A string $s$ is said to be a palindrome if $s = s^R$. A string $s$ said to be a gapped palindrome if $s = xux^R$ for some string $u \in \Sigma^+$. Note that $u$ may or may not be a palindrome. The prefix $x$ (resp. suffix $x^R$) of $xux^R$ is called the left arm (resp. right arm) of gapped palindrome $xux^R$. If $|u| = g$, then $xux^R$ is said to be a $g$-gapped palindrome. We denote a maximal $g$-gapped palindrome in a string $s$ by a pair $(b, e)_g$ such that $s[b..e]$ is a $g$-gapped palindrome and $s[b-1..e+1]$ is not. Let $\text{gPals}(s)$ denote the set of all maximal $g$-gapped palindromes in $s$.

Given a text string $s \in \Sigma^+$ and a pattern string $p \in \Sigma^+$, we say that $p$ occurs at position $i$ ($1 \leq i \leq |s| - |p| + 1$) iff $s[i..i+|p|-1] = p$. Let $\text{Occ}(s, p)$ denote the set of positions where $p$ occurs in $s$. For a pair of integers $1 \leq b \leq e$, $[b, e] = \{b, b+1, \ldots, e\}$ is called an interval.

**Lemma 1 ([15]).** For any strings $s, p \in \Sigma^+$ and any interval $[b, e]$ with $1 \leq b \leq e \leq b + |p|$, $\text{Occ}(s, p) \cap [b, e]$ forms a single arithmetic progression if $\text{Occ}(s, p) \cap [b, e] \neq \emptyset$.

### 2.2 Straight-line programs

A straight-line program (SLP) $\mathcal{S}$ of size $n$ is a set of productions $\mathcal{S} = \{X_i \rightarrow \text{expr}_i\}^{n}_{i=1}$, where each $X_i$ is a distinct variable and each $\text{expr}_i$ is either $\text{expr}_i = X_iX_j$ ($1 \leq i < j$), or $\text{expr}_i = a$ for some $a \in \Sigma$. Note that $X_n$ derives only a single string and, therefore, we view the SLP as a compressed representation of the string $s$ that is derived from the variable $X_n$. Recall that the length $N$ of the string $s$ can be as large as $O(2^n)$. However, it is always the case that $n \geq \log N$. For any variable $X_i$, let $\text{val}(X_i)$ denote the string that is derived from variable $X_i$. Therefore, $\text{val}(X_n) = s$. When it is not confusing, we identify $X_i$ with the string represented by $X_i$.

Let $T_i$ denote the derivation tree of a variable $X_i$ of an SLP $\mathcal{S}$. The derivation tree of $\mathcal{S}$ is $T_n$ (see also Fig. 5 in Appendix C). Let $\text{height}(X_i)$ denote the height of the derivation tree $T_i$ of $X_i$ and $\text{height}(\mathcal{S}) = \text{height}(X_n)$. We associate each leaf of $T_i$ with the corresponding position of the string $\text{val}(X_i)$. For any node $z$ of the derivation tree $T_i$, let $\ell_z$ be the number of leaves to the left of $z$ in $T_i$. The position of $z$ in $T_i$ is $\ell_z + 1$.

Let $[u, v]$ be any integer interval with $1 \leq u \leq v \leq |\text{val}(X_i)|$. We say that the interval $[u, v]$ crosses the boundary of node $z$ in $T_i$, if the lowest common ancestor of the leaves $u$ and $v$ in $T_i$ is $z$. We also say that the interval $[u, v]$ touches the boundary of node $z$ in $T_i$, if either $[u - 1, v]$ or $[u, v + 1]$ crosses the boundary of $z$ in $T_i$. Assume $p = w[u..u + |p| - 1]$ and interval $[u, u + |p| - 1]$ crosses or touches the boundary of node $z$ in $T_i$. When $z$ is labeled by $X_j$, then we also say that the occurrence of $p$ starting at position $u$ in $\text{val}(X_i)$ crosses or touches the boundary of $X_j$.  

3
Lemma 2 ([4]). Given an SLP $S$ of size $n$ describing string $w$ of length $N$, we can pre-process $S$ in $O(n)$ time and space to answer the following queries in $O(\log N)$ time:

- Given a position $u$ with $1 \leq u \leq N$, answer the character $w[u]$.
- Given an interval $[u, v]$ with $1 \leq u \leq v \leq N$, answer the node $z$ the interval $[u, v]$ crosses, the label $X_i$ of $z$, and the position of $z$ in $T_S = T_n$.

For any production $X_i \rightarrow X_l X_r$ and a string $p$, let $Occ^\xi(X_i, p)$ be the set of occurrences of $p$ which begin in $X_l$ and end in $X_r$. Let $S$ and $T$ be SLPs of sizes $n$ and $m$, respectively. Let the AP-table for $S$ and $T$ be an $n \times m$ table such that for any pair of variables $X \in S$ and $Y \in T$ the table stores $Occ^\xi(X, Y)$. It follows from Lemma 1 that $Occ^\xi(X, Y)$ forms a single arithmetic progression which requires $O(1)$ space, and hence the AP-table can be represented in $O(nm)$ space.

Lemma 3 ([10]). Given two SLPs $S$ and $T$ of sizes $n$ and $m$, respectively, the AP-table for $S$ and $T$ can be computed in $O(nmh)$ time and $O(nm)$ space, where $h = \text{height}(S)$.

Lemma 4 ([10], local search (LS)). Using AP-table for $S$ and $T$ that describe strings $p$ in $s$, we can compute, given any position $b$ and constant $\alpha > 0$, $Occ(s, p) \cap [b, b + \alpha|p|]$ as a form of at most $\lceil \alpha \rceil$ arithmetic progressions in $O(h)$ time, where $h = \text{height}(S)$.

