Department of Computer Science
Series of Publications A
Report A-2022-9

Lower and Upper Bounds for String Matching in Labelled Graphs

Massimo Equi

Doctoral dissertation, to be presented for public examination with the permission of the Faculty of Science of the University of Helsinki in Hall A129, Chemicum, on June 22, 2022, at 13 o'clock.

University of Helsinki
Finland

Supervisor
Veli Mäkinen, University of Helsinki, Finland
Alexandru I. Tomescu, University of Helsinki, Finland

Pre-examiners
Chirag Jain, Indian Institute of Science, India
Dominik Kempa, Stony Brook University, New York, USA

Opponent
Nicola Prezza, Ca' Foscari University of Venice, Italy

Custos
Veli Mäkinen, University of Helsinki, Finland

Contact information

Department of Computer Science
P.O. Box 68 (Pietari Kalmin katu 5)
FI-00014 University of Helsinki
Finland

Email address: info@cs.helsinki.fi
URL: http://cs.helsinki.fi/
Telephone: +358 2941 911

ISSN 1238-8645 (print)
ISSN 2814-4031 (online)
ISBN 978-951-51-8216-6 (paperback)
ISBN 978-951-51-8217-3 (PDF)
Helsinki 2022
Unigrafia

Lower and Upper Bounds for String Matching in Labelled Graphs

Massimo Equi

Department of Computer Science
P.O. Box 68, FI-00014 University of Helsinki, Finland
massimo.equi@helsinki.fi

PhD Thesis, Series of Publications A, Report A-2022-9
Helsinki, June 2022, 78+76 pages
ISSN 1238-8645 (print)
ISSN 2814-4031 (online)
ISBN 978-951-51-8216-6 (paperback)
ISBN 978-951-51-8217-3 (PDF)

Abstract

String Matching in Labelled Graphs (SMLG) is a generalisation of the classic problem of finding a match for a string into a text. In SMLG, we are given a pattern string and a graph with node labels, and we want to find a path whose node labels match the pattern string. This problem has been studied since 1992, and it was initially intended to model the problem of finding a link in a hypertext. Recently, the problem received attention due to its applications in bioinformatics, but all of the solutions, old and new, failed to run in truly sub-quadratic time.

In this work, based on four published papers, we study SMLG from different angles, first proving conditional lower bounds, and then proposing efficient algorithms for special classes of graphs.

In the first paper, we unveil the reason behind the hardness of SMLG, showing a quadratic conditional lower bound based on the Orthogonal Vectors Hypothesis and the Strong Exponential Time Hypothesis. The techniques that we employ come from the fine-grained complexity, and involve finding linear-time reductions from the Orthogonal Vectors problem to different variations of SMLG.

In the second paper, we strengthen our findings by showing that an indexing data structure built in polynomial time is not enough to provide

subquadratic time queries for SMLG. We devise a general framework for obtaining indexing lower bounds out of regular lower bounds, and we prove the indexing lower bound for SMLG as an application of this technique.

In the third paper, we surpass the limitations of our lower bounds by identifying a class of graphs, called founder block graphs, which support linear time queries after subquadratic indexing. This class of graph effectively represents collections of strings called multiple sequence alignments, if gap characters are not present.

In the fourth paper, we significantly improve our previous results on efficiently indexable graphs. We propose elastic founder graphs, a superset of founder block graphs, that are able to represent multiple sequence alignments with gaps. Moreover, we propose algorithms for constructing elastic founder graph, indexing them, and perform queries in linear time.

Computing Reviews (2012) Categories and Subject Descriptors:

Theory of computation → Problems, reductions and completeness
Theory of computation → Graph algorithms analysis
Theory of computation → Pattern matching
Theory of computation → Sorting and searching
Theory of computation → Dynamic programming
Applied computing → Genomics

General Terms:
algorithms, graphs, strings, bioinformatics

Additional Key Words and Phrases:
exact pattern matching, indexing, orthogonal vectors, complexity theory, reductions, lower bounds, edit distance, graph query, lower bounds, fine-grained complexity, string matching, multiple-sequence alignment, exact pattern matching, graph query, graph search, labelled graphs, string matching, string search, strong exponential time hypothesis, heterogeneous networks, variation graphs

Acknowledgements

This thesis was possible thanks to the collaboration with esteemed colleagues, whom I would like to thank. The first on the list cannot be other than my dear supervisor Veli Mäkinen, who helped me in finding my path to develop my own research, always taking the time to listen to my ideas, and never limiting my freedom of exploring even the bizarre ones. Co-supervisor Alexandru I. Tomescu, co-author of all papers, has been a key collaborator in finding new results, and guided me especially during the beginning of my doctoral studies. I thus thank the Academy of Finland and the European Research Council for funding my PhD studies via Mäkinen's project and Tomescu's starting grant, respectively.

Paper I is the result of a collaboration with Roberto Grossi, who mentored me while taking my first steps in the field of research, supervising my master's thesis and following me during my transition to doctoral studies. I thank Bastien Cazaux, co-author of Paper III and IV, for challenging my ideas also while I was working on Paper I and II.

I finally thank Lapo Frati for stimulating conversations that led me to a deeper understanding of the results of Paper IV.

Helsinki, June 2022
Massimo Equi

Original Papers

I **On the Complexity of String Matching for Graphs.**
Equi, M., Grossi, R., Mäkinen, V. & Tomescu, A. I.,
46th International Colloquium on Automata, Languages, and Programming (ICALP 2019). Leibniz International Proceedings in Informatics (LIPIcs); vol. 132, p. 55:1–55:15.

II **Graphs Cannot Be Indexed in Polynomial Time for Subquadratic Time String Matching, Unless SETH Fails**
Equi, M., Mäkinen, V. & Tomescu, A. I.,
47th International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 2021). Lecture Notes in Computer Science (LNCS); vol. 12607 p. 608–622.

III **Linear Time Construction of Indexable Founder Block Graphs**
Mäkinen, V., Cazaux, B., Equi, M., Norri, T. & Tomescu, A. I.,
20th International Workshop on Algorithms in Bioinformatics (WABI 2020). Leibniz International Proceedings in Informatics (LIPIcs); vol. 172 p. 7:1–7:18.

IV **Algorithms and Complexity on Indexing Elastic Founder Graphs**
Equi, M., Norri, T., Alanko, J., Cazaux, B., Tomescu, A. I. & Mäkinen, V.,
32nd International Symposium on Algorithms and Computation (ISAAC 2021). Leibniz International Proceedings in Informatics (LIPIcs); vol. 212 p. 20:1–20:18.

List of Errata

A conceptual mistake in Paper IV was brought to our attention after having already printed the thesis, as well as two technical typos.

- Check `https://arxiv.org/abs/2102.12822v6`, Section 7.2, for a revised version of our results. This affects the indexing and querying algorithms explained in Section 8.2 of the thesis, which has to be revised for the semi-repeat-free case. The repeat-free case is unaffected.
- Page 10, Definition 2.5:"m" should be "n".
- Page 48: "$O(N^4\lceil n/M \rceil)$" should be "$O(N^4\lceil n/N \rceil)$".

