String Matching in (Block) Graphs: A Full Classification by Walk Length

Sebastian Angrick
Karlsruhe Institute of Technology, Germany

Ben Bals
CWI, Amsterdam, The Netherlands
Vrije Universiteit Amsterdam, The Netherlands

Paweł Gawrychowski
University of Wrocław, Poland

Solon P. Pissis
The Cyprus Institute, Nicosia, Cyprus

Yuki Yonemoto
Kyushu University, Fukuoka, Japan

Abstract

We consider directed graphs in which the nodes are labeled with strings. A walk in such a graph naturally corresponds to the concatenation of the visited nodes' labels. These graphs are widely used in bioinformatics to compactly describe large collections of highly similar genomes. Given such a graph $G = (V, E)$ and a pattern of length m, we seek a walk whose corresponding string has an occurrence of the pattern. We call this the SMLG problem. Amir et al. [J. Algorithms, 2000] showed that SMLG can be solved in $\mathcal{O}(m\,|E| + N)$ time, where N is the total length of all node labels. Equi et al. [ACM Trans. Algorithms, 2023] showed that this is essentially optimal (under SETH).

The existing lower bound assumes that the sought walk is of length $\Theta(|V|)$. Thus, we might be able to bypass this lower bound by restricting the walk length to $b - 1$, which naturally reduces to having as input a directed graph whose set of nodes is partitioned into b *blocks*. Then, we seek a walk in this graph that starts in the first block and ends in the last block. We call this the b-SMBG problem. Equi et al. [Algorithmica, 2023] showed that, if we impose no restriction on b, the existing algorithm of Amir et al. is essentially optimal for b-SMBG (again under SETH). We provide a more fine-grained classification that essentially settles the complexity of b-SMBG parameterized by b:

1. For $b = 2$, Pissis [SOSA 2025] already provided a simple $\mathcal{O}(m + |E| + N)$-time algorithm.
2. We design a new $\widetilde{\mathcal{O}}(m + |E| + N)$-time algorithm for $b = 3$. As a direct implication of this result, the SMLG problem for $b \leq 3$ (walks of length at most 2) also admits near-linear-time complexity.
3. There is no $\mathcal{O}((m\,|E|)^{1-\varepsilon} + N)$-time *combinatorial* algorithm, for any $b \geq 4$ and $\varepsilon > 0$.
4. There is an algorithm working in $\mathcal{O}\left(\max(|V|, m)^{\omega} + N\right)$ time, where ω is the matrix multiplication exponent, which is conditionally optimal for graphs with $b \geq 4$ blocks.
5. Under SETH, no $\mathcal{O}((m\,|E|)^{1-\varepsilon} + N)$-time algorithm exists, for any $b = \omega(\log |V|)$ and $\varepsilon > 0$.

Although our motivation is primarily of a theoretical nature, we stress that our algorithms are simple to implement. As such, they may contribute to practical advancements in applications where the SMLG problem is an important primitive, such as in the analysis of pangenome graphs.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases string algorithms, pattern matching, lower bounds, fine-grained complexity

Digital Object Identifier 10.4230/LIPIcs.ESA.2026.105

Related Version *Full Version*: https://arxiv.org/abs/2607.28159

Funding *Paweł Gawrychowski*: Partially supported by the Polish National Science Centre grant number 2023/51/B/ST6/01505.

1 Introduction

String matching is the classical problem of finding the occurrences of a *pattern* $P = P[1\,..\,m]$ of length m in a *text* $T = T[1\,..\,n]$ of length n. The problem is by now well-understood: there exist $\mathcal{O}(n)$-time algorithms to solve the problem (e.g., [16, 28]), even when using only $\mathcal{O}(1)$ extra space (e.g., [13, 17]). Since much of today's data is interconnected, it is natural to study string matching not only in linear texts but also in *labeled graphs*. Indeed, large-scale labeled graphs are now prevalent across diverse areas, including graph databases [5], graph mining [14], and, more recently, bioinformatics [8]. While most applications require sophisticated operations on these graphs, they often rely on primitives that locate graph walks whose node labels match a given pattern [7, 34, 35].[1] See Figure 1 for an example.

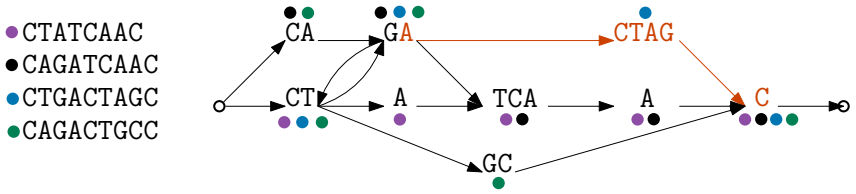

Figure 1 The DNA sequences on the left are represented by the node-labeled graph on the right. The pattern $P = P[1\,..\,6] =$ `ACTAGC` is matched in a walk of length 2 (in orange). This type of graph is known as *variation graph* [8], and it is a widely used representation for analyzing pangenomes.

1.1 String Matching in Graphs and the State of the Art

Let $\Sigma = \{1, 2, \ldots, \sigma\} = [1, \sigma]$ be an *alphabet* of σ elements that we call *characters*. A *node-labeled graph* $G = (V, E, \lambda)$ is a directed graph (V, E) equipped with a *labeling function* $\lambda : V \to \Sigma^*$ that defines which string over Σ is assigned to each node as a *label*. We set $N_G \coloneqq \sum_{v \in V} |\lambda(v)|$, the total length of the node labels in G; we may drop the subscript G from N_G when the context is clear. Given a pattern $P = P[1\,..\,m]$ of length m over Σ, we say that P has a *match* in G if there is a walk $v_1, \ldots, v_h$ in G such that $P = \alpha \cdot \lambda(v_2) \cdots \lambda(v_{h-1}) \cdot \beta$, α is a suffix of $\lambda(v_1)$ and β is a prefix of $\lambda(v_h)$; we also say that P *occurs* in G, and that $v_1, \ldots, v_h$ is an *occurrence* of P in G.

We next define the problem of string matching in node-labeled graphs (cf. [4, 18]).

String Matching in Labeled Graphs (SMLG)
Input: A node-labeled graph $G = (V, E, \lambda)$ and a pattern $P = P[1\,..\,m]$.
Output: *True* if and only if there is at least one occurrence of P in G.

The following upper bound is known due to Amir, Lewenstein, and Lewenstein [4].

▶ **Theorem 1** ([4]). *The SMLG problem can be solved in $\mathcal{O}(m\,|E| + N)$ time.*

The following lower bound, conditioned on the Strong Exponential Time Hypothesis (SETH) [27], is known due to Equi, Mäkinen, Tomescu, and Grossi [18].

▶ **Theorem 2** ([18]). *Under SETH, the SMLG problem cannot be solved in $\mathcal{O}((m\,|E|)^{1-\varepsilon} + N)$ time, for any constant $\varepsilon > 0$, even if every node label is a single character of a binary alphabet and G is a DAG, where the sum of the out- and in-degree of any node is at most 3.*

[1] Indeed, string matching in node-labeled graphs is a central topic of large bioinformatics consortia such as ALPACA (`https://alpaca-itn.eu/`) or PANGAIA (`https://www.pangenome.eu/`).

1.2 Our Motivation and Parameterization

Given Theorem 1 and Theorem 2, we were thus motivated to ask whether SMLG instances restricted to walks of *bounded length* can be solved significantly faster than $\mathcal{O}(m\,|E| + N)$ time. More concretely, what if we are looking for occurrences of P in G of the form $v_1, \ldots, v_h$ for a fixed $h \in \mathbb{N}$? Namely, we parameterize the SMLG problem by walk length $(h-1)$. Pissis showed that this problem can be solved in $\mathcal{O}(m + |E| + N)$ time for $h = 2$ [33] (for an earlier, slower solution for $h = 2$, see [37, Lemma 8]). Let us now formalize the problem.

h-String Matching in Labeled Graphs (h-SMLG)
Input: A node-labeled graph $G = (V, E, \lambda)$ and a pattern $P = P[1\,..\,m]$.
Output: *True* if and only if there is at least one occurrence of P in G of the form $v_1, \ldots, v_h$.

Furthermore, real-world instances of this problem (e.g., the ones arising from pangenome graphs [8]) often include large node labels. For instance, a recent survey [30] shows that the average node label length is in the order of 10^3 and for some graphs it is close to 10^6. Intuitively, this is because these graphs are constructed over a collection of *highly similar* genomes. Thus, when the pattern P is relatively short, it is reasonable to expect that an occurrence of P may span only a few nodes in G (i.e., that the sought walk is somewhat short). Note that patterns may be short in practice for two reasons: (i) they are either short DNA sequencing reads; or (ii) they are short DNA fragments used as anchors for alignments.

For algorithmic and application domain reasons, it is common to consider the more structured *block graphs*. Let us give a formal definition from [32]; see Figure 2 for an example. The definitions and notation introduced for SMLG carry over to block graphs.

▶ **Definition 3.** *A* block graph $G = (V, E, \lambda)$ *is a node-labeled graph satisfying:*
1. *The set V can be partitioned into a sequence $V_1, V_2, \ldots, V_b$ of* blocks*;*
2. *If $(u, v) \in E$ then $u \in V_i$ and $v \in V_{i+1}$, for some $i \in [1, b)$.*

When in addition to properties (1) and (2), we have the property that, if for all $u, v \in V_i$, $i \in [b]$, we have $\lambda(u) \neq \lambda(v)$ for $u \neq v$, we call G a block graph with unique labels.

