On the Complexity of String Matching for Graphs

MASSIMO EQUI, VELI MÄKINEN, and ALEXANDRU I. TOMESCU, University of Helsinki
ROBERTO GROSSI, Università di Pisa

Exact string matching in labeled graphs is the problem of searching paths of a graph $G = (V, E)$ such that the concatenation of their node labels is equal to a given pattern string $P[1..m]$. This basic problem can be found at the heart of more complex operations on variation graphs in computational biology, of query operations in graph databases, and of analysis operations in heterogeneous networks.

We prove a conditional lower bound stating that, for any constant $\epsilon > 0$, an $O(|E|^{1-\epsilon} m)$ time, or an $O(|E| m^{1-\epsilon})$time algorithm for exact string matching in graphs, with node labels and pattern drawn from a binary alphabet, cannot be achieved unless the Strong Exponential Time Hypothesis (SETH) is false. This holds even if restricted to undirected graphs with maximum node degree 2—that is, to *zig-zag matching in bidirectional strings*, or to *deterministic* directed acyclic graphs whose nodes have maximum sum of indegree and outdegree 3. These restricted cases make the lower bound stricter than what can be directly derived from related bounds on regular expression matching (Backurs and Indyk, FOCS'16). In fact, our bounds are tight in the sense that lowering the degree or the alphabet size yields linear time solvable problems.

An interesting corollary is that exact and approximate matching are equally hard (i.e., quadratic time) in graphs under SETH. In comparison, the same problems restricted to strings have linear time vs quadratic time solutions, respectively (approximate pattern matching having also a matching SETH lower bound (Backurs and Indyk, STOC'15)).

CCS Concepts: • **Theory of computation → Problems, reductions and completeness**; **Pattern matching**; **Graph algorithms analysis**; • **Applied computing → Computational genomics;**

Additional Key Words and Phrases: Exact pattern matching, graph query, graph search, labeled graphs, string matching, string search, Strong Exponential Time Hypothesis, heterogeneous networks, variation graphs

ACM Reference format:
Massimo Equi, Veli Mäkinen, Alexandru I. Tomescu, and Roberto Grossi. 2023. On the Complexity of String Matching for Graphs. *ACM Trans. Algor.* 19, 3, Article 21 (April 2023), 25 pages.
https://doi.org/10.1145/3588334

1 INTRODUCTION

String matching is the classical problem of finding the occurrences of a pattern as a substring of a text [36]. As most of today's data is linked, it is natural to investigate string matching not only in

A preliminary version of this article appeared in ICALP 2019 [22].
This work was partially funded by the Academy of Finland under grants 309048 (M. Equi, V. Mäkinen), 322595 (A. I. Tomescu), and 328877 (A. I. Tomescu), and by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement 851093, SAFEBIO).
Authors' addresses: M. Equi, V. Mäkinen, and A. I. Tomescu, Department of Computer Science, University of Helsinki, P. O. Box 68, Pietari Kalmin katu 5, Helsinki, 00014, Finland; emails: {massimo.equi, veli.makinen, alexandru.tomescu}@helsinki.fi; R. Grossi, Dipartimento di Informatica, Universitá di Pisa, Largo B. Pontecorvo, 3, Pisa, 56127, Italy; email: grossi@di.unipi.it.

1549-6325/2023/04-ART21
https://doi.org/10.1145/3588334

text strings but also in labeled graphs. Indeed, large-scale labeled graphs are becoming ubiquitous in several areas, such as graph databases, graph mining, and computational biology. Applications require sophisticated operations on these graphs, and they often rely on primitives that locate paths whose nodes have labels matching a pattern given at query time. The most basic pattern to search in a graph is a string, and in this article we will prove that performing string matching in graphs is computationally challenging, even on very restricted graph classes.

In graph databases, query languages provide the user with the ability to select paths based on the labels of their nodes or edges. In this way, graph databases explicitly lay out the dependencies between the nodes of data, whereas these dependencies are implicit in classical relational databases [7]. Although a standard query language has not been yet universally adopted (as it occurred for SQL in relational databases), popular query languages such as Cypher [26], Gremlin [46], and SPARQL [43] offer the possibility of specifying paths by matching the labels of their nodes.

In graph mining and machine learning for network analysis, heterogeneous networks specify the type of each node [48]. A basic task related to graph kernels [33] and node similarity [16] is to find paths whose label matches a specific pattern. For example, in the DBLP network [53], the nodes for authors can be marked with the letter 'A,' and the nodes for papers can be marked with the letter 'P,' and edges connect authors to their papers. For example, coauthors can be identified by the pattern 'APA' if the two 'A' letters match two different nodes.

In genome research, the very first step of many standard analysis pipelines of high-throughput sequencing data has been to align sequenced fragments of DNA (called *reads*) on a reference genome of a species. Further analysis reveals a set of positions where the sequenced individual differs from the reference genome. After years of such studies, there is now a growing dataset of frequently observed differences between individuals and the reference. A natural representation of this gained knowledge is a *variation graph* in which the reference sequence is represented as a backbone path and variations are encoded as alternative paths [47]. Aligning reads (i.e., string matching) on this labeled graph gives the basis for the new paradigm called *computational pangenomics* [15]. There are already practical tools that use such ideas (e.g., [28]).

The string matching problem that we consider in this article is defined as follows. Given an alphabet Σ of symbols, consider a labeled graph $G = (V, E, L)$, where (V, E) represents a directed or undirected graph and $L : V \to \Sigma$ is a function that defines which symbol from Σ is assigned to each node as label.[1] A node labeled with $\sigma \in \Sigma$ is called a *σ-node*, and an edge whose endpoints are labeled σ_1 and σ_2, respectively, is called a *$\sigma_1\sigma_2$-edge*. If G is a directed graph, we say that G is *deterministic* if, for any two out-neighbors of the same node, their labels are different. In the following, we introduce the acronym *3-DDAG* to indicate a *deterministic* **Directed Acyclic Graph (DAG)** such that the sum of the indegree and outdegree of each node is at most 3.

Given a pattern string $P[1..m]$ over Σ, we say that P has a *match* in G if there is a path $u_1, \ldots, u_m$ in G such that $P = L(u_1) \cdots L(u_m)$ (we also say that P *occurs* in G, and that $u_1, \ldots, u_m$ is an *occurrence* of P).

Problem 1 (String Matching in Labeled Graphs (SMLG)).
input: A labeled graph $G = (V, E, L)$ and a pattern string P, both over alphabet Σ.
output: True *if and only if there is at least one occurrence of P in G.*

1.1 Our Results

We give conditional bounds for the **String Matching in Labeled Graphs (SMLG)** problem using the **Orthogonal Vectors (OV)** hypothesis [52]. The latter states that for any constant $\epsilon > 0$, no

[1]Note that we can also define the node labels as nonempty strings, but it suffices to use single symbols to show that string matching in graphs is challenging.

algorithm can solve in $O(n^{2-\epsilon}\text{poly}(d))$ time the OV problem: given two sets $X, Y \subseteq \{0,1\}^d$ such that $|X| = |Y| = n$ and $d = \omega(\log n)$, decide whether there exist $x \in X$ and $y \in Y$ such that x and y are orthogonal, namely, $x \cdot y = 0$. We observe that it is common practice to use the **Strong Exponential Time Hypothesis (SETH)** [34], but since SETH implies the OV hypothesis [52], it suffices to use the OV hypothesis in the bounds, as they hold also for SETH.

First, we consider the SMLG problem on directed graphs. Their weakest form is a 3-DDAG, for which we prove in Section 3 that subquadratic time for exact string matching cannot be achieved unless the OV hypothesis is false.

Theorem 1.1. *For any constant $\epsilon > 0$, the SMLG problem for labeled deterministic DAGs cannot be solved in either $O(|E|^{1-\epsilon}\, m)$ or $O(|E|\, m^{1-\epsilon})$ time unless the OV hypothesis fails. This holds even if restricted to a binary alphabet, and to DAGs in which the sum of outdegree and indegree of any node is at most 3 (i.e., 3-DDAGs).*

Next, we consider the SMLG problem on undirected graphs and introduce the *zig-zag* pattern matching problem in strings, which models searching a string P along a path of an undirected graph. An exact occurrence of P in a text string is found by scanning the text forward for increasing positions in P; however, a *zig-zag* occurrence of P in a text can be found by partially scanning forward and backward adjacent text positions, as many times as needed (e.g., for an edge $\{u, v\}$ with $L(u) = \mathsf{a}$ and $L(v) = \mathsf{b}$, all patterns of the form $\mathsf{a}, \mathsf{ab}, \mathsf{aba}, \mathsf{abab}, \ldots$ occur starting from u). We prove in Section 4 the following result.

Theorem 1.2. *The conditional lower bound stated in Theorem 1.1 holds even if it is restricted to undirected graphs whose nodes have degree at most 2, where the pattern and the node labels are drawn from a binary alphabet.*

Our results can cover arbitrary graphs in this way. Interpreting the graphs from Theorem 1.2 as directed, we observe that they have nodes with both indegree 2 and outdegree 2. Looking at Theorem 1.1, we observe that it involves directed graphs with both nodes of indegree at most 1 and outdegree 2, and nodes with outdegree at most 1 and indegree 2. Thus, the only uncovered case is that of directed graphs with only nodes of indegree at most 1 or directed graphs with only nodes of outdegree at most 1. For such graphs, observe that their edges can be decomposed into forests of directed trees (arborescences), whose roots may be connected in a directed cycle (at most one cycle per forest). We show in Section 5.1 that the **Knuth-Morris-Pratt (KMP)** algorithm [36] can be easily extended to solve exact string matching for these special directed graphs in linear time, thus completing the picture of the complexity of the SMLG problem.

1.2 History and Implications

The idea of extending the problem of string matching to graphs, as given in SMLG, is not new. If the nodes $u_1, \ldots, u_m$ are required to be distinct (i.e., to be a *simple* path), this problem is NP-hard as it solves the well-known Hamiltonian Path problem, so this requirement is removed for this reason. The SMLG problem was studied more than 25 years ago as a search problem for hypertext by Manber and Wu [38]. The history of key contributions is given in Table 1, where a common feature of the reported bounds is the appearance of the quadratic term $m\,|E|$ (except for some special cases). Specifically, Amir et al. [5, 6] gave the first quadratic time solution for exact string matching in $O(N + m \cdot |E|)$ time, where $N = \sum_{u \in V} |L(u)|$.

In the approximate matching case, allowing errors in the graph makes the problem NP-hard [6], so onward we consider errors only in the pattern. In such case, the quadratic cost of the approximate matching in graphs is asymptotically optimal under SETH since (i) it solves the approximate string matching as a special case, since a graph consisting of just one directed path of $|E| + 1$ nodes

Table 1. State of the Art for SMLG

Year	Authors	Graph	Exact/ Approximate	Time
1992	Manber and Wu [38]	DAG	Approximate$^{(1)}$	$O(m\|E\| + occ \lg\lg m)$
1993	Akutsu [2]	Tree	Exact	$O(N)$
1995	Park and Kim [40]	DAG	Exact$^{(3)}$	$O(N + m\|E\|)$
1997	**Amir et al. [6]**	**General**	**Exact**	$\mathbf{O(N + m\|E\|)}$
1997	Amir et al. [6]	General	Approximate$^{(2)}$	NP-hard
1997	Amir et al. [6]	General	Approximate$^{(1)}$	$O(Nm \lg N + m\|E\|)$
1998	Navarro [39]	General	Approximate$^{(1)}$	$O(Nm + m\|E\|)$
2017	**Rautiainen and Marschall [45]**	**General**	**Approximate$^{(1)}$**	$\mathbf{O(N + m\|E\|)}$
2019	Jain et al. [35]	General binary alphabet	Approximate$^{(2)}$	NP-hard

V, set of nodes; E, set of edges; occ, number of matches for the pattern in the graph; m, pattern length; N, total length of text in all nodes; *(1)*, errors only in the pattern; *(2)*, errors in the graph; *(3)*, matches span only one edge. The two rows highlighted in bold report the best known bounds for exact and approximate string matching, respectively.

and $|E|$ edges is a text string of length $n = |E| + 1$, and (ii) it has been recently proved that the edit distance of two strings of length n cannot be computed in $O(n^{2-\epsilon})$ time, for any constant $\epsilon > 0$, unless SETH is false [10]. This conditional lower bound explains why the $O(m|E|)$ barrier has been difficult to cross in the approximate case. Rautiainen and Marschall [45] and Jain et al. [35] recently gave the best bound for errors in pattern only, $O(N + m \cdot |E|)$ time, same as the exact string matching. The two best results for exact and approximate pattern matching, both taking quadratic time in the worst case, are highlighted in Table 1.

