Compressed Consecutive Pattern Matching

Pawel Gawrychowski, Garance Gourdel, Tatiana Starikovskaya, Teresa Anna Steiner

Compressed Consecutive Pattern Matching

Paweł Gawrychowski[1], Garance Gourdel[2,3], Tatiana Starikovskaya[3], and Teresa Anna Steiner[4]

[1] University of Wrocław, Poland

[2] Université Rennes IRISA Inria Rennes, France

[3] DI/ENS, PSL Research University, France

[4] DTU Compute, Denmark

Abstract

Originating from the work of Navarro and Thankanchan [TCS 2016], the problem of consecutive pattern matching is a variant of the fundamental pattern matching problem, where one is given a text and a pair of patterns p_1, p_2, and must compute consecutive occurrences of p_1, p_2 in the text. Assuming that the text is given as a straight-line program of size g, we develop an algorithm that computes all consecutive occurrences of p_1, p_2 in optimal $O(g + |p_1| + |p_2| + \text{output})$ time. As a corollary, we also derive an algorithm that reports all co-occurrences separated by a distance $d \in [a, b]$ in $O(g + |p_1| + |p_2| + \text{output})$ time and an algorithm that reports the top-k closest co-occurrences in $O(g + |p_1| + |p_2| + k)$ time.

1 Introduction

In the classical pattern matching problem, one is given a pattern and a text, and must find all substrings of the text equal to the pattern. However, considering potential applications, it is advantageous to enable queries beyond simply identifying exact matches of a given pattern in the preprocessed text.

Recently, Navarro and Thankanchan [1] suggested a generalisation of the pattern matching problem, where in addition to the pattern and the text one is given two integers a, b, and must find all pairs of consecutive occurrences of the pattern in the text separated by a distance $d \in [a, b]$. They showed that there is a $O(n \log n)$-space index for this problem with optimal query time $O(m + \text{output})$, where n is the length of the text and m of the pattern.

Following their work, indexing for consecutive occurrences is receiving growing attention in the literature [2, 3, 4].

Bille et al. [3] considered an even more general case of the problem, where a query consists of two different patterns p_1, p_2 of total length m and two integers a, b, and one must find all pairs of consecutive occurrences of p_1, p_2 in the text separated by a distance $d \in [a, b]$. For reporting the occurrences, they showed a linear-space data structure with $\tilde{O}(m + n^{2/3}\text{output})$ query time. On the other hand, by reduction from the set intersection problem, they showed a lower bound suggesting that achieving query time better than $\tilde{O}(m + \sqrt{n})$ for indexes of size $\tilde{O}(n)$ is impossible, even if $a = 0$ is fixed.

Gawrychowski et al. [4] showed that one can circumvent this lower bound in the case $a = 0$ assuming that the text is very compressible: assuming that the text is

represented by a straight-line program (SLP) of size g, they showed a $\tilde{O}(g^5)$-space data structure with $\tilde{O}(m + \text{output})$ query time, where m is the total length of the patterns.

Unfortunately, the above upper bounds, despite their theoretical interest, are still far from providing an efficient solution. Motivated by this, we study the dual variant of the problem, where one must process the text and the patterns simultaneously. Note that if the text is uncompressed and has length n, the problem can be solved by classical online pattern matching in $O(n + m + \text{output})$ time by keeping the most recent occurrences of p_1 and p_2. In this work, we show that for very compressible texts which can be represented by an SLP of size $g \ll n$, we get a better runtime $O(g + m + \text{output})$ by extending the ideas of [5]:

Theorem 1. *Given a text of length n on a constant-size alphabet represented by an SLP G of size g and patterns p_1, p_2 of total length m, all consecutive occurrences of p_1, p_2 in the text can be found in $O(g + m + \text{output})$ time assuming the word-RAM model with a machine word of size $w = \Omega(\log n)$.*

Using similar techniques, we derive an algorithm to output all consecutive occurrences with a bounded distance between them:

Corollary 2. *Given a text of length n on a constant-size alphabet represented by an SLP G of size g and patterns p_1, p_2 of total length m, all consecutive occurrences of p_1, p_2 in the text separated by a distance $d \in [a, b]$ can be found in $O(g + m + \text{output})$ time assuming the word-RAM model with a machine word of size $w = \Omega(\log n)$.*

Finally, our techniques allow to derive an efficient solution for the variant of the problem suggested by Bille et al. [2], where one must report the top-k consecutive occurrences of p with smallest distances between them.

Corollary 3. *Given an integer k, a text of length n on a constant-size alphabet represented by an SLP G of size g and patterns p_1, p_2 of total length m, the k closest consecutive occurrences of p_1, p_2 in the text can be found in $O(g+m+k)$ time assuming the word-RAM model with a machine word of size $w = \Omega(\log n)$.*

Technical Overview. Our algorithm builds on and extends the work by Ganardi and Gawrychowski [5], who gave the first $O(g + m)$-time algorithm for deciding whether a pattern p occurs in a string s represented by a context-free grammar G of size g. Their algorithm relies on the fact that for any occurrence of pattern p of length at least 2 in s, there exists a production $A \to BC$ in G, such that a suffix of the extension of B is a prefix of p and and a prefix of the extension of C is a suffix of p. They show how to efficiently compute information about prefixes (resp. suffixes) of non-terminals which contain the *longest possible* suffix (resp. prefix) of p. This information is called *boundary information* and is constructed for G bottom-up, starting at the terminals. To do so efficiently, the authors give a new data structure for efficiently answering batches of substring concatenation queries. Our algorithm extends this approach to reporting consecutive occurrences of patterns p_1 and p_2.