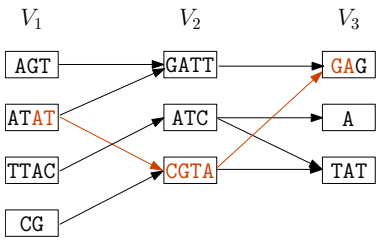

Figure 2 A block graph with an occurrence of $P = P[1\,..\,8] =$ `ATCGTAGA` (in orange).

String Matching in Block Graphs (SMBG)
Input: A block graph $G = (V, E, \lambda)$ with b blocks and a pattern $P = P[1\,..\,m]$.
Output: *True* if and only if there is at least one occurrence of P in G.

When h is bounded, we can efficiently reduce h-SMLG on $G = (V, E, \lambda)$ to string matching in a block graph $G' = (V', E', \lambda')$ with $b := h$ blocks of nodes; that is, $V' = V'_1 \sqcup \ldots \sqcup V'_b$. Informally, each block V'_i is set to V (G' gets h copies of V), there is an edge in E' from

$v \in V'_i$ to $v' \in V'_{i+1}$ if and only if $(v, v') \in E$ (G' gets $h-1$ copies of E), and the λ' values are inherited from λ. Then, an occurrence $v_1, \ldots, v_h$ of P exists in G if and only if an occurrence $u'_1, \ldots, u'_b$ of P exists in G'.

SMBG can be solved in $\mathcal{O}(m\,|E| + N)$ time using Theorem 1. Ascone et al. [7] showed how to solve SMBG in $\mathcal{O}(bm + N + |E|)$ time for a specific class of block graphs: when all labels in each V_i have the same length k_i. Another specific class of block graphs has also been investigated: when, for all pairs (V_i, V_{i+1}), all possible edges are present. In this case, the set E becomes implicit, and the resulting graph is known as an *elastic-degenerate string* [26]. Many algorithms have been proposed for SMBG on this class of block graphs [6,9,22]. Similar to Theorem 2, Equi et al. [19] showed that SMBG cannot be solved in $\mathcal{O}((m\,|E|)^{1-\varepsilon} + N)$ time, for any $\varepsilon > 0$, unless SETH fails.

As we are interested in the parameterized version h-SMLG of the more general SMLG problem, we define the corresponding version of string matching in block graphs.

b-String Matching in Block Graphs (b-SMBG)
Input: A block graph $G = (V, E, \lambda)$ with b blocks and a pattern $P = P[1\,..\,m]$.
Output: *True* if and only if there is at least one occurrence of P in G of the form $v_1, \ldots, v_b$.

Since we seek an occurrence of P in G of the form $v_1, \ldots, v_b$, it must form a walk in G of length exactly $(b-1)$, which means that the occurrence $v_1, \ldots, v_b$ spans all b blocks of G. Our algorithms and lower bounds can be adapted to lift this restriction to allow occurrences using some of the blocks. To simplify the presentation, we stick to this version of the problem.

Recall that h-SMLG for $h = 2$ can be solved in linear time [33]. This algorithm solves b-SMBG for $b = 2$ in linear time and then applies the above reduction. The contrast between this result and the general $(m\,|E|)^{1-\varepsilon}$ lower bound brings us to the following basic question:

Can we solve b-SMBG in $\mathcal{O}((m\,|E|)^{1-\varepsilon} + N)$ time for some fixed $b \geq 3$ and some $\varepsilon > 0$?

1.3 Our Contributions and Techniques

If we consider the set of all possible instances of the b-SMBG problem, the algorithm by Amir et al. [4] is essentially optimal under SETH (up to subpolynomial improvements) [19]. We thus take a finer look at the complexity landscape of b-SMBG. Notably, the instances underlying this hardness [19] are (1) **wide** (the number of blocks is in $\Theta(|V|)$) and (2) **sparse** (they have a constant number of edges per node). Thus, it is natural to ask whether we can improve on the state of the art for (1) **narrower**, or (2) **denser** instances. Towards these directions, a simple linear-time algorithm is known for $b = 2$ blocks [33]. We extend this regime by showing a near-linear-time algorithm for $b = 3$ blocks.

▶ **Theorem 18.** *The b-SMBG problem for $b = 3$ can be solved in $\widetilde{\mathcal{O}}(m + |E| + N)$ time. This directly implies that the h-SMLG problem for $h = 3$ can be solved in $\widetilde{\mathcal{O}}(m + |E| + N)$ time.*

We emphasize that obtaining a near-linear-time algorithm for $b = 3$ was *not* a straightforward extension of the $b = 2$ case. First, note that, for $b = 1$, we can simply employ any linear-time string matching algorithm (e.g., [28]). Then, for $b = 2$, the primary challenge presented in [33] was not to obtain a near-linear-time algorithm, but rather to obtain a *linear-time* algorithm. In particular, we can first compute, for every node v in the first block, the longest prefix of P matching a suffix of $\lambda(v)$, and then compute the symmetric information for the second block. Finally, we can simulate the prefix-suffix "concatenation" per edge using existing data structures [21, 23, 33]. Unfortunately, when we move from $b = 2$ to $b = 3$, the existence of the middle block renders this technique inapplicable.

To overcome this challenge, we rely on a well-known combinatorial fact about the *borders* of a string (i.e., about prefixes that are also suffixes of a string): even though there can be a linear number of borders in a string (e.g., consider the string $\mathtt{aa}\ldots\mathtt{a}$), they can be compactly represented as a logarithmic number of *arithmetic progressions* (APs) [10, 12, 23, 24, 28, 29]. These APs are sets of indices of the form $\ell, \ell + x, \ell + 2x, \ldots, \ell + kx = r$; each set has a periodic structure and can thus be represented by a tuple (ℓ, r, x) of the first element ℓ, the last element r, and the common difference x.

For each node in the first and last blocks, we compactly represent the prefixes (or suffixes) of the pattern that match a node label as APs. For middle-block nodes, we compute candidate matches by intersecting the APs of their k in-neighbors with all occurrences of the middle node's label, doing so *implicitly* – without computing this set. Next, using a carefully designed intersection operation, we intersect these candidates with the suffix APs from the k' out-neighbors. To achieve this efficiently, we exploit periodicity information encoded in the APs. Crucially, these steps are no longer *linear* in m: they output only a logarithmic (in m) number of APs per edge. By combining AP-based compression, efficient intersection, and constant-time verification, the algorithm achieves near-linear time.

As a second main result, we show a lower bound by reducing from Boolean matrix multiplication (BMM) to b-SMBG with $b \geq 4$ blocks. As a consequence, a near-linear-time algorithm for this setting exists only if one exists for BMM. This pinpoints the threshold number of blocks for which near-linear-time algorithms are likely. Additionally, under the popular BMM conjecture [3], there can be no combinatorial algorithm polynomially faster than the state-of-the-art SMBG algorithm.

▶ **Intuitive Statement of Theorem 23.** *Unless the combinatorial BMM conjecture is false, we cannot solve b-SMBG in $\mathcal{O}((m\,|E|)^{1-\varepsilon} + N)$ time, for any $b \geq 4$ and constant $\varepsilon > 0$ using a combinatorial algorithm.*

For block graphs with unique labels, our lower bound holds when there are at least six blocks, thus leaving the corresponding problem open for the cases $b = 4$ and $b = 5$.

Towards (2), our goal for developing more efficient algorithms for dense graphs, we give a matrix-multiplication-based algorithm that can be much faster in this setting.

▶ **Corollary 21.** *The b-SMBG problem can be solved in $\mathcal{O}\left(\max(|V|, m)^{\omega} + N\right)$ time, where ω is the matrix multiplication exponent.*

As our previous lower bound reduces to BMM, this algorithm is optimal up to subpolynomial factors. Thus, our upper and lower bounds together essentially settle the complexity landscape of b-SMBG parameterized by the block number b. For more blocks, this algorithm can be analyzed more subtly (see Theorem 20). For example, if there are $\sqrt{|V|}$ blocks, each having equally many nodes, and $m \in \mathcal{O}(|V|)$, then the problem can be solved in $\mathcal{O}(|V|^{2.126})$ time, using the current best bound $\omega \leq 2.371552$ on the matrix multiplication exponent ω [38], thus in less than $|V|^{\omega} \approx |V|^{2.372}$ time.

Interestingly, by relying on rectangular and sparse matrix multiplication, the algorithm also efficiently handles wider and sparser instances and in the worst case matches the state-of-the-art $\mathcal{O}(m\,|E|)$ running time. For example, for the lower bound instances by Equi et al. [19], since they are so wide and sparse, there will be only a constant number of edges between any two consecutive blocks, thus leaving no room for algebraic improvement of this part of the algorithm. Also note that in these instances, $|E| \in \mathcal{O}(|V|)$, meaning they provide little insight on the optimal dependence on $|E|$ versus $|V|$. In some sense, this suggests the state of the art is the *correct* algorithm, at least as long as we only consider the parameters m, $|E|$ and ignore $|V|$, b, which is the setting our work improves on.

We provide upper and lower bounds expressed in terms of $|V|$ or $|E|$. Because the interplay between these parameters depends on graph density, direct comparisons between different bounds – and with the state of the art – can be nuanced. Often, we can rephrase our results dependent on the density (e.g., by using a sparse matrix multiplication routine or relying on the Strong Triangle conjecture [2]). We prioritize readability by focusing on the endpoints of the density spectrum: $|E| \in \widetilde{\mathcal{O}}(|V|)$ and $|E| \in \widetilde{\Omega}(|V|^2)$. We highlight broader density-dependent generalizations only where they are particularly salient.