Contents

Chapter 1

Introduction

In this thesis we present conditional lower bounds as well as efficient algorithms for the problem of *String Matching in Labelled Graphs* (*SMLG*). Finding a match for a pattern string inside a text is a fundamental problem of theoretical computer science, representing the inner core of many algorithms. In its simplest forms, the problem has been undoubtedly solved, while some variations and generalisations still remain open. This thesis explores a generalisation of this problem to graphs, explaining why this problem is hard in general, and identifying special cases that are easier to solve.

Given string T, the simplest form of string matching consists in finding a substring of T that equals a given pattern string P. One of the most celebrated solutions for this problem is the algorithm based on the KMP function [30], a classical result proved in the '70s that solves the problem in linear time $O(|T|+|P|)$. We call this type of string matching *exact*, and we can see it as a special case of the more general *approximate* string matching, which is the problem of finding a substring of T of minimum distance with P. The distance measure typically adopted is the *edit distance*, that is the number of single-character insertions, deletions and substitutions needed to turn one string into the other.

Computing the edit distance between string A and string B is a quadratic problem, solvable with dynamic programming approaches in time $O(|A||B|)$ [20, 21]. Contrary to exact string matching, for which the linear-time algorithm is clearly optimal, the question of whether the quadratic algorithms were optimal remained open for several decades. Backurs and Indyk gave a final answer to this question in 2015 [10, 12], when they proved that it is not possible to improve over a quadratic time complexity, exploiting a new technique for proving conditional lower bounds for polynomial problems. This technique relies on the *strong exponential time hypothesis* (SETH),

which also implies the *orthogonal vector hypothesis* (OVH). The former is a hypothesis on the complexity of solving SAT [28], the latter is a hypothesis on the complexity of solving the orthogonal vectors problem, and they are both used in the field of *fine-grained complexity* for proving conditional lower bounds. The strength of this technique resides in allowing to study the exponent of polynomial time complexities, something that is hardly achieved by methods for unconditional lower bounds or for conditional lower bounds based on the $NP \neq P$ hypothesis. In the case of edit distance, given strings A and B such that $|A| = |B| = n$, the lower bound states that no algorithm can compute $\mathsf{EDIT}(A, B)$ in time $O(n^{2-\epsilon})$, unless SETH is false.

Approximate string matching is one possible generalisation of exact string matching, but we can push the limits even further and wonder how much harder can it be to match a string into a graph. We call this problem *string matching in labelled graphs* (SMLG) and, as standard string matching, it can be divided in *exact* SMLG and *approximate* SMLG. Given node-labelled graph G and pattern string P, exact SMLG asks to find a path in G whose concatenation of node labels, namely its path label, matches string P exactly. Approximate SMLG consists in finding the path label in G of minimum edit distance with P.

Manber and Wu [31] pioneered this line of research in 1992, proposing the first algorithm for approximate string matching in labelled graphs. This first algorithm was slightly slower than quadratic, because it featured a small logarithmic factor. In the following years, the research community proposed many other solutions to different variations of the problem, both for exact and approximate SMLG. Unfortunately, nobody was able to break thought the quadratic barrier, with the best results being achieved by Amir et al. [7, 8] in 1997 for the exact version of the problem, and one year later by Navarro [34, 35] for the approximate version. Thus, we could say that approximate SMLG had become not harder than computing edit distance, catching up with the string-versus-string case, but for exact SMLG the performance difference with the linear KMP algorithm remained substantial.

With no improvement over these results, the scientific focus for SMLG progressively shifted away, also because the main practical application was pattern matching in the internet hypertext, which was being solved by other means. In recent years, bioinformatics sparked new interest in this problem, for it naturally models several biological scenarios, especially in the field of *pangenomics* [19]. Thus, in the absence of an efficient online (i.e. not indexed) solution for the problem, researchers started to look

into alternatives that involved indexing the graph to achieve fast queries for the pattern. This led to new complications, as subquadratic queries and reasonable indexing time were not willing to marry. Using the index proposed by Thachuk [41], there still remain worst cases in which queries are quadratic; the approach of Sirén et al. [40] succeed in providing linear time queries, but the procedure for building the index, although it is expected to be linear, may require exponential time in the worst case.

Given these difficulties on both the online and the indexed fronts, research started to look for compromises. If generalising the problem from strings to a string and a graph is so much harder, maybe there is still hope for those structures that are, in a sense, between a string and a graph. Such structures can be, for example, special classes of graphs, like Wheeler graphs [26], or special collections of strings, like *generalized degenerate strings* (GDSs) [6] and *elastic degenerate strings* (EDSs) [9, 14] (and references therein). Wheeler graphs are graphs whose nodes can be sorted to support efficient string matching, while GDSs and EDSs are a sequence of sets of strings with specific properties, where a match for a pattern string span several sets, one string per set. Among all of these data structures, EDSs are the ones that are most suitable for pangenomics applications, that is, those applications in which we represent several similar genomic sequences as a *multiple sequence alignment* (MSA). Indeed, a MSA is a collection of sequences disposed in a matrix-like fashion, one sequence for each row, possibly with gap characters to better align the same characters on the same columns. Thus, the MSA can be divided into segments, that is, sets of columns, that can be turned into the sets of strings of an EDS. In doing so, we are virtually representing a larger set of sequences than those in the original MSA, and this often is a desired feature in biology contexts, as it models *recombination*. Nevertheless, the way in which EDSs control recombination is quite rigid, because we cannot decide what to include or exclude from the representation at the level of single sequences. Moreover, the time complexity for querying an EDS is subquadratic, but this result is achieved only with a randomized approach, and seems to be unlikely otherwise [14].

1.1 Overview of the Thesis

For SMLG, there are some questions that remain open, both in terms of lower bounds and upper bounds. Are the quadratic algorithms for exact SMLG optimal? Is it possible to index a graph in polynomial time and obtain subquadratic time queries? Are there graphs for which SMLG is a

subquadratic problem, that also are a representation of an MSA with an accurate control of recombination? In this thesis, we provide answers for all of these questions.

Chapter 1 is the introduction you are reading. Chapter 2 introduces the formal concepts and tools needed to understand the results presented in this thesis. The subsequent two chapters give an overview of the landscape of algorithms and lower bounds prior to our work. In particular, Chapter 3 presents the main design details of the first algorithm for SMLG, of the optimal online algorithms for exact and approximate SMLG, and of the indexed algorithms for exact SMLG. Chapter 4 discusses the origin of SETH, why it is considered to be reliable, and its ties with OVH. Chapters 5-8 are dedicated to the original publications.

Chapter 5 presents Paper I. We address the question of whether online exact SMLG could be solved in subquadratic time with a conditional lower bound for the problem. We show how it is possible to define a linear time reduction from OV to SMLG, proving that a subquadratic time algorithm for SMLG would contradict OVH. This also holds under SETH, thanks to its relation with OV. We extend this result for very simple graph structures, and in the case of undirected graphs we prove that the lower bound holds even for a chain of nodes, a result yet unpublished.