Note that, given any $1 \leq i \leq j \leq |s|$, we are able to build an SLP of size $O(n)$ that generates substring $s[i..j]$ in $O(n)$ time. Hence, by computing the AP-table for $S$ and the new SLP, we can conduct the local search LS operation on substring $s[i..j]$ in $O(n^2 h)$ time.

For any variable $X_i$ of $S$ and positions $1 \leq k_1, k_2 \leq |X_i|$, we define the “right-right” longest common extension query by

$$\text{LCE}(X_i, k_1, k_2) = \text{lcp}(X_i[k_1..|X_i|], X_i[k_2..|X_i|]).$$

Using a technique of [15] in conjunction with Lemma 3 it is possible to answer the query in $O(n^2 h)$ time for each pair of positions, with no pre-processing. We will later show our new algorithm which, after $O(n^2 h)$-time pre-processing, answers to the LCE query for any pair of positions in $O(h \log N)$ time.

3 Finding runs

In this section we propose an $O(n^3 h)$-time and $O(n^2)$-space algorithm to compute $O(n \log N)$-size representation of all runs in a text $s$ of length $N$ represented by SLP $S = \{X_i \rightarrow \text{expr}_i\}_{i=1}^n$ of height $h$.

For each production $X_i \rightarrow X_{\ell(i)} X_{r(i)}$ with $i \leq n$, we consider the set $\text{Run}^\xi(X_i)$ of runs which touch or cross the boundary of $X_i$ and are completed in $X_i$, i.e., those that are not prefixes nor suffixes of $X_i$. Formally,

$$\text{Run}^\xi(X_i) = \{(b, e, c) \in \text{Run}(X_i) \mid 1 \leq b - 1 \leq |X_{\ell(i)}| < e + 1 \leq |X_i|\}.$$
It is known that for any interval $[b, e]$ with $1 \leq b \leq e \leq |s|$, there exists a unique occurrence of a variable $X_i$ in the derivation tree of SLP, such that the interval $[b, e]$ crosses the boundary of $X_i$. Also, wherever $X_i$ appears in the derivation tree, the runs in $\text{Run}_X(X_i)$ occur in $s$ with some appropriate offset, and these occurrences of the runs are never contained in $\text{Run}_X(X_j)$ with any other variable $X_j$ with $j \neq i$. Hence, by computing $\text{Run}_X(X_i)$ for all variables $X_i$ with $i \leq n$, we can essentially compute all runs of $s$ that are not prefixes nor suffixes of $s$. In order to detect prefix/suffix runs of $s$, it is sufficient to consider two auxiliary variables $X_{n+1} \rightarrow X_S X_n$ and $X_{n+2} \rightarrow X_{n+1} X_{S'}$, where $X_S$ and $X_{S'}$ respectively derive special characters $\$ and $\$ that are not in $s$ and $\$ that are not in $s$. Hence, the problem of computing the runs from an SLP $S$ reduces to computing $\text{Run}_X(X_i)$ for all variables $X_i$ with $i \leq n + 2$.

Our algorithm is based on the divide-and-conquer method used in \cite{3} and also \cite{8}, which detect squares crossing the boundary of each variable $X_i$. Roughly speaking, in order to detect such squares we take some substrings of $\text{val}(X_i)$ as seeds each of which is in charge of distinct squares, and for each seed we detect squares by using LS and LCE constant times. There is a difference between \cite{3} and \cite{8} in how the seeds are taken, and ours is rather based on that in \cite{3}.

In the next subsection, we briefly describe our basic algorithm which runs in $O(n^3 h \log N)$ time.

### 3.1 Basic algorithm

Consider runs in $\text{Run}_X(X_i)$ with $X_i \rightarrow X_t X_r$. Since a run in $\text{Run}_X(X_i)$ contains a square which touches or crosses the boundary of $X_i$, our algorithm finds a run by first finding such a square, and then computing the maximal extension of its period to the left and right of its occurrence.

We divide each square $ww$ by its length and how it relates to the boundary of $X_i$. When $|w| > 1$, there exists $1 \leq t < \log |\text{val}(X_i)|$ such that $2^t \leq |w| < 2^{t+1}$ and there are four cases (see also Fig. 1): (1) $|w_\ell| \geq \frac{3}{2}|w|$, (2) $\frac{3}{2}|w| > |w_\ell| \geq |w|$, (3) $|w_\ell| > |w_\ell| \geq \frac{3}{2}|w|$, (4) $\frac{3}{2}|w| > |w_\ell|$, where $w_\ell$ is a prefix of $ww$ which is also a suffix of $\text{val}(X_i)$.

The point is that in any case we can take a substring $p$ of length $2^{t-1} - 1$ of $s$ which touches the boundary of $X_i$, and is completely contained in $w$. By using $p$ as a seed we can detect runs by the following steps:

**Step 1:** Conduct local search of $p$ in an “appropriate range” of $X_i$, and find a copy $p'$ ($= p$) of $p$.

**Step 2:** Compute the length $\text{plen}$ of the longest common prefix to the right of $p$ and $p'$, and the length $\text{slen}$ of the longest common suffix to the left of $p$ and $p'$, then check that $\text{plen} + \text{slen} \geq d - |p|$, where $d$ is the distance between the beginning positions of $p$ and $p'$.

Notice that Step 2 actually computes maximal extension of the repetition.

Since $d = |w|$, it is sufficient to conduct local search in the range satisfying $2^t \leq d < 2^{t+1}$, namely, the width of the interval for local search is smaller than
2|p|, and all occurrences of p' are represented by at most two arithmetic progressions. Although exponentially many runs can be represented by an arithmetic progression, its periodicity enables us to efficiently detect all of them, by using LCE only constant times, and they are encoded in $O(1)$ space. We put the details in Appendix A since the employed techniques are essentially the same as in \cite{8}.