Finally, we show that solving b-SMBG in $\mathcal{O}((m\,|E|)^{1-\varepsilon} + N)$ time for a polylogarithmic number of blocks is not possible unless the Orthogonal Vectors Hypothesis (OVH), which is implied by SETH [40], is refuted. This lower bound holds for all algorithms (including non-combinatorial ones), illuminating the limits of algebraic techniques for this problem.

▶ **Theorem 26.** *Unless the OVH fails, we cannot solve the b-SMBG problem in $\mathcal{O}((m\,|E|)^{1-\varepsilon} + N)$ time for any $b = \omega(\log |V|)$ and constant $\varepsilon > 0$.*

In a nutshell, we obtain significant improvements for graphs that have a small number of blocks ($b \leq 3$), or are sufficiently dense. Even for dense graphs with a rather large number of blocks, our matrix-multiplication-based algorithm is still beneficial. Given the $\sqrt{|V|}$-block example from above, assuming that consecutive blocks are fully connected, our algorithm runs in time $\mathcal{O}(|V|^{2.126} + N)$, which is faster than $\Theta(m\,|E| + N) = \Theta(|V|^{2.5} + N)$.

Finally, we stress that our algorithms are simple to implement, and as such, they may contribute to practical advancements in applications where the SMLG problem is an important primitive (e.g., in pangenome analyses [34, 35]).

Complexity Landscape of h-SMLG. As long as h is subpolynomial (e.g., constant or logarithmic), the above reduction from h-SMLG to b-SMBG shows that the complexity results we have for b-SMBG directly translate to h-SMLG (up to subpolynomial factors). As the results required to establish the landscape of b-SMBG require only $b \in \Theta(1)$ or $b \in \omega(\log |V|)$, we also settle the landscape for h-SMLG. Namely, there is a near-linear-time algorithm for $h = 3$; starting at $h = 4$ the problem becomes hard under the BMM conjecture; and at $h \in \omega(\log |V|)$ the problem becomes hard under SETH. Also, there is a matrix-multiplication-based algorithm that is conditionally optimal for all h at least 4 and at most subpolynomial. In short, while the reduction from h-SMLG to b-SMBG is efficient only for small values of h, surprisingly, the two problems still have the same (up to subpolynomials) complexity when parameterized by h and b, respectively. This is largely due to the fact that our lower bounds show that b-SMBG (and thus h-SMLG) become hard before the reduction breaks down. After the reduction breaks down, both problems can be solved in the same $\mathcal{O}(m\,|E| + N)$ time and not faster.

Computational Model. We assume the standard word RAM model with machine words consisting of $\Omega(\log n)$ bits, where n is the input size [15]. Basic operations on machine words, such as indirect addressing and arithmetic operations, are thus assumed to take $\mathcal{O}(1)$ time.

Paper Organization. In Section 2, we provide the necessary notation and definitions as well as a few tools that we use in our algorithms. In Section 3, we recap a version of the algorithm by Amir et al. for solving b-SMBG. In Section 4, we present our near-linear-time algorithm for $b = 3$ blocks (Theorem 18). In Section 5, we present our algorithm employing fast matrix multiplication (Corollary 21). Finally, in Section 6, we present our conditional lower bounds for $b \geq 4$ and $b \in \omega(\log |V|)$ blocks (Theorem 23 and Theorem 26, respectively). Any materials (e.g., proofs) omitted from this version can be found in the full version.

2 Preliminaries

For all $n \in \mathbb{N}^+$, we set $[n] \coloneqq \{1, \ldots, n\}$. For two sets X, Y, we write $X \sqcup Y$ for their disjoint union. For a third set Z, we also write $Z = X \sqcup Y$ to indicate that X and Y form a partition of Z. The notation $\widetilde{\mathcal{O}}(f(n))$ denotes $\mathcal{O}(f(n) \cdot \mathrm{polylog}(n))$.

For a directed graph $G = (V, E)$ and a node $v \in V$, we write $\delta^-(v)$ for its in-neighbors and $\delta^+(v)$ for its out-neighbors. In the Landau notation, we also use $\delta^-(v)$ and $\delta^+(v)$ for the respective sizes of these sets, that is, the in- and out-degree of node v.

Arithmetic Progressions. We represent an *arithmetic progression* (AP) by the tuple $(\ell, r, x) \in \mathbb{N}^3$; it represents the set $\{\ell + ix \mid i \in \mathbb{N}, \ell + ix \leq r\}$. Call x its *common difference*. We will treat (ℓ, r, x) as a set when using the $\in$ operator (that is, when we write $i \in (\ell, r, x)$, we mean that i is in the represented set, and not that i is one of ℓ, r, or x). In the word RAM model, any AP can be represented using a constant number of machine words. The *intersection* of two APs is the intersection of the sets they represent. In contrast, we say that two APs for sets X and Y *overlap* if the intervals $[\min X, \max X]$ and $[\min Y, \max Y]$ intersect. We say a set of APs is *overlap-free* if its APs are pairwise non-overlapping. Similarly, we call a set of APs *synchronized* if each pair of non-singleton APs is non-overlapping or shares the same common difference. Note that for singleton APs, the common difference is arbitrary (i.e., they still describe the same singleton set regardless of which x we choose). Thus, for the purposes of synchronization, we disregard them.

We will use the following lemma (whose proof is deferred to the full version).

▶ **Lemma 4.** *The intersection of two APs is empty or a single AP. It can be computed in $\mathcal{O}(1)$ time after $\mathcal{O}(m)$-time preprocessing if the largest index in any of the APs is at most m.*

Strings. An *alphabet* Σ is a finite set of σ elements called *characters*. We consider an *integer* alphabet $\Sigma = [1, \sigma]$. For a string $S = S[1]S[2]\ldots S[n]$ over Σ, we denote its length n by $|S|$ and its i-th character by $S[i]$. A *fragment* of S starting at position i and ending at position j of S is denoted by $S[i\,..\,j]$. A *prefix* of S is a fragment of the form $S[1\,..\,j]$, and a *suffix* of S is a fragment of the form $S[i\,..\,n]$. A fragment $S[i\,..\,j]$ of S corresponds to a *substring* P of S: the string composed of $S[i]\ldots S[j]$. A substring P of S may have many occurrences in S. We characterize an *occurrence* of P in S by its *starting position* $i \in [n]$; that is, $P = S[i\,..\,i + |P| - 1]$. The set of occurrences of P in S is denoted by $\mathrm{Occ}(P, S)$. For two strings S and T, we write ST or $S \cdot T$ for their concatenation.

A positive integer p is called a *period* of a string S, if $S[i] = S[i + p]$, for all $i \in [1, |S| - p]$. The length of a nonempty string is a period of this string, so every nonempty string has at least one period. We define *the period* of a nonempty string S as the smallest of its periods, denoted by $\mathsf{per}(S)$. A *border* of a nonempty string S is a substring S' of S with $|S'| < |S|$ that is both a prefix and a suffix of S. We refer to the longest border of S as *the border* and denote it by $\mathsf{border}(S)$. A well-known fact (cf. [16]) is the following duality between periods and borders: for any string S, $\mathsf{per}(S) + |\mathsf{border}(S)| = |S|$.

▶ **Lemma 5** (Periodicity lemma [20]). *Let S be a string with periods p and q. If $p + q - \gcd(p, q) \leq |S|$, then $\gcd(p, q)$ is also a period of S.*

▶ **Lemma 6** ([28]). *Let $P \in \Sigma^m$. After $\mathcal{O}(m)$-time preprocessing, given $S \in \Sigma^*$, we can compute $\mathrm{Occ}(P, S)$ or the prefixes of P that are a suffix of S in $\mathcal{O}(|S|)$ time.*

▶ **Lemma 7** (Suffix tree [39]). *Let $P \in \Sigma^m$. After $\mathcal{O}(m \log m)$-time preprocessing, given $S \in \Sigma^\ell$, we can compute one of its occurrences in P (if any exists) in $\mathcal{O}(\ell \log \ell)$ time.*

▶ **Lemma 8** (Suffix tree + LCP queries [31]). *Let $P \in \Sigma^m$. After $\mathcal{O}(m \log m)$-time preprocessing, given i and j, we can compute the longest common prefix of $P[i\,..\,m]$ and $P[j\,..\,m]$ in $\mathcal{O}(1)$ time.*

▶ **Lemma 9** ([33]). *The b-SMBG problem for $b = 2$ can be solved in $\mathcal{O}(m + |E| + N)$ time.*

3 Recap: State-of-the-Art Algorithm for b-SMBG

Algorithm 1 restates the state of the art for b-SMBG as, in many ways, it is the natural algorithm for this problem. This is a version of the algorithm by Amir et al. [4] to specifically solve b-SMBG. We have adapted the algorithm's presentation to fit the rest of our paper; the algorithm itself is essentially unchanged. We thus omit a formal time or correctness analysis. Our algorithms will follow the same framework, but speed up crucial operations performed by the algorithm to obtain our improvements.

Algorithm 1 The state-of-the-art algorithm for b-SMBG.