In this scenario and the application domains mentioned at the beginning, our results have a number of implications:

- Although we can explain the complexity of *approximate* string matching in graphs, not much is known about the complexity of *exact* string matching in graphs. The classical exact string matching can be solved in linear time [36], so one could expect the corresponding problem on graphs to be easier than approximate string matching. A lower bound (i.e., NP-hard, as mentioned earlier) exists only in the case when the pattern is restricted to match only simple paths in the graph. Extensions of this type of matching for special graph classes were studied in the work of Limasset et al. [37]. Here we study the general case, where paths can pass through nodes multiple times. Somewhat surprisingly, Theorems 1.1 and 1.2 imply that *exact and approximate pattern matching are equally hard in graphs*, even if they are 3-DDAGs.
- Our results imply that the algorithm for directed graphs by Amir et al. [5, 6] is essentially the best we can hope for asymptotic bounds unless the OV hypothesis is false. This also applies to the case of undirected graphs by the simple transformation so that each edge $\{u, v\}$ is transformed into a pair of arcs (u, v) and (v, u). Note that we need also Theorem 1.2 to explicitly state that this is the best possible also for undirected graphs of maximum degree 2. To complete the picture, we show how to get linear time for the preceding special case of directed graphs where each node has indegree at most 1 or directed graphs whose nodes have outdegree at most 1.
- Our results also explain why it has been difficult to find indexing schemes for fast exact string matching in graphs, with other than best-case or average-case guarantees [27, 49],

except for limited search scenarios [50]. They complement recent findings about *Wheeler graphs* [3, 27, 31]. Wheeler graphs are a class of graphs admitting an index structure that supports linear time exact pattern matching. Gibney and Thankachan [31] show that it is NP-complete to recognize whether a (non-deterministic) DAG is a Wheeler graph. Alanko et al. [3] give a linear time algorithm for recognizing whether a deterministic automaton is a Wheeler graph. They also give an example where the minimum size Wheeler graph is exponentially smaller than an equivalent deterministic automaton. Theorem 1.1 shows that converting an arbitrary deterministic DAG into an equivalent (not necessarily minimum size) Wheeler graph should take at least quadratic time unless the OV hypothesis is false; moreover, later refinement of this result [24] shows that exponential time for the conversion is needed under the OV hypothesis. In particular, the 3-DDAG obtained in the reduction from OV in the proof of Theorem 1.1 is not a Wheeler graph.

- We describe a simple transformation in Section 5.2 so that we can see our 3-DDAG and the pattern P as two **Deterministic Finite Automata (DFAs)** so that our SMLG problem reduces to the emptiness intersection for the string sets recognized by these two DFAs. This highlights a connection between the two problems, and immediately provides a quadratic conditional lower bound using OV for the latter problem. However, this might not be the best that can be obtained for the emptiness intersection problem, as ongoing work attempts to prove a quadratic lower bound, under SETH [51], already when the two DFAs are trees. Nevertheless, our algorithm from Section 5.1 shows that emptiness intersection between a tree and a chain of nodes is solvable in linear time.

Our reductions share some similarities with those for string problems [1, 8–11, 13, 41]. The closest connection is with a conditional hardness of several forms of regular expression matching [9]. We describe these similarities in Section 2, highlighting the main limitations of this reduction scheme. (For the interested reader, we went through the details of such reduction in an early version of this work [21].) Later we explain why our strategy is crucial in achieving stronger results such as covering the case of deterministic directed graphs with bounded degree. This strategy yields a graph of small degree and enables local merging of non-deterministic subgraphs into deterministic counterparts. Such locality feature of our reduction is of cardinal importance, since converting a **Non-Deterministic Finite Automaton (NFA)** into a DFA can take exponential time [44]. Finally, although this reduction works also for undirected graphs of small degree, it does not cover undirected graphs of degree 2. For this case (zig-zag matching in a bidirectional string), we need a more intricate reduction as the underlying graph has less structure.

2 OVERVIEW OF THE REDUCTION AND CONNECTIONS WITH REGULAR EXPRESSIONS

As mentioned in Section 1, our lower bounds have deep connections with previous results on regular expressions matching. We use these connections to conceptually introduce some internal components of our reductions before proceedings to their formal definitions. Additionally, this allows us to point out why a simple modification of an earlier reduction is not sufficient for our purposes.

Backurs and Indyk [9] analyzed which types of regular expressions are hard to match, and one of their lower bounds can be adapted to address the SMLG problem in the case of a *non-deterministic DAG*. The type of regular expressions in question is $|\cdot|$—that is, a composition of two *or* operations. An example of such regular expression is $[(a|b)(b|c)]|[(a|c)b]$. Given a regular expression p of this type and a text t, determining whether or not a substring of t can be generated by p requires quadratic time, unless there exists a subquadratic time algorithm for OV. The reduction adopted

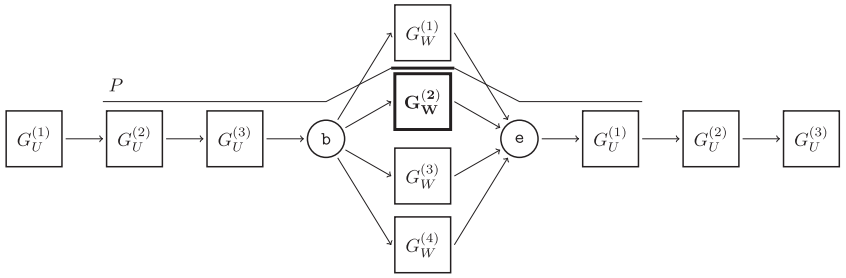

Fig. 1. A sketch of the structure of the reduction for non-deterministic graphs. Pattern P can shift to select a matching subpattern, shown in bold.

to prove such result consists in defining text $t = t_1 2 t_2 2 \ldots 2 t_n$ as the concatenation of all the binary vectors $t_1, \ldots, t_n$ of X, placing the separator character 2 between them. Regular expression $p = G_W^{(1)} \mid G_W^{(2)} \mid \cdots \mid G_W^{(n)}$ is an *or* of n gadgets, one for each vector in the set Y. Moreover, gadget $G_W^{(j)}$ is designed in such a way that it accepts substring t_i if and only if the i-th vector of X and the j-th vector of Y are orthogonal. Hence, it is fairly straightforward to prove that a substring of t is accepted by p if and only if there exists a pair of OVs in X and Y, respectively.

The idea behind this reduction can be modified for the SMLG problem as follows. In the SMLG problem, we need to construct a pattern P and a graph G such that P has a match in G if and only if there is a vector in X orthogonal to a vector in Y. Consider the NFA that accepts the same language as the regular expression p defined earlier, and call b and e its start and accepting states, respectively.

We can enrich such automaton with a universal gadget $G_U^{(j)}$, which accepts any binary vector of length d. We place $n-1$ universal gadgets on each side of the $G_W^{(i)}$'s, to allow P to shift, as shown in Figure 1. Pattern P is again defined as the concatenation of the vectors in X with separator characters. (Again, see the next section for formal definition.) Due to the fact that we placed only $n-1$ universal gadgets on each side, pattern P matches in G if and only if a subpattern of P matches in one of the $G_W^{(j)}$ gadgets, which can happen if and only if there exists a pair of OVs.

Observe that this reduction builds a non-deterministic graph because of the out-neighbors of node b. This non-deterministic feature appears inherent to this type of construction. Our contribution is a heavy restructuring of this reduction, whose two main ideas can be intuitively summarized as follows. First, instead of placing the $G_W^{(j)}$ gadgets on a "column," we place them on a "row." We then place the left universal gadgets on a "row" on top of this one, the right universal gadgets on a "row" below this one, and force the pattern to have a match starting in the top row and ending in the bottom row. See Section 3.3 and Figure 4 presented later in the article. This allows us to restrain the non-deterministic parts of the graph to nodes having only *two* out-neighbors with the same label. Second, we then show how to remove this non-determinism by locally merging the parts of the graph labeled with the same letter while still maintaining the properties of the graph. See Section 3.4 and Figure 6 presented later in the article.

3 DETERMINISTIC DAGS

In this section, we reduce the OV problem to the SMLG problem for the restricted case of 3-DDAGs. In this scenario, 3-DDAGs are the most restricted case, as otherwise the SMLG problem can be solved in linear time (see Section 5.1).

Given an OV instance with sets $X = \{x_1, \ldots, x_n\}$ and $Y = \{y_1, \ldots, y_n\}$ of d-dimensional binary vectors, we show how to build a pattern P and a 3-DDAG G such that P will have a match in G if

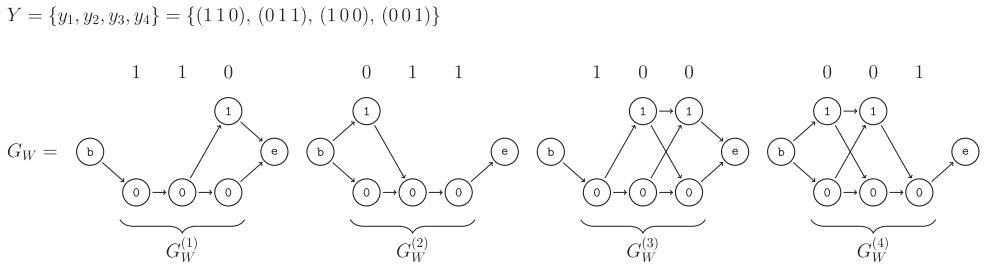

Fig. 2. Gadget G_W.

and only if there exists a vector in X orthogonal to one in Y. We first describe how to build P and how to obtain a directed graph whose nodes are labeled with a constant-sized alphabet. Then we discuss how to turn such a graph into the 3-DDAG G.

3.1 Pattern

Pattern P is over the alphabet $\Sigma = \{\mathsf{b}, \mathsf{e}, \mathtt{0}, \mathtt{1}\}$, has length $|P| = O(nd)$, and can be built in $O(nd)$ time from the first set of vectors $X = \{x_1, \ldots, x_n\}$. Namely, we define

$$P = \mathsf{bb}P_{x_1}\mathsf{e}\,\mathsf{b}P_{x_2}\mathsf{e}\ldots\mathsf{b}P_{x_n}\mathsf{ee},$$

where P_{x_i} is a string of length d that is associated with $x_i \in X$, for $1 \le i \le n$. The h-th symbol of P_{x_i} is either $\mathtt{0}$ or $\mathtt{1}$, for each $h \in \{1, \ldots, d\}$, such that $P_{x_i}[h] = \mathtt{1}$ if and only if $x_i[h] = 1$.[2] We thus view the vectors in X as subpatterns P_{x_i}s which are concatenated by placing separator characters eb. Note that P starts with bb and ends with ee: such strings are found nowhere else in P, marking thus its beginning and its end.

3.2 Graph Gadgets

The gadget implementing the main logic of the reduction is a directed graph $G_W = (V_W, E_W, L_W)$, illustrated in Figure 2. Starting from the second set of vectors Y, set V_W can be seen as n disjoint *groups* of nodes $V_W^{(1)}, V_W^{(2)}, \ldots, V_W^{(n)}$ (plus some extra nodes), where the nodes in $V_W^{(j)}$ are uniquely associated with vector $y_j \in Y$, for $1 \le j \le n$. The corresponding induced subgraph $G_W^{(j)} = (V_W^{(j)}, E_W^{(j)})$ will contain an occurrence of a subpattern P_{x_i} if and only if $x_i \cdot y_j = 0$. We give more details in the following.

The nodes in $V_W^{(j)}$ are defined as follows. For $1 \le h \le d$, we consider entry $y_j[h]$ of vector $y_j \in Y$. If $y_j[h] = 1$, we place just a $\mathtt{0}$-node w_{jh}^0 to indicate that we only accept $P_{x_i}[h] = \mathtt{0}$ for this h coordinate. Instead, if $y_j[h] = 0$, we place both a $\mathtt{0}$-node w_{jh}^0 and a $\mathtt{1}$-node w_{jh}^1 to indicate that the value of $P_{x_i}[h]$ does not matter. The nodes in $V_W^{(j)}$ are preceded by a special begin b-node $b_W^{(j)}$ and succeeded by a special end e-node $e_W^{(j)}$. The overall nodes are thus $V_W = \bigcup_{1 \le j \le n}(V_W^{(j)} \cup \{b_W^{(j)}, e_W^{(j)}\})$, and it holds that $|V_W| = O(nd)$.

As for the edges in $E_W^{(j)}$, they properly connect the nodes inside each group $V_W^{(j)}$. Specifically, node $b_W^{(j)}$ is connected to w_{j1}^0 and, if it exists, to w_{j1}^1. Additionally, we place edges connecting both nodes w_{jd}^0 and w_{jd}^1 (if this exists) to node $e_W^{(j)}$. Moreover, there is an edge for every pair of nodes that are consecutive in terms of h coordinate, for $1 \le h < d$ (e.g., w_{jh}^1 is connected to $w_{j\,h+1}^0$). The overall edges are thus $E_W = \bigcup_{1 \le j \le n} E_W^{(j)}$, where $|E_W| = O(nd)$.

[2]Note that $\mathtt{1}$ is a symbol of Σ, whereas 1 is the truth value in x_i.