Similarly to above, for any consecutive occurrence of p_1 and p_2, there exists some production $A \to BC$, such that the *first position* of the occurrence of p_1 is in the extension of B, and the *last position* of the occurrence of p_2 is in the extension of C. However, there are different cases to consider, depending on whether the occurrences of p_1 resp. p_2 are fully in the extension of B resp. C, or whether one or both of them start in the extension of B and end in the extension of C (called a *crossing occurrence*). To handle the different cases, we need to compute additional information for every non-terminal, including the left- and rightmost occurrence of both patterns in the extension of any non-terminal, all crossing occurrences of both patterns, and a two-level version of the boundary information. The two-level boundary information is necessary for the case where p_2 is a crossing occurrence, and we need to find its "predecessor occurrence" of p_1, i.e. the rightmost occurrence of p_1 starting before p_2. We show how to compute this information efficiently for all non-terminals via a bottom-up approach. For computing all crossing occurrences, we extend the work by Ganardi and Gawrychowski [5], who only showed how to answer whether or not a crossing occurrence of a pattern exists in any non-terminal. We can find all crossing occurrences in the same time complexity, even if there are many in the same non-terminal, by using periodicity arguments. Then we show how to use this information to report all co-occurrences, which requires a careful case analysis and further periodicity arguments.

2 Preliminaries

A *string* s of length $|s| = n$ is a sequence $s[0]s[1]\dots s[n-1]$ of characters from a finite alphabet Σ. In this work, we assume that the size of the alphabet is constant. A *substring* of a string s is a pair (i, j) where $0 \le i \le j < |s|$ and is identified with the string $s[i \dots j] = s[i] \dots s[j]$. We also use the notation $s[i \dots j)$ and $s(i \dots j]$ which stand for the substring $s[i \dots j-1]$ and $s[i-1 \dots j]$ respectively. We say that a substring $s[i \dots j]$ is fully contained in another substring $s[i' \dots j']$ if $i' \le i \le j \le j'$. We call a substring $s[0 \dots i]$ *a prefix* of s and use a simplified notation $s[\dots i]$, and a substring $s[i \dots n-1]$ *a suffix* of s denoted by $s[i \dots]$. We say that x occurs in s at position i if $x = s[i \dots i + |x|)$, alternatively we say i is an occurrence of x in s. Additionally, an occurrence i is fully included in a substring f of s if $s[i \dots i + |x|)$ is fully included in f.

An occurrence k_1 of p_1 together with an occurrence k_2 of p_2 form a *consecutive occurrence (co-occurrence)* (k_1, k_2) of the strings p_1, p_2 in a string s if there are no occurrences of p_1 in $(k_1, k_2]$ and no occurrences of p_2 in $[k_1, k_2)$. The distance $k_2 - k_1$ is sometimes referred to as a *gap*.

An integer π is a *period* of a string s of length n if $s[i] = s[i + \pi]$ for all $i = 0, \dots, n - 1 - \pi$. We say that s is *periodic* if its smallest period, referred to as *the period* of s, is at most $|s|/2$. We also exploit the well-known corollary of the Fine and Wilf's periodicity lemma [6]:

Corollary 4. *Let x, y be strings such that $|x| \le 2|y|$. If there are at least three occurrences of y in x, then all occurrences of y in x form an arithmetic progression*

with difference equal to the period of y.

Proposition 5. *Let p be an m-length string over a constant-size alphabet. Then, one can preprocess p in $O(m)$ time to maintain the following queries in constant time: Given a substring $p[i...j]$, find its leftmost and rightmost occurrences in p, as well as the total number of occurrences. Given two substrings $p[i...j]$ and $p[k...l]$, compute their longest common prefix and their longest common suffix.*

Proof. We first construct the suffix tree of p in $O(m)$ time [7]. Belazzougui et al. [8] showed that the suffix tree can be preprocessed in linear time so that, given a substring $p[i...j]$, one can find the node u of the suffix tree labeled by $p[i...j]$ in constant time. The leaves of the subtree of u correspond to the occurrences of $p[i...j]$ in p. For each node, we can precompute in linear time in the size of its subtree, the leftmost and rightmost occurrences of its label. This is done by simply traversing the tree from the bottom to the top and propagating the information. We also preprocess the suffix trees in linear time so that it is possible to find the lowest common ancestor between two nodes in constant time [9]. Given two nodes u and v labelled by substrings $p[i...j]$ and $p[k...l]$, the length of the label of their common ancestor is the longest common prefix of $p[i...j]$ and $p[k...l]$. Analogously, by building the suffix tree for the reverse of p, we can compute the longest common suffix of two substrings in constant time. □

Corollary 6 (of Corollary 4 and Proposition 5)**.** *One can preprocess an m-length string p over a constant-size alphabet in $O(m)$ time to maintain the following queries in constant time: Given a substring (i, j), such that $j - i \geq m/2$, one can output the arithmetic progression of the occurrences of $p[i...j]$ in p in constant time.*

2.1 Grammars

Definition 1 (Straight-line program [10])**.** *A* straight-line program *(SLP) is a context-free grammar (CFG) consisting of a set of non-terminals $X_1, \ldots, X_q$, a set of terminals, an initial symbol X_q, and a set of productions, satisfying the following properties:*

- *A production consists of a left-hand side and a right-hand side, where the left-hand side is a non-terminal X_i and the right-hand side is a sequence X_jX_k, where $j, k < i$, or a terminal;*
- *Every non-terminal is on the left-hand side of exactly one production.*

The *expansion* $\overline{S}$ of a sequence of terminals and non-terminals S is the string that is obtained by iteratively replacing non-terminals by the right-hand sides in the respective productions, until only terminals remain. We say that G *represents* the expansion $\overline{G}$ of its initial symbol.

Definition 2 (Parse tree)**.** *The* parse tree *of a SLP is defined as follows:*

- *The root is labeled by the initial symbol;*
- *Each internal node is labeled by a non-terminal;*

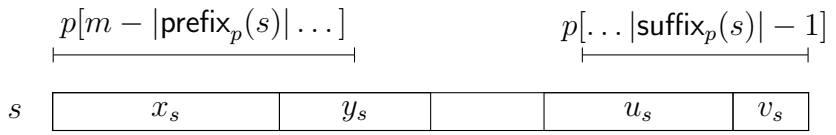

Figure 1: A p-boundary information for a string s that is not a substring of p.