By varying $t$ from 1 to $\log N$, we can obtain an $O(\log N)$-size compact representation of $\text{Run}^t(X_i)$ in $O(n^2h \log N)$ time. More precisely, we get a list of $O(\log N)$ quintuplets $(\delta_1, \delta_2, \delta_3, c, k)$ such that the union of sets $\bigcup_{j=0}^{k-1}(\delta_1 - cj, \delta_2 + cj, \delta_3 + cj)$ for all elements of the list equals to $\text{Run}^t(X_i)$ without duplicates. By applying the above procedure to all the $n$ variables, we can obtain an $O(n \log N)$-size compact representation of all runs in $s$ in $O(n^3h \log N)$ time. The total space requirement is $O(n^2)$, since we need $O(n^2)$ space at each step of the algorithm.

In order to improve the running time of the algorithm to $O(n^3h)$, we will use new techniques of the two following subsections.

### 3.2 Longest common extension

In this subsection we propose a more efficient algorithm for LCE queries.

**Lemma 5.** We can pre-process an SLP $S$ of size $n$ and height $h$ in $O(n^2h)$ time and $O(n^2)$ space, so that given any variable $X_i$ and positions $1 \leq k_1, k_2 \leq |X_i|$, $\text{LCE}(X_i, k_1, k_2)$ is answered in $O(h \log N)$ time.
To compute \( \text{LCE}(X_i, k_1, k_2) \) we will use the following function: For an SLP \( S = \{X_i \rightarrow expr_i\}_{i=1}^n \), let \( \text{Match} \) be a function such that

\[
\text{Match}(X_i, X_j, k) = \begin{cases} 
\text{true} & \text{if } k \in \text{Occ}(X_i, X_j), \\
\text{false} & \text{if } k \notin \text{Occ}(X_i, X_j).
\end{cases}
\]

**Lemma 6.** We can pre-process a given SLP \( S \) of size \( n \) and height \( h \) in \( O(n^2h) \) time and \( O(n^2) \) space so that the query \( \text{Match}(X_i, X_j, k) \) is answered in \( O(\log N) \) time.

**Proof.** We apply Lemma 2 to every variable \( X_i \) of \( S \), so that the queries of Lemma 2 is answered in \( O(\log N) \) time on the derivation tree \( T_i \) of each variable \( X_i \) of \( S \). Since there are \( n \) variables in \( S \), this takes a total of \( O(n^2) \) time and space. We also apply Lemma 3 to \( S \), which takes \( O(n^2h) \) time and \( O(n^2) \) space. Hence the pre-processing takes a total of \( O(n^2h) \) time and \( O(n^2) \) space.

To answer the query \( \text{Match}(X_i, X_j, k) \), we first find the node of \( T_i \) the interval \( [k, k+|X_j|-1] \) crosses, its label \( X_q \), and its position \( r \) in \( T_i \). This takes \( O(\log N) \) time using Lemma 2. Then we check in \( O(1) \) time if \( (k-r) \in \text{Occ}(X_q, X_j) \) or not, using the arithmetic progression stored in the AP-table. Thus the query is answered in \( O(\log N) \) time.

The following function will also be used in our algorithm: Let \( \text{FirstMismatch} \) be a function such that

\[
\text{FirstMismatch}(X_i, X_j, k) = \begin{cases} 
|\text{lcp}(X_i|k..|X_i|], X_j)| & \text{if } |X_i| - k + 1 \leq |X_j|, \\
\text{undefined} & \text{otherwise}.
\end{cases}
\]

Using Lemma 3, we can establish the following lemma. See Appendix B for a full proof.

**Lemma 7.** We can pre-process a given SLP \( S \) of size \( n \) and height \( h \) in \( O(n^2h) \) time and \( O(n^2) \) space so that the query \( \text{FirstMismatch}(X_i, X_j, k) \) is answered in \( O(h\log N) \) time.

We are ready to prove Lemma 3.

**Proof.** Consider to compute \( \text{LCE}(X_i, k_1, k_2) \). Without loss of generality, assume \( k_1 \leq k_2 \). Let \( z \) be the lca of the \( k_1 \)-th and \( (k_2-k_1+|X_i|) \)-th leaves of the derivation tree \( T_i \). Let \( P_z \) be the path from \( z \) to the \( k_1 \)-th leaf of the derivation tree \( T_i \), and let \( L \) be the list of the right child of the nodes in \( P_z \) sorted in increasing order of their position in \( T_i \). The number of nodes in \( L \) is at most \( \text{height}(X_i) \leq h \), and \( L \) can be computed in \( O(\text{height}(X_i)) = O(h) \) time. Let \( P_r \) be the path from \( z \) to the \( (k_2-k_1+|X_i|) \)-th leaf of the derivation tree \( T_i \), and let \( R \) be the list of the left child of the nodes in \( P_r \) sorted in increasing order of their position in \( T_i \). \( R \) can be computed in \( O(h) \) time as well. Let \( U = L \cup R = \{X_{u(1)}, X_{u(2)}, \ldots, X_{u(m)}\} \) be the list obtained by concatenating \( L \) and \( R \). For each \( X_{u(p)} \) in increasing order of \( p = 1, 2, \ldots, m \), we perform query \( \text{Match}(X_i, X_{u(p)}, k_1 + \sum_{q=1}^{p-1} |X_{u(q)}|) \) until
either finding the first variable $X_{u(p')}$ for which the query returns false (see also Fig. 4 in Appendix C), or all the queries for $p = 1, \ldots, m$ have returned true. In the latter case, clearly $\text{LCR}(X_i, k_1, k_2) = |X_i| - k_1 + 1$. In the former case, the first mismatch occurs between $X_i$ and $X_{u(p')}$, and hence $\text{LCR}(X_i, k_1, k_2) = \sum_{q = 1}^{p' - 1} |X_{u(q')}| + \text{FirstMismatch}(X_i, X_{u(p')}, k_1 + \sum_{q = 1}^{p' - 1} |X_{u(q')}|)$.