Input: Block graph $G = (V = V_1 \sqcup \cdots \sqcup V_b, E, \lambda)$, pattern P of length m

1. For all $v \in V$, we maintain a set $M_v \subseteq [m]$ that represents the set of prefixes of P that can be matched by a walk ending in v. In particular, $i \in M_v$ represents $P[1\,..\,i]$.
2. For each node $v \in V_1$, we compute the borders of string $P\#\lambda(v)$, where $\# \notin \Sigma$ is a separator. We add the lengths of those borders to M_v.
3. For each node $v \in V_j$, for $j \in [2, b)$, we compute the union of all the match sets of the predecessors of v, that is $M'_v \coloneqq \bigcup_{v' \in \delta^-(v)} M_{v'}$. We compute all occurrences of $\lambda(v)$ in P. If there is an occurrence starting at position i and $(i-1) \in M'_v$, we add $i + |\lambda(v)| - 1$ to M_v.
4. For each node $v \in V_b$, compute M'_v as in the previous step. Compute the borders of string $\lambda(v)\#P$, where $\# \notin \Sigma$ is a separator. If there is a border of length k and $(m-k) \in M'_v$, add m to M_v.
5. Report all nodes $v \in V_b$ such that $m \in M_v$. Every such node is the endpoint of a matching walk.

▶ **Theorem 10** ([4]). *Algorithm 1 solves the b-SMBG problem in $\mathcal{O}(m\,|E| + N)$ time.*

Let us remark that the direct reduction to standard string matching by building the concatenation of all strings given by paths of length $(b-1)$, and then searching P, might give $\Omega(|V|^b)$ many such strings, making this strategy unattractive for any $b \geq 2$.

4 Exploiting Periodicity Yields Near-Linear Time for Three Blocks

In this section, we focus on 3-block graphs $G = (V = V_1 \sqcup V_2 \sqcup V_3, E, \lambda)$. The core bottleneck in the state-of-the-art algorithm (Algorithm 1) is maintaining a set of size m to represent how each string $\lambda(v)$, for $v \in V$, interacts with the length-m pattern P. This means that for some middle node $v \in V_2$, we explicitly consider all occurrences of $\lambda(v)$ in P. Similar considerations apply for the suffixes and prefixes of P matched by the nodes of the first and last block, respectively. Fundamentally, an approach like that cannot directly lead to an algorithm faster than $\Theta(m\,|V|)$ as this is the size of the data just described. Our goal in this section is to circumvent exactly this bottleneck. We do this by representing how each node interacts with P more efficiently. This will then allow us to combine that information more efficiently, solving b-SMBG for $b = 3$ significantly faster than the state of the art.

4.1 How the First and Last Block Interact with the Pattern

We take a closer look at how we can represent prefixes, suffixes, and borders of P. A prefix of a string is characterized by its ending position, and a suffix of a string is characterized by its starting position. A set of prefixes of a string can thus be characterized as a set of natural numbers, which can always be written as a union of APs. We next prove several combinatorial properties showing that, for the cases we care about, relatively few such APs suffice and they behave nicely, in a sense that we will formalize shortly. By symmetry, the same holds for a set of suffixes of a string. We will use this AP representation extensively. We similarly treat sets of borders of a string. As a border is both a prefix and a suffix, we (choose to) identify it with the ending position of the prefix, which coincides with its length. We start with the following well-known result that underlies our algorithmic improvement.

▶ **Lemma 11.** *For any $P \in \Sigma^m$, after $\mathcal{O}(m)$-time preprocessing, given $S \in \Sigma^*$, all prefixes of P that are a suffix of S can be computed as $\mathcal{O}(\log m)$ overlap-free APs in $\mathcal{O}(|S|)$ time.*

Proof. We employ Lemma 6. The prefixes of P are encoded from $[\min(m, |S|)]$, and so they can be sorted in $\mathcal{O}(|S|)$ time using bucket sort. It is well-known that the sorted sequence of periods of a length-n string can be cut into $\mathcal{O}(\log n)$ (overlap-free) APs [28]. By the duality between periods and borders, this also holds for the prefixes of P that are a suffix of S. ◀

The symmetric claim (Corollary 12) holds too (e.g., by reversing the strings P and S).

▶ **Corollary 12.** *For any $P \in \Sigma^m$, after $\mathcal{O}(m)$-time preprocessing, given $S \in \Sigma^*$, all suffixes of P that are a prefix of S can be computed as $\mathcal{O}(\log m)$ overlap-free APs in $\mathcal{O}(|S|)$ time.*

We will apply Lemma 11 to the pattern P paired with every node label in V_1 (and Corollary 12 with every node label in V_3). To efficiently work with their output, we need the following structural fact about the interaction of two APs *for the same P but a different S*.

▶ **Lemma 13.** *Given $P \in \Sigma^m$ and $S_1, S_2 \in \Sigma^*$, let $\mathcal{A}_1$ be the set of lengths of all prefixes of P that are suffixes of S_1, and $\mathcal{A}_2$ be the set of lengths of all prefixes of P that are suffixes of S_2. Given an integer $j \geq 0$, let (ℓ_1, r_1, x_1) be an AP that contains exactly all elements from $\mathcal{A}_1 \cap [2^j, 2^{j+1}) \cap [\ell_1, r_1]$ and (ℓ_2, r_2, x_2) be an AP that contains exactly all elements from $\mathcal{A}_2 \cap [2^j, 2^{j+1}) \cap [\ell_2, r_2]$. If both APs contain at least 3 elements, then $x_1 = x_2$.*

We will make sure that in our algorithm, the APs match the preconditions of this lemma and we can thus apply this result. The fact that overlapping APs from different S_1, S_2 with at least three elements share their common difference will allow us to restore overlap-freeness (which is the nice behavior mentioned above) after combining two sets of APs.

Proof. Let u_1, u_2, u_3 be three consecutive elements in (ℓ_1, r_1, x_1) with common difference x_1. Let v_1, v_2, v_3 be three consecutive elements in (ℓ_2, r_2, x_2) with common difference x_2.

These elements establish $P[1\,..\,u_3]$ has period x_1 and $P[1\,..\,v_3]$ has period x_2. Since $u_1, u_3 \in [2^j, 2^{j+1})$, $u_3 - u_1 = 2x_1 < 2^j$, so $x_1 < 2^{j-1}$. Similarly, $x_2 < 2^{j-1}$. Let $L = \min(u_3, v_3)$. Since $L \geq 2^j$ and $x_1 + x_2 < 2^j$, the prefix $P[1\,..\,L]$ satisfies the condition of the periodicity lemma $L \geq x_1 + x_2 - \gcd(x_1, x_2)$. Therefore, $P[1\,..\,L]$ has period $g := \gcd(x_1, x_2)$.

Since $P[1\,..\,u_3]$ has period g, the prefix $P[1\,..\,u_3 - g]$ is a suffix of $P[1\,..\,u_3]$. Therefore, $P[1\,..\,u_3 - g]$ is a suffix of S_1. We verify that $u_3 - g$ lies within the interval $\mathcal{I} = [2^j, 2^{j+1})$:

1. $u_3 - g < u_3 < 2^{j+1}$ (true since $g \geq 1$).
2. $u_3 - g \geq u_3 - x_1 = u_2 \geq 2^j$ (true since $g = \gcd(x_1, x_2) \leq x_1$).

Since $u_3 - g \in \mathcal{A}_1 \cap \mathcal{I} \cap [\ell_1, r_1]$ and the AP (ℓ_1, r_1, x_1) is exhaustive for that interval (i.e., it contains all elements from $\mathcal{A}_1 \cap [\ell_1, r_1]$), $u_3 - g$ must be an element of the AP. In an AP with common difference x_1, if two elements A and B exist, then $|A - B|$ must be a multiple of x_1. Here, $|u_3 - (u_3 - g)| = g$. Thus, if g is a multiple of x_1, then $g \geq x_1$. Combined with the fact that $g = \gcd(x_1, x_2) \leq x_1$, it must be that $g = x_1$. By symmetry for the second AP, $g = x_2$. We conclude that $x_1 = x_2 = \gcd(x_1, x_2)$. ◀

In the following lemma, we show how, by iterative application of Lemma 13, we combine the sets of APs for many different strings into one large set of APs that is still "well-behaved".

▶ **Lemma 14.** *Given $P \in \Sigma^m$ and $S_1, \ldots, S_k \in \Sigma^*$, let $\{A_i\}_{i \in [k]}$ be such that each A_i is the set of APs from Lemma 11 for P and S_i.*

Given all A_i as input, we can compute a synchronized set of $\mathcal{O}(k \log m)$ APs representing all elements in the union of all AP elements over all A_i in $\mathcal{O}(k \log m)$ total time. Moreover, the non-singleton APs have at most $\mathcal{O}(\log m)$ unique common differences.

Proof. The size of our input is $\mathcal{O}(k \log m)$ by Lemma 11. Our only operation will be splitting APs. By this, we mean replacing one AP (ℓ, r, x) by two APs (ℓ_1, r_1, x) and (ℓ_2, r_2, x) that represent the same elements and have the property $\ell_1 \leq r_1 < \ell_2 \leq r_2$.

Let $\mathcal{A} \coloneqq \bigcup_{i \in [k]} A_i$. First, split all APs so that each resulting AP is a subset of an interval of the form $[2^j, 2^{j+1})$, for $j = 0, 1, \ldots, \lfloor \log m \rfloor$. That is, split each AP at every dyadic boundary 2^j it intersects, so that each resulting AP is fully contained in a single interval $[2^j, 2^{j+1})$. For each A_i, this increases its size to at most $2|A_i| + \log m$ because A_i was overlap-free. Therefore, in total, the size of $\mathcal{A}$ increases to at most $(2|\mathcal{A}| + k\lceil \log m \rceil) \in \mathcal{O}(k \log m)$. Second, we split each AP that now contains exactly two elements into two singletons, again at most doubling the size of $\mathcal{A}$. The total size remains in $\mathcal{O}(k \log m)$. As we spent constant time per split, the time is bounded by the input and output size.