- *If S is the expansion of the initial symbol, then the i^{th} leaf of the parse tree is labeled by a terminal $S[i]$;*
- *A node labeled with a non-terminal A that is associated with a production $A \to BC$ has two children labeled by B and C, respectively.*

The *size* of a grammar is the total size of all right-hand sides of all productions. The *height* of a grammar is the height of the parse tree.

3 Boundary information

Fix a pattern p of length m. In this section, we remind the notion of the p-boundary information for a string s, introduced by Ganardi and Gawrychowski [5]. It constitutes information about the longest prefix resp. suffix of s containing a suffix resp. prefix of a pattern p. This information, if computed for the extension of a non-terminal A in G, can then be used to decide whether there is a crossing occurrence of p in A. The information is defined in such a way that given boundary information for strings s and t, the boundary information for st can be constructed in constant time, given additional data structures (see Algorithm 1). This allows efficiently computing boundary information for all non-terminals in G.

For a string s, let $\mathsf{prefix}_p(s)$ be the longest prefix of s which is a suffix of p and $\mathsf{suffix}_p(s)$ the longest suffix of s which is a prefix of p. If s occurs in p at position i, meaning $s = p[i...i+|p|-1]$, then we define *p-substring information* for s as $(i, i+|p|-1)$, and otherwise p-substring information for s is undefined (if p occurs multiple times in s, any occurrence defines a valid *p-substring information* for s). For a string s, a *p-boundary information* is defined as follows:

1. If s occurs in p, then it is simply p-substring information for s;
2. Otherwise, it is two substrings of p, x_s and y_s such that $\mathsf{prefix}_p(s)$ is a prefix of $x_s y_s$ which in turn is a prefix of s (*p-prefix information*), and two substrings of p, u_s, v_s such that $\mathsf{suffix}_p(s)$ is a suffix of $u_s v_s$ which in turn is a suffix of s (*p-suffix information*). (See Fig. 1.)

For a string s, there are multiple ways *p-boundary information* can be constructed. We apply a recursive approach: for two strings s, t, assume to be given p-boundary information for s, t. Algorithm 1 (first described in [5]) correctly constructs a p-boundary information for $c = st$.

Algorithm 1 A boundary information of $c = st$

1. If s is a substring of p and t is not, then the p-suffix information of c is the p-suffix information of t and we define the p-prefix information for c as follows:
   (a) If sx_t is a substring of p, then $x_c = sx_t$ and $y_c = y_t$;
   (b) Otherwise, $x_c = s$ and $y_c = x_t$.
2. If t is a substring of p and s is not, then the p-prefix information of c is the p-prefix information of s and we define the suffix information for c as follows:
   (a) If $v_s t$ is a substring of p then $v_c = v_s t$ and $u_c = u_s$;
   (b) Otherwise, $u_c = v_s$ and $v_c = t$;
3. If s and t are both substrings of p, and c is a substring $p[i \dots j]$ of p, then the p-boundary information is p-substring information for c, and we define it equal to (i, j). Otherwise, we put $x_c = u_c = s$ and $y_c = v_c = t$.
4. If neither s nor t is a substring of p, then one can take p-prefix information for c equal to p-prefix information for s and p-suffix information to p-suffix information for t.

Definition 3 (Crossing occurrences)**.** *Let s, t be two strings. We say that a position i is a* crossing occurrence *of p in a string $c = st$ if i is an occurrence of p in c and $i \leq |s| \leq i+|p|-1$. By extension, an occurrence i of p in the expansion $\overline{A}$ is a* crossing occurrence *of p for a non-terminal A of a grammar G, or simply a crossing occurrence for A, if A is associated with a production $A \to BC$ and $i \leq |B| \leq i + |p| - 1$.*

Ganardi and Gawrychowski [5] showed that given a p-boundary information of two strings s, t one can efficiently decide whether there is a crossing occurrence of p in $c = st$. By extending their solution, we can report all such occurrences. Note that by Corollary 4, all crossing occurrences form an arithmetic progression.

Lemma 7. *Assume to be given a p-boundary information of strings s_k, t_k for $k \in [1, q]$. One can compute all crossing occurrences of p in all strings $s_k t_k$, for $k \in [1, q]$, in $O(q+m)$ time. For each k, the output is represented as an arithmetic progression.*

The proof of this lemma and our algorithm exploit the following fact:

Fact 8 (see [5, Lemma 2.2, Theorem 1.3])**.** *Let w be the size of the machine word. A string p of length m can be preprocessed in $O(m)$ time so that:*

- q substring concatenation queries *on p can be answered in $O(q + m/w)$ time. A substring concatenation query on a string p asks: Given two substrings $u = p[i \dots j]$ and $v = p[k \dots \ell]$ of p, check whether uv occurs in p and, if so, return an occurrence;*

- *Given q substrings $u_1, \ldots, u_q$ of p one can compute $\mathsf{prefix}_p(u_i)$ and $\mathsf{suffix}_p(u_i)$ in $O(q + m/w)$ time.*

4 Compressed consecutive pattern matching

We are now ready to prove Theorem 1. Recall that the text is an n-length string over a constant size alphabet represented by an SLP G of size $g \ll n$, and the patterns p_1, p_2 have total length m. By scanning the patterns in $O(m)$ time we can decide whether they contain characters outside of the alphabet of the text. If this is the case, there are no co-occurrences of p_1, p_2 in the text. Below we assume that p_1, p_2 are strings over the constant-size alphabet of the text.

The main strategy is to first compute all *primary co-occurrences* of p_1 and p_2:

Definition 4 (Primary co-occurrence)**.** *Let A be a non-terminal of G associated with a production $A \to BC$. We say that a co-occurrence (i, j) of p_1, p_2 in $\overline{A}$ is* primary *if $i \leq |\overline{B}| \leq j + |p_2| - 1$.*

We handle primary co-occurrences depending on whether the occurrence of p_1 is fully contained in $\overline{B}$ or a crossing occurrence, and whether the occurrence of p_2 is fully contained in $\overline{C}$ or a crossing occurrence. To do so, we preprocess the non-terminals of G to compute the boundary and secondary boundary information for p_1 and p_2 (see Section 4.1), all crossing occurrences, the leftmost and rightmost occurrences of p_1 and p_2 in any non-terminal. The secondary boundary information will be used in the case where there is a crossing occurrence of p_2 and we need to find the rightmost occurrence of p_1 starting before that occurrence.