Chapter 6 presents Paper II. We provide another lower bound for exact SMLG, this time addressing the indexed case. We prove that indexing the graph in polynomial time is not enough to achieve subquadratic time queries for the pattern, under OVH and SETH. Instead of devising an ad-hoc reduction, we present this result as a special case of a more general fact. We define the concept of *linear independent-components* (*lic*) reduction and prove that, for every problem with such a reduction from OV, both an online and indexing conditional lower bounds hold. This is achieved by reducing a generalised version of OV to many smaller instances of OV itself. We also show how this allows us to obtain indexing lower bounds for other problems like edit distance basically for free, if they already have a *lic* reduction from OV.

Chapter 7 presents Paper III. We introduce the notion of *founder block graph* (FBG), a special class of graphs built on top of an MSA without gap characters. In a FBG, nodes are organised in subsequent blocks, which allow us to selectively decide which sequences to express by placing or not placing edges between nodes in consecutive blocks. We show that, by adding a feature called *repeat-free* property, FBGs can be indexed in polynomial to solve SMLG queries in linear time. We also give an algorithm to build such FBGs from a gapless MSA in linear time. Moreover, as novel content,

we present a property discovered after the publication of Paper III, which simplifies and strengthens the techniques.

Chapter 8 presents Paper IV. We significantly improve the previous results on FBGs by generalising them to *elastic founder graphs* (EFGs). The advantage is that we retain all the useful properties of FBGs, but now we can also handle MSAs with gap characters, and we relax the repeat-free property to *semi-repeat-free*. We start by proving that SMLG on EFGs is hard in general, finding a reduction from OV and applying the techniques of Paper II to automatically obtain an indexing lower bound as well. Then, we achieve similar results as for FBGs, proving that indexed exact SMLG on the special class of semi-repeat-free EFGs can still be solved with linear time queries, after polynomial indexing. The techniques used are more involved, since managing gap characters and the semi-repeat-free property require additional care. For constructing such EFGs, we offer two solutions, optimising two different objective functions, and we provide the details of one of these algorithms. Additionally, as novel content, we discuss whether there can be EFGs of different orders.

Chapter 9 assesses what we were able to achieve, where there is still room for improvement, and possible future directions.

1.2 Personal Contribution to the Original Publications

What follows is a list of my personal contributions to each original publication.

Paper I. This paper provides the conditional lower bound for online exact SMLG. I devised the first version of the reduction for OV to SMLG mostly by myself, and I have been one of the main contributors of every improvement on that model.

Paper II. This paper provides the conditional lower bound for indexed exact SMLG and proves that a similar result holds for every problem with a *lic*-reduction from OVH. I suggested the lines to follow and proposed the main ideas, which have been developed jointly with the other authors.

Paper III. This paper introduces FBGs and provides an efficient indexing scheme for them, as well as an algorithm to construct them from a gapless MSA. The main ideas were devised by the first author, while I checked the correctness of the formal definitions and algorithms, and I contributed to

the writing of some parts of the paper. The software was developed by the first and fourth author, I did not contribute to it.

Paper IV. This paper improves over Paper III, introducing EFGs, proving a conditional lower bound and efficient indexing solutions for the special case of semi-repeat-free EFGs, as well as an algorithm to construct them from an gapped MSA. I have been one of the main contributors in finding the reduction from OV for the special case of EFGs, and I participated in the reasoning process discovering the novel property described in Chapter 7, which served as a building block for the design for the matching algorithm.

Chapter 2

Preliminaries

We now introduce the notation and the formal definitions of key concepts adopted throughout the thesis. We also present and discuss some background concepts that are fundamental in order to understand and appreciate our results. This chapter is intended to be a glossary that the reader can refer to, in case a technical detail is unclear or forgotten.

2.1 Strings and Labelled Graphs

Strings and labelled graphs are the core concepts of this work. In this thesis, we will adopt the following definitions.

Pattern string. Given a string P of m characters from alphabet Σ, we denote its length as $|P| = m$. The i-th character in P is $P[i]$, with indexes starting from 1, and $P[i..j]$ is the substring of P that starts at position i and ends at position j, $P[i]$ and $P[j]$ included.

Graph. A node-labelled graph $G = (V, E, \ell)$ has $|V|$ nodes, $|E|$ edges and labelling function $\ell : V \to \Sigma$, which assigns to every node $v \in V$ a label $\ell(v)$ over alphabet Σ. We may refer to nodes labelled with $\sigma \in \Sigma$ as σ-nodes. A walk is a sequence of nodes $\pi = v_1, v_2, \ldots, v_k$ such that $(v_z, v_{z+1}) \in E$ for $1 \leq z \leq k-1$. A path is a walk where no node is repeated. The label of path or walk $\pi = v_1, v_2, \ldots, v_k$ is the concatenation of its node labels, that is $\ell(\pi) = \ell(v_1) \circ \ell(v_2) \circ \cdots \circ \ell(v_k)$, where $\circ$ indicates string concatenation. The set of in-neighbours and the in-degree of node $v \in V$ are $in(v)$ and $indeg(v)$, respectively; for out-neighbours and out-degree the notation is $out(v)$ and $outdeg(v)$.

Throughout this thesis, if not better specified, we always use the following conventions: P and $G = (V, E, \ell)$ refer to this definition of pattern and labelled graph, index i refers to positions in the pattern, index j refers to nodes in the graph.

We formally define the core problem of this thesis, that is, matching a pattern string into a node-labelled graph.

Problem 2.1 *Exact String Matching in Labelled Graphs (Exact SMLG)*

***Input*:** *Graph $G = (V, E, \ell)$ and pattern string $P \in \Sigma^+$ over alphabet Σ.*

***Output*:** True *if and only if there is a walk $(v_1, v_2, \ldots, v_{|P|})$ in G such that $P[i] = \ell(v_i)$ holds for all $1 \leq i \leq |P|$.*

We observe that the node labels could be defined using strings instead of single characters, namely $\ell : V \to \Sigma^+$, but this does not make a significant difference. On one hand, single-character node labels can be seen as a special case of string node labels, and since our lower bounds work for the former, they consequently work for the latter. On the other hand, we can always replace a string-labelled node with a chain of nodes labelled with single characters, and then connect the original in-neighbours to the first node of the chain and the last node to the original out-neighbours [8]. Thus, using this transformation, every algorithm for string node labels can run also for single-character node labels, and the opposite is also trivially true. For some algorithms, it might be the case that term $|V|$ turns into N in the time complexity, where N is the total numbers of characters in all node labels, but this does not significantly affect the asymptotic behaviour.

As mentioned in the introduction, we can define a more general version of SMLG in which we try to find approximate matches for a pattern string into a graph. Since matches are not exact, scientific literature sometimes phrases the problem as aligning a sequence against a graph, especially in more applied bioinformatics contexts. There are several ways of defining the approximate match of a sequence in a graph, because there are several ways of defining an approximate match already between two strings. Here, we adopt the edit distance as a measure of the quality of a match, and we discuss why alternative and more general definitions might pose a problem. We recall that the edit distance between two strings A and B is the minimum number of single-character insertions, deletions and substitutions needed to turn A into B. In this thesis, notation $\mathsf{EDIT}(A, B)$ is the value of the edit distance between strings A and B, while EDIT is the problem of computing such values.