In this way, we define the directed graph $G_W = (V_W, E_W, L_W)$, which can be built in $O(nd)$ time from set Y and consists of n connected components $G_W^{(j)}$, one for each vector $y_j \in Y$.

We observe that pattern occurrences in G_W have some useful combinatorial properties. The following lemma is an immediate observation, which follows from the fact that each $G_W^{(j)}$ is acyclic and not connected to any other $G_W^{(j')}$.

Lemma 3.1. *If subpattern* $\mathsf{b}P_{x_i}\mathsf{e}$ *has a match in* G_W*, then the nodes matching* P_{x_i} *share the same* j *coordinate and have distinct and consecutive* h *coordinates.*

The following lemma instead relates the occurrence of a subpattern to the OV problem.

Lemma 3.2. *Subpattern* $\mathsf{b}P_{x_i}\mathsf{e}$ *has a match in* G_W *if and only if there exist* $y_j \in Y$ *such that* $x_i \cdot y_j = 0$.

Proof. Recall that, by construction, $w_{jh}^0 \in V_W^{(j)}$ and $w_{jh}^1 \in V_W^{(j)}$ hold for those h such that $y_j[h] = 0$, whereas $w_{jh}^0 \in V_W^{(j)}$ and $w_{jh}^1 \notin V_W^{(j)}$ hold in case $y_j[h] = 1$. We handle the two implications of the statement individually.

($\Rightarrow$) By Lemma 3.1, we can focus on the d distinct and consecutive nodes of $G_W^{(j)}$ that match P_{x_i}. In particular, we know that each character $P_{x_i}[h]$ is matched by either w_{jh}^0 or w_{jh}^1. Consider vectors $x_i \in X$ and $y_j \in Y$. If $P_{x_i}[h] = 1$ has a match in $G_W^{(j)}$, it means that node w_{jh}^1 exists and hence $y_j[h] = 0$, implying $x_i[h] \cdot y_j[h] = 0$. If $P_{x_i}[h] = \mathtt{0}$, by construction we know that $x_i[h] = 0$ and, no matter whether node w_{jh}^1 exists or not, the pattern will match w_{jh}^0, and it clearly holds that $x_i[h] \cdot y_j[h] = 0$. At this point, we can conclude that $x_i[h] \cdot y_j[h] = 0$ for every $1 \le h \le d$, thus $x_i \cdot y_j = 0$.

($\Leftarrow$) Consider vectors $x_i \in X$ and $y_j \in Y$ that are such that $x_i \cdot y_j = 0$. For $h = 1, 2, \ldots, d$, if $y_j[h] = 0$ then $w_{jh}^0, w_{jh}^1 \in V_W^{(j)}$ and $P_{x_i}[h]$ can match either w_{jh}^0 or w_{jh}^1 in $G_W^{(j)}$. If $y_j[h] = 1$ it must be $x_i[h] = 0$ since $x_i \cdot y_j = 0$, thus $P_{x_i}[h] = \mathtt{0}$ and it can match node w_{jh}^0, which is always present in $G_W^{(j)}$. Finally, characters b and e can match nodes $b_W^{(j)}$ and $e_W^{(j)}$, respectively. All characters of $\mathsf{b}P_{x_i}\mathsf{e}$ have now a matching node and the definition of the edges in E_W allows to visit all such nodes via a matching path starting at $b_W^{(j)}$ and ending at $e_W^{(j)}$. □

In the following, we will also use gadget $G_U = (V_U, E_U, L_U)$, the degenerate case of G_W with $2n-2$ (instead of just n) connected components $G_U^{(j)}$ where, for all $1 \le j \le 2n-2$ and $1 \le h \le d$, we place both a $\mathtt{0}$-node and a $\mathtt{1}$-node: we call these two nodes u_{jh}^0 and u_{jh}^1, respectively, to distinguish them from those in G_W. Moreover, every e-node of this gadget is connected with the next b-node, in terms of the j coordinate (Figure 3). As it can be seen, any subpattern P_{x_i} occurs in G_U, so it can be used as a "jolly" gadget.

3.3 Non-Deterministic Graph

A possible approach is based on suitably combining one instance of gadget G_W and two instances of gadgets G_U, named G_{U1} and G_{U2}. The idea is that when $x_i \cdot y_j = 0$, we want P to occur in G, so that the following three conditions hold:

- *Instance* G_{U1}: P_{x_1} occurs in $G_{U1}^{(n-1+j-(i-1))}$, …, $P_{x_{i-1}}$ occurs in $G_{U1}^{(n-1+j-1)}$.
- *Instance* G_W: P_{x_i} occurs in $G_W^{(j)}$.
- *Instance* G_{U2}: $P_{x_{i+1}}$ occurs in $G_{U2}^{(j)}$, …, P_{x_n} occurs in $G_{U2}^{(j+n-i-1)}$.

However, when $x_i \cdot y_j \neq 0$, we do not want P_{x_i} to occur in $G_W^{(j)}$. We can suitably link the instances G_W, G_{U1}, and G_{U2} so that we get the preceding conditions. We connect the e-nodes in G_{U1}

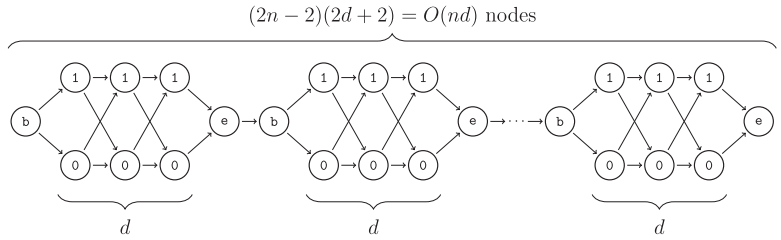

Fig. 3. Gadget G_U.

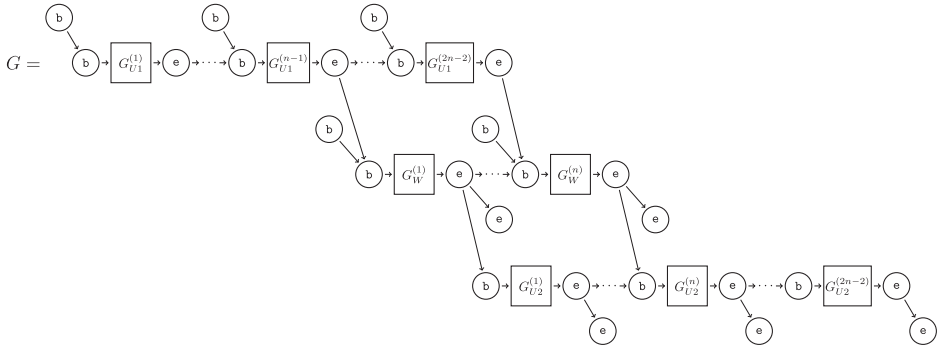

Fig. 4. Non-deterministic graph G.

to b-nodes in G_W and connect the e-nodes in G_W to b-nodes in G_{U2}. Additionally, we place additional starting b-nodes and additional ending e-nodes, to properly match the bb and ee prefix and suffix of P, respectively. More precisely, for every b-node in G_{U1} and G_W, we add a new b-node as an in-neighbor of it, and for every e-node in G_W and G_{U2}, we add a new e-node as an out-neighbor of it. Such construction is depicted in Figure 4.

However, even if G_W, G_{U1}, and G_{U2} are deterministic, their resulting composition is not so, because of the out-neighbors of the e-nodes.[3] In the following, we show how to obtain a deterministic graph by suitably merging G_W with portions of G_U.

3.4 Deterministic Graph

To obtain a *deterministic* DAG, we need to suitably combine one instance of gadget G_W with the two instances G_{U1} and G_{U2} (recall that both G_{U1} and G_{U2} have instances of gadget $G_U^{(j)}$, for all $1 \leq j \leq 2n-2$). Although G_{U2} will be used as is, G_{U1} needs to be partially merged with G_W to obtain determinism. We start building our final graph G from G_W by adding parts of G_{U1} when needed, obtaining a deterministic graph called G_{U1W}, as shown in Figure 5. Consider subgraph $G_W^{(j)}$ and assume that the first position in which the 1-node is lacking is h. We place a partial version of subgraph $G_{U1}^{(j')}$, $j' := n-1+j$, by adding to the graph the nodes and edges of $G_{U1}^{(j')}$ that are located between position $h+1$ and node $e_{U1}^{(j')}$ (included). If $h = d$, we place only node $e_{U1}^{(j')}$. We also place 1-node u_{jh}^1 and connect the 0-node and the 1-node (if any) of $G_W^{(j)}$ in position $h-1$ to it (if $h > 1$), or we connect $b_W^{(j)}$ to it (if $h = 1$). Moreover, we connect node u_{jh}^1 to the first 0- and 1-node of partial $G_{U1}^{(j')}$. If $h = d$, we connect u_{jh}^1 to $e_{U1}^{(j')}$. Then we scan $G_W^{(j)}$ from left to right looking for those positions h', $h \leq h' < d$, such that there is no 1-node in position $h'+1$. We connect the 0-node

[3] An e-node can have two b-nodes as out-neighbors when linking G_{U1} to G_W (see [23]).

$Y = \{y_1, y_2, y_3, y_4\} = \{(1\,1\,0), (0\,1\,1), (1\,0\,0), (0\,0\,1)\}$

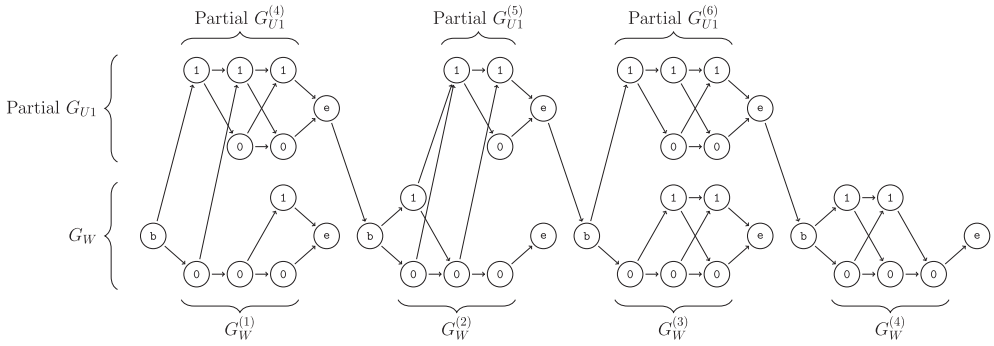

Fig. 5. Graph G_{U1W} after merging G_{U1} (from Figure 3) with G_W (from Figure 2).

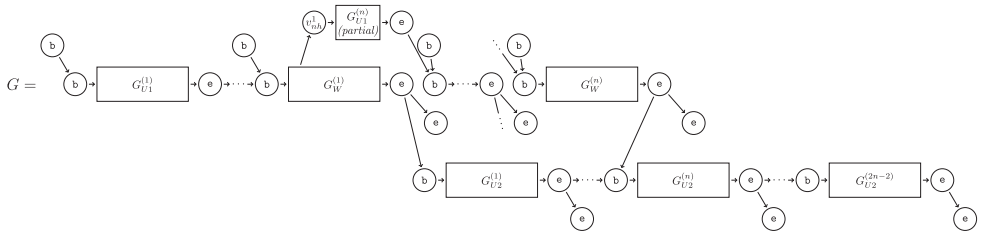

Fig. 6. Final deterministic DAG G.

and the 1-node (if any) of $G_W^{(j)}$ in position h' to the 1-node of $G_{U1}^{(j')}$ in position $h'+1$. Finally, we place edge $(e_{U1}^{(j')}, b_W^{(j+1)})$. To complete the merging task, we apply the preceding modification to all $G_W^{(j)}$, for $1 \le j \le n-1$, and thus obtain gadget G_{U1W}.

At this point, we place gadget G_{U2} and connect G_{U1W} to it by placing edges $(e_W^{(j)}, b_{U2}^{(j)})$, for all $1 \le j \le n$. Additionally, for every b-node of G_{U1W}, we place an additional b-node as in-neighbor. We do the same for every e-node of G_{U2}, placing an e-node as out-neighbor. Adding subgraphs $G_{U1}^{(1)}, \ldots, G_{U1}^{(n-1)}$ with one additional b-node as in-neighbor of their b-nodes, and connecting the e-node of $G_{U1}^{(n-1)}$ to the b-node of $G_W^{(1)}$, completes the transformation into the wanted deterministic DAG, which we call G. Figure 6 gives an overall picture of G.

It is easy to verify that every b- and e-node in G can have no more than two out-neighbors, and in such case, they have different labels. This shows that graph G is deterministic.

The deterministic DAG G has a crucial property which, combined with Lemma 3.1 and Lemma 3.2, is essential to ensure the correctness of our reduction.