4.1 Computing boundary information and crossing occurrences

We first use a linear-time pattern matching algorithm (e.g. the Knuth-Morris-Pratt algorithm [11]) to check whether p_2 is a substring of p_1. If it is, then every co-occurrence of p_1, p_2 in the text is a pair $(i, i + \ell)$, where i is an occurrence of p_1 in the text and ℓ is the leftmost occurrence of p_2 in p_1. (By definition, there is no occurrence of p_2 in $[i, i + \ell)$. Note also that there cannot exist an occurrence $i < i' \leq i + \ell$ of p_1, because then $i + \ell - i' < \ell$ would be an occurrence of p_2 in p_1, a contradiction with the choice of ℓ.) In other words, to find all co-occurrences of p_1, p_2 in the text it suffices to find all occurrences of p_1 in the text, which can be done in $O(g + m + \text{output})$ time [5].

Below we assume that p_2 is not a substring of p_1. Define an array P such that for every $j \in [0, |p_2| - 1]$, $P[j]$ is the rightmost occurrence of p_1 in p_2 to the left of j if their is one, else -1. We call P the *predecessor array*. P can be computed in $O(m)$ time by a linear-time pattern matching algorithm.

By [12], we can restructure the SLP G representing the text in $O(g)$ time to ensure that its height is $O(\log n)$, while its size increases by only a constant factor.

For every symbol A of G (a non-terminal or a terminal) associated with a production $A \to BC$, we compute:

1. a p_1-boundary information and a p_2-boundary information for $\overline{A}$;
2. All crossing occurrences of p_1 and p_2 for A, as well as the rightmost and the leftmost occurrences of p_1 and p_2 in $\overline{A}$;
3. Furthermore, if p_2-suffix information for $\overline{A}$ is (u_A, v_A), then we compute:
   (a) a p_1-boundary information of u_A and v_A, which we refer to as *secondary boundary information*;
   (b) All crossing occurrences of p_1 for u_A, v_A, as well as the rightmost occurrence of p_1 in $\overline{A}$ starting before u_A.

Proposition 9. *There is a $O(g)$-time algorithm that computes boundary and secondary boundary information for all symbols of G.*

Proof. We first compute boundary information. For a terminal, it suffices to check if it occurs in p. If it does and i is one of the occurrences, we define the p-substring information to be (i, i), else we define p-prefix information and the p-suffix information to be the empty strings. Let the k-th level L_k of G be the set of its symbols whose parse tree has height k. We apply Algorithm 1 to compute boundary information for the symbols of each level in turn, starting from level 0. Processing L_k takes $|L_k|$ substring concatenation queries and $O(|L_k|)$ extra time. Since the height of G is $O(\log n)$, by Fact 8 we obtain $O(\sum_k(|L_k| + m/w)) = O(g + m)$ total time.

The secondary boundary information is computed by applying Algorithm 1 on the substrings constituting the boundary information. In more detail, note that for any non-terminal A of G associated with a production $A \to BC$ Algorithm 1 computes the boundary information of $\overline{A}$ by either copying the boundary information of $\overline{B}$ or $\overline{C}$ or by concatenating the substrings constituting the boundary information of $\overline{B}$ and $\overline{C}$ a constant number of times. It follows that in total there are $O(|L_k|)$ copying and concatenation operations at level k. For each concatenation operation, we apply Algorithm 1 to update the boundary information. Processing L_k hence takes $O(|L_k|)$ substring concatenation queries and $O(|L_k|)$ extra time. As above, this leads to $O(\sum_k(|L_k| + m/w)) = O(g + m)$ total time. □

We now apply Lemma 7 to compute all crossing occurrences for the non-terminals of G in $O(g + m)$ time. By a second application of Lemma 7, we compute, for every non-terminal A of G, all crossing occurrences for pairs u_A, v_A, which constitute p_2-suffix information for $\overline{A}$, using the same amount of time.

Proposition 10. *Given the boundary information and the crossing occurrences for all symbols of G, one can compute the rightmost and leftmost occurrences of p_1 and p_2 in the expansion of every symbol of G in $O(g)$ time.*

Proof. We explain how to compute the rightmost occurrences of p_1, the rest can be computed analogously. The algorithm processes the symbols of G bottom-up. Consider a symbol A of G. If it is a terminal, then we can find the rightmost occurrence

of p_1 in its expansion (if it exists) in $O(1)$ time. Otherwise, assume that A is associated with a production $A \to BC$. If $\overline{C}$ contains an occurrence of p_1, then the rightmost occurrence of p_1 in $\overline{A}$ is the rightmost occurrence of p_1 in $\overline{C}$ (and we have already computed it). Otherwise, if there are crossing occurrences of p_1 for $\overline{B}, \overline{C}$, it is the rightmost such occurrence. And finally, if $\overline{C}$ does not contain an occurrence of p_1 and there are no crossing occurrences, then we copy the rightmost occurrence of p_1 in $\overline{B}$. □

We will need the following auxiliary claim:

Observation 11. *Given an arithmetic progression by its starting position, difference, and length, we can find the predecessor of a number z in that progression in constant time.*

Proposition 12. *There is a $O(g+m)$-time algorithm that computes, for each symbol A of G, the rightmost occurrence of p_1 in $\overline{A}$ before u_A, where (u_A, v_A) is p_2-suffix information for $\overline{A}$.*

Proof. The algorithm processes the symbols of G bottom-up. Consider a symbol A of G. If it is a terminal, then (u_A, v_A) is either not defined if A occurs in p_2, or $u_A = v_A$ are both equal to the empty string. In the second case, the rightmost occurrence of p_1 in $\overline{A}$ before u_A is 0, if $A = p_1$, otherwise no occurrence of p_1 exists in $\overline{A}$. Now, if there is a production $A \to BC$, let (u_B, v_B) (resp., (u_C, v_C)) be the p_2-suffix information for $\overline{B}$ (resp., $\overline{C}$) that we computed using Algorithm 1. Note that they might not be defined, if $\overline{B}$ or $\overline{C}$ is a substring of p_2. We review the cases of Algorithm 1:

In Cases 1 and 4, the p_2-suffix information of $\overline{A}$ is the p_2-suffix information of $\overline{C}$. Therefore, the rightmost occurrence of p_1 before u_A in $\overline{A}$ is either the last occurrence before $u_C = u_A$ in $\overline{C}$, or the rightmost crossing occurrence of p_1 for A, or the rightmost occurrence of p_1 in $\overline{B}$, and we can compute it in constant time.