Now we want to check the property that the non-singleton APs in $\mathcal{A}$ do not overlap unless they share their common difference. Although each A_i is overlap-free, APs originating from different sets A_i may overlap. To address this, we treat overlaps in two cases:

Case (1): if two APs overlap and at least one of them contains at most two elements, then by our splitting, it is a singleton. By definition, overlaps between singletons and any other AP do not violate the synchronized definition.

Case (2): assume that two overlapping APs (ℓ_i, r_i, x_i), (ℓ_j, r_j, x_j) each have length at least three. By Lemma 11, both APs exhaustively cover all elements in the range $[\ell_i, r_i], [\ell_j, r_j]$ of their respective A_i, A_j. Thus, we apply Lemma 13, proving that $x_i = x_j$, and do nothing further. In particular, Lemma 13 also shows that if any two APs of length at least three appear in the same dyadic interval, they share the same common difference, proving that there can be at most $\mathcal{O}(\log m)$ unique common differences. ◀

The symmetric result holds for the APs from Corollary 12.

4.2 How to Match with the Middle Block

Now that we have shown how to represent the parts of the pattern matched by the first and last block and how to access that information efficiently, we use this tool for solving b-SMBG for $b = 3$ faster than $\mathcal{O}(m|E| + N)$ time. Our next lemma shows that we can efficiently do the following: Given AP positions for the prefixes of the pattern P matched by the in-neighbors of a middle-block node, check which of these are followed by the label of the middle node. In the standard Algorithm 1, this step requires, for all $v^2 \in V_2$, explicitly

computing all occurrences of $\lambda(v^2)$ in P and then comparing this to the prefixes matched by nodes in V_1. We cannot afford this as the string matching between all the labels in V_2 and P requires $\Omega(m\,|V_2|)$ time. Fortunately, we can use the periodicity information about P encoded in the APs representing the prefixes matched by V_1 to perform this important step in the algorithm much more efficiently. The core fact underlying the following lemma is this.

▶ **Observation 15.** *If (ℓ, r, x) is an AP of prefixes of P that are suffixes of some string S and if the AP has at least 2 elements, then x is a period of $P[1\,..\,r]$. We define the* extended periodic region *of (ℓ, r, x) as the longest prefix $P[1\,..\,r']$ of P, with $r' \geq r$, that has period x.*

The symmetric version holds for an AP of suffixes of P that are prefixes of some S.

▶ **Lemma 16.** *We can preprocess a string $P \in \Sigma^m$ and a block graph $G = (V, E, \lambda)$ in time $\widetilde{\mathcal{O}}(m + N_G)$ into a data structure that can answer the following queries in $\mathcal{O}(1)$ time:*

Given an AP (ℓ, r, x), representing the prefixes of P that are a suffix of a label $\lambda(v^1)$, and some (other) label $\lambda(v^2)$, output all occurrences i of $\lambda(v^2)$ in P such that $i - 1 \in (\ell, r, x)$. These occurrences are output as $\mathcal{O}(1)$ APs, each of which is a sub-AP with common difference x of the input AP (ℓ, r, x). Also, any occurrence of $\lambda(v^2)$ that would end after the extended periodic region is output as a singleton AP.

Proof. As preprocessing, we construct the LCP data structure from Lemma 8 on P. In addition, for each node label $\lambda(v)$ we use Lemma 7 to compute one occurrence of $\lambda(v)$ in P, if such an occurrence exists. The resulting position is stored for $\mathcal{O}(1)$-time access. This information enables us to verify (other) occurrences of $\lambda(v)$ in P in $\mathcal{O}(1)$ time by asking LCP queries on P. The total preprocessing time is thus $\widetilde{\mathcal{O}}(m + N_G)$.

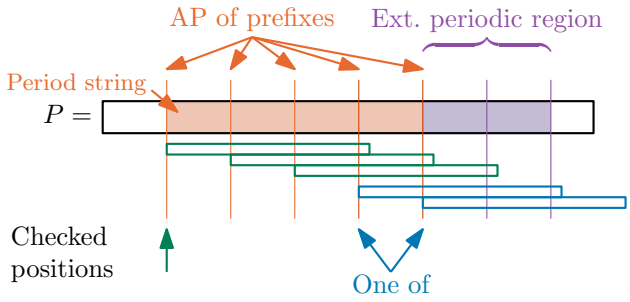

Figure 3 Illustration of Lemma 16. Orange lines depict an AP of prefixes of P. As purple, further repetitions of the *period string*. Below, all possible occurrences of a string $\lambda(v^2)$ directly *after* an AP element. These fall into the two categories used in the proof: entirely within the periodic region (green), and partially or entirely after (blue). For green, we only need to check a single starting position (indicated by an arrow) to deduce all occurrences. For blue, we only need to check one of the starting positions in the indicated range.

Now consider a query consisting of an arithmetic progression (ℓ, r, x) and a label $\lambda(v^2)$. Assume r is the AP's largest element and let $\alpha := \lfloor (r - \ell)/x \rfloor + 1$ denote the number of its elements. Let $\alpha' \geq \alpha$ be the maximum integer such that we have α' full repetitions in the extended periodic region (which is defined as long as $\alpha \geq 1$), that is, the fragments $P[(\ell + tx + 1)\,..\,(\ell + (t+1)x)]$ are equal for every $t = 0, 1, \ldots, \alpha' - 1$, but $P[(\ell + \alpha' x + 1)\,..\,(\ell + (\alpha' + 1)x)]$ is either different or undefined. If $\alpha \leq 2$, then these positions can be checked directly using LCP queries (Lemma 8). Henceforth, assume that $\alpha \geq 3$. Using one LCP query, we can check how far this periodic region extends to the right (computing α'). We distinguish possible occurrences of $\lambda(v^2)$ into two categories; see Figure 3.

First (green in the figure), consider possible occurrences of $\lambda(v^2)$ that start at or before $(r+1)$ and are fully contained within the periodic region. We check whether $\lambda(v^2)$ occurs at position $(\ell+1)$. If so, then by periodicity, it also occurs starting at every AP position $\ell+tx+1$ as long as the occurrence is fully contained in the periodic region $P[\ell+1\,..\,\ell+\alpha' x]$. These occurrences therefore form a single AP.

Second (blue in the figure), consider possible occurrences of $\lambda(v^2)$ that start at or before $(r+1)$ and end after the periodic region. If such an occurrence succeeds the i-th-before-last AP position (i.e., $(r+1)$ succeeds $i=0$ and so on), then $\lambda(v^2)$ must start with $\min(i+\alpha'-\alpha+1, \lfloor |\lambda(v^2)| / x \rfloor)$ repetitions of the period string. By computing the maximum number of period-string repetitions at the start of $\lambda(v^2)$ using an LCP query, we can deduce i from α, α', and $|\lambda(v^2)|$. Knowing i, we verify the occurrence using a single additional LCP query. Since there are exactly α' repetitions of the period string, there is at most one such alignment i. Thus, we add either nothing or one singleton AP to the output.

Each case contributes at most one AP and all checks take $\mathcal{O}(1)$ time after preprocessing. ◀

4.3 Putting Everything Together

Now that we have shown how to combine multiple sets of APs into a single well-behaved set, we define the intersection operation. We provide an intuition right after the statement.

▶ **Lemma 17.** *Let $P \in \Sigma^m$, and let $S_1, \ldots, S_k \in \Sigma^*$ with corresponding AP sets A_i^1 that come from applying Lemma 16 to Lemma 11 for some $v^2 \in V_2$. Also, let $S'_1, \ldots, S'_{k'} \in \Sigma^*$ with corresponding AP sets A_j^3 from Corollary 12. Given all A_i^1, all A_j^3 and v^2 as input, we can compute in $\mathcal{O}((k+k')\log^2 m)$ time if there is a value z such that z is in some AP in some A_i^1 and $z + |\lambda(v^2)| + 1$ is in some AP in some A_j^3.*

One should think about the strings $S_1, \ldots, S_k$ as node labels from the first block of G, and the strings $S'_1, \ldots, S'_{k'}$ as node labels from the last block of G. This middle-block node is connected to the respective nodes from the first and last block. We may also assume that the label of the middle node appears in P directly after every position listed in all A_i^1 by Lemma 16. We now want to determine the positions in P where this is completed into a full match of P by the connected first and last block nodes.

Proof. We may assume Lemma 14 had been applied to $\{A_i^1\}_{i\in[k]}$ and the symmetric version of the lemma to $\{A_j^3\}_{j\in[k']}$. Let $\mathcal{A}^1 \coloneqq \bigcup_{i\in[k]} A_i^1$ and $\mathcal{A}^3 \coloneqq \bigcup_{j\in[k']} A_j^3$. Next, shift all APs in $\mathcal{A}^3$ by $(-|\lambda(v^2)|-1)$; that is, decrease both the lower and upper bounds of each AP by $|\lambda(v^2)|+1$. To distinguish between the shifted and unshifted version, we add a $\hat{}$ symbol on the shifted version. This preserves the common difference and the number of APs, and can be done in $\mathcal{O}(|\mathcal{A}^3|) = \mathcal{O}(k' \log m)$ time. After this transformation, the desired output is $\bigcup \mathcal{A}^1 \cap \bigcup \hat{\mathcal{A}}^3$. To determine if this intersection is empty efficiently, proceed as follows.

We extract as singletons the last element of every AP in $\mathcal{A}^1$ and the first element of each AP in $\hat{\mathcal{A}}^3$. If, after this, any AP has fewer than three elements, split it into singleton APs. This increases the number of APs by a constant factor. Sort the singleton APs by value. Since $|\mathcal{A}^1| + |\hat{\mathcal{A}}^3| = \mathcal{O}((k+k')\log m)$, sorting takes $\mathcal{O}((k+k')\log^2 m)$ time. In this sorted list, check if there are two singleton APs that intersect.