Lemma 3.3. *Pattern P has a match in G if and only if a subpattern* $\mathsf{b}P_{x_i}\mathsf{e}$ *of P has a match in the underlying subgraph G_W of G_{U1W}.*

Proof. For the ($\Rightarrow$) implication, because of the directed eb-edges, each distinct subpattern $\mathsf{b}P_{x_i}\mathsf{e}$ matches a path from either a distinct portion of G_{U1W} (or from the $G_{U1}^{(j)}$ subgraphs, $1 \le j \le n-1$, before it) or G_{U2}. Moreover, each occurrence of P must begin with bb and end with ee. String bb can be matched only in G_{U1W} (or in the $G_{U1}^{(j)}$ subgraphs before it), hence the match must start here. However, string ee is found either in G_{U1W} or in G_{U2}. Observe that, by construction, once a match for pattern P is started in G_{U1W} (or in the $G_{U1}^{(j)}$ subgraphs before it), the only way to successfully

conclude it is either by matching ee within G_{U1W}, or by matching also a portion of G_{U2} and then ee. Because of the structure of the graph, in both cases a subpattern $\mathsf{b}P_{x_i}\mathsf{e}$ of P must match one of the subgraphs $G_W^{(j)}$ that are present in G_{U1W}.

The ($\Leftarrow$) implication is trivial. In fact, if $\mathsf{b}P_{x_i}\mathsf{e}$ has a match in one subgraph $G_W^{(j)}$, then by construction we can match $\mathsf{b}P_{x_1}\mathsf{e} \ldots \mathsf{b}P_{x_{i-1}}\mathsf{e}$ possibly in the $G_{U1}^{(j)}$ subgraphs before G_{U1W}, then possibly in the partial $G_{U1}^{(j)}$ subgraphs of G_{U1W}. We can then match $\mathsf{b}P_{x_{i+1}}\mathsf{e} \ldots \mathsf{b}P_{x_n}\mathsf{e}$ in G_{U2} and thus have a full match for P in G. □

We conclude this section by proving the following weaker version of Theorem 1.1. In the next two sections, we show how to obtain the full proof of Theorem 1.1, by transforming G to have maximum sum of indegree and outdegree 3, and how to reduce the alphabet to binary.

Theorem 3.4. *For any constant $\epsilon > 0$, the SMLG problem for a labeled deterministic DAG cannot be solved in either $O(|E|^{1-\epsilon}\, m)$ or $O(|E|\, m^{1-\epsilon})$ time unless the OV hypothesis fails. This holds even if restricted to an alphabet of size* 4.

Proof. First, we argue that the reduction given in this section is correct. Then we analyze its cost and argue how a subquadratic time algorithm for SMLG would contradict the OV hypothesis.

Correctness. We need to ensure that pattern P has a match in G if and only if there exist vectors $x_i \in X$ and $y_j \in Y$ which are orthogonal. This follows from Lemma 3.3, which guarantees that P has a match in G if and only if a subpattern P_{x_i} has a match in G_W, and the fact that, by Lemma 3.2, this holds if and only if $x_i \cdot y_j = 0$.

Cost. As observed during the construction in Sections 3.1 and 3.2, both pattern P and graph G have size $O(nd)$. Indeed, for each one of the n vectors $x_i \in X$, we place in P characters b and e plus d characters that can be either 0 or 1. In graph G, the size of each subgraph is proportional to the dimension d of the vectors, and we place $O(n)$ of them.

Using the OV Hypothesis. The last step is to show that any $O(|E|^{1-\epsilon}\, m)$-time or $O(|E|\, m^{1-\epsilon})$ time algorithm A for SMLG contradicts the OV hypothesis. Given two sets of vectors X and Y, we can perform our reduction obtaining pattern P and graph G in $O(nd)$ time while observing that $|E| = O(nd)$ and $m = O(nd)$. No matter whether A has $O(|E|^{1-\epsilon} m)$ or $O(|E|\, m^{1-\epsilon})$ time complexity, we will end up with an algorithm deciding if there exists a pair of OVs between X and Y in $O(nd \cdot (nd)^{1-\epsilon}) = O(n^{2-\epsilon}\mathrm{poly}(d))$ time, which contradicts the OV hypothesis. □

3.5 Reduced Degree

In this section, we show how to transform the deterministic graph G from the previous section to be a 3-DDAG.

Observe that every node in G can have at most two in-neighbors and two out-neighbors. An emblematic case is that of four nodes, say v, w, v', and w', with edges (v, w), (v, w'),(v', w), and (v', w'). To reduce to 1 the outdegree of v and v', and the indegree of w and w', the idea is to add two dummy nodes $\bar{v}$ and $\bar{w}$ connected by an edge $(\bar{v}, \bar{w})$, then replace the four preceding edges with $(v, \bar{v})$, $(v', \bar{v})$, $(\bar{w}, w)$, and $(\bar{w}, w')$. The dummy nodes can be labeled, for example, with 0, then one can do a symmetric modification in the pattern. One needs to apply such transformations between any two consecutive columns of G.

To be more precise, we need to consider four node configurations. The first three, shown in Figure 7, are slightly simpler than the fourth one, in Figure 8. The final result is achieved by applying these adjustments among (sequences of) consecutive columns of G, observing that these four cases cover all possible configurations in the graph.

Since we always insert a pair of two new 0-nodes, or a b- and an e-node, between prescribed columns in G, then we can analogously modify the pattern to match the new structure of G.

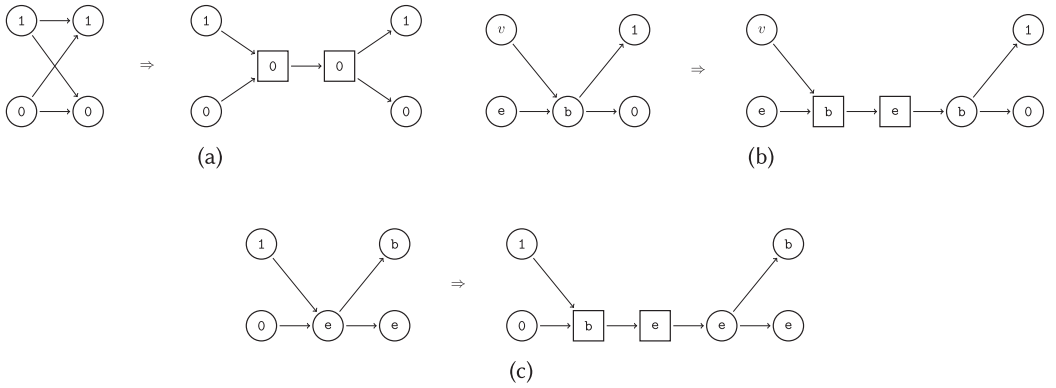

Fig. 7. Adjustments to the graph needed for achieving maximum sum of indegree and outdegree 3 for every node. The squared nodes are the new artificial nodes added to reach this goal. (a) This is the general case that captures the main idea. Notice that one or both 1-nodes may be missing in G, but we apply the transformation nonetheless. (b) A special case of (a). Node v can be either a b- or an e-node. Node v and the 1-node may be missing in G. (c) The other special case of (a). The 1-node, the b-node, or the e-node on the right may be missing in G.

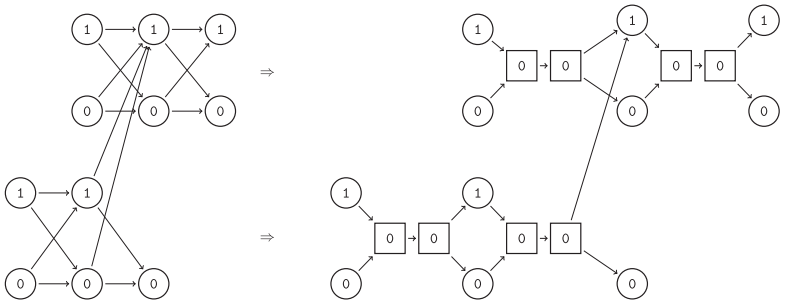

Fig. 8. An example of the special case that occurs in gadget G_{U1W}. Notice that some nodes may be missing in G.

The encoding that we present next to obtain a binary alphabet can be safely applied after reducing the degree of the nodes of G with this technique.

3.6 Binary Alphabet

The size of the alphabet used until this point is 4. One can reduce the alphabet size to binary using the following encoding,

$$\alpha(\mathtt{0}) = \mathtt{0000}, \quad \alpha(\mathtt{1}) = \mathtt{1111}, \quad \alpha(\mathtt{b}) = \mathtt{10}, \quad \alpha(\mathtt{e}) = \mathtt{01},$$

for both the pattern and the graph. Given any string $x = x[1..m]$, we define its binary encoding $\alpha(x) := \alpha(x[1]) \cdots \alpha(x[m])$. In the graph, we replace each σ-node with a path of as many nodes as characters in $\alpha(\sigma)$.

To make this encoding work, we need to additionally make the pattern start with characters ebb (instead of just bb) and end with characters eeb (instead of just ee) to exploit the properties of sequence eb. Moreover, this entails that also in the graph we have to place and connect a new e-node to each b-node used to mark the beginning of a viable match, and in the same manner, we need to add a new b-node after every e-node used to mark the end of a match.

We can now assume that the graph and the pattern have been changed as described in the previous section so that the graph has the maximum sum of indegree and outdegree 3. The goal is to show that there is a bijection from matches before and after applying such encoding and reduction adjustments.

At this point, we apply the α encoding, and nodes with labels of length 2 and 4 will be replaced by chains of nodes labeled by single characters each. Note that in graph G, the only out-neighbors of a node can be 0 and 1, or b and e, respectively, hence this encoding keeps the graph deterministic. We now prove some key properties of the chosen encoding.

Observe that even if we modified the graph to reduce the degree, it still holds that the subgraphs of G where matches of some subpattern can be present are separated by an eb-edge (recall Figure 7(b) and (c)). Thus, the following synchronizing property is useful.

Lemma 3.5. *For any string $x \in \Sigma^+$, its binary encoding $\alpha(x)$ contains* 0110 *if and only if x contains* eb.

Proof. We observe that e and b are encoded by two bits each, whereas 0 and 1 are encoded by four bits each. Hence, 0110 can appear by concatenating the binary encoding of two or three symbols. However, eb occurs in x if and only if it occurs in a substring of length 3 of x. Consequently, it suffices to check the claim by inspection of all the 64 substrings of x of length 3, 000, ..., eee, and their encodings to see that the property holds. □

An immediate consequence of Lemma 3.5 is that the encoding preserves the occurrences. Let $G^{(ex)}$ be the deterministic DAG reduced to have the maximum sum of indegree and outdegree 3, extended with the extra b- and e-nodes, and let $P^{(ex)}$ be the pattern corresponding to this reduced graph, extended with the b and e characters. Let $\alpha(G^{(ex)})$ denote the graph obtained from $G^{(ex)}$ by relabeling its nodes with the binary encoding α of their labels and substituting such nodes that now have labels of length 2 and 4 with undirected paths of length 2 and 4, respectively, whose nodes are labeled with single characters.

Lemma 3.6. *In the reduction, $P^{(ex)}$ has a match in $G^{(ex)}$ if and only if $\alpha(P^{(ex)})$ has a match in $\alpha(G^{(ex)})$.*

Proof. The forward implication is trivial. For the reverse implication, observe that by Lemma 3.5, in any match of $\alpha(P^{(ex)})$ in $\alpha(G^{(ex)})$, the encoding of the string eb in the pattern is aligned with the encoding of the eb-edges in the graph. As such, the encoding of all other characters of the pattern are aligned with the encoding of their corresponding nodes of the graph, and thus $P^{(ex)}$ has a match in $G^{(ex)}$. □

4 UNDIRECTED GRAPHS: ZIG-ZAG MATCHING

In this section, we prove Theorem 1.2. To this end, we need to modify the previous reduction, defining a new alphabet, pattern, and graph. The main ideas will be the same, but since the graph will now be a single undirected path, some key changes will be needed. In Section 4.1, we introduce a reduction in which the alphabet has cardinality 6, and in Section 4.2, we show how to reduce the alphabet to binary.

4.1 Non-Binary Alphabet

The original alphabet $\Sigma = \{\mathsf{b}, \mathsf{e}, \mathsf{0}, \mathsf{1}\}$ is replaced with $\Sigma' = \{\mathsf{b}, \mathsf{e}, \mathsf{A}, \mathsf{B}, \mathsf{s}, \mathsf{t}\}$. Characters 0 and 1 are encoded in the following manner:

$$\mathsf{0} = \mathsf{ABABABA} \qquad \text{and} \qquad \mathsf{1} = \mathsf{ABA}.$$

(a) (b)

Fig. 9. New substructures. (a) The old substructure is replaced by an undirected path that can match either sABAs (which represents 1) by going forward only, or sABABABAs (which represents 0) by going forward, backward, and forward again. (b) An undirected path replacing a 0-node can match only the string sABABABAs.