Since $U$ contains at most $2 \cdot \text{height}(X_i)$ variables, we perform $O(h)$ Match queries. We perform at most one FirstMismatch query. Thus, using Lemmas 9 and 7 we can compute $\text{LCR}(X_i, k_1, k_2)$ in $O(h \log N)$ time after $O(n^2 h)$-time $O(n^2)$-space pre-processing.

We can use Lemma 5 to also compute “left-left”, “left-right”, and “right-right” longest common extensions on the uncompressed string $s = \text{val}(\mathcal{S})$: We compute in $O(n)$ time an SLP $\mathcal{S}^R$ of size $n$ which represents the reversed string $s^R$. We then construct a new SLP $\mathcal{S}'$ of size $2n$ and height $h + 1$ by concatenating the last variables of $\mathcal{S}$ and $\mathcal{S}^R$, and apply Lemma 6 to $\mathcal{S}'$.

### 3.3 Approximate doubling

Here we show how to reduce the number of AP-table computation required in Step 1 of the basic algorithm, from $O((\log N)^2)$ to $O(1)$ times per variable.

Consider any production $X_i \rightarrow X_iX_r$. If we build a new SLP which contains variables that derive the prefixes of length $2^t$ of $X_r$ for each $0 \leq t < \log |X_r|$, we can obtain the AP-tables for $X_i$ and all prefix seeds of $X_r$ by computing the AP-table for $X_i$ and the new SLP. Unfortunately, however, the size of such a new SLP can be as large as $O(n \log N)$. Here we notice that the lengths of the seeds do not have to be exactly doublings, i.e., the basic algorithm of Section 3.1 works fine as long as the following properties are fulfilled: (a) the ratio of the lengths for each pair of consecutive seeds is constant; (b) the whole string is covered by the $O((\log N)^2)$ seeds.\footnote{A minor modification is that we conduct local search for a seed $p$ at Step 1 with the range satisfying $2|p| \leq d < 2|q|$, where $q$ is the next longer seed of $p$.}

We show in the next lemma that we can build an approximate doubling SLP of size $O(n)$.

**Lemma 8.** Let $\mathcal{S} = \{X_i \rightarrow \text{expr}_i\}_{i=1}^n$ be an SLP that derives a string $s$. We can build in $O(n)$ time a new SLP $\mathcal{S}' = \{Y_i \rightarrow \text{expr}_i'\}_{i=1}^n$ with $n' = O(n)$ and height($\mathcal{S}'$) = $O(\text{height}(\mathcal{S}))$, which derives $s$ and contains $O(\log N)$ variables $Y_{a_1}, Y_{a_2}, \ldots, Y_{a_k}$ satisfying the following conditions:

- For any $1 \leq j \leq k$, $Y_{a_j}$ derives a prefix of $s$, $|Y_{a_1}| = 1$ and $|Y_{a_k}| = |s|$.
- For any $1 \leq j < k$, $|Y_{a_j}| < |Y_{a_{j+1}}| \leq 2|Y_{a_j}|$.

**Proof.** First, we copy the productions of $\mathcal{S}$ into $\mathcal{S}'$. Next we add productions needed for creating prefix variables $Y_{a_1}, Y_{a_2}, \ldots, Y_{a_k}$ in increasing order. We consider separating the derivation tree $T_n$ of $X_n$ into segments by a sequence of nodes $v_1, v_2, \ldots, v_k$ such that the $i$-th segment enclosed by the path from $v_i$ to $v_{i+1}$ represents the suffix of $Y_{a_{i+1}}$ of length $|Y_{a_{i+1}}| - |Y_{a_i}|$, namely, $Y_{a_{i+1}} \rightarrow Y_{a_i}Y_{b_i}$.
For each of the $O$ satisfies one of the following conditions; (1) the rightmost position of $N$ of length $1 \leq X_2 n$ moving down due to Condition (2), the number of the traversed edges as well as Fig. 7 in Appendix C). Since each variable root to $v$ of $X$ such nodes. Consider all the edges we have traversed in the derivation tree new variables needed for representing the segment is bounded by the number of the $X$ have respectively approximately doubling suffix variables of $X_\ell$ and prefix variables of $X_r$, and compute two AP-tables for $S$ and each of them in $O(n^2 h)$ time. For each of the $O(\log N)$ prefix/suffix variables, we use it as a seed and find

- If Condition (1) holds, $v_{i+1}$ is set to be an l-node. It is clear that the length of the $i$-th segment is exactly $|Y_a_i|$ and $|Y_{a+i-1}| = 2|Y_a_i|$. 
- If Condition (1) does not hold but Condition (2) holds, $v_{i+1}$ is set to be an r-node. Since $v_{i+1}$ contains position $2|Y_a_i|$, the length of the $i$-th segment is less than $|Y_a_i|$ and $|Y_{a+i-1}| < 2|Y_a_i|$. We remark that since $X_j$ appears in $Y_{a+i+1}$, then $|Y_{a+i+1}|+|X_j| \leq 2|Y_{a+i+1}|$, and therefore, we never move down $v_{i+1}$ for the segments to follow.