Problem 2.2 *Approximate String Matching in Labelled Graphs (Approximate SMLG)*

***Input*:** *Graph $G = (V, E, \ell)$ and pattern string $P \in \Sigma^+$ over alphabet Σ.*

***Output*:** *Value $\mathsf{EDIT}(P, \ell(\pi))$, and possibly π itself, where $\pi = (v_1, v_2, \ldots, v_{|P|})$ is a walk in G such that, for every other walk π' of length $|P|$ in G, $\mathsf{EDIT}(P, \ell(\pi)) \leq \mathsf{EDIT}(P, \ell(\pi'))$.*

A more general way of defining approximate SMLG is by allowing errors also in the graph. In the definition we just gave, we allow errors only in the pattern, that is we apply edit operations to the pattern to make it match a walk in the graph. We could also define edit operations to apply to the graph so that it provides a match for the pattern. The reason why we do not consider this possibility is because Amir et al. [8] already proved that allowing errors only in the graph or in the graph and the pattern makes the problem NP-complete for a generic alphabet. This result has been recently strengthened by Jain et al. [29], extending it to binary alphabet and hamming distance.

2.2 Hypothesis for Conditional Lower Bounds

The conditional lower bound for EDIT, and thus for approximate SMLG, is based on an assumption on the time complexity needed for solving CNF-SAT. In a CNF-SAT problem we are given a Boolean formula F in conjunctive normal form over n Boolean variables $v_1, \ldots, v_n$, and we are asked to determine whether F is statisfiable, i.e. if there exists a truth assignment for the n variables that makes F true.

The aforementioned assumption on the complexity of SAT is the so called *strong exponential time hypothesis* (SETH) [28]. Here we present the original definition of SETH, which provides the tools to explain why such a hypothesis is reasonable in the first place. Later we show a simpler version of this definition, which is the most commonly used in the proofs of conditional lower bounds.

Definition 2.1 (SETH - Original) *Let* q-SAT *be an instance of CNF-SAT with n Boolean variables and at most* q *literals per clause.*
Given $s_q = inf\{\alpha : $ There is an algorithm solving q-SAT *in time $O(2^{\alpha n})\}$, SETH states* $s_\infty = \lim_{q \to \infty} s_q = 1$.

In other words, this means that the exponent in the $O(2^n)$ time complexity needed to solve a SAT problem cannot be improved, not even by a

constant factor. Hence, SETH can be defined in a different way, which is more suitable for proving conditional lower bounds.

Definition 2.2 (SETH - Alternative) *For every $\epsilon > 0$, a generic instance of* SAT *with n variables cannot be solved in $O(2^{(1-\epsilon)n})$ time.*

Closely related to SETH is also the even more reliable *exponential time hypothesis* (ETH), where by "more reliable" we mean that SETH $\Rightarrow$ ETH, but not the contrary.

Definition 2.3 (ETH) *For $q \geq 3$, $s_q > 0$.*

When proving lower bounds for problems that are solvable in polynomial time, the proofs based on SETH often make use of another problem named OV (*Orthogonal Vectors*) and of the related hypothesis OVH (*Orthogonal Vectors Hypothesis*).

Definition 2.4 (OV) *Let $X, Y \subseteq \{0,1\}^d$ be two sets of n binary vectors of length d. Determine whether there exist $x \in X, y \in Y$ such that $x \cdot y = \sum_{i=1}^{d} x[i] \cdot y[i] = 0$.*

Notation $x[i] \cdot y[i]$ indicates the scalar product when used for two single entries of vectors x and y, while it refers to the dot product $x \cdot y$ when applied on the vectors themselves.

Definition 2.5 (OVH) *For any constant $\epsilon > 0$, no algorithm can solve the OV problem in $O(m^{2-\epsilon} poly(d))$ time.*

2.3 Multiple Sequence Alignment and Block Graphs

A *multiple sequence alignment* $\mathsf{MSA}[1 \ldots m, 1 \ldots n]$ is a matrix with m strings drawn from $\Sigma \cup \{\text{-}\}$, each of length n, as its rows. Here - $\notin \Sigma$ is the *gap* symbol. For a string $X \in (\Sigma \cup \{\text{-}\})^*$, we denote $\mathsf{spell}(X)$ the string resulting from removing the gap symbols from X.

The MSA serves as the basis for the construction of special classes of graphs called block graphs. Here we give a definition of such graphs, and in Chapters 7 and 8 we see how to refine this definition to construct these graphs on top of MSAs using the concept of segmentation.

Definition 2.6 (Block Graph) *A* block graph *is a graph* $G = (V, E, \ell)$ *where* $\ell : V \to \Sigma^+$ *is a function that assigns a string label to every node and for which the following properties hold.*

1. *Set* V *can be partitioned into a sequence of* b blocks $V^1, V^2, \ldots, V^b$, *that is,* $V = V^1 \cup V^2 \cup \cdots \cup V^b$ *and* $V^i \cap V^j = \emptyset$ *for all* $i \neq j$;
2. *If* $(v, w) \in E$ *then* $v \in V^i$ *and* $w \in V^{i+1}$ *for some* $1 \leq i \leq b-1$; *and*
3. *if* $v, w \in V^i$ *then* $|\ell(v)| = |\ell(w)|$ *for each* $1 \leq i \leq b$ *and if* $v \neq w$, $\ell(v) \neq \ell(w)$.

To construct block graphs out of MSAs, we take advantage of several data structures: tries, suffix trees, suffix arrays and the Burrows-Wheeler transform.

A *trie* [15] of a set of strings is a rooted directed tree with outgoing edges of each node labelled by distinct characters such that there is a root to leaf path spelling each string in the set; the shared part of the root to leaf paths to two different leaves spell the common prefix of the corresponding strings. Such a trie can be computed in $O(N \log \sigma)$ time, where N is the total length of the strings, and it supports string queries that require $O(q \log \sigma)$ time, where q is the length of the queried string. In a *compact trie* the maximal non-branching paths of a trie become edges labeled with the concatenation of labels on the path.

A *suffix tree* is the compact trie of all suffixes of string $T\$$. In this case, the edge labels are substrings of T and can be represented in constant space as an interval. Such a tree takes linear space and can be constructed in linear time [25] so that when reading the leaves from left to right, the suffixes are listed in their lexicographic order.

The *suffix array* [32] of string T is an array $\mathtt{SA}[1..n+1]$ such that $\mathtt{SA}[i] = j$ if $T[j..n+1]\$$ is the i-th smallest suffix of string $T\$$ in lexicographic order. A *generalized suffix tree or array* is one built on a set of strings. In this case, string T above is the concatenation of the strings with symbol \$ between each.

Burrows-Wheeler transform $\mathrm{BWT}[1..n+1]$ [17] of string T is such that $\mathrm{BWT}[i] = T'[\mathtt{SA}[i]-1]$, where $T' = T\$$ and $T'[-1]$ is regarded as $T'[n+1] = \$$.

Finally, we explain our indexing algorithms for block graphs using an Aho-Corasick automaton. An *Aho-Corasick automaton* [3] is a trie of a set of strings with additional pointers (fail-links). While scanning a text string, these pointers (and some shortcut links on them) allow to identify all the positions in the text at which a match for any of the strings occurs.

The construction of the automaton takes the same time as that of the trie. Queries for text string T take $O(|T| \log \sigma + \texttt{occ})$ time, where $\texttt{occ}$ is the number of matches.