When such encoding is applied, character s will be used as a separator marking the beginning and the end of the old characters. As an example, the subpattern

$$P_{x_i} = 1\ 0\ 1 \qquad \text{will be encoded as} \qquad P'_{x_i} = \texttt{s ABA s ABABABA s ABA s}.$$

A new pattern P' is built applying this encoding to each one of the subpatterns P_{x_i}, thus obtaining new subpatterns P'_{x_i}. We then concatenate all the subpatterns P'_{x_i} by placing the new character t to separate them, instead of eb. Finally, we place characters bt at the beginning of the new pattern and te at the end. We have the following example.

$$P = \texttt{bb 100 e b 101 ee}$$

$$P' = \texttt{b t s ABA s ABABABA s ABABABA s t s ABA s ABABABA s ABA s t e}$$

Note that for each subpattern, we are introducing a constant number of new characters, hence the size of the entire pattern P' still is $O(nd)$.

An analogous encoding will be applied to the graph. The strategy is to encode G_W in an undirected path by concatenating subpaths representing each $G_W^{(j)}$, one after another.

The positions h in which both a 0- and a 1-node are present in $G_W^{(j)}$ are replaced by a path that can be matched both by 0 = ABABABA and 1 = ABA. Positions h with only a 0-node and no 1-node are encoded instead with a path that can be matched only by 0 = ABABABA (Figure 9). We use s-nodes to separate these paths. We denote by $LG_W^{(j)}$ (*Linear* $G_W^{(j)}$) this linearized version of $G_W^{(j)}$. Moreover, given subgraph $G_W^{(j)}$, two new t-nodes will mark the beginning and the ending of its encoding. Figure 10 illustrates this transformation for $G_W^{(j)}$.

In a similar manner, G_U is also encoded as a path. We do not need to encode all its $2n - 2$ subgraphs: since the matching path can go through nodes more than once, we only need to encode one of these subgraphs, in the same manner as done for $G_W^{(j)}$. Let LG_U be the linearized version of only one of the "jolly" gadgets that were composing the original G_U.

Then, for each $1 \leq j \leq n$, we build structure $LG^{(j)}$ by placing t-nodes, LG_U instances, $LG_W^{(j)}$, a b-node on the left, and an e-node on the right, as in Figure 11. In such structure, the b-node and the e-node delimit the beginning and the end of a viable match for a pattern. The t-nodes are separating the LG_U structures from $LG_W^{(j)}$, and in general, they are marking the beginning and the end of a match for a subpattern P'_{x_i}. The idea behind $LG^{(j)}$ is that a match of P can traverse LG_U from the beginning to the end, backward and forward as many times as needed, before starting a match of some subpattern P'_{x_i} inside $LG_W^{(j)}$. Notice also that this allows only subpatterns on even

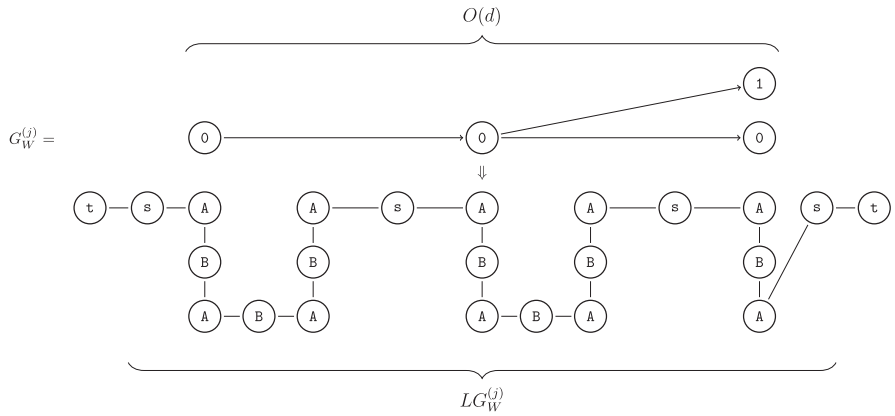

Fig. 10. A subgraph $G_W^{(j)}$ is converted into a linear structure $LG_W^{(j)}$ using s as the separator.

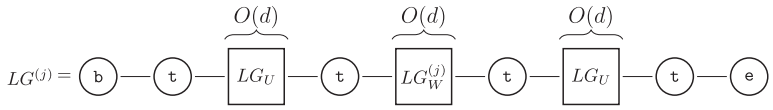

Fig. 11. The $LG_W^{(j)}$ structure surrounded by two instances of LG_U. The t-nodes establish the beginning and the end of a match for a subpattern $tP'_{x_i}t$ while the b- and e-nodes are the starting and ending point for a match of the whole pattern P'.

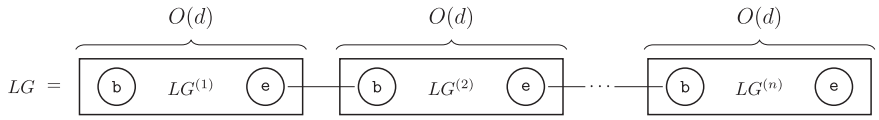

Fig. 12. The final graph LG.

positions i to match inside $LG_W^{(j)}$. We will address this minor issue at the end see the paragraph following the proof of Lemma 4.3.

To construct the final graph LG, we concatenate all $LG^{(1)}, LG^{(2)}, \ldots, LG^{(n)}$ into a single undirected path. Figure 12 gives a picture of the end result.

No issues arise regarding the size of the graph, since we are replacing every 0-node, or every pair of a 0-node and a 1-node, with a constant number of new nodes. By construction, the two gadgets LG_U and $LG_W^{(j)}$ both have size $O(d)$, since for each one of the d entries of a vector we place one of the two possible encodings. In LG, there are n instances of $LG_W^{(j)}$, each one surrounded by two LG_U instances. Hence, the total size of the graph remains $O(nd)$.

To prove the correctness of the reduction, we will show some properties on LG by introducing the following lemmas. We use $t_l LG_W^{(j)} t_r$ to refer to $LG_W^{(j)}$ extended with the t-nodes on its left and on its right. When referring to the k-th s-character in P'_{x_i}, we mean the k-th s-character found scanning P'_{x_i} from left to right; in the same manner, we refer to the k-th s-node in $LG_W^{(j)}$.

Lemma 4.1. *If subpattern $tP'_{x_i}t$ has a match in $t_l LG_W^{(j)} t_r$ starting at t_l and ending at t_r, then the k-th s-character in P'_{x_i} matches the k-th s-node in $LG_W^{(j)}$, for all $1 \leq k \leq d+1$.*

Proof. First we prove that all the s-nodes in $t_l LG_W^{(j)} t_r$ are matched exactly once by $\mathtt{t}P'_{x_i}\mathtt{t}$. By construction, subpattern P'_{x_i} has $d+1$ s-characters, and $LG_W^{(j)}$ has $d+1$ s-nodes. Since we are working on a chain of nodes and the match is starting at t_l and ending at t_r, all the nodes between t_l and t_r have to be matched at least once by P'_{x_i}. Assume by contradiction that one such s-node is matched more than once. Subpattern P'_{x_i} is left with strictly less than d s-characters available for matching the other d s-nodes, and we reach a contradiction. Now we can prove the statement of the lemma by induction on k—that is, the index of the s-characters and s-nodes. Let $\mathtt{s}_k^{(P'_{x_i})}$ denote the k-th s-character in P'_{x_i}, and let $s_k^{(LG_W^{(j)})}$ denote the k-th s-node in $LG_W^{(j)}$.

Base Case $k = 1$. The match starts at t_l, hence the only node that $\mathtt{s}_1^{(P'_{x_i})}$ can match is the first s-node to the right on t_l—that is, $s_1^{(LG_W^{(j)})}$.

Inductive Case $k > 1$. The inductive hypothesis tells us that all the nodes up to $s_k^{(LG_W^{(j)})}$ have been matched by consecutive s-characters of P'_{x_i} up to $\mathtt{s}_k^{(P'_{x_i})}$. We have to prove the statement for $k+1$. Starting from node $s_k^{(LG_W^{(j)})}$, the next s-nodes that can be matched by $\mathtt{s}_{k+1}^{(P'_{x_i})}$ are $s_{k-1}^{(LG_W^{(j)})}$ and $s_{k+1}^{(LG_W^{(j)})}$. Character $\mathtt{s}_{k+1}^{(P'_{x_i})}$ cannot match node $s_{k-1}^{(LG_W^{(j)})}$ since it has already been matched by $s_{k-1}^{(P'_{x_i})}$ and, as argued earlier, every s-node can be matched only once. Thus, $\mathtt{s}_{k+1}^{(P'_{x_i})}$ has to match $s_{k+1}^{(LG_W^{(j)})}$. □

Lemma 4.2. *Subpattern* $\mathtt{t}P'_{x_i}\mathtt{t}$ *has a match in* $t_l LG_W^{(j)} t_r$ *starting at* t_l *and ending at* t_r *if and only if there exist* $y_j \in Y$ *such that* $x_i \cdot y_j = 0$.

Proof. This property has already been proved for gadget G_W in Lemma 3.2, thus what we are left to prove is that $LG_W^{(j)}$ behaves the same as the subgadget $G_W^{(j)}$. First recall that in the construction of $LG_W^{(j)}$, we placed an encoded 1 if in $G_W^{(j)}$ we had both a 0-node and a 1-node in the same position, whereas we placed an encoded 0 if we had only a 0-node. Lemma 4.1 guarantees that the encoding in P' of a single character of P is aligned with the encoding in $LG_W^{(j)}$ of a single node of G_W, preventing (the encoding of) a character of P from matching (the encoding of) multiple nodes of G_W and vice versa. By construction, 1 = ABA can match the encoding of a 1-node while it fails to match the encoding of the 0-nodes, since their encoding involves too many characters. However, 0 = ABABABA can match an encoded 0-node with a natural alignment, but it can also match the encoding of a 1-node by scanning it forward, backward, and forward again. Therefore, the logic behind $LG_W^{(j)}$ safely implements the one of $G_W^{(j)}$, and from this point onward, one can follow the same reasoning as in Lemma 3.2 to complete the proof. □

The main difference with the original proof resides in assuming that a match for P'_{x_i} starts at t_l and ends at t_r. This feature is crucial for the correctness of the reduction and can be safely exploited since, as shown in the following, the b- and e-nodes guarantee that in case of a match for P' we will cross the $LG_W^{(j)}$ gadget from left to right at least once.

Lemma 4.3. *Pattern* P' *has a match in* LG *if and only if there exist* i *and* j *such that* i *is even and subpattern* $\mathtt{t}P'_{x_i}\mathtt{t}$ *has a match in* $t_l LG_W^{(j)} t_r$ *starting at* t_l *and ending at* t_r.

Proof. For the ($\Rightarrow$) implication, first observe that the b- and e-nodes in LG are forcing a direction to follow. Let $LG_{Ul}^{(j)}$ and $LG_{Ur}^{(j)}$ be the LG_U gadgets to the left and to the right of $LG_W^{(j)}$, respectively. Since pattern P' starts with a b and ends with an e, a match can only start at the b-node on the left of $LG_{Ul}^{(j)}$ and end at the e-node on the right of $LG_{Ur}^{(j)}$, for some j. Hence, $LG_W^{(j)}$

needs to be crossed by a match from left to right at least once. Thus, there must exist a subpattern $\mathtt{t}P'_{x_i}\mathtt{t}$ that has a match starting at t_l and ending at t_r. For such a pattern, Lemma 4.2 applies. Moreover, because of our construction, only a subpattern on even position can achieve such a match.

The ($\Leftarrow$) implication is immediate since given a subpattern $\mathtt{t}P'_{x_i}\mathtt{t}$ that has a match in $t_l LG_U^{(j)} t_r$ one can match $\mathtt{bt}P'_{x_1}\mathtt{t}\ldots\mathtt{t}P'_{x_{i-1}}\mathtt{t}$ in $LG_{Ul}^{(j)}$ and $\mathtt{t}P'_{x_{i+1}}\mathtt{t}\ldots\mathtt{t}P'_{x_n}\mathtt{te}$ in $LG_{Ur}^{(j)}$ and have a full match for P' in LG. □

Since Lemma 4.3 gives us a property that holds only if a subpattern is in an even position, we need to tweak pattern P' to make the reduction work. Indeed, we define two patterns. The first pattern $P'^{(1)}$ is P' itself; the second pattern $P'^{(2)}$ is obtained by swapping the subpatterns P'_{x_i} on odd position with the next subpatterns $P'_{x_{i+1}}$ on even position, for every $i = 1, 3, \ldots$. For example, if n is even, we will have the following.

$$P'^{(1)} = \mathtt{bt}\ P'_{x_1}\ \mathtt{t}\ P'_{x_2}\ \mathtt{t}\ P'_{x_3}\ \mathtt{t}\ P'_{x_4}\ \mathtt{t}\ \ldots\ \mathtt{t}\ P'_{x_{n-1}}\ \mathtt{t}\ P'_{x_n}\ \mathtt{te} = P'$$
$$P'^{(2)} = \mathtt{bt}\ P'_{x_2}\ \mathtt{t}\ P'_{x_1}\ \mathtt{t}\ P'_{x_4}\ \mathtt{t}\ P'_{x_3}\ \mathtt{t}\ \ldots\ \mathtt{t}\ P'_{x_n}\ \mathtt{t}\ P'_{x_{n-1}}\ \mathtt{te}$$

While $P'^{(1)}$ checks the even positions of P', $P'^{(2)}$ checks the odd ones. If n is even, then neither $P'^{(1)}$ nor $P'^{(2)}$ would be able to have a match in LG, since after matching an even number of subpatterns it is not possible to match any e-node. In such case, we can simply add a dummy subpattern $\bar{P} = \mathtt{s}\,\mathtt{ABA}\,\mathtt{s}\,\mathtt{ABA}\,\mathtt{s}\ldots\mathtt{s}\,\mathtt{ABA}\,\mathtt{s}$ (with d repetitions of ABA) at the end of P as it were its last subpattern so that the number of subpatterns becomes odd. Indeed, observe that $\bar{P}$ corresponds to vector $\bar{x} = (11\ldots1)$, which has null product only with vector $\bar{y} = (00\ldots0)$. Hence, if $\bar{y} \notin Y$, then $\bar{P}$ does not have a match in any $LG^{(j)}$, whereas if $\bar{y} \in Y$, every subpattern P'_{x_i} has a match in the $LG^{(j)}$ built on top of $\bar{y}$. This means that $\bar{P}$ does not disrupt our reduction.[4]

Now we are ready to present the end result.