In Case 3, either (u_A, v_A) is undefined or $u_A = \overline{B}$, and therefore the rightmost occurrence of p_1 is undefined.

In Case 2a, $u_A = u_B$, and therefore the rightmost occurrence of p_1 in $\overline{A}$ before u_A is either the rightmost occurrence of p_1 in $\overline{B}$ before u_B or the rightmost crossing occurrence of p_1 for A that precedes u_B, which can be found in constant time by Observation 11.

In Case 2b, we have $u_A = v_B$. Thus, the rightmost occurrence before u_A equals to one of the following:

1. The rightmost crossing occurrence of p_1 for u_B and v_B;
2. The rightmost occurrence of p_1 fully contained in u_B;
3. The rightmost occurrence of p_1 preceding u_B which is fully in $\overline{B}$;
4. The rightmost crossing occurrence of p_1 for A preceding u_B.

We can find the rightmost occurrence fully contained in $u_B = p_2[i \dots j]$ by querying the predecessor array P for $j - |p_1| + 1$, and the three other candidate occurrences have been already computed. □

4.2 Reporting co-occurrences

We now show how to quickly report the co-occurrences.

Proposition 13. *Consider a symbol A of G associated with a production $A \to BC$. Let $j < |\overline{B}| \leq j + p_2 - 1$ be an occurrence of p_2 in $\overline{A}$. One can find the rightmost occurrence $i \leq j$ of p_1 such that $i + |p_1| - 1 < |\overline{B}|$ in $O(1)$ time.*

Proof. Let (u_B, v_B) be p_2-suffix information for $\overline{B}$. By definition, j belongs to $u_B v_B$. If j belongs to v_B, then i is the rightmost existing one of the following candidates:

1. The rightmost occurrence $i' \leq j$ of p_1 such that $\overline{A}[i'...i' + |p_1|)$ is fully contained in v_B (which we can find in $O(1)$ time using the predecessor array P);
2. The rightmost crossing occurrence $i' \leq j$ of p_1 for u_B, v_B (which we can find in $O(1)$ time by Observation 11);
3. The rightmost occurrence of p_1 that is fully in u_B (which we can find in $O(1)$ time using the predecessor array P);
4. The rightmost occurrence of p_1 in $\overline{B}$ starting before u_B (which we have precomputed).

If j is in u_B, then i is the rightmost existing one of the following candidates:

- The rightmost occurrence $i' \leq j$ of p_1 such that $\overline{A}[i'...i' + |p_1|)$ is fully contained in u_B (which we can find in $O(1)$ time using the predecessor array P);
- The rightmost crossing occurrence $i' \leq j$ of p_1 for u_B, v_B (which we can find in $O(1)$ time by Observation 11);
- The rightmost occurrence of p_1 in $\overline{B}$ starting before u_B (which we have precomputed).

It follows that i can be computed in constant time. □

For a node u of the parse tree of G, denote by $\mathrm{off}(u)$ the number of leaves to the left of the subtree rooted at u.

Observation 14. *Assume that p_2 is not a substring of p_1, and let (i, j) be a co-occurrence of p_1, p_2 in the text. In the parse tree of G, there exists a unique node u such its label A is associated with a production $A \to BC$, and $(i - \mathrm{off}(u), j - \mathrm{off}(u))$ is a primary co-occurrence of p_1, p_2 in $\overline{A}$.*

Lemma 15. *Assume that p_2 is not a substring of p_1 and that we are given the information computed in Section 4.1. There is a $O(g + m)$-time algorithm that reports all primary co-occurrences of p_1 and p_2 in the expansions of the non-terminals of G. If there is more than one primary co-occurrence in the expansion of a non-terminal, they are output as $O(1)$ arithmetic progressions.*

Proof. Let A be a non-terminal associated with a production $A \to BC$. We consider three types of co-occurrences of p_1, p_2 in $\overline{A}$:

1. The occurrence of p_1 is fully contained in $\overline{B}$ and the occurrence of p_2 in $\overline{C}$;
2. The occurrence of p_1 is a crossing occurrence for $\overline{B}, \overline{C}$ and the occurrence of p_2 is not;

3. The occurrence of p_2 is a crossing occurrence for $\overline{B}, \overline{C}$.

Let (i, j) be a co-occurrence of Type 1. It must have the property that i is the rightmost occurrence of p_1 fully contained in $\overline{A}[\dots|\overline{B}|-1]$, j is the leftmost occurrence of p_2 in $\overline{A}[|\overline{B}|\dots]$, and there are no occurrences of p_1, p_2 in between. As we store all crossing occurrences for B, C, the rightmost occurrences of p_1, p_2 in $\overline{B}$ and the leftmost occurrences of p_1, p_2 in $\overline{C}$, we can check whether (i, j) exists and compute it in $O(1)$ time by Observation 11.

A co-occurrence (i, j) of of Type 2 must satisfy the following properties:

1. j cannot be in $\overline{A}[\dots|\overline{B}| - 1]$, since it is not a crossing occurrence and p_2 is not a substring of p_1;
2. Since i and j are consecutive, and i is a crossing occurrence, i must be the rightmost crossing occurrence and j must the leftmost occurrence of p_2 in $\overline{A}[|\overline{B}|\dots] = \overline{C}$.