We iterate the above procedures until we obtain a prefix variable $Y_{a_k-1}$ that satisfies $|X_n| \leq 2|Y_{a_{k-1}}|$. We let $u_k$ be the deepest node on the path from the root to $v_{k-1}$ which contains position $|s|$, and let $v_k$ be the right child of $u_k$. Since $|Y_{a_k}| < 2|Y_{a_{k+1}}|$ for any $1 \leq i < k$, $k = O(\log N)$ holds.

We note that the $i$-th segment can be represented by the concatenation of “inner” nodes attached to the path from $v_i$ to $v_{i+1}$, and hence, the number of new variables needed for representing the segment is bounded by the number of such nodes. Consider all the edges we have traversed in the derivation tree $T_n$ of $X_n$. Each edge contributes to at most one new variable for some segment (see also Fig. 4 in Appendix C). Since each variable $X_j$ is used constant times for moving down due to Condition (2), the number of the traversed edges as well as $n'$ is $O(n)$. Also, it is easy to make the height of $Y_{a_k}$ be $O(\text{height}(S')) = O(\log N + \text{height}(S')) = O(\text{height}(S))$. □

3.4 Improved algorithm

Using Lemmas 5 and 8 we get the following theorem.

**Theorem 1.** Given an SLP $S$ of size $n$ and height $h$ that describes string $s$ of length $N$, an $O(n \log N)$-size compact representation of all runs in $s$ can be computed in $O(n^3 h)$ time and $O(n^2)$ working space.

**Proof.** Using Lemma 5 we first pre-process $S$ in $O(n^3 h)$ time so that any “right-right” or “left-left” LCE query can be answered in $O(h \log N)$ time. For each variable $X_i \rightarrow X_\ell X_r$, using Lemma 8 we build temporal SLPs $T$ and $T'$ which have respectively approximately doubling suffix variables of $X_\ell$ and prefix variables of $X_r$, and compute two AP-tables for $S$ and each of them in $O(n^2 h)$ time. For each of the $O(\log N)$ prefix/suffix variables, we use it as a seed and find
all corresponding runs by using LS and LCE queries constant times. Hence the time complexity is $O(n^2h + n(n^2h + (h + h \log N) \log N)) = O(n^3h)$. The space requirement is $O(n^2)$, the same as the basic algorithm.

4 Finding $g$-gapped palindromes

A similar strategy to finding runs on SLPs can be used for computing a compact representation of the set $g\text{Pals}(s)$ of $g$-gapped palindromes from an SLP $S$ that describes string $s$. As in the case of runs, we add two auxiliary variables $X_{n+1} \rightarrow X_{n}$ and $X_{n+2} \rightarrow X_{n+1}X_{g}$. For each production $X_i \rightarrow X_{\ell}X_r$ with $i \leq n + 2$, we consider the set $g\text{Pals}^g(X_i)$ of $g$-gapped palindromes which touch or cross the boundary of $X_i$ and are completed in $X_i$, i.e., those that are not prefixes nor suffixes of $X_i$. Formally,

$$g\text{Pals}^g(X_i) = \{(b, e)_g \in g\text{Pals}(X_i) \mid 1 \leq b - 1 \leq |X_\ell| < e + 1 \leq |X_i|\}.$$

Each $g$-gapped palindrome in $X_i$ can be divided into three groups (see also Fig. 2): (1) its right arm crosses or touches with its right end the boundary of $X_i$, (2) its left arm crosses or touches with its left end the boundary of $X_i$, (3) the others.

For Case (3), for every $|X_\ell| - g + 1 \leq j < |X_\ell|$ we check if $\text{lcp}(X_{\ell}[1..j], X_{\ell}[j+g+1..|X_i|]) > 0$ or not. From Lemma 5 it can be done in $O(gh \log N)$ time for any variable by using “left-right” LCE (excluding pre-processing time for LCE). Hence we can compute all such $g$-gapped palindromes for all productions in $O(n^2h + gh \log N)$ time, and clearly they can be stored in $O(ng)$ space.

For Case (1), let $w_\ell$ be the prefix of the right arm which is also a suffix of $val(X_\ell)$. We take approximately doubling suffixes of $X_\ell$ as seeds. Let $p$ be the longest seed that is contained in $w_\ell$. We can find $g$-gapped palindromes by the following steps:

---

Fig. 2. Three groups of $g$-gapped palindromes to be found in $X_i$. 

---
Step 1: Conduct local search of $p' = p^R$ in an “appropriate range” of $X_i$ and find it in the left arm of palindrome.

Step 2: Compute “right-left” LCE of $p'$ and $p$, then check that the gap can be $g$. The outward maximal extension can be obtained by computing “left-right” LCE queries on the occurrences of $p'$ and $p$.

As in the case of runs, for each seed, the length of the range where the local search is performed in Step 1 is only $O(|p|)$. Hence, the occurrences of $p'$ can be represented by a constant number of arithmetic progressions. Also, we can obtain $O(1)$-space representation of $g$-gapped palindromes for each arithmetic progression representing overlapping occurrences of $p'$, by using a constant number of LCE queries. Therefore, by processing $O(\log N)$ seeds for every variable $X_i$, we can compute in $O(n^2h + n(n^2h + (h + h\log N)\log N)) = O(n^3h)$ time an $O(n\log N)$-size representation of all $g$-gapped palindromes for Case (1) in $s$.

In a symmetric way of Case (1), we can find all $g$-gapped palindromes for Case (2). Putting all together, we get the following theorem.

Theorem 2. Given an SLP of size $n$ and height $h$ that describes string $s$ of length $N$, and non-negative integer $g$, an $O(n\log N + ng)$-size compact representation of all $g$-gapped palindromes in $s$ can be computed in $O(n^3h + gnh\log N)$ time and $O(n^2)$ working space.