Afterwards, we do a second pass in which we check whether a singleton AP of $\mathcal{A}^1$ intersects a non-singleton AP of $\hat{\mathcal{A}}^3$. We then repeat the process with switched roles of $\mathcal{A}^1$ and $\hat{\mathcal{A}}^3$. We do a sorted scan of the APs and maintain a balanced search tree (BST) [15] with the currently active non-singleton APs $(\hat{\ell}_3, \hat{r}_3, x_3)$ (i.e., those with active interval $[\hat{\ell}_3, \hat{r}_3]$), indexed

by $\hat{\ell}_3$ mod x_3. If there are collisions in the BST, we only need to keep the AP that extends the furthest to the right. Since all non-singleton APs are synchronized by Lemma 14, all APs active at the same time have the same common difference x_3. When scanning a singleton AP, consisting of a single element ℓ_1, we query the BST with the value for ℓ_1 mod x_3. Observe that ℓ_1 intersects an active AP $(\hat{\ell}_3, \hat{r}_3, x_3)$ if and only if $\ell_1 \equiv \hat{\ell}_3 \pmod{x_3}$, thus we find an intersection if one exists. Since all keys are in $\mathcal{O}(m)$, BST operations take $\mathcal{O}(\log m)$ time, the sorting takes $\mathcal{O}((k+k')\log^2 m)$ time, and so the total time is $\mathcal{O}((k+k')\log^2 m)$.

We are left to argue that if there is an intersection between two non-singleton APs, then there is an intersection involving a singleton AP. For this assume that $a^1 = (\ell_1, r_1, x_1)$ from $\mathcal{A}^1$ and $\hat{a}^3 = (\hat{\ell}_3, \hat{r}_3, x_3)$ from $\hat{\mathcal{A}}^3$ are non-singleton APs that intersect. If a^1 and $\hat{a}^3$ intersect, they must overlap. Also, since we extracted the last element of a^1 and first element of $\hat{a}^3$, the original APs overlapped by at least $x_1 + x_3 + 1$. Let r_1' be the last element of a^1 before the extraction. We know the extended periodic region of a^1 extends at least $\left|\lambda(v^2)\right|$ to the right after r_1' as otherwise r_1' would have been output as a singleton by Lemma 16. Thus, the extended periodic region of a^1 overlaps with $\hat{a}^3$ by at least $x_1 + x_3 + \left|\lambda(v^2)\right| + 1$. After undoing the shift, the extended periodic regions overlap by at least $x_1 + x_3$. To apply the periodicity lemma, we now also need that x_1 and x_3 are periods of this overlap. This follows from Observation 15, which we can apply since all APs we have are cut up APs of those originally produced using Lemma 11. In conclusion, $\gcd(x_1, x_3)$ is a period of the overlap of the extended periodic regions and thus of P.

Now, let y be an element of the intersection of a^1 and $\hat{a}^3$, then any $y + z \cdot \gcd(x_1, x_3)$ must be in $\hat{\mathcal{A}}^3$. This holds since if $P[s\,..\,m]$ is a suffix of P that is a prefix of some S_j', the same is true for $P[s + z \cdot p\,..\,m]$, where p is any period of $P[s\,..\,m]$, so in particular for $s = y + \left|\lambda(v^2)\right| + 1$ and $p = \gcd(x_1, x_3)$. Clearly, r_1' can be written as $\ell_1 + z \cdot x_1$ and thus as $y + z' \cdot \gcd(x_1, x_3)$. Therefore, a singleton from $\mathcal{A}^1$ intersects an AP from $\hat{\mathcal{A}}^3$. ◀

Together, Lemmas 16 and 17 enable the core matching step of our approach. Our improved algorithm (stated as Algorithm 2) retains the overall structure of Algorithm 1 for $b = 3$, but leverages the data structures and combinatorial insights developed in this section to carry out these core steps significantly more efficiently.

Algorithm 2 Our improved algorithm for b-SMBG and $b = 3$.

Input: Block graph $G = (V_1 \sqcup V_2 \sqcup V_3, E, \lambda)$, pattern P of length m

1. Preprocess P and G using Lemma 16. Preprocess P using Lemma 11 and Corollary 12.
2. For each $v^1 \in V_1$, compute A_{v^1} using Lemma 11 on P and $\lambda(v^1)$.
3. For each $v^3 \in V_3$, compute A_{v^3} using Corollary 12 on P and $\lambda(v^3)$.
4. For each $v^2 \in V_2$:
   - **(a)** Let $\mathcal{A}^1 \coloneqq \bigcup_{v^1 \in \delta^{\urcorner}(v^2)} A_{v^1}$ represent the set of prefixes matched by in-neighbors of v^2.
   - **(b)** Let $\mathcal{A}^3 \coloneqq \bigcup_{v^3 \in \delta^{+}(v^2)} A_{v^3}$ represent the set of suffixes matched by out-neighbors of v^2.
   - **(c)** For each AP from $\mathcal{A}^1$, use the query from Lemma 16 to restrict to the prefixes of P that are directly followed by an occurrence of $\lambda(v^2)$.
   - **(d)** Apply Lemma 17 to (the restricted) $\mathcal{A}^1$ and to $\mathcal{A}^3$ to compute if there is a position in P where an occurrence of $\lambda(v^2)$ can be extended to a full match of P in G.

▶ **Theorem 18.** *The b-SMBG problem for $b = 3$ can be solved in $\widetilde{\mathcal{O}}(m + |E| + N)$ time. This directly implies that the h-SMLG problem for $h = 3$ can be solved in $\widetilde{\mathcal{O}}(m + |E| + N)$ time.*

Proof. For b-SMBG, we analyze Algorithm 2. The implication for h-SMLG for $h = 3$ follows from the reduction outlined in the introduction.

First, observe that in Step 4(c), every AP queried using Lemma 16 is a subset of an AP computed in Step 4(a). Hence, all elements of these APs are prefixes of P that are suffixes of some label from the first block, satisfying the precondition of Lemma 16.

The running time follows by simply adding the running times from Lemmas 11, 14, 16, and 17 and Corollary 12. For correctness, observe that Algorithm 2 computes exactly the same sets of indices as the Algorithm 1 does, just that here we represent them as APs. ◀

Reporting All Occurrences. It is possible to extend Algorithm 2 to report all occurrences of P in G in the form $(v_1, v_2, v_3, i_1, i_3)$, where $P = \lambda(v_1)[i_1 \,..\, |\lambda(v_1)|] \cdot \lambda(v_2) \cdot \lambda(v_3)[1 \,..\, i_3]$, making it more useful for the applications outlined in the introduction (as opposed to a purely decision algorithm). The required changes are described in the full version. In this case, the runtime of the algorithm gains an additional +output term.

Solving SMBG for $b = 3$. To solve SMBG, the more general problem where matches are not required to span all $b = 3$ blocks, we do the following. Let $G = (V = V_1 \sqcup V_2 \sqcup V_3, E, \lambda)$ and P be the input to SMBG. We apply Lemma 6 using P and all the node labels $\lambda(v)$ for $v \in V_1 \sqcup V_2 \sqcup V_3$ with $|\lambda(v)| \geq m$. The total time is $\mathcal{O}(m + N)$. We then apply Lemma 9 two times: once on the graph induced by $V_1 \sqcup V_2$ from G; and once on the graph induced by $V_2 \sqcup V_3$ from G. The total time is $\mathcal{O}(m + |E| + N)$. Finally, we apply Theorem 18 on G. The time is $\widetilde{\mathcal{O}}(m + |E| + N)$. We have arrived at the following result.

▶ **Corollary 19.** *The SMBG problem for $b = 3$ can be solved in $\widetilde{\mathcal{O}}(m + |E| + N)$ time.*

5 An Algorithm Based on Matrix Multiplication

In this section, we give an alternative algorithm for the b-SMBG problem that works for an arbitrary number b of blocks.

When solving the b-SMBG problem, a core challenge arises. For each node v in our block graph G, we need to merge the occurrences of $\lambda(v)$ in P with the prefixes of P matched by the predecessors of v in G. Algorithm 1 considers all predecessors of v in G individually, introducing the $\mathcal{O}(|E|)$ factor to the algorithm's running time. We avoid performing this costly operation individually for each node and compute them for a complete block simultaneously using matrix multiplication.

We use $\omega \in [2, 2.372)$ as the running time exponent of matrix multiplication of square matrices [38]. We will also make use of the notation $\omega(a, b, c)$ to denote the running time exponent of rectangular matrix multiplication of an $n^a \times n^b$ by an $n^b \times n^c$ matrix, and we use [11] to calculate the concrete value of this function. We show the following result.

▶ **Theorem 20.** *The b-SMBG problem can be solved in $\mathcal{O}\left(\left(\sum_{i\in[b-1]} n^{\omega(x,y_i,y_{i+1})}\right) + N\right)$ time, where $n^x = m$ and $n^{y_i} = |V_i|$, for each $i \in [b]$.*

We defer the proof of Theorem 20 to the full version. We get the following, easier to read corollary – we again defer its proof to the full version.