We retrieve i, the rightmost crossing occurrence of p_1, and j, the leftmost occurrence of p_2 in $\overline{A}[|\overline{B}|\dots]$. It remains to check that there is no occurrence of p_1 in $(i, j]$ and no occurrence of p_2 in $[i, j)$. If there is an occurrence of p_1 in $(i, j]$, it can only be the leftmost occurrence of p_1 in $\overline{A}[|\overline{B}|\dots] = \overline{C}$. If there is an occurrence of p_2 in $[i, j)$, it can only be a crossing occurrence for A. Both conditions can be tested in constant time by Observation 11.

For Type 3, consider two cases. First, consider the case when j is the leftmost crossing occurrence. Let $i' \leq j$ be the rightmost occurrence of p_1 such that $i' + |p_1| - 1 < |\overline{B}|$, which we can find in $O(1)$ time by Proposition 13, and i'' be the rightmost crossing occurrence of p_1 that is at most j, which we can find in $O(1)$ time as well using Observation 11. By definition, the only candidate for a co-occurrence containing j is $(i = \max\{i', i''\}, j)$. By construction of i' and i'', there can't be any occurrence of p_1 in $(i, j]$ and it suffices to check whether there are occurrences of p_2 in $[i, j)$. If there is such an occurrence, it must be the rightmost occurrence of p_2 in $\overline{B}$, and we can check if it is the case in constant time.

Consider now the case when j is not the leftmost crossing occurrence of p_2 for A. By Corollary 4, all crossing occurrences of p_2 form an arithmetic progression. Let j_0 be the leftmost occurrence, d be the difference and ℓ the length of this progression. By Corollary 4, $\overline{A}[j_0 \dots j_0 + \ell \cdot d + |p_2| - 1]$ is periodic with period d. Let $1 \leq k \leq \ell$ be the largest such that the occurrence $j^* = j_0 + k \cdot d$ forms a co-occurrence with an occurrence i of p_1. By definition of a co-occurrence, $j_0 \leq j^* - d < i \leq j^*$. Furthermore, since p_2 is not a substring of p_1, we have $i + |p_1| - 1 < j^* + |p_2| - 1$. Hence, by periodicity, $(i - k' \cdot d, j^* - k' \cdot d)$ is a co-occurrence for *all* $1 - k \leq k' \leq \ell - k$. (In particular, by maximality of j^*, we have $j^* = j_0 + \ell \cdot d$.) Hence, it suffices to find the co-occurrence for $k' = 1 - k$, i.e. to find the occurrence of p_1 preceding $j_0 + d$. This can be done in constant time by Proposition 13. This case is illustrated in Figure 2.

By Fact 8, the algorithm takes $O(g + m)$ time. □

Finally, we report all co-occurrences of p_1, p_2 in the text given the primary co-occurrences for the non-terminals of G. Using the approach of [13, Section 6.4]), it can be done in $O(g + \text{output})$ time. Observation 14 guarantees that we report all co-occurrences.

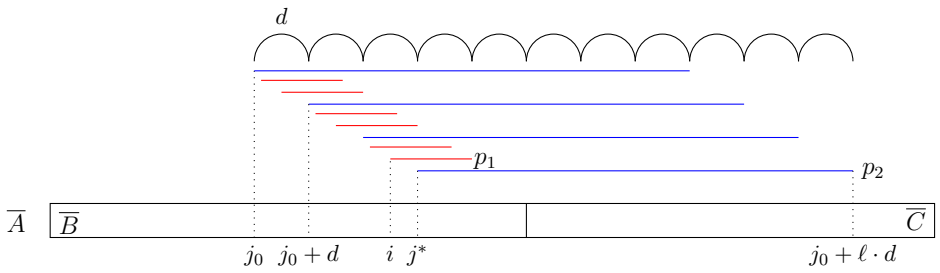

Figure 2: Co-occurrences with crossing occurrences of p_2 forming an arithmetic progression.

5 Gapped and top-k consecutive pattern matching

We now explain how to modify the algorithm to report only the co-occurrences with a bounded gap (Corollary 2) and only the top-k co-occurrences in the text (Corollary 3).

Bounded-gap co-occurrences. We run the algorithm of Section 4 in $O(g + m)$ time to generate a description of all primary co-occurrences (the elements of this description are single co-occurrences and arithmetic progressions of co-occurrences with a fixed gap) and select the elements of this description with a gap in $[a, b]$. For each selected element, we apply the approach of [13, Section 6.4]) to generate all co-occurrences with a gap in the interval $[a, b]$ in time $O(g + m + \text{output})$.

Top-k co-occurrences. To report the top-k co-occurrences, we first generate a description of all primary co-occurrences (the elements of this description are single co-occurrences and arithmetic progressions of co-occurrences with a fixed gap) in $O(g + m)$ time as in Section 4. Second, we arrange the elements of the description in a heap in $O(g)$ time sorted by the gaps. Then, conceptually, we attach to each node of the heap a path containing the secondary co-occurrences that originate from the element stored in the node. We finally select the k co-occurrences with smallest gaps in $O(k)$ time using Frederickson's heap selection algorithm [14]. The algorithm and its analysis requires the min-heap property, the fact that all nodes have constant degree, and to have quick access to the children of an already visited node. The first two properties are guaranteed by construction, and the method of [13, Section 6.4]) guarantees that the children of an already visited node can be accessed in constant amortised time.

References

[1] Gonzalo Navarro and Sharma V. Thankachan, "Reporting consecutive substring occurrences under bounded gap constraints," *Theor. Comput. Sci.*, vol. 638, pp. 108–111, 2016.

[2] Philip Bille, Inge Li Gørtz, Max Rishøj Pedersen, Eva Rotenberg, and Teresa Anna Steiner, "String indexing for top-k close consecutive occurrences," *Theor. Comput. Sci.*, vol. 927, pp. 133–147, 2022.