Lemma 4.4. *Either $P'^{(1)}$ or $P'^{(2)}$ has a match in LG if and only if there exist vectors $x_i \in X$ and $y_j \in Y$ which are orthogonal.*

Proof. For ($\Rightarrow$), we assume that either $P'^{(1)}$ or $P'^{(2)}$ have a match in LG. By Lemma 4.3, this means that there exists a subpattern $P'^{(q)}_{x_i}$, $q \in \{1, 2\}$ that has a match in $LG_W^{(j)}$, for some j. Lemma 4.2 then ensures that $x_i \cdot y_j = 0$, thus x_i and y_j are orthogonal. For the other implication ($\Leftarrow$), we assume that there exist two OVs $x_i \in X$ and $y_j \in Y$. Thanks to Lemma 4.2, we find a subpattern P'_{x_i} matching $LG_W^{(j)}$. By construction, P'_{x_i} has to be in an even position either in $P'^{(1)}$ or in $P'^{(2)}$. By Lemma 4.3, this means that either $P'^{(1)}$ or $P'^{(2)}$ has a match in LG. □

Theorem 1.2 follows directly from the correctness of these constructions, except for the alphabet size reduction to binary, which we cover in the next section.

[4]An alternative strategy is to use only one pattern P'' instead of two, defined as

$$P'' = \mathtt{bt}\ \bar{P}\ \mathtt{t}\ P'_{x_1}\ \mathtt{t}\ \bar{P}\ \mathtt{t}\ P'_{x_2}\ \mathtt{t}\ \bar{P}\ \ldots\ \mathtt{t}\ \bar{P}\ \mathtt{t}\ P'_{x_n}\ \mathtt{t}\ \bar{P}\ \mathtt{te}.$$

The "dummy" subpatterns $\bar{P}$ encode a 1 in every position and guarantee that we always have an odd number of subpatterns in P''. Moreover, every actual subpattern P'_{x_i} has a chance to be matched in $LG_W^{(j)}$, for some j, since every such subpattern occurs in an even position.

4.2 Binary Alphabet

In this section, we explain how to reduce the alphabet from the reduction in Section 4.1 to be binary. For this purpose, we apply the following encoding α to the characters:

$$\alpha(\mathtt{A}) = \mathtt{A}, \quad \alpha(\mathtt{B}) = \mathtt{B}, \quad \alpha(\mathtt{s}) = \mathtt{AAA}, \quad \alpha(\mathtt{t}) = \mathtt{BBB}, \quad \alpha(\mathtt{b}) = \alpha(\mathtt{e}) = \mathtt{ABBAAB}.$$

Denote by $\alpha(P')$ and $\alpha(LG)$ the encoded pattern and graph, respectively. Note that when applying the encoding to LG, we replace each σ-node with a sequence of nodes labeled with the characters of the encoding of σ. Thus, we maintain the property that the label of each node is a single character. To prove correctness, it suffices to prove the following two lemmas.

Lemma 4.5. *If P' has a match in LG, then $\alpha(P')$ has a match in $\alpha(LG)$.*

Proof. Since the encoding replaces single symbols with multiple symbols, the difficulties arise when a match of P' in LG performs a change of direction. To understand how to handle this issue, let us follow a match of P' in LG from left to right. As long as such match has no zig-zags (i.e., it does not change direction in LG), then it trivially holds that we can construct a match of $\alpha(P')$ in $\alpha(LG)$. Suppose now that a change of direction happens. We first match node v, followed by w, followed by v again (i.e., it changes direction at w).

If w is an old A- or B-node, then the encoding did not change w and $\alpha(P')$ can still match w. Observe also that w cannot be a b- or an e-node, by construction. The remaining case is when w is an s- or a t-node. If w is a t-node, then v cannot be a b-node, because sequence b t b never occurs in the pattern. Note however that the encodings of s- and t consist of three identical characters. Thus, the match of $\alpha(P')$ in $\alpha(LG)$ can be made to use the border node of the encoding of w (the one adjacent to the encoding of v), then the middle node, and then the same border node again (i.e., to change direction at the middle node of the encoding of w).

At this point the match will continue in the reverse direction. Notice that the encodings of A, B, s, and t are all palindrome strings. Hence, all the previous reasoning for matching following the forward direction also applies for the reverse direction. □

Lemma 4.6. *If $\alpha(P')$ has a match in $\alpha(LG)$, then P' has a match in LG.*

Proof. To simplify notation, in this proof we treat 1 = ABA and 0 = ABABABA as single characters of P'. To prove the lemma, it suffices to prove that in any match of $\alpha(P')$ in $\alpha(LG)$, the encodings of b, e, s, t, and 1 = ABA and 0 = ABABABA in the encoded pattern are precisely aligned with encoded b-, e-, s-, and t-nodes, and with nodes encoding 1 and 0, respectively, in the encoded graph. When saying that such encodings are aligned, we mean that the first and last characters of an encoding $\alpha(\sigma)$ in the pattern must match either the first or the last node (irrespectively of which) of an encoding of the same character σ in $\alpha(LG)$. For example, when character $\alpha(\mathtt{s}) = \mathtt{AAA}$ separates 0- or 1-nodes, it can be properly aligned to the graph as shown in Figures 13 and 14.

We organize the proof of this lemma in two parts, proving separately two claims. The goal is to show that the encoding of the characters preserves the properties already proven for the non-binary case.

Claim 1. *Encodings $\alpha(\mathtt{b})$ and $\alpha(\mathtt{e})$ in the pattern can only be exactly aligned with encodings $\alpha(\mathtt{b})$ and $\alpha(\mathtt{e})$ in the graph, respectively, from left to right.*

Proof. First, note that the substrings $\alpha(\mathtt{b}) = \alpha(\mathtt{e})$ of the encoded pattern cannot have a match in the encoded graph starting anywhere else than in the encoding of a b- or e-node. Indeed, $\alpha(\mathtt{b})$ contains BB, which appears in the graph only in the encoding of a t-node (apart from the encoding of b- or e-nodes). Suppose for a contradiction that a match of $\alpha(\mathtt{b})$ matches two B characters from an encoding of a t-node in the graph; in particular, $\alpha(\mathtt{b})$ starts with ABB, and this prefix of

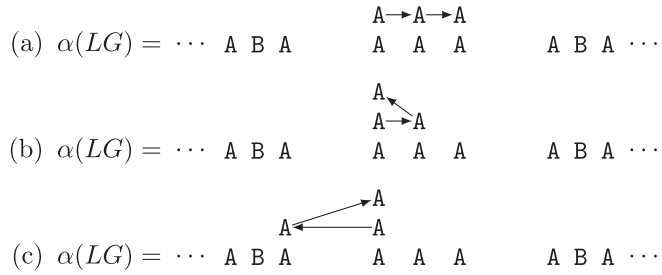

Fig. 13. The three possible alignments for $\alpha(\mathtt{s}) = \mathtt{AAA}$ that can be obtained starting in the first position of $\alpha(\mathtt{s})$ in the graph.

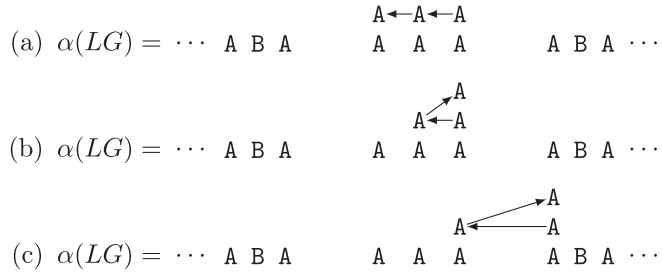

Fig. 14. The three possible alignments for $\alpha(\mathtt{s}) = \mathtt{AAA}$ that can be obtained starting in the last position of $\alpha(\mathtt{s})$ in the graph.

$\alpha(\mathtt{b})$ must end at the middle B-node of $\alpha(\mathtt{t})$. However, the character following ABB in $\alpha(\mathtt{b})$ is A, whereas any neighbor of the middle B-node of $\alpha(\mathtt{t})$ is labeled with B, a contradiction.

We now prove that $\alpha(\mathtt{b})$ in the pattern can only be exactly aligned with $\alpha(\mathtt{b})$ in the graph, from left to right. Suppose for a contradiction that this is not the case. We draw here below the configuration in LG and $\alpha(LG)$ at the border between $LG^{(j)}$ and $LG^{(j+1)}$ (the beginning and end of LG are the same, but missing te, and bt, respectively).

LG :	···	t	e	b	t	···
$\alpha(LG)$:	···	BBB	ABBAAB	ABBAAB	BBB	···

Following the contradiction reasoning, there must be a way of aligning $\alpha(\mathtt{b})$ to the graph other than using an exact match from left to right with $\alpha(\mathtt{b})$ in the graph. Indeed, we can analyze the alternative ways of aligning $\alpha(\mathtt{b})$ to the graph by considering the possible starting position for a potential alignment. Since $\alpha(\mathtt{b})$ starts with AB, a potential alignment in the graph might start in the second or third A character of $\alpha(\mathtt{b})$, or in the first, second, or third A character of $\alpha(\mathtt{e})$. In Figure 15, we analyze all of these five cases, concluding that at some point they will all fail. Completely symmetrically, we can argue that $\alpha(\mathtt{e})$ in the pattern can only be exactly aligned with $\alpha(\mathtt{e})$ in the graph, from left to right. □

Claim 2. *Encodings of* t, s, *and of the substrings* 1 = ABA *and* 0 = ABABABA *in the encoded pattern are aligned with the corresponding encoded nodes in* $\alpha(LG)$.

Proof. We prove this claim by induction on the position of the current character in P'.

Substrings $\alpha(\mathtt{s})$ and $\alpha(\mathtt{t})$ of the encoded pattern are allowed to change direction inside the encoded graph, but they must still start and end at an extremity of the occurrences of $\alpha(\mathtt{s})$ and $\alpha(\mathtt{t})$ in the encoded graph, respectively.

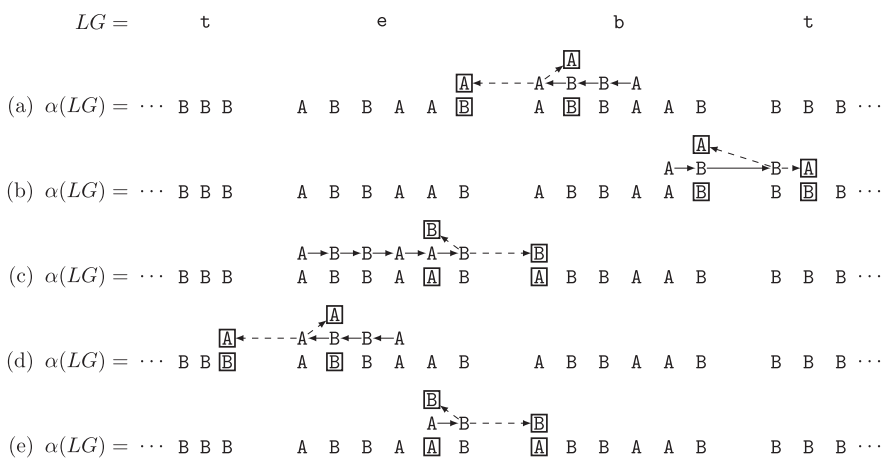

Fig. 15. The five potential alignments for string $\alpha(\mathtt{b}) = \mathtt{ABBAAB}$ that do not start in the first position of $\alpha(\mathtt{b})$ in the graph. The squares around the characters highlight the mismatches. Cases (a) and (b) take into account potential alignments starting at the fourth and fifth position of $\alpha(\mathtt{b})$ in the graph, respectively. Cases (c), (d), and (e) depict potential alignments starting at the first, fourth, or fifth position of $\alpha(\mathtt{e})$ in the graph, respectively. In case (c), the mismatch occurs on the first B character of $\alpha(\mathtt{t})$, which we know always follows $\alpha(\mathtt{b})$.