▶ **Corollary 21.** *The b-SMBG problem can be solved in $\mathcal{O}\left(\max(|V|, m)^\omega + N\right)$ time, where ω is the matrix multiplication exponent.*

Algorithm 3 iteratively computes the matching prefix of the pattern P to the first i blocks of the input graph. Inside the algorithm, we use the matrix $A'_{V_j} \in \{0,1\}^{m \times |V_j|}$ for which $A'_{V_j}[i,v] = 1$ if and only if there exists a path in $V_1 \times \ldots \times V_j$ ending in v that matches the prefix $P[1\,..\,i]$ of P (as a suffix). The matrix $A_{V_j} \in \{0,1\}^{m \times |V_j|}$ at cell (i,v) indicates whether the label $\lambda(v)$ matches P at $P[i\,..\,i+|\lambda(v)|-1]$. The intermediate matrix $T_j \in \{0,1\}^{m \times |V_j|}$ stores for each prefix $P[1\,..\,i]$ and each node v whether a predecessor of v already matches $P[1\,..\,i]$. We denote binary matrix multiplication using the notation $(\cdot)$.

Algorithm 3 Our algorithm for b-SMBG based on matrix multiplication.

Input: Block graph $G = (V = V_1 \sqcup \cdots \sqcup V_b, E, \lambda)$, pattern P of length m

1. For every node v in V_1, compute a binary column vector $\vec{M}_v$ of length m s.t. $\vec{M}_v[i] = 1$ if $P[1\,..\,i]$ is a suffix of $\lambda(v)$. Assemble these into a $\{0,1\}^{m \times |V_1|}$ matrix A_{V_1}.
2. Construct the adjacency matrix $\mathrm{Adj}_{V_{1\to 2}}$ between V_1 and V_2. Compute $T_2 \coloneqq A_{V_1} \cdot \mathrm{Adj}_{V_{1\to 2}}$.
3. For every node u in V_2, compute a binary column vector $\vec{M}_u$ of length m s.t. $\vec{M}_u[i] = 1$ if $i \in \mathrm{Occ}(\lambda(u), P)$. Assemble these into a match matrix $A_{V_2} \in \{0,1\}^{m \times |V_2|}$.
4. Compute the matrix A'_{V_2} of prefix matches of P by V_1 and V_2. Set $A'_{V_2}[i,u] \coloneqq T_2[i',u] \wedge A_{V_2}[i'+1,u]$ for all $i \in [m], u \in V_2$ with $i' = i - |\lambda(u)|$. If $i' \leq 0$, the matrix value is treated as 0.
5. Repeat Steps 2 to 4 for all blocks $j \in [3,b)$, using $A'_{V_{j-1}}$ in place of A_{V_1}.
6. For the last block b, similar to the first, construct $A_{V_b} \in \{0,1\}^{m \times |V_b|}$ such that $A_{V_b}[i,v] = 1$ for any $v \in V_b$ if $P[i\,..\,m]$ is a prefix of $\lambda(v)$. Repeat Steps 2 and 4, using this A_{V_b} instead in Step 4.
7. If A'_{V_b} contains at least one 1, we have found a matching walk $v_1, \ldots, v_b$.

Note that if our block graph is sufficiently sparse and our pattern is sufficiently small with $m\,|E| \ll \max(|V|\,, m)^\omega$, Algorithm 1 ([4]) is faster than the algorithm underlying Theorem 20. However, using sparse (rectangular) matrix multiplication instead [1, 25], we always match or improve upon the $\mathcal{O}(m\,|E|)$ state-of-the-art running time.

Reporting All Occurrences. Algorithm 3 decides whether an occurrence exists. To output any one such walk, we trace back which values of T_j and thus $A'_{V_{j-1}}$ were non-zero, thus constructing an occurrence.

Furthermore, Algorithm 3 allows us to count the number of occurrences. Note that the intermediate matrix T_j does not need to be binary. Using the standard matrix product instead of the binary matrix multiplication, it instead computes the number of prefix block matches for each node. Further, using multiplication instead of a logical operation in Step (4) yields the number of prefix matches of P for A'_{V_b}.

Solving SMBG. In fact, with some extra care, the algorithm underlying Theorem 20 solves the more general SMBG problem, where matches must not span all blocks, within the same time complexity. The proof of the following corollary is deferred to the full version.

▶ **Corollary 22.** *The SMBG problem can be solved in* $\mathcal{O}\left(\left(\sum_{i\in[b-1]} n^{\omega(x,y_i,y_{i+1})}\right) + N\right)$ *time, where* $n^x = m$ *and* $n^{y_i} = |V_i|$*, for each* $i \in [b]$.

6 Lower Bounds

In this section, we give an overview of our lower bounds; we defer their proofs to the full version. Comparing our results for $b \leq 3$ and $b \geq 4$ blocks, we see a jump in the time complexity. While for $b \leq 3$ we have near-linear-time algorithms ([33] and Algorithm 2), we only have an algebraic $\mathcal{O}(\max(|V|, m)^\omega + N)$-time or a combinatorial $\mathcal{O}(m|E| + N)$-time algorithm for $b \geq 4$ (Corollary 21 and [4]), which for $\omega > 2$ represents a significant increase.

We show that these algorithms are conditionally optimal by giving a matching lower bound for $b = 4$ blocks, implying that the observed difference in complexity is inherent to b-SMBG. We reduce from the Triangle Detection (TD) problem, which consists of determining whether there exists a triangle (i.e., a clique of size 3) in an (undirected) graph. We give its accompanying hardness hypothesis below; a short discussion is included in the full version.

▶ **Triangle Detection Hypothesis.** *Let $\varepsilon > 0$ be a constant. There is no combinatorial algorithm that solves the TD problem on n-node graphs in $\mathcal{O}(n^{3-\varepsilon})$ time. Likewise, there is no (algebraic) algorithm that solves the TD problem in $\mathcal{O}(n^{\omega-\varepsilon})$ time, where ω is the matrix multiplication exponent.*

▶ **Theorem 23.** *If we can solve b-SMBG with $b \geq 4$ blocks on a binary alphabet in time $T_4(|V|, N, m)$, we can solve TD in a graph with n nodes in time $\mathcal{O}(T_4(n, n^2, n) + n^2)$.*

Under the Triangle Detection hypothesis, we get the following lower bounds for b-SMBG if the matrix multiplication exponent ω is greater than 2. Note that, in case $\omega = 2$, we get a lower bound for b-SMBG from reading all, potentially $\Theta(|V|^2)$ many, input edges.

▶ **Corollary 24.** *There is no $\mathcal{O}(\max(|V|, m)^{\omega-\varepsilon} + N)$-time algorithm for b-SMBG for $b \geq 4$ blocks and a constant $\varepsilon > 0$ under the Triangle Detection hypothesis.*

▶ Remark 25. There is no combinatorial $\mathcal{O}(\max(|V|, m)^{3-\varepsilon} + N)$-time or $\mathcal{O}((m|E|)^{1-\varepsilon} + N)$-time algorithm for b-SMBG for $b \geq 4$ and a constant $\varepsilon > 0$ under the combinatorial BMM conjecture [2, 36].

This shows that Algorithm 3 is optimal up to subpolynomial factors for $b \geq 4$ blocks. We note that our reduction uses *duplicate* labels in the constructed block graph; another construction with $b = 6$ blocks and *unique* labels is deferred to the full version. It remains an interesting open question whether there exists a faster, $\mathcal{O}(\max(|V|, m)^{\omega-\varepsilon} + N)$-time algorithm, for any $\varepsilon > 0$, for b-SMBG with unique labels.

For the combinatorial $\mathcal{O}(m|E| + N)$-time algorithm [4], we give a matching lower bound for polylogarithmically many blocks based on the Orthogonal Vectors (OV) problem, which is defined as follows.

Given two sets of n vectors $A, B \subset \{0,1\}^d$, decide whether $\exists \alpha \in A, \exists \beta \in B : \langle \alpha, \beta \rangle = \sum_{i=1}^{d} \alpha[i]\beta[i] = 0$.

It is well-known that SETH implies the following hypothesis [40].

▶ **Orthogonal Vectors Hypothesis (OVH).** *For $d \in \omega(\log n)$, there is no constant $\varepsilon > 0$ such that the OV problem can be solved in $\mathcal{O}(n^{2-\varepsilon})$ time.*

Our next lower bound thus rules out any (also non-combinatorial) improvements over the $\mathcal{O}(m|E| + N)$ algorithm. We note that a similar lower bound exists in [19]; however, their construction requires linearly many blocks in the reduced instance.