[3] Philip Bille, Inge Li Gørtz, Max Rishøj Pedersen, and Teresa Anna Steiner, "Gapped indexing for consecutive occurrences," *Algorithmica*, vol. 85, no. 4, pp. 879–901, 2023.

[4] Pawel Gawrychowski, Garance Gourdel, Tatiana Starikovskaya, and Teresa Anna Steiner, "Compressed indexing for consecutive occurrences," in *CPM*, 2023, vol. 259 of *LIPIcs*, pp. 12:1–12:22.

[5] Moses Ganardi and Pawel Gawrychowski, "Pattern matching on grammar-compressed strings in linear time," in *SODA*, 2022, pp. 2833–2846.

[6] Nathan J. Fine and Herbert S. Wilf, "Uniqueness theorems for periodic functions," *Proc. Am. Math. Soc.*, vol. 16, no. 1, pp. 109–114, 1965.

[7] Martin Farach-Colton, Paolo Ferragina, and S. Muthukrishnan, "On the sorting-complexity of suffix tree construction," *J. ACM*, vol. 47, no. 6, pp. 987–1011, 2000.

[8] Djamal Belazzougui, Dmitry Kosolobov, Simon J. Puglisi, and Rajeev Raman, "Weighted ancestors in suffix trees revisited," in *CPM*, 2021, vol. 191 of *LIPIcs*, pp. 8:1–8:15.

[9] Michael A. Bender and Martin Farach-Colton, "The LCA problem revisited," in *LATIN*. Springer, 2000, pp. 88–94.

[10] John C. Kieffer and En-Hui Yang, "Grammar-based codes: A new class of universal lossless source codes," *IEEE Trans. Inf. Theory*, vol. 46, no. 3, pp. 737–754, 2000.

[11] Donald E. Knuth, James H. Morris, Jr., and Vaughan R. Pratt, "Fast pattern matching in strings," *SIAM J. on Computing*, vol. 6, no. 2, pp. 323–350, 1977.

[12] Moses Ganardi, Artur Jez, and Markus Lohrey, "Balancing straight-line programs," in *FOCS*. 2019, pp. 1169–1183, IEEE Computer Society.

[13] Anders Roy Christiansen, Mikko Berggren Ettienne, Tomasz Kociumaka, Gonzalo Navarro, and Nicola Prezza, "Optimal-time dictionary-compressed indexes," *ACM Trans. Algorithms*, vol. 17, no. 1, pp. 8:1–8:39, 2021.

[14] Greg N. Frederickson, "An optimal algorithm for selection in a min-heap," *Information and Computation*, vol. 104, no. 2, pp. 197–214, 1993.

A Proof of Lemma 7

Consider two strings s, t. Let u, v be p-suffix information of s and x, y p-prefix information of t. To find all crossing occurrences of p in a string st, it suffices to look at occurrences in $uvxy$ as uv contains $\mathsf{suffix}_p(s)$ and xy contains $\mathsf{prefix}_p(t)$.

We can also assume that u is a prefix of p and y a suffix of p because there is an occurrence of p in $uvxy$ if and only if there is an occurrence of p in $\mathsf{suffix}_p(u)vx\mathsf{prefix}_p(y)$. Here and below, whenever we replace a string u with $\mathsf{suffix}_p(u)$ and or a string y with $\mathsf{prefix}_p(y)$, we assume to compute them using Fact 8.

By Corollary 4, the crossing occurrences of p form a single arithmetic progression. We will consider several cases and for each case will report an arithmetic progression of occurrences, but in the end they can be merged into a single one. We repeatedly make use of the following procedure:

Proposition 16. *Let ℓ be a prefix of p, r a suffix of p, and c a concatenation of at most three substrings of p. After $O(m)$-time shared preprocessing of p, one can report all occurrences of p in ℓc starting at positions $i \le |\ell|/2$ and all occurrences in cr ending at positions $j \ge |c| + |r|/2$ in constant time. The occurrences are output as an arithmetic progression.*

Proof. We show how to proceed for the occurrences in $s = \ell c$, the proof for the occurrences in cr is symmetric. Assume that there is an occurrence of p at position $i \leq |\ell|/2$. As ℓ is a prefix of p, $i = \alpha \cdot d$, where $0 \leq \alpha \leq |\ell|/2d$ is an integer and d is the period of ℓ.

After a classical shared $O(m)$-time preprocessing the period of any prefix of p can be extracted in $O(1)$ time [11]. If $d > |\ell|/2$, then the only candidate is $i = 0$ and we can test whether p occurs at this position using a constant number of longest common extension queries. We now assume $d \leq |\ell|/2$. Let $\alpha_{\max} \leq |\ell|/(2d)$ be the rightmost position such that $\alpha_{\max} \cdot d + m \leq |s|$. If there are none, then $|s| < m$ and there are no occurrences of p in s. Let $k \geq |\ell|$ be the rightmost position such that $p[0 \dots k-1]$ has period d; k can be computed by one longest common prefix and suffix query on p and $p[d \dots m-1]$. Furthermore, using $O(1)$ more longest common prefix and suffix queries, one can check if $p[0 \dots k-1]$ occurs at position $\alpha_{\max} \cdot d$ and if not, compute the first mismatching position.