Chapter 3

Known Solutions for SMLG

Many algorithms have been proposed for solving SMLG in quadratic time. This chapter showcases key approaches representative of different techniques.

3.1 The First Algorithm for SMLG

The first attempt of solving SMLG dates back to 1992, when Manber and Wu published their pioneer study [31]. The goal was to find an approximate match between a pattern string and a graph whose nodes are labelled with strings; in other words, they wanted to solve approximate SMLG. They adopted this representation because the graph was meant to model a hypertext but, as discussed earlier, this problem is equivalent to the one in which we have single characters as node labels. The proposed algorithm can handle insertions and deletions and, in the worst case, runs in $O(N + R \log\log |P| + |P| \sum_{v \in G}(indeg(v) - 1))$ time, where N is the total number of characters in the graph and R is the total number of ordered pairs of positions and nodes at which the pattern and the graph match. The crucial terms are R and the last one, which make the algorithm run in time $O(|P||V| \log\log |P|)$ or $O(|P||E|)$, respectively, in the worst case. Their technique relies on keeping updated, for each node, a set of indexes that represent potential matches. When a node has multiple in-neighbours, their sets of indexes, potentially each of size $O(|P|)$, have to be merged together to compute the set of indexes for the current node. This operation is what leads to the quadratic time complexity $O(|P||E|)$, since in general this is the dominant term.

3.2 The Optimal Online Algorithm for Exact SMLG

A few years later, Amir et al. [7, 8] focused on finding exact matches between a pattern string and a graph, namely exact SMLG, following a different approach. Their algorithm is based on a product operation between the graph and the pattern, and we find this technique worth explaining for its originality.

Consider graph $G = (V, E, \ell)$ and pattern string P. Compute the Cartesian product of the nodes with the positions of the pattern, that is,

$$V' = \{v^i | \, v \in V, \, 1 \leq i \leq |P|\} \cup \{s, f\}.$$

What we are doing is replicating each node $|P|$ times, to represent every position of the pattern. Set V' constitutes the nodes of a new graph $G' = (V', E')$, whose edges E' are defined as follows. First, we connect s to every v^1, and every $v^{|P|}$ to f if and only if $\ell(v) = P[|P|]$. To better picture the resulting graph, we also add the extra condition $\ell(v) = P[1]$ for the first case, not present in the original work [8]. Then, we connect node v^i with node w^{i+1} if and only if there was an edge between v and w in the original graph, and if v^i represents a match for the i-th character of the pattern, that is $\ell(v) = P[i]$. Again, we add the additional condition $\ell(w) = P[i+1]$. Figure 3.1 shows an example of this transformation, which can be formally defined as

$$\begin{aligned} E' = &\{(s, v^1) | \, v \in V, \, \ell(v) = P[1]\} \cup \{(v^{|P|}, f) | \, v \in V, \, \ell(v) = P[|P|]\} \cup \\ &\{(v^i, w^{i+1}) | \, (v, w) \in E, \, \ell(v) = P[i], \, \ell(w) = P[i+1]\}. \end{aligned}$$

Existential queries can now be answered by running a DFS starting from s. If we can reach f there is at least one match, otherwise there is no match, and the DFS clearly runs in $O(|P|)$ time. Reporting the matches themselves requires few additional steps. When running the DFS, mark every node v^i such that there is a path from that node to f. Then, scan each node v of the original graph, and if the corresponding v^i is marked then report v as part of a match for P. Marking nodes v^i with a DFS can be done in $O(|V'|)$ time, since we can decided whether or not to mark a node by computing an *or* of the recursive calls on its children. The final scan clearly takes $O(|V|)$ time. The overall complexity is given by the most costly operation, computing the pattern-graph product, which takes both time and space $O(|P||E|)$. In Chapter 5, we prove that this is optimal under OVH and SETH.

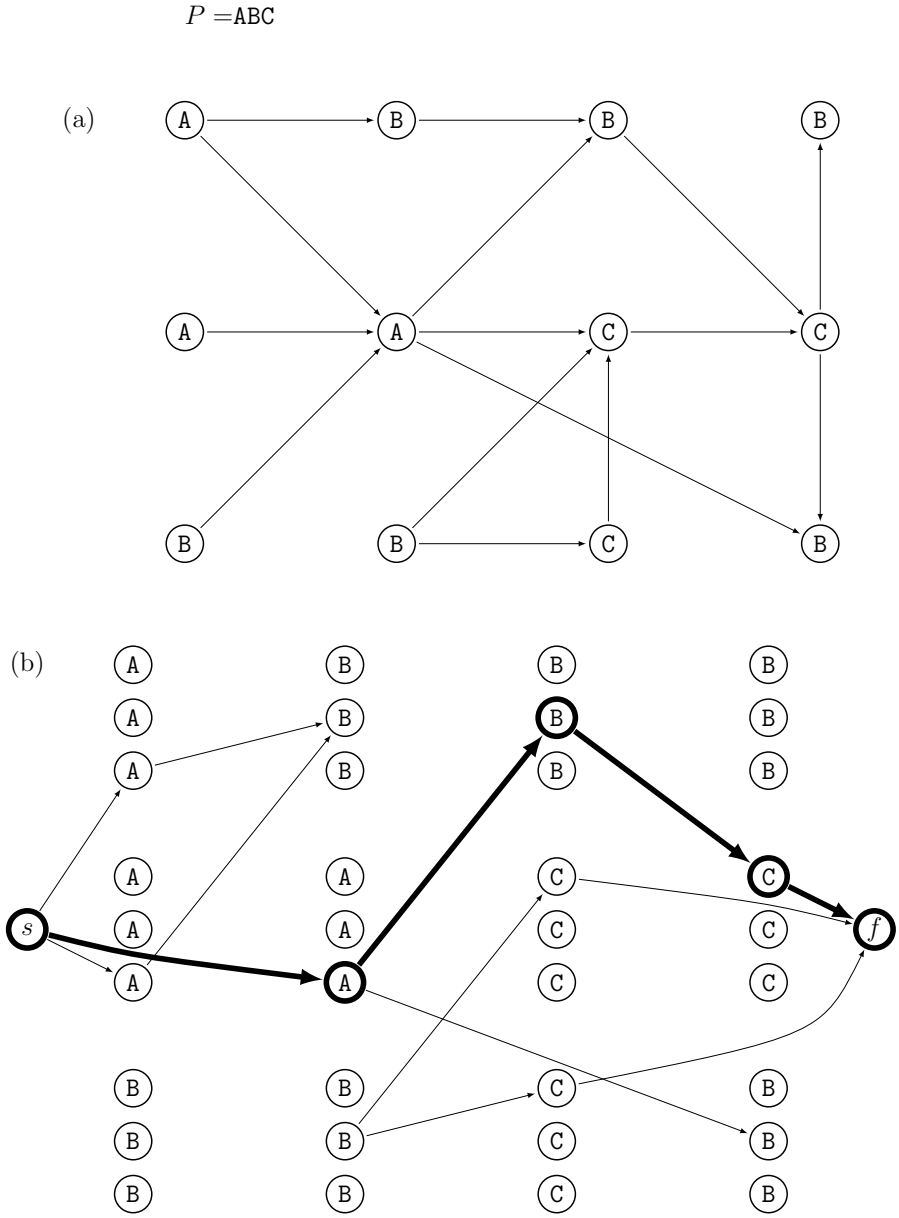

Figure 3.1: An execution of the algorithm of Amir et al. [8] to find a match for pattern $P =$ ABC. The graph at the top (a) is the original graph in which we want to find the match, the graph at the bottom (b) is built as the "product" of the pattern and the original graph. A DFS search starting from node s and ending at node f finds the path in bold, which corresponds to a match for P in the original graph.