Suppose that the prefix $\alpha(\mathtt{b})$ of $\alpha(P')$ matches $\alpha(\mathtt{b})$ in the substructure $\alpha(LG^{(j)})$ of $\alpha(LG)$. As the base case, observe that the characters in $\alpha(P')$ following $\alpha(\mathtt{b})$ are $\alpha(\mathtt{t})\alpha(\mathtt{s})$, followed by $\mathtt{1} = \mathtt{ABA}$ or $\mathtt{0} = \mathtt{ABABABA}$. It can be easily checked that they must match from left to right those nodes that follow $\alpha(\mathtt{b})$ in the encoded graph.

For the inductive case, suppose first that the current character of P' is $\mathtt{1} = \mathtt{ABA}$ (or $\mathtt{0} = \mathtt{ABABABA}$). By construction, the character of P' preceding it can only be s, and by induction, we have that $\alpha(\mathtt{s}) = \mathtt{AAA}$ is aligned with the nodes encoding s in the graph. The match of $\alpha(P')$ cannot go back using A-nodes of $\alpha(\mathtt{s})$ because it would not have a B-node to continue the match. Thus, it must use the nodes of the encoded graph corresponding to $\mathtt{1} = \mathtt{ABA}$ (or $\mathtt{0} = \mathtt{ABABABA}$). Moreover, it cannot go on using A-nodes from the next occurrence of $\alpha(\mathtt{s})$ in the graph, because they are all A-nodes. Therefore, this proves that if the current character of P' is $\mathtt{1} = \mathtt{ABA}$ or $\mathtt{0} = \mathtt{ABABABA}$, then it is aligned with the corresponding nodes in the encoded graph.

Suppose now that the current character of P' is s. The character of P' preceding it can be t, $\mathtt{1} = \mathtt{ABA}$, or $\mathtt{0} = \mathtt{ABABABA}$. In case the preceding character is t, the match of $\alpha(P')$ in the encoded graph cannot go back to using nodes encoding t because they are all B-nodes. Suppose thus that the preceding character is $\mathtt{1} = \mathtt{ABA}$ or $\mathtt{0} = \mathtt{ABABABA}$ and the encoding of s in the pattern goes back to use nodes from the encoding of such preceding character. This means it can only match the first A-node of $\alpha(\mathtt{s})$, then go back to the A node of the encoding of this previous character, and then use the same first A-node of $\alpha(\mathtt{s})$. Notice that this is allowed by our notion of alignment.

The last remaining case is when that the current character of P' is t. Since this t occurrence is not the first one (which was handled in the base case), the character preceding it in P' is always s, and recall that $\alpha(\mathtt{s}) = \mathtt{AAA}$. Also in this case, the encoding $\alpha(\mathtt{t}) = \mathtt{BBB}$ cannot go back and use such A-nodes of $\alpha(\mathtt{s})$, thus it must align to the encoding of a t-node. □

Claims 1 and 2 presented previously complete the proof of this lemma since they allow us to apply the same reasoning of the non-binary case. □

5 ADDITIONAL RESULTS

5.1 A Linear Time Algorithm for Almost Trees

Directed pseudo forests are directed graphs whose nodes have outdegree at most 1, and their transpose are graphs whose nodes have indegree at most 1. Both of these types of graphs are structures lying between our conditional hardness results and the linear time solvable string matching case. Such structures are forests of directed trees whose roots may be connected in a directed cycle (at most one cycle per forest).

Exact string matching in a tree whose edges are directed from root to leaves (graphs whose nodes have indegree at most 1) can be solved in linear time. One such algorithm [2] works on constant alphabet, but there is a folklore alphabet-independent solution through a simple variation of the KMP algorithm [36]: recall that after linear time preprocessing of the pattern $P[1..m]$, KMP scans through the text string T, updating index i in the pattern in amortized constant time to find the longest prefix $P[1..i]$ that matches suffix $T[j-i+1..j]$ of the current position j in the text. One can simulate this algorithm on a tree by just storing the current value of index i at each node before branching.

One can reduce our special case to the tree case as follows. Cut the cycle at any edge (v, w) to form a tree rooted at w. Read the cycle from v backward (possibly many times) to form a string $S[1..m]$, where m is the pattern length. Create a path matching the reverse of $S[1..m]$ and connect this path to the root w forming a new tree. Pattern matching on this tree takes linear time [2].

To see that the reduction works correctly, consider root r of some tree hanging from the cycle. Let S^r be the infinite string formed by reading the cycle starting at r backward. For searching a pattern of length m spanning r, it is sufficient to add a path spelling reverse of $S^r[1..m]$ on top of r and use the linear time solution for trees [2]. Furthermore, observe that the infinite strings S^r for all roots r along the cycle overlap, so it is sufficient to linearize the cycle until each root is preceded by a length m part of the reverse of their infinite string S^r. To cover also matches inside the cycle, one can consider similarly any node on a cycle as a root. The reduction covers these cases.

Finally, the symmetric case of a cycle containing roots of upward directed trees (graphs whose nodes have outdegree at most 1) can be reduced to the symmetric case by reversing all edges and the pattern.

5.2 Language Intersection of Two DFAs

We can show a connection between SMLG and the emptiness intersection problem by turning a deterministic DAG and a pattern into two DFAs. We do so by modifying the graph of our reduction so that we also obtain a reduction from OV to the emptiness intersection.

Let G be the 3-DDAG obtained in the reduction of Section 3. We can obtain a DFA D_1 from G as follows. First, the nodes in G become the states of D_1, and each arc (u, v) in G gives a transition from state u to state v in D_1 with symbol $L(v)$. Also let S be the states in D_1 that correspond to b-nodes in G with zero indegree. We add $O(|S|)$ states to D_1 forming a tree whose root becomes the initial state of D_1, and the leaves of this tree have transition to the states in S with symbol b. Each transition from each of these new states to its left child is labeled with L and to its right child is labeled with R.

The other DFA D_2 is obtained from P as follows. We employ the same tree with $|S|$ leaves as earlier, except the transitions from these leaves with b go the same state: from this state, we have a simple chain of states that spells P. We can observe that P occurs in G if and only if the languages of D_1 and D_2 have a nonempty intersection, as this amounts to find an occurrence of P starting from one of the b-nodes in G corresponding to a state in S.

6 DISCUSSION

The lower bounds that we presented for directed deterministic graphs are tight with regard to the structure of the graph, in the sense that lowering the degree or the alphabet size makes the problem solvable in subquadratic time. Lowering the degree from 3 makes the problem fall into the almost-tree category that we dealt with in Section 5.1. Lowering the alphabet size to unary means that the graph can only consist of a set of paths or cycles. If there is a cycle in the graph, the pattern always matches, and otherwise one can easily check in linear time if there is a long enough path for the pattern to match. Similar trivial or esoteric cases occur when considering the same for directed non-deterministic, undirected deterministic, and undirected non-deterministic graphs.

Our reductions create *sparse graphs* $G = (V, E)$ with $|E| = O(|V|)$, and hence the results are covering also the difficulty of finding $O(|V|^{1-\epsilon}\, m)$ or $O(|V|\, m^{1-\epsilon})$ time algorithms for SMLG. This difficulty carries over to non-deterministic *subdense graphs* with $|E| = O(|V|^{2-\epsilon})$ and alphabet size at least 3: given a sparse graph $G' = (V, E')$ and pattern P of length m from binary alphabet, convert G' into a subdense graph $G = (V, E)$ adding $|E|$ spurious arcs labeled with a third symbol. In other words, unless the OV hypothesis fails, there is no $O(|E|+|V|^{1-\epsilon}\, m+|E|^{\frac{1}{2}}\, m)$ time algorithm for SMLG on subdense graphs G for $m = O(|V|)$. However, for *dense graphs* with $|E| = \omega(|V|^{2-\epsilon})$, there is room to improve the bounds.

Open Problem 1. *Is there an* $O(|E| + |V|\, m + |E|^{\frac{1}{2}}\, m)$ *time algorithm for SMLG on dense graphs?*

Other natural directions to continue the study include the tradeoff between indexing and query time on string matching for graphs, as well as a closer examination of other possible string-alike graph classes than those already covered here.

For the former, a slight modification of the proof of Theorem 1.1 results in conditional hardness of finding $O(|E|^{\alpha} m^{\beta})$ time algorithms for SMLG for any $\alpha, \beta > 0$ with $\alpha + \beta < 2$. This observation can then be exploited in a self-reduction [24], showing that one cannot achieve subquadratic search times using polynomial time for indexing (under the OV hypothesis).

For the latter, one possible direction is to consider *degenerate generalized strings* [4]: a sequence $S = S_1, S_2, \ldots, S_n$ is a degenerate generalized string if set S_i consist of strings of fixed-length n_i for all i. When interpreted as an automaton, the language of S is the Cartesian product of its sets. It was recently shown that language intersection emptiness on two degenerate generalized strings can be decided in linear time in the total size of the sets [4]. However, if the requirement of equal length strings is relaxed, the complexity of string matching on such *elastic degenerate strings* has shown to have tight connection with fast matrix multiplication [12]. Naturally, our reductions do not cover graphs representing degenerate generalized strings. They also do not cover the elastic case, but another relaxation of degenerate generalized strings: consider that the Cartesian product taking all combinations of consecutive sets is replaced by an arbitrary selection of subsets of combinations of consecutive sets. A characteristic feature of graphs resulting from this relaxation is that all paths from one node to another are of the same length. This is also a feature of our reduction graphs. Hence, other features need to be identified to close this gap between linear time solvability and conditional quadratic time hardness; interestingly, conditional hardness of indexing elastic degenerate strings has been established without a direct link to the complexity of the online version [29].

After our last submission of this work for review, many new research directions have emerged around the topic. Some of these are already covered in a survey [42]. In the following, we briefly discuss some recent directions.

The conditional lower bounds have been strengthened to consider how many logarithmic factors can be shaved off from the quadratic complexity [30]. The conclusion is that if the denominator of the time complexity feature is a $O(\log^c m)$ or $O(\log^c |E|)$ term, the exponent c is bounded by

a constant. New graphs properties have been identified that make them amenable to indexing: graphs that can be partially sorted [18], graphs parameterized by the maximum width of their co-lexicographic relation [17], and graphs induced from suitable segmentation of multiple sequence alignments [25] admit efficient indexing schemes. The latter work adapts a reduction technique from this work to show that an arbitrary segmentation of a multiple sequence alignment does not break the conditional lower bound, but one needs a stronger property. Further complexity results have also been derived for online exact and approximate matching on different graph classes [14, 20, 32]. Finally, SMLG has been studied also under the model of computation of quantum computing [19], achieving a subquadratic solution for non-sparse graphs.

ACKNOWLEDGMENTS

We would like to acknowledge the contribution of Alessio Conte, Luca Versari and Bastien Cazaux in useful and inspirational conversations. Also, we would like to thank an anonymous reviewer of a previous version of this article for pointing out the open problem on dense graphs.

Even if the work of Backurs and Indyk [9] represents the closest connection with our results, a folklore proof by Russell Impagliazzo about the hardness of the NFA acceptance problem was also known. We would like to thank Karl Bringmann for bringing such proof to our attention.[5]

REFERENCES

[1] Amir Abboud, Arturs Backurs, and Virginia Vassilevska Williams. 2015. Tight hardness results for LCS and other sequence similarity measures. In *Proceedings of the IEEE 56th Annual Symposium on Foundations of Computer Science (FOCS'15)*. IEEE, Los Alamitos, CA, 59–78. https://doi.org/10.1109/FOCS.2015.14

[2] Tatsuya Akutsu. 1993. A linear time pattern matching algorithm between a string and a tree. In *Combinatorial Pattern Matching*. Lecture Notes in Computer Science, Vol. 684. Springer, 1–10. https://doi.org/10.1007/BFb0029792

[3] Jarno Alanko, Giovanna D'Agostino, Alberto Policriti, and Nicola Prezza. 2020. Regular languages meet prefix sorting. In *Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms (SODA'20)*. 911–930. https://doi.org/10.1137/1.9781611975994.55

[4] Mai Alzamel, Lorraine A. K. Ayad, Giulia Bernardini, Roberto Grossi, Costas S. Iliopoulos, Nadia Pisanti, Solon P. Pissis, and Giovanna Rosone. 2018. Degenerate string comparison and applications. In *Proceedings of the 18th International Workshop on Algorithms in Bioinformatics (WABI'18)*. Leibniz International Proceedings in Informatics, Vol. 113. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, Article 21, 14 pages. https://doi.org/10.4230/LIPIcs.WABI.2018.21

[5] Amihood Amir, Moshe Lewenstein, and Noa Lewenstein. 1997. Pattern matching in hypertext. In *Algorithms and Data Structures*. Lecture Notes in Computer Science, Vol. 1272. Springer, 160–173. https://doi.org/10.1007/3-540-63307-3_56

[6] Amihood Amir, Moshe Lewenstein, and Noa Lewenstein. 2000. Pattern matching in hypertext. *J. Algorithms* 35, 1 (2000), 82–99. https://doi.org/10.1006/jagm.1999.1063

[7] Renzo Angles and Claudio Gutierrez. 2008. Survey of graph database models. *ACM Comput. Surv.* 40, 1 (Feb. 2008), Article 1, 39 pages. https://doi.org/10.1145/1322432.1322433

[8] Arturs Backurs and Piotr Indyk. 2015. Edit distance cannot be computed in strongly subquadratic time (unless SETH is false). In *Proceedings of the 47th Annual ACM Symposium on Theory of Computing (STOC'15)*. ACM, New York, NY, 51–58. https://doi.org/10.1145/2746539.2746612

[9] Arturs Backurs and Piotr Indyk. 2016. Which regular expression patterns are hard to match? In *Proceedings of the IEEE 57th Annual Symposium on Foundations of Computer Science (FOCS'16)*. IEEE, Los Alamitos, CA, 457–466. https://doi.org/10.1109/FOCS.2016.56

[10] Arturs Backurs and Piotr Indyk. 2018. Edit distance cannot be computed in strongly subquadratic time (unless SETH is false). *SIAM J. Comput.* 47, 3 (2018), 1087–1097. https://doi.org/10.1137/15M1053128

[5] An example of this proof can be found in chapter 1, page 6, of the lecture notes of the course Fine-Grained Complexity Theory, run by Karl Bringmann and Marvin Künneman in 2019 for the Max Plank Institute Informatik. The lecture notes are available online at https://www.mpi-inf.mpg.de/departments/algorithms-complexity/teaching/summer19/fine-complexity/.