Consider first the case where $p[0 \dots k-1]$ occurs at position $\alpha_{\max} \cdot d$. If $k = m$, then p occurs at every position $\alpha \cdot d$ with $0 \leq \alpha \leq \alpha_{\max}$. If $k < m$, then p cannot occur at a position αd with $\alpha \leq \alpha_{\max}$ by the maximality of k. It suffices to check if p occurs at position $\alpha_{max} \cdot d$ using $O(1)$ longest common prefix and suffix queries and report it accordingly. Now assume that $p[0 \dots k-1]$ does not occur at position $\alpha_{\max} \cdot d$ and let $p[0 \dots i-1]$ be the longest prefix starting at position $\alpha_{\max} \cdot d$ in s, meaning $p[i] \neq s[\alpha_{\max} \cdot d + i]$. By construction, d is a period of $s[0 \dots \alpha_{\max} \cdot d + i - 1]$. Consequently, no occurrence of $p[0 \dots k-1]$ in ℓc can cross position i, meaning there is no occurrence of p in ℓc starting at position $\alpha \cdot d$ with $\alpha \cdot d + k > \alpha_{\max} \cdot d + i$. Thus, occurrences can only be at positions $\alpha \cdot d \leq \alpha_{\max} \cdot d + i - k$. If $k = m$, by the d-periodicity of $s[0 \dots \alpha_{\max} d + i - 1]$, any such position is valid and we can report the occurrences as a single arithmetic progression. If $k < m$, the only possible candidate is the maximal $\alpha \cdot d$ such that $\alpha \cdot d + k < \alpha_{\max} + i$ (by maximality of k), and we can check whether there is an occurrence of p via $O(1)$ longest common prefix and suffix queries as above, and report it accordingly. □ □

We start by applying Proposition 16 on $\ell = u$ and $c = vxy$ to report all occurrences starting before $|u|/2$ and then apply it again on $\ell = \mathsf{suffix}_p(u[|u|/2 \dots])$ and $c = vxy$, which gives all occurrences of p in $uvxy$ starting before $3|u|/4$. Symmetrically, we can report all occurrences of p ending after $|uvx| + |y|/4$.

It remains to report the occurrences in $u'vxy'$, where $u' = \mathsf{suffix}_p(u[3|u|/4 \dots])$ and $y' = \mathsf{prefix}_p(y[\dots |y|/4])$. As $|u|, |y| \leq m$, we have $|u'|, |y'| \leq m/4$. For an occurrence i of p in $u'vxy'$, consider three (overlapping) cases: 1. The occurrence is fully contained in $u'vx$; 2. The occurrence fully contains vx; 3. The occurrence is fully contained in vxy'. Consider Case 1. By applying Proposition 16 on $c = u'v$ and $r = x$, we can assume that $|x| \leq m/2$. We then have three subcases: (a) either an occurrence of p is fully contained in $u'v$, or (b) it contains v, or (c) it is fully contained in vx. In Case 1(a), as $|u'| \leq m/4$ and $|v| \leq m$, any occurrence of p in $u'v$ ends in the second half of v and hence we can report all occurrences by applying Proposition 16 once to u' and $\mathsf{prefix}_p(v)$. Case 1(c) is analogous. We repeat the argument for Case 3. Thus, it remains to report all occurrences of p in a string $h = efg$ fully containing f, where

$|e|, |g| \leq m/4$, e is a prefix of p, g is a suffix of p, and f is a concatenation of at most two substrings of p.

If the length of f is smaller than $m/2$, then $|h| < m$ and there are no occurrences of p in h. Assume now that $|f| \geq m/2$.

Consider first the case when f is a substring of p given by its starting and ending positions, i.e. let $f = p[i \dots j]$. By Corollary 6, after $O(m)$-time shared preprocessing we can find the arithmetic progression of the occurrences of f in p in constant time. If there are only two occurrences, we test if they extend in e and g to an occurrence of p using two longest common prefix and suffix queries.

Assume now that there are at least three occurrences. Let p_{mid} be the minimal substring of p which contains all occurrences of f. By Corollary 4, the period of p_{mid} equals the period of f, d. Let $p = p_{\text{left}}p_{\text{mid}}p_{\text{right}}$. Next, using two longest prefix and suffix queries, we compute the maximal substring f' of h that starts and ends with an occurrence of f. Namely, by Corollary 4, it suffices to check how far the periodicity in f extends beyond f: f' must be periodic with period d, must fully contain f, and must start at a position $|e| - \alpha \cdot d$ and end at a position $|e| + |f| + \alpha \cdot \beta$ for some integers α, β. Let $h = e'f'g'$. By Corollary 4, the occurrences of f in h start at positions $|e'| + \alpha \cdot d$ for integer $0 \leq \alpha \leq (|f'| - |f|)/d$. Hence, if p_{left} is not empty, then the only possible position where p can occur in h is $|e'| - |p_{\text{left}}|$, and we can test whether it is the case using $O(1)$ longest common prefix and suffix queries. If p_{right} is not empty, then the only possible position where p can occur in h is $|e'| + |f'| - |p_{\text{left}}p_{\text{mid}}|$. Otherwise, if both p_{left} and p_{right} are empty, the arithmetic progression of occurrences of $p = p_{\text{mid}}$ in h is simply $|e'| + \alpha \cdot d$ for $0 \leq \alpha \leq (|f'| - |p_{\text{mid}}|)/d$.

Now consider the case when f is given as the concatenation of two substrings of p, i.e. $f = yu$. We would like to reduce this case to the first one. For that, we need to find an occurrence of f in p. As $|f| \geq m/2$, at least one of y, u has length greater or equal to $m/4$. Assume w.l.o.g. $|y| \geq m/4$, the other case is symmetric. By Corollary 6, after $O(m)$-time shared preprocessing we can find a constant number of arithmetic progressions representing the set of all occurrences of y in p in constant time, as well as the period d of y equal to the difference of the progressions. In more detail, if $|y| \geq m/2$, we apply Corollary 6 on p and y directly. If $m/2 \geq |y| \geq 3m/8$, we represent p as two $3m/4$-length blocks overlapping by $m/2$ characters and apply Corollary 6 to each block and y (then $|y|$ is at least half the length of a block, and every occurrence is definitely contained in one of the blocks). Finally, if $3m/8 \geq |y| \geq m/4$, we consider $m/2$-length blocks overlapping by $3m/8$ characters.

By one longest common extension query, we find the longest prefix of u periodic with period d, let its length be ℓ. Fix an arithmetic progression of occurrences of y and let q_{last} be the last position in it. Let q be the leftmost position in this progression such that $q + \ell \geq q_{last}$. Similarly to above, it can be shown that there is an occurrence of y in the progression followed by an occurrence of u iff $q + |y|$ is followed by an occurrence of u. We can decide whether this is the case in constant time by one longest common extension query and if it is report q as an occurrence of yu thus reducing to the first case.

□