5 Discussions

Let $\mathbb{R}$ and $\mathbb{G}$ denote the output compact representations of the runs and $g$-gapped palindromes of a given SLP $S$, respectively, and let $|\mathbb{R}|$ and $|\mathbb{G}|$ denote their size. Here we show an application of $\mathbb{R}$ and $\mathbb{G}$; given any interval $[b, e]$ in $s$, we can count the number of runs and gapped palindromes in $s[b..e]$ in $O(n + |\mathbb{R}|)$ and $O(n + |\mathbb{G}|)$ time, respectively. We will describe only the case of runs, but a similar technique can be applied to gapped palindromes. As is described in Section 5.2, $s[b..e]$ can be represented by a sequence $U = (X_{u(1)}, X_{u(2)}, \ldots, X_{u(m)})$ of $O(h)$ variables of $S$. Let $T$ be the SLP obtained by concatenating the variables of $U$. There are three different types of runs in $\mathbb{R}$: (1) runs that are completely within the subtree rooted at one of the nodes of $U$; (2) runs that begin and end inside $[b, e]$ and cross or touch any border between consecutive nodes of $U$; (3) runs that begin and/or end outside $[b, e]$. Observe that the runs of types (2) and (3) cross or touch the boundary of one of the nodes in the path from the root to the $b$-th leaf of the derivation tree $T_S$, or in the path from the root to the $e$-th leaf of $T_S$. A run that begins outside $[b, e]$ is counted only if the suffix of the run that intersects $[b, e]$ has an exponent of at least 2. The symmetric variant applies to a run that ends outside $[b, e]$. Thus, the number of runs of types (2) and (3) can be counted in $O(n + 2|\mathbb{R}|)$ time. Since we can compute in a total of $O(n)$ time the number of nodes in the derivation tree of $T$ that are labeled by $X_i$ for all variables $X_i$, the number of runs of type (1) for all variables $X_{u(j)}$ can be counted in $O(n + |\mathbb{R}|)$ time. Noticing that runs are compact representation
of squares, we can also count the number of occurrences of all squares in $s[b..e]$ in $O(n + |R|)$ time by simple arithmetic operations.

The approximate doubling and LCE algorithms of Section 3 can be used as basis of other efficient algorithms on SLPs. For example, using approximate doubling, we can reduce the number of pairs of variables for which the AP-table has to be computed in the algorithms of Lifshits [19], which compute compact representations of all periods and covers of a string given as an SLP. As a result, we improve the time complexities from $O(n^2 h \log N)$ to $O(n^2 h)$ for periods, and from $O(n^2 h \log^2 N)$ to $O(nh(n + \log^2 N))$ for covers.

References

1. Apostolico, A., Breslauer, D.: An optimal $O(\log \log N)$-time parallel algorithm for detecting all squares in a string. SIAM Journal on Computing 25(6), 1318–1331 (1996)
2. Apostolico, A., Breslauer, D., Galil, Z.: Parallel detection of all palindromes in a string. Theor. Comput. Sci. 141(1&2), 163–173 (1995)
3. Bannai, H., Gagie, T., I, T., Inenaga, S., Landau, G.M., Lewenstein, M.: An efficient algorithm to test square-freeness of strings compressed by straight-line programs. Inf. Process. Lett. 112(19), 711–714 (2012)
4. Bille, P., Landau, G.M., Raman, R., Sadakane, K., Satti, S.R., Weimann, O.: Random access to grammar-compressed strings. In: Proc. SODA 2011. pp. 373–389 (2011)
5. Crochemore, M., Ilie, L., Rytter, W.: Repetitions in strings: Algorithms and combinatorics. Theor. Comput. Sci. 410(50), 5227–5235 (2009)
6. Crochemore, M., Rytter, W.: Efficient parallel algorithms to test square-freeness and factorize strings. Information Processing Letters 38(2), 57–60 (1991)
7. Jansson, J., Peng, Z.: Online and dynamic recognition of squarefree strings. International Journal of Foundations of Computer Science 18(2), 401–414 (2007)
8. Khvorost, L.: Computing all squares in compressed texts. In: Proceedings of the 2nd Russian Finnish Symposium on Discrete Mathematics. vol. 17, pp. 116–122 (2012)
9. Kolpakov, R.M., Kucherov, G.: Finding maximal repetitions in a word in linear time. In: FOCS. pp. 596–604 (1999)
10. Lifshits, Y.: Processing compressed texts: A tractability border. In: Proc. CPM 2007. LNCS, vol. 4580, pp. 228–240 (2007)
11. Main, M.G.: Detecting leftmost maximal periodicities. Discrete Applied Mathematics 25(1-2), 145–153 (1989)
12. Main, M.G., Lorentz, R.J.: An $O(n \log n)$ algorithm for finding all repetitions in a string. Journal of Algorithms 5(3), 422–432 (1984)
13. Manacher, G.K.: A new linear-time “on-line” algorithm for finding the smallest initial palindrome of a string. J. ACM 22(3), 346–351 (1975)
14. Matsubara, W., Inenaga, S., Ishino, A., Shinohara, A., Nakamura, T., Hashimoto, K.: Efficient algorithms to compute compressed longest common substrings and compressed palindromes. Theoretical Computer Science 410(8–10), 900–913 (2009)
15. Miyazaki, M., Shinohara, A., Takeda, M.: An improved pattern matching algorithm for strings in terms of straight-line programs. In: Proceedings of the 8th Annual Symposium on Combinatorial Pattern Matching. pp. 1–11 (1997)
Appendix A: Details of the algorithm to find runs

In this section, we describe how we process occurrences of $p'$ at Step 2 of the basic algorithm. To handle occurrences of $p'$ that are represented by an arithmetic progression, we make use of its periodicity.