3.3 The Optimal Online Algorithm for Approximate SMLG

The algorithm of Manber and Wu [31] was already able to handle approximate matches that allowed insertions and deletions in the pattern. However, this solution, besides not dealing with substitutions, features factor $R \log \log |P|$ in the time complexity. Since R is the number of position-node pairs with matching characters, it can be the case that $R = O(|V||P|)$. This term dominates over $O(|P||E|)$ when $|E| = O(|V|)$, meaning that in some cases the overall time complexity can be worse than quadratic, even if by just a double-logarithmic factor.

These difficulties are instead not present in the newer algorithm proposed by Navarro [35] in 2000, later refined by Rautiainen and Marschall [38] in 2017. This algorithm (in the version of Rautiainen and Marschall [38]) solves approximate SMLG in time $O(|V| + |E||P|)$, and requires space $O(|V|)$ on directed graph $G = (V, E)$ and pattern string P. Cycles are supported, and so are undirected graphs, because undirected edges can be seen as two directed edges with opposite orientation. Since exact SMLG is a special case of approximate SMLG, this is also an improvement over the algorithm of Amir et al. [8], because the space complexity is linear. Moreover, notice that EDIT is a special case of approximate SMLG in which the graph is a chain of nodes, thus the quadratic conditional lower bound for the former holds also for the latter, making this quadratic algorithm optimal under OVH and SETH.

The idea of this solution is to use dynamic programming to compute the minimum edit distance between the pattern and the graph, allowing errors only in the pattern. This means finding a path in the graph whose label is at minimum edit distance with the pattern. In the dynamic programming table, each row represents a position of the pattern, while graph nodes are on the columns. The algorithm updates the table row by row, thus processing one pattern character per iteration. The edges of the graph define the dependencies between the cells of the table, namely which cells have to be considered when updating the current cell. As illustrated in Figure 3.2, the cell in position (i, j) stores the cost $c_{i,j}$, which represents the fact of matching $P[i]$ against node label $\ell(v_j)$, possibly performing a substitution. Each $c_{i,j}$ is defined as

$$c_{i,j} = \min \begin{cases} c_{i-1,k} + \Delta_{i,j}, & \text{for } v_k \in in(v_j) \\ c_{i,k} + 1, & \text{for } v_k \in in(v_j) \\ c_{i-1,j} + 1. \end{cases}$$

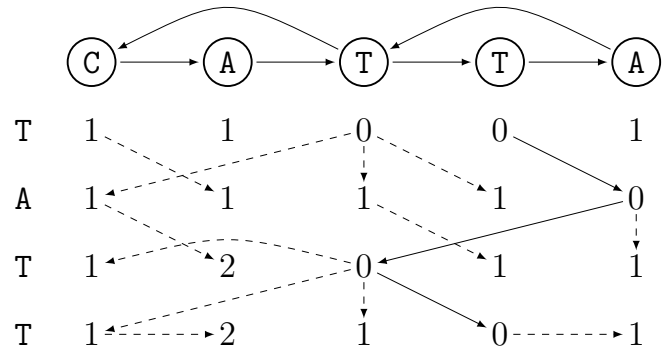

Figure 3.2: An example of the dynamic programming table used to solve approximate SMLG for pattern $P =$ TATT. The dashed arrows are the backtrace, that is, they show which cell was used to update the next one. The solid arrows highlight an optimal alignment of cost 0, which is also a solution for exact SMLG. This figure is taken from the work of Rautiainen and Marschall [38].

As the recurrence shows, there are diagonal dependencies when considering $c_{i-1,k}$, horizontal dependencies when considering $c_{i,k}$, as well as vertical dependencies when considering $c_{i-1,j}$. Similarly to edit distance, a diagonal dependency represents two substitutions, a vertical dependency represents an insertion and a substitution, and a horizontal dependency represents a deletion and a substitution. Hence, each row depends on the previous ones, but there are also horizontal dependencies among cells of the same row. This raises the issue of having cyclic dependencies within the same row, if the graph has cycles. To handle such dependencies, the authors define partial costs as follows:

$$p_{i,j} = \min \begin{cases} c_{i-1,k} + \Delta_{i,j}, & \text{for } v_k \in in(v_j) \\ c_{i-1,j} + 1 \end{cases}$$

In other words, $p_{i,j}$ is the value of the cost computed by ignoring horizontal dependencies. These partial costs can be used by the algorithm to update each row in two steps. First, a partial cost $p_{i,j}$ for all the cells in a row is computed; then, each $p_{i,j}$ is updated to $c_{i,j}$ by visiting the cells in ascending partial-cost order. After finishing this process, the minimum edit distance cost is found as the minimum value in the last row of the table. To justify why this procedure is correct, the authors prove the following property.

Lemma 3.1 *$p_{i,j} - c_{i,j} \in \{0, 1\}$, for all $i \in \{2, \ldots, |P|\}$ and $j \in \{1, \ldots, |V|\}$*

Now, consider partial costs p_{i,j_1} and p_{i,j_2}. If $p_{i,j_1} \leq p_{i,j_2}$, then Lemma 3.1 implies that final cost c_{i,j_1} is such that $c_{i,j_1} \leq p_{i,j_2}$. Thus, p_{i,j_2} will never be chosen as the best choice for computing c_{i,j_1}, simply because the choice that we made when computing p_{i,j_1} was already better. Thus, when scanning the cells in a row in ascending partial-cost order, we can ignore all the dependencies coming from cells with greater partial costs, and compute the final costs considering only the other dependencies.

3.4 Indexed Algorithms for Exact SMLG

As we have seen, researchers struggled to find solutions for SMLG that run in less than quadratic time, even considering the exact variation of the problem. Indeed, even without a formally proven lower bound, this quadratic barrier might seem unbreakable. However, all the solutions for SMLG that we explored so far follow the same approach, that is, they always try to solve the problem from scratch every time a new input is given. An alternative strategy is to first build and index structure on top of the graph, and then use such an index to handle queries for different patterns. If we follow this approach, how much time do we have to spend indexing the graph in order to achieve reasonably fast queries for the pattern?

Thachuk developed the first solution in this direction [41] considering the case of having strings as node labels, over alphabet Σ, and showing that it is possible, via succinct data structures, to run queries for pattern P in graph $G = (V, E, \ell)$ in time $O(|P| \log |\Sigma| + |P| \frac{\log |V|}{\log \log |V|} + \gamma^2 + \gamma \frac{\log |V|}{\log \log |V|})$, after indexing the graph in time $O(|E| \log |E|)$. Here, γ is the number of occurrences of nodes as substrings of the pattern, and γ^2 is the crucial factor in the time complexity, because in the worst case $\gamma^2 = O(|V|^2)$. Nevertheless, Thachuk underlines that if every node label is the prefix of at most $O(1)$ other node labels, then the query time reduces to $O(|P| \log \Sigma + |P| \frac{\log |V|}{\log \log |V|})$. We come back to this fact in Chapters 7 and 8, where we introduce the *(semi)-repeat-free* property, similar but not equivalent to Thachuk's. Thanks to this property, we achieve linear time queries for a special class of graphs.