[11] Arturs Backurs and Christos Tzamos. 2017. Improving Viterbi is hard: Better runtimes imply faster clique algorithms. In *Proceedings of the 34th International Conference on Maching Learning (ICML'17)*. Proceedings of Machine Learning Research, Vol. 70., 311–321. http://proceedings.mlr.press/v70/backurs17a.html.

[12] Giulia Bernardini, Pawel Gawrychowski, Nadia Pisanti, Solon P. Pissis, and Giovanna Rosone. 2019. Even faster elastic-degenerate string matching via fast matrix multiplication. In *Proceedings of the 46th International Colloquium on Automata, Languages, and Programming (ICALP'19)*. Leibniz International Proceedings in Informatics, Vol. 132. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, Article 21, 15 pages. https://doi.org/10.4230/LIPIcs.ICALP.2019.21

[13] Karl Bringmann and Marvin Künnemann. 2015. Quadratic conditional lower bounds for string problems and dynamic time warping. In *Proceedings of the IEEE 56th Annual Symposium on Foundations of Computer Science (FOCS'15)*. IEEE, Los Alamitos, CA, 79–97. https://doi.org/10.1109/FOCS.2015.15

[14] Manuel Caceres. 2022. Parameterized algorithms for string matching to DAGs: Funnels and beyond. *arXiv:2212.07870* (2022). https://doi.org/10.48550/ARXIV.2212.07870

[15] The Computational Pan-Genomics Consortium. 2018. Computational pan-genomics: Status, promises and challenges. *Brief Bioinformatics* 19, 1 (2018), 118–135. https://doi.org/10.1093/bib/bbw089

[16] Alessio Conte, Gaspare Ferraro, Roberto Grossi, Andrea Marino, Kunihiko Sadakane, and Takeaki Uno. 2018. Node similarity with q-grams for real-world labeled networks. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'18)*. ACM, New York, NY, 1282–1291. https://doi.org/10.1145/3219819.3220085

[17] Nicola Cotumaccio. 2022. Graphs can be succinctly indexed for pattern matching in $O(|E|^2 + |V|^{5/2})$ time. In *Proceedings of the Data Compression Conference (DCC'22)*. IEEE, Los Alamitos, CA, 272–281.

[18] Nicola Cotumaccio and Nicola Prezza. 2021. On indexing and compressing finite automata. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA'21)*. 2585–2599. https://doi.org/10.1137/1.9781611976465.153

[19] Parisa Darbari, Daniel Gibney, and Sharma V. Thankachan. 2022. Quantum time complexity and algorithms for pattern matching on labeled graphs. In *String Processing and Information Retrieval*. Lecture Notes in Computer Science, Vol. 13617. Springer, 303–314. https://doi.org/10.1007/978-3-031-20643-6_22

[20] Riccardo Dondi, Giancarlo Mauri, and Italo Zoppis. 2022. On the complexity of approximately matching a string to a directed graph. *Information and Computation* 288 (2022), 104748. https://doi.org/10.1016/j.ic.2021.104748

[21] Massimo Equi, Roberto Grossi, and Veli Mäkinen. 2019. On the complexity of exact pattern matching in graphs: Binary strings and bounded degree. *arXiv e-prints, arXiv:1901.05264 [cs.CC]* (2019).

[22] Massimo Equi, Roberto Grossi, Veli Mäkinen, and Alexandru I. Tomescu. 2019. On the complexity of string matching for graphs. In *Proceedings of the 46th International Colloquium on Automata, Langages, and Programming (ICALP'19)*. Leibniz International Proceedings in Informatics, Vol. 132. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, Article 55, 15 pages. https://doi.org/10.4230/LIPIcs.ICALP.2019.55

[23] Massimo Equi, Roberto Grossi, Alexandru I. Tomescu, and Veli Mäkinen. 2019. On the complexity of exact pattern matching in graphs: Determinism and zig-zag matching. *arXiv e-prints, arXiv:1902.03560 [cs.CC]* (2019).

[24] Massimo Equi, Veli Mäkinen, and Alexandru I. Tomescu. 2021. Graphs cannot be indexed in polynomial time for sub-quadratic time string matching, unless SETH fails. In *SOFSEM 2021: Theory and Practice of Computer Science*. Lecture Notes in Computer Science, Vol. 12607. Springer, 608–622. https://doi.org/10.1007/978-3-030-67731-2_44

[25] Massimo Equi, Tuukka Norri, Jarno Alanko, Bastien Cazaux, Alexandru I. Tomescu, and Veli Mäkinen. 2022. Algorithms and complexity on indexing founder graphs. *Algorithmica*. Published online, July 28, 2022. https://doi.org/10.1007/s00453-022-01007-w

[26] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. 2018. Cypher: An evolving query language for property graphs. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD'18)*. 1433–1445. https://doi.org/10.1145/3183713.3190657

[27] Travis Gagie, Giovanni Manzini, and Jouni Sirén. 2017. Wheeler graphs: A framework for BWT-based data structures. *Theor. Comput. Sci.* 698 (2017), 67–78. https://doi.org/10.1016/j.tcs.2017.06.016

[28] Garrison Erik, Jouni Sirén, Adam M. Novak, Glenn Hickey, Jordan M. Eizenga, Eric T. Dawson, William Jones, et al. 2018. Variation graph toolkit improves read mapping by representing genetic variation in the reference. *Nat. Biotechnol.* 36 (Aug. 2018), 875. https://doi.org/10.1038/nbt.422710.1038/nbt.4227

[29] Daniel Gibney. 2020. An efficient elastic-degenerate text index? Not likely. In *String Processing and Information Retrieval*. Lecture Notes in Computer Science, Vol. 12303. Springer, 76–88. https://doi.org/10.1007/978-3-030-59212-7_6

[30] Daniel Gibney, Gary Hoppenworth, and Sharma V. Thankachan. 2021. Simple reductions from formula-SAT to pattern matching on labeled graphs and subtree isomorphism. In *Proceedings of the 4th Symposium on Simplicity in Algorithms (SOSA'21)*. 232–242. https://doi.org/10.1137/1.9781611976496.26

[31] Daniel Gibney and Sharma V. Thankachan. 2019. On the hardness and inapproximability of recognizing wheeler graphs. In *Proceedings of the 27th Annual European Symposium on Algorithms (ESA'19)*. Leibniz International Proceedings in Informatics, Vol. 144. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, Article 51, 16 pages. https://doi.org/10.4230/LIPIcs.ESA.2019.51

[32] Daniel Gibney, Sharma V. Thankachan, and Srinivas Aluru. 2022. On the hardness of sequence alignment on de Bruijn graphs. *J. Comput. Biol.* 29, 12 (2022), 1377–1396. https://doi.org/10.1089/cmb.2022.0411

[33] Shohei Hido and Hisashi Kashima. 2009. A linear-time graph kernel. In *Proceedings of the 9th IEEE International Conference on Data Mining (ICDM'09)*. IEEE, Los Alamitos, CA, 179–188.

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

[35] Chirag Jain, Haowen Zhang, Yu Gao, and Srinivas Aluru. 2019. On the complexity of sequence to graph alignment. In *Research in Computational Molecular Biology*, Lenore J. Cowen (Ed.). Springer International Publishing, Cham, Switzerland, 85–100.

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

[37] Antoine Limasset, Bastien Cazaux, Eric Rivals, and Pierre Peterlongo. 2016. Read mapping on de Bruijn graphs. *BMC Bioinform.* 17 (2016), 237. https://doi.org/10.1186/s12859-016-1103-9

[38] Udi Manber and Sun Wu. 1992. Approximate string matching with arbitrary costs for text and hypertext. In *Advances in Structural and Syntactic Pattern Recognition*. World Scientific, 22–33. https://doi.org/10.1142/9789812797919_0002

[39] Gonzalo Navarro. 2000. Improved approximate pattern matching on hypertext. *Theor. Comput. Sci.* 237, 1-2 (2000), 455–463. https://doi.org/10.1016/S0304-3975(99)00333-3

[40] Kunsoo Park and Dong Kyue Kim. 1995. String matching in hypertext. In *Combinatorial Pattern Matching*. Lecture Notes in Computer Science, Vol. 937. Springer, 318–329. https://doi.org/10.1007/3-540-60044-2_51

[41] Aaron Potechin and Jeffrey Shallit. 2020. Lengths of words accepted by nondeterministic finite automata. *Inform. Process. Lett.* 162 (2020), 105993. https://doi.org/10.1016/j.ipl.2020.105993

[42] Nicola Prezza. 2021. Subpath queries on compressed graphs: A survey. *Algorithms* 14, 1 (2021), 14.

[43] Eric Prud'hommeaux and Andy Seaborne. 2008. *SPARQL Query Language for RDF*. World Wide Web Consortium Recommendation REC-rdf-sparql-query-20080115. W3C.

[44] M. O. Rabin and D. Scott. 1959. Finite automata and their decision problems. *IBM J. Res. Dev.* 3, 2 (April 1959), 114–125. https://doi.org/10.1147/rd.32.0114

[45] Mikko Rautiainen and Tobias Marschall. 2017. Aligning sequences to general graphs in $O(V + mE)$ time. *bioRxiv* (2017). https://doi.org/10.1101/216127

[46] Marko A. Rodriguez. 2015. The Gremlin graph traversal machine and language (invited talk). In *Proceedings of the 15th Symposium on Database Programming Languages*. 1–10. https://doi.org/10.1145/2815072.2815073

[47] Korbinian Schneeberger, Jörg Hagmann, Stephan Ossowski, Norman Warthmann, Sandra Gesing, Oliver Kohlbacher, and Detlef Weigel. 2009. Simultaneous alignment of short reads against multiple genomes. *Genome Biol.* 10 (2009), R98. https://doi.org/10.1186/gb-2009-10-9-r98

[48] Chuan Shi, Yitong Li, Jiawei Zhang, Yizhou Sun, and Philip S. Yu. 2017. A survey of heterogeneous information network analysis. *IEEE Trans. Knowl. Data Eng.* 29, 1 (2017), 17–37. https://doi.org/10.1109/TKDE.2016.2598561

[49] Jouni Sirén, Niko Välimäki, and Veli Mäkinen. 2014. Indexing graphs for path queries with applications in genome research. *IEEE/ACM Trans. Comput. Biol. Bioinform.* 11, 2 (March 2014), 375–388. https://doi.org/10.1109/TCBB.2013.2297101

[50] Chris Thachuk. 2013. Indexing hypertext. *J. Discrete Algorithms* 18 (2013), 113–122. https://doi.org/10.1016/j.jda.2012.10.001

[51] Michael Wehar. 2016. *On the Complexity of Intersection Non-Emptiness Problems*. Ph.D. Dissertation. University at Buffalo, State University of New York. http://www.michaelwehar.com/documents/mwehar_dissertation.pdf.

[52] Ryan Williams. 2005. A new algorithm for optimal 2-constraint satisfaction and its implications. *Theor. Comput. Sci.* 348, 2 (2005), 357–365. https://doi.org/10.1016/j.tcs.2005.09.023

[53] Jaewon Yang and Jure Leskovec. 2015. Defining and evaluating network communities based on ground-truth. *Knowl. Inf. Syst.* 42, 1 (2015), 181–213. https://doi.org/10.1007/s10115-013-0693-z

Received 10 March 2020; revised 31 August 2021; accepted 7 March 2023