For any string $s$ and positive integer $c \leq |s|$, let $\hat{r}_c \cdot p_c(s)$ (resp. $\hat{r}_c \cdot p_c(s)$) denote the length of the longest prefix (resp. suffix) of $s$ having period $c$.

Lemma 9. Let $s, p \in \Sigma^+$ and $\{a_0, a_1, \ldots, a_k\}$ be consecutive occurrences of $p$ in $s$ that form a single arithmetic progression with common difference $c \leq |p|$. Let $z_j = s[a_j + |p|..|s|]$ and $z_j' = s[1.a_j - 1]$ for any $0 \leq j \leq k$. For any non-empty strings $x, x' \in \Sigma^+$, it holds that

$$lcp(z_j, x) = \begin{cases} 
\min\{\alpha - c_j, \beta\} & \text{if } \alpha - c_j \neq \beta; \\
\beta + lcp(z_0[\beta + 1..|z_0|], x[\beta + 1..|x|]) & \text{otherwise, and}
\end{cases}$$

$$lcs(z_j', x') = \begin{cases} 
\min\{\overline{\alpha} + c_j, \overline{\beta}\} & \text{if } \overline{\alpha} + c_j \neq \overline{\beta}; \\
\beta + lcs(z_0'[1..|z_0'| - \overline{\beta}], x'[1..|x'| - \overline{\beta}]) & \text{otherwise},
\end{cases}$$

where $\overline{\alpha} = \hat{r}_c \cdot p_c(z_0) - |p|$, $\overline{\beta} = \hat{r}_c \cdot p_c(px) - |p|$, $\overline{\alpha} = \hat{r}_c \cdot p_c(z_0p) - |p|$ and $\overline{\beta} = \hat{r}_c \cdot p_c(x'p)$.

Proof. Since $\hat{r}_c \cdot p_c(z_j) = \overline{\alpha} - c_j + |p|$, both $pz_j$ and $px$ have a prefix of length $\min\{\overline{\alpha} - c_j, \overline{\beta}\} + |p|$ with period $c$ (see also Fig. 3). If $\overline{\alpha} - c_j \neq \overline{\beta}$, either $pz_j$ or $px$ has a prefix of length $\min\{\overline{\alpha} - c_j, \overline{\beta}\} + |p| + 1$ with period $c$ while the other does not, and hence $lcp(z_j, x) = lcp(pz_j, px) - |p| = \min\{\overline{\alpha} - c_j, \overline{\beta}\}$. Only when the period breaks the periodicity, i.e., $\overline{\alpha} - c_j = \overline{\beta}$, $lcp(z_j, x)$ could expand. Note that such expansion occurs at most once. Similarly, since $\hat{r}_c \cdot p_c(x'_j) = \overline{\alpha} + c_j$ we get the statement for $lcs(z_j', x')$. \qed

![Fig. 3. Illustration for Lemma 9](image)

In the next lemma, we show how to handle one of the arithmetic progressions computed in Step 2 of Case (3).
Lemma 10. Let $X_i \rightarrow X_k X_r$ be a production of an SLP of size $n$ and $p$ be the suffix of $val(X_i)$ of length $2^{l-1}$. Let $\{a_0, a_1, \ldots, a_k\}$ be consecutive occurrences of $p'$ in $val(X_i)$ which form a single arithmetic progression, which are computed in Step 2 of Case (3). We can detect all runs corresponding to the occurrences of $p'$ by using LCE constant times. Also, such runs are represented in constant space.

Proof. We apply Lemma 9 by setting $s = val(X_i)$, $x = val(X_r)$ and $x' = val(X_k) [1..|val(X_i)| - |p|]$. First we compute $\vec{\alpha} = lcp(pz_0, p[c+1..|p|]z_0) + c - |p|$, $\vec{\beta} = lcp(px, p[c+1..|p|]x) + c - |p|$, $\vec{\gamma} = lcs(z_0^p, z_0^p[1..|p| - c]) + c - |p|$ and $\vec{\delta} = lcs(x'p, x'p[1..|p| - c]) + c - |p|$ by using $lcp$ and $lcs$ four times.

Claim. If $\vec{\beta} + \vec{\gamma} \geq a_0 - 1 + c$, the root of any repetition detected from $a_j$ is not primitive.

Proof of Claim. If $\vec{\beta} + \vec{\gamma} \geq a_0 - 1 + c$, py must have period $c$, where $y$ is the prefix of length $a_1 - 1$ of $x$. Since $pyp[c+1..c+p] = p$, $|yp| - c$ is a period of $yp$. It follows from the periodicity lemma that $py$, as well as every $a_j + |p| - 1$, is divisible by greatest common divisor of $c$ and $|yp| - c$, and hence the root of any repetition detected from $a_j$ is not primitive.

From the above claim, in what follows we assume that $\vec{\beta} + \vec{\gamma} < a_0 - 1 + c$. Let $d_j = a_j - 1 + |p| = a_0 - 1 + |p| + cj$, and then we want to check if $lcp(z_j, x) + lcs(z_j', x') \geq d_j - |p| = a_0 - 1 + cj$, or equivalently, $lcp(z_j, x) + lcs(z_j', x') - cj \geq a_0 - 1$. We show that the root of such repetition $(\delta_1, \delta_2 + cj, \delta_3 + cj)$ appears iff $\vec{\beta} + \vec{\gamma} \geq a_0 - 1$, where $\delta_1 = |x'| + 1 - \vec{\alpha}$, $\delta_2 = a_0 + |p| + \vec{\beta} - 1$ and $\delta_3 = a_0 + |p| - 1$ are constants.