A different type of result was achieved by Sirén et al. [40], that followed the strategy of potentially spending a lot of time indexing the graph to achieve subquadratic time queries for the pattern. This indexing scheme is called *Generalised Compressed Suffix Array* (GCSA), and it is based on the properties of the BWT and of the concept of *prefix-sorted* nodes of a finite

automaton. Assume to be given finite automaton A with nodes V where every node v lies on a path from v_1 to $v_{|V|}$ and $\ell(v_1) = \#$, $\ell(v_{|V|}) = \$$, and $\# \leq \sigma \leq \$$ for every $c \in \Sigma$. Then, *prefix-sorted* nodes are defined as follows.

Definition 3.1 *Node $v \in V$ of finite automaton A is* prefix-sorted *by prefix $p(v)$, if the labels of all paths from v to $v_{|V|}$ share a common prefix $p(v)$, and no path from any other node $u \neq v$ to $v_{|V|}$ has $p(v)$ as a prefix of its label. Automaton A is prefix-sorted if all nodes are prefix-sorted.*

In a prefix-sorted automaton, prefix-sorted nodes allow us to index node prefixes instead of listing all the paths for locating pattern matches. The goal is to have a structure that generalizes the BWT. One of the key properties of the BWT is that suffixes of the text starting with character c are sorted in the same way as suffixes preceded by character c. In case of a prefix-sorted automaton, we have prefixes $p(v)$ in place of suffixes, and the same relationship holds between the prefixes of nodes starting with character c and the prefixes of nodes that are out-neighbours of nodes with label c. This property allows indexing the graph so that it is possible to determine if pattern P matches or not in time linear in $|P|$. However, if our input graph is not prefix-sorted, we have to convert it into such form, and this is the crucial operation. For some graphs (or automata), the size of an equivalent prefix-sorted graph (or automaton) that accepts the same language is expected to be linear, but in the worst case there could be an exponential blow-up in the number of nodes. Thus, from the point of view of a worst-case analysis, this indexing scheme provides linear time queries for the pattern at the cost of exponential time for indexing the graph. Moreover, this technique works for DAGs but not for general graphs.

3.5 Algorithms for Alternative Graph Types

There are many different angles from which we can tackle the difficulty of solving SMLG in less than quadratic time. Indexing the graph is one way of making additional assumptions on how we want to handle queries, that is, we assume that preprocessing and querying can be done at two different times, with the former taking much longer than the latter. We could make alternative and complementary assumptions, to find faster algorithms for at least a simpler version of the problem. This is the motivation behind the study of *Generalized Degenerate Strings* (GDS) [5] and *Elastic Degenerate Strings* (EDS) [14]. These structures can be seen as DAGs with very specific properties, thanks to which string matching can be performed in subquadratic time.

A GDS is a sequence of n sets of strings, where the i-th set contains strings of the same length k_i, but this length can vary between different sets. The sum $\sum_{i=1}^{n} k_i$ of the lengths of each set is the width of the entire GDS. An EDS is a GDS where the strings in the i-th set can be of different lengths. Given a GDS of width W, its language is the set of all the possible strings of length W obtained by scanning the GDS from left to right, and concatenating a string of set i with one of set $i + 1$, for every $1 \leq i < n$. If we are given two GDSes of the same width, it is possible to determine whether or not the intersection of their languages is empty, in linear time in the number of characters in each GDS [5]. For EDSes instead, we know that it is possible to find a match for a string in expected $O(n|P|^{1.385} + N)$ time, where n is the number of sets of strings and N the total number of characters. In EDSes, a match for a string is allowed to start at any position of any string in any set, can span strings of multiple consecutive sets, and can end at any position of any string in the last consecutive set. In Chapters 7 and 8, we identify alternative classes of graphs for which it is possible to query the pattern in truly linear time, possibly after polynomial time preprocessing.

Chapter 4

New Lower-Bound Techniques

As already mentioned, the edit distance lower bound is conditioned on SETH, a hypothesis stating that, for any $\epsilon > 0$, CNF-SAT cannot be solved in time $O(2^{(1-\epsilon)n})$. During the last years, SETH has been extensively used for proving several other conditional lower bounds. The idea of such a hypothesis comes from the work of Impagliazzo and Paturi [27, 28] in 1999, where they proved a property on the constant $\alpha > 0$ involved in the time complexity $O(2^{\alpha n})$ of CNF-SAT. In their work, they concluded that it is reasonable to believe that SETH is true, since α increases as the number of literals per clause increases. Thus, they proposed the open problem of providing a formal proof for SETH. A few years later Williams [45] reduced CNF-SAT to a problem basically equivalent to the OV problem. This gave the tools to conclude that a strictly subquadratic time algorithm for OV would provide an improved algorithm for CNF-SAT, i.e. an algorithm running in $O(2^{(1-\epsilon)n})$ time. Since this would contradict SETH, it is clear that there is a strong connection between Impagliazzo and Paturi's [27, 28] work and Williams' [45] work; in particular, SETH implies OVH.

Nowadays, it is common practice to combine these two results to prove conditional lower bounds for polynomial time problems, and the edit distance problem is no exception. Indeed, the strategy adopted for proving the lower bound for EDIT [10] consisted in reducing OV to EDIT, which automatically proved that a strongly subquadratic time algorithm for computing edit distance would contradict SETH. The lower bound for the edit distance problem has been a cardinal step in pattern matching and it was suddenly followed by other strongly related results. Following the same strategy, the same lower bound was proven for computing the *longest common subsequence* and the *dynamic time warping distance* [1], which are both alternative definitions of distance between strings and curves, respectively. Moreover, the lower bounds for the edit distance and for the dy-

namic time warping distance have also been improved by Bringmann and Künnemann [16]. In fact, they proved that the lower bound still holds even when the sequences are drawn from a binary alphabet, while the original lower bound was valid only for an alphabet of size at least 7.

4.1 Reliability of SETH

The lower bound for EDIT heavily relies on SETH, which is a hypothesis believed to be true although no mathematical proof has been provided yet. It is reasonable to question how reliable it is, hence we need to explain for which reason such a hypothesis is supported. All the needed information is to be found in the original work from Impagliazzo and Paturi [28].

Consider the original statement of SETH given in Definition 2.1. Impagliazzo and Paturi first observed that the best known algorithms that had been proposed for solving CNF-SAT were running in $O(2^{(1-d/O(q))n})$ time, where d is a constant and q is the number of literals per clause. Hence, the s_q constant of their definition clearly had to be $s_q \leq 1$, and they wondered if such a constant could be exactly 1, when $q \to \infty$. Even if showing that $s_\infty = 1$ still is an open challenge, they proved that assuming ETH it holds true that $s_q \leq (1 - d/q)s_\infty$. The consequence is that sequence $\{s_q\}$ is increasing infinitely often when $q \to \infty$.