▶ **Theorem 26.** *Unless the OVH fails, we cannot solve the b-SMBG problem in $\mathcal{O}((m|E|)^{1-\varepsilon} + N)$ time for any $b = \omega(\log |V|)$ and constant $\varepsilon > 0$.*

References

1. Amir Abboud, Karl Bringmann, Nick Fischer, and Marvin Künnemann. The time complexity of fully sparse matrix multiplication. In David P. Woodruff, editor, *Proceedings of the 2024 ACM-SIAM Symposium on Discrete Algorithms, SODA 2024, Alexandria, VA, USA, January 7-10, 2024*, pages 4670–4703. SIAM, 2024. doi:10.1137/1.9781611977912.167.
2. Amir Abboud and Virginia Vassilevska Williams. Popular conjectures imply strong lower bounds for dynamic problems. In *55th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2014, Philadelphia, PA, USA, October 18-21, 2014*, pages 434–443. IEEE Computer Society, 2014. doi:10.1109/FOCS.2014.53.
3. Amir Abboud, Virginia Vassilevska Williams, and Huacheng Yu. More applications of the polynomial method to algorithm design. In *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015, San Diego, CA, USA, January 4-6, 2015*, pages 218–230. Society for Industrial and Applied Mathematics, 2015. doi:10.1137/1.9781611973730.17.
4. Amihood Amir, Moshe Lewenstein, and Noa Lewenstein. Pattern matching in hypertext. *J. Algorithms*, 35(1):82–99, 2000. doi:10.1006/jagm.1999.1063.
5. Renzo Angles and Claudio Gutierrez. Survey of graph database models. *ACM Comput. Surv.*, 40(1):1:1–1:39, 2008. doi:10.1145/1322432.1322433.
6. Kotaro Aoyama, Yuto Nakashima, Tomohiro I, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Faster online elastic degenerate string matching. In Gonzalo Navarro, David Sankoff, and Binhai Zhu, editors, *Annual Symposium on Combinatorial Pattern Matching, CPM 2018, Qingdao, China, July 2-4, 2018*, volume 105 of *LIPIcs*, pages 9:1–9:10. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPIcs.CPM.2018.9.
7. Rocco Ascone, Giulia Bernardini, Alessio Conte, Massimo Equi, Estéban Gabory, Roberto Grossi, and Nadia Pisanti. Pattern matching with elastic-degenerate strings and elastic-founder graphs. *Algorithms Mol. Biol.*, 21(1):8, 2026. doi:10.1186/s13015-025-00289-3.
8. Jasmijn A. Baaijens, Paola Bonizzoni, Christina Boucher, Gianluca Della Vedova, Yuri Pirola, Raffaella Rizzi, and Jouni Sirén. Computational graph pangenomics: a tutorial on data structures and their applications. *Nat. Comput.*, 21(1):81–108, 2022. doi:10.1007/s11047-022-09882-6.
9. Giulia Bernardini, Pawel Gawrychowski, Nadia Pisanti, Solon P. Pissis, and Giovanna Rosone. Elastic-degenerate string matching via fast matrix multiplication. *SIAM J. Comput.*, 51(3):549–576, 2022. doi:10.1137/20m1368033.
10. Itai Boneh and Shay Golan. Covers in optimal space. In Paola Bonizzoni and Veli Mäkinen, editors, *36th Annual Symposium on Combinatorial Pattern Matching, CPM 2025, Milan, Italy, June 17-19, 2025*, volume 331 of *LIPIcs*, pages 5:1–5:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025. doi:10.4230/LIPIcs.CPM.2025.5.
11. Jan van den Brand. Complexity term balancer. jvdbrand.com/complexity/. Tool to balance complexity terms depending on fast matrix multiplication.
12. Dany Breslauer and Zvi Galil. Real-time streaming string-matching. *ACM Trans. Algorithms*, 10(4):22:1–22:12, 2014. doi:10.1145/2635814.
13. Dany Breslauer, Roberto Grossi, and Filippo Mignosi. Simple real-time constant-space string matching. *Theor. Comput. Sci.*, 483:2–9, 2013. doi:10.1016/j.tcs.2012.11.040.
14. Deepayan Chakrabarti and Christos Faloutsos. Graph mining: Laws, generators, and algorithms. *ACM Comput. Surv.*, 38(1):2, 2006. doi:10.1145/1132952.1132954.
15. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 4th Edition*. MIT Press, 2022. URL: https://mitpress.mit.edu/9780262046305/introduction-to-algorithms/.
16. Maxime Crochemore, Christophe Hancart, and Thierry Lecroq. *Algorithms on strings*. Cambridge University Press, 2007.
17. Maxime Crochemore and Dominique Perrin. Two-way string matching. *J. ACM*, 38(3):651–675, 1991. doi:10.1145/116825.116845.

18 Massimo Equi, Veli Mäkinen, Alexandru I. Tomescu, and Roberto Grossi. On the complexity of string matching for graphs. *ACM Trans. Algorithms*, 19(3):21:1–21:25, 2023. doi:10.1145/3588334.

19 Massimo Equi, Tuukka Norri, Jarno Alanko, Bastien Cazaux, Alexandru I. Tomescu, and Veli Mäkinen. Algorithms and complexity on indexing founder graphs. *Algorithmica*, 85(6):1586–1623, 2023. doi:10.1007/s00453-022-01007-w.

20 Nathan J. Fine and Herbert S. Wilf. Uniqueness theorems for periodic functions. *Proceedings of the American Mathematical Society*, 16(1):109–114, 1965. doi:10.1090/S0002-9939-1965-0174934-9.

21 Pawel Gawrychowski. Optimal pattern matching in LZW compressed strings. *ACM Trans. Algorithms*, 9(3):25:1–25:17, 2013. doi:10.1145/2483699.2483705.

22 Roberto Grossi, Costas S. Iliopoulos, Chang Liu, Nadia Pisanti, Solon P. Pissis, Ahmad Retha, Giovanna Rosone, Fatima Vayani, and Luca Versari. On-line pattern matching on similar texts. In Juha Kärkkäinen, Jakub Radoszewski, and Wojciech Rytter, editors, *28th Annual Symposium on Combinatorial Pattern Matching, CPM 2017, Warsaw, Poland, July 4-6, 2017*, volume 78 of *LIPIcs*, pages 9:1–9:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.CPM.2017.9.

23 Ming Gu, Martin Farach, and Richard Beigel. An efficient algorithm for dynamic text indexing. In Daniel Dominic Sleator, editor, *Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms. 23-25 January 1994, Arlington, Virginia, USA*, pages 697–704. ACM/SIAM, 1994. URL: http://dl.acm.org/citation.cfm?id=314464.314675.

24 Leonidas J. Guibas and Andrew M. Odlyzko. Periods in strings. *J. Comb. Theory A*, 30(1):19–42, 1981. doi:10.1016/0097-3165(81)90038-8.

25 Fred G. Gustavson. Two fast algorithms for sparse matrices: Multiplication and permuted transposition. *ACM Trans. Math. Softw.*, 4(3):250–269, September 1978. doi:10.1145/355791.355796.

26 Costas S. Iliopoulos, Ritu Kundu, and Solon P. Pissis. Efficient pattern matching in elastic-degenerate strings. *Inf. Comput.*, 279:104616, 2021. doi:10.1016/j.ic.2020.104616.

27 Russell Impagliazzo and Ramamohan Paturi. On the complexity of k-SAT. *J. Comput. Syst. Sci.*, 62(2):367–375, 2001. doi:10.1006/JCSS.2000.1727.

28 Donald E. Knuth, James H. Morris Jr., and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(2):323–350, 1977. doi:10.1137/0206024.

29 Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, and Tomasz Walen. Internal pattern matching queries in a text and applications. *SIAM J. Comput.*, 53(5):1524–1577, 2024. doi:10.1137/23m1567618.

30 Venkataramana Kopalli, Kübra Arslan, Noemia Morales-Díaz, Silvia F Zanini, and Agnieszka A Golicz. Toward a standardized framework for pangenome graph evaluation: assessing crop plant pangenome variation graph construction from multiple assemblies. *GigaScience*, 14:giaf121, January 2025. doi:10.1093/gigascience/giaf121.

31 Gad M. Landau and Uzi Vishkin. Efficient string matching with k mismatches. *Theor. Comput. Sci.*, 43:239–249, 1986. doi:10.1016/0304-3975(86)90178-7.

32 Veli Mäkinen, Bastien Cazaux, Massimo Equi, Tuukka Norri, and Alexandru I. Tomescu. Linear time construction of indexable founder block graphs. In Carl Kingsford and Nadia Pisanti, editors, *20th International Workshop on Algorithms in Bioinformatics, WABI 2020, September 7-9, 2020, Pisa, Italy (Virtual Conference)*, volume 172 of *LIPIcs*, pages 7:1–7:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.WABI.2020.7.

33 Solon P. Pissis. Optimal prefix-suffix queries with applications. In Ioana Oriana Bercea and Rasmus Pagh, editors, *2025 Symposium on Simplicity in Algorithms, SOSA 2025, New Orleans, LA, USA, January 13-15, 2025*, pages 166–171. SIAM, 2025. doi:10.1137/1.9781611978315.13.

34 Mikko Rautiainen, Veli Mäkinen, and Tobias Marschall. Bit-parallel sequence-to-graph alignment. *Bioinform.*, 35(19):3599–3607, 2019. doi:10.1093/bioinformatics/btz162.

35 Mikko Rautiainen and Tobias Marschall. Graphaligner: rapid and versatile sequence-to-graph alignment. *Genome Biology*, 21(1):253, 2020. doi:10.1186/s13059-020-02157-2.

36 Liam Roditty and Uri Zwick. On dynamic shortest paths problems. *Algorithmica*, 61(2):389–401, 2011. doi:10.1007/s00453-010-9401-5.

37 Chris Thachuk. Indexing hypertext. *J. Discrete Algorithms*, 18:113–122, 2013. doi:10.1016/j.jda.2012.10.001.

38 Virginia Vassilevska Williams, Yinzhan Xu, Zixuan Xu, and Renfei Zhou. New bounds for matrix multiplication: from alpha to omega. In David P. Woodruff, editor, *Proceedings of the 2024 ACM-SIAM Symposium on Discrete Algorithms, SODA 2024, Alexandria, VA, USA, January 7-10, 2024*, pages 3792–3835. SIAM, 2024. doi:10.1137/1.9781611977912.134.

39 Peter Weiner. Linear pattern matching algorithms. In *14th Annual Symposium on Switching and Automata Theory, Iowa City, Iowa, USA, October 15-17, 1973*, pages 1–11. IEEE Computer Society, 1973. doi:10.1109/SWAT.1973.13.

40 Ryan Williams. A new algorithm for optimal 2-constraint satisfaction and its implications. *Theor. Comput. Sci.*, 348(2-3):357–365, 2005. doi:10.1016/J.TCS.2005.09.023.