We show that the root of such repetition $(\delta_1, \delta_2 + cj, \delta_3 + cj)$ is primitive. Assume on the contrary that it is not primitive, namely, $s' \equiv s[\delta_1 - cj, \delta_2 + cj] = u^q$ with $|u| \leq (\delta_3 + cj)/2$ and $q \geq 4$. Evidently, $\vec{\mathbb{r}}(s') = \vec{\beta} + lcs(z_j', x') + |p| = \vec{\beta} + \vec{\alpha} + |p| + cj$. It follows from $a_0 - 1 \leq \vec{\beta} + \vec{\alpha} < a_0 - 1 + c$ that $\delta_3 + cj \leq \vec{\mathbb{r}}(s') < \delta_3 + cj + c < |s'|$. Since $2|u| \leq \vec{\mathbb{r}}(s')$ and $c \leq |p| \leq (\delta_3 + cj)/2 \leq \delta_3 + cj - |u| \leq \vec{\mathbb{r}}(s') - |u|$, $\vec{\mathbb{r}}(s'[1..|s'| - |u|]) = \vec{\mathbb{r}}(s'[|u| + 1..|s'|]) + |u|$, however both $s'[1..|s'| - |u|]$ and $s'[|u| + 1..|s'|]$ are $u^{q-1}$, a contradiction. Therefore, for all $0 \leq j < \min\{j', j''\}$, $(\delta_1 - cj, \delta_2 + cj, \delta_3 + cj)$ are runs, and they can be encoded by a quintuplet $(\delta_1, \delta_2, \delta_3, c, \min\{j', j''\})$.

For any $\min\{j', j''\} \leq j \leq k$, except for $j = j'$ or $j''$, $lcp(z_j, x) + lcs(z_j', x') - cj$ is monotonically decreasing by at least $c$ and satisfies $lcp(z_j, x) + lcs(z_j', x') - cj < \vec{\beta} + \vec{\gamma} - c < a_0 - 1$, and hence, no repetition appears. For $j'$ and $j''$, we can check whether these two occurrences become runs or not by using LCE constant times. □
Fig. 4. Illustration for Lemma 10. Four runs are found. Here $j' = 3$ and $j'' = 2$. The runs from $p_0$ and $p_1$ are encoded by a quintuplet. For each $j'$ and $j''$, the run is separately encoded by a quintuplet that shows a single run.

The other cases can be processed in a similar way.

A minor technicality is that we may redundantly find the same run in different cases. However, we can avoid duplicates by simply looking into the currently computed runs when we add new runs, spending $O(\log N)$ time. Also, we can remove repetitions whose root are not primitive by just choosing the smallest period among the repetitions with the same interval.
Appendix B: Proof of Lemma 7

Proof. The outline of our algorithm to compute FirstMismatch follows [15] which used a slower algorithm for Match. Assume $|X_i| - k + 1 \leq |X_j|$ holds.

If $X_j \rightarrow a$ with $a \in \Sigma$, then

$$\text{FirstMismatch}(X_i, X_j, k) = \begin{cases} 1 & \text{if } \text{Match}(X_i, X_j, k) = \text{true}, \\ 0 & \text{if } \text{Match}(X_i, X_j, k) = \text{false}. \end{cases}$$

If $X_j \rightarrow X_{r(j)} X_{l(j)}$, then we can recursively compute FirstMismatch($X_i, X_j, k$) as follows:

$$\text{FirstMismatch}(X_i, X_j, k) = \begin{cases} \text{FirstMismatch}(X_i, X_{r(j)}, k + |X_{l}|) & \text{if } \text{Match}(X_i, X_{l(j)}, k) = \text{true}, \\ \text{FirstMismatch}(X_i, X_{l(j)}, k) & \text{if } \text{Match}(X_i, X_{l(j)}, k) = \text{false}. \end{cases} \tag{1}$$

We apply Lemma [6] to $S$, pre-processing SLP $S$ in $O(n^2h)$ time and $O(n^2)$ space, so that query Match($X_i, X_{j'}, k'$) is answered in $O(\log N)$ time for any variable $X_{j'}$ and integer $k'$. Note that in either case of Equation (1) the height of the second variable decreases by 1. Hence we can compute FirstMismatch($X_i, X_j, k$) in $O(h \log N)$ time, after the $O(n^2h)$-time $O(n^2)$-space pre-processing. $\square
Appendix C: Figures

Fig. 5. The derivation tree of SLP $S = \{X_1 \to a, X_2 \to b, X_3 \to X_2 X_2, X_4 \to X_1 X_2, X_5 \to X_1 X_3, X_6 \to X_4 X_3, X_7 \to X_3 X_6, X_8 \to X_7 X_7\}$, representing string $s = \text{abbabbbabbb}$.

Fig. 6. Lemma 5 Illustration for computing $\text{LCE}(X_i, k_1, k_2)$. The roots of the gray subtrees are labeled by the variables in $U$. We find the first variable $X_{u(p')}$ in the list $U$ with which the Match query returns false. We then perform the FirstMismatch query for $X_i$ and $X_{u(p')}$ using the appropriate offset.
Lemma 8: Illustration for approximate doubling. The prefix variables up to $Y_{a_5}$ have been created. The traversals for $v_2$, $v_3$, $v_4$ end due to Condition 1 and that for $v_5$ ends due to Condition 2. Each traversed edge (depicted in bold) contributes to at most one new variable for some segment. Next, we will resume the traversal from $v_5$ targeting position $2|Y_{a_5}|$, and iterate the procedure until we get the last variable $Y_{a_k}$. The total number of bold edges can be bounded by $O(n)$ thanks to Condition 2.