In order to understand why such inequality is making sequence s_q increase infinitely often, consider q and q' such that $q < q'$. Hence we have $(1 - d/q)s_\infty < (1 - d/q')s_\infty$, as shown in Figure 4.1.

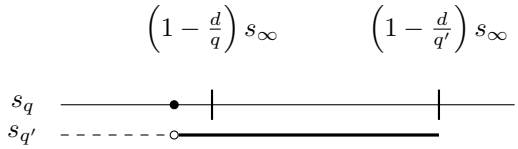

Figure 4.1: The solid line at the bottom shows the possible values for $s_{q'}$.

We then reason about where $s_{q'}$ could lie on the line of real numbers. Recall the definition of s_q. Since we are assuming ETH, that is, $s_q > 0$ for $q > 3$, there exist constants $s_q > 0$ and $s_{q'} > 0$. It cannot be the case that $s_{q'} < s_q$, because in that case we would have found a better constant for q-SAT, contradicting the very definition of s_q. Can it be that $s_q = s_{q'}$? To explain this, let us consider q_i such that $q < q_i < q'$. There must be only a finite number of constants q_i such that $s_q = s_{q_i}$. In fact, suppose by contradiction that $s_q = s_{q_i}$ for $i \to \infty$. In that case $s_q = s_\infty$, while we know that $s_q \leq (1 - d/q)s_\infty$. Thus, there must be a s_{q_i} such that $s_q < s_{q_i}$

and we consider such a constant to be our $s_{q'}$. Since we now proved that for every q there is a $q' > q$ such that $s_q < s_{q'}$, we conclude that sequence $\{s_q\}$ is increasing infinitely often.

Recall that the time complexity of the best known algorithms for CNF-SAT is $O(2^{(1-d/O(q))n})$. Given such time complexity and defining $s_q = 1 - d/O(q)$ we notice that $s_q \to 1$ when $q \to \infty$, that is, $s_\infty = 1$. We conclude that it is reasonable to believe that $s_\infty = 1$, namely SETH is true, because $s_\infty \leq 1$, sequence $\{s_q\}$ is increasing infinitely often and $s_\infty = 1$ is true for the known algorithms.

4.2 Reducing SAT to OV

When proving conditional lower bounds, it is common practice to reduce OV to the problem in question via a subquadratic time reduction. Then the proof is concluded by showing that a subquadratic time algorithm for the problem we reduced to would provide a subexponential time algorithm for CNF-SAT, contradicting SETH. Even if reducing directly from CNF-SAT is also possible, it is preferable to reduce from OV. This makes the proof easier to present, because we always work with polynomial time complexities, without having to deal with exponential time complexities. Indeed, this is possible since CNF-SAT can be reduced to OV. This relationship between these two problems allows us to use OVH in place of SETH, without weakening our results. We now explain that OVH is true unless SETH fails, presenting the following lemma.

Lemma 4.1 *SETH $\Rightarrow$ OVH.*

Let us go through a detailed sketch of the proof. The strategy is to start from an instance of CNF-SAT, reduce it to an instance of OV, and then show that a $O(n^{2-\epsilon_1})$ algorithm for OV would lead to a $O(2^{(1-\epsilon_2)n})$ algorithm for CNF-SAT, where $\epsilon_1 > 0$ and $\epsilon_2 > 0$. Since such a conclusion would contradict SETH, our claim would be proven.

Consider a generic instance of CNF-SAT where formula F is drawn from n variables $v_1, \ldots, v_n$ and consists of k clauses $c_1, \ldots, c_k$. Split the variables in two groups $v_1, \ldots, v_{\frac{n}{2}}$ and $v_{\frac{n}{2}+1}, \ldots, v_n$, assuming n to be even w.l.o.g., and define sets X and Y as all of the $2^{\frac{n}{2}}$ possible truth assignments for the first and second group of variables, respectively.

$$X = \{x_i \mid x_i = \langle b_1^{(i)}, \ldots, b_{\frac{n}{2}}^{(i)} \rangle \text{ is a truth assignment for } v_1, \ldots, v_{\frac{n}{2}}\}$$

$$Y = \{y_j \mid y_j = \langle b_{\frac{n}{2}+1}^{(j)}, \ldots, b_n^{(j)} \rangle \text{ is a truth assignment for } v_{\frac{n}{2}+1}, \ldots, v_n\}.$$

$$F = \underbrace{(v_1 \vee \neg v_3)}_{c_1} \wedge \underbrace{(\neg v_1 \vee v_2)}_{c_2} \wedge \underbrace{(\neg v_1 \vee v_3 \vee \neg v_4)}_{c_3} \wedge \underbrace{(v_2 \vee v_4)}_{c_4}$$

$$2^{\frac{n}{2}} \left\{ \begin{array}{lll} x_1 = & (0\ 0) & (1\ 0\ 0\ 1) = u_1 \\ x_2 = & (0\ 1) & (1\ 0\ 0\ 0) = u_2 \\ x_3 = & (1\ 0) & (0\ 1\ 1\ 1) = u_3 \\ x_4 = & (1\ 1) & (0\ 0\ 1\ 0) = u_4 \end{array} \right\} m \qquad \begin{array}{c} u_i[h] = 0 \\ \Leftrightarrow \\ x_i \models c_h \end{array}$$

$$2^{\frac{n}{2}} \left\{ \begin{array}{lll} y_1 = & (0\ 0) & (0\ 1\ 0\ 1) = w_1 \\ y_2 = & (0\ 1) & (0\ 1\ 1\ 0) = w_2 \\ y_3 = & (1\ 0) & (1\ 1\ 0\ 1) = w_3 \\ y_4 = & (1\ 1) & (1\ 1\ 0\ 0) = w_4 \end{array} \right\} m \qquad \begin{array}{c} w_j[h] = 0 \\ \Leftrightarrow \\ y_j \models c_h \end{array}$$

$v_1\ v_2$ over $\frac{n}{2}$; $c_1\ c_2\ c_3\ c_4$ over k

Figure 4.2: The reduction from CNF-SAT to OV. In this example we assume to have 4 variables ($n = 4$) and a CNF-SAT formula F with 4 clauses ($k = 4$). For each partial truth assignment, a vector is placed. In each vector, the entries set to 0 represent those clauses that can be satisfied by the corresponding partial truth assignment.

We call *partial* truth assignments the elements in X and Y. Each of these partial truth assignments can be related with a vector in an OV problem. Consider two sets of binary vectors U and W. Both U and W have $m = 2^{\frac{n}{2}}$ vectors each, one for every partial truth assignment. Moreover, such vectors are of length $d = k$, that is, they have one entry for each clause in F. For each vector $u_i \in U$, partial truth assignment $x \in X$ and clause c_h, set the h-th entry of the vector $u_i[h] = 0$ if and only if partial truth assignment x_i satisfies clause c_h ($x_i \models c_h$), for $1 \leq i \leq 2^{\frac{n}{2}}$ and $1 \leq h \leq d$. Proceed in the same way for all the vectors w_j of set W, but for the fact that we use the partial truth assignments y_j of set Y. See Figure 4.2 for an example.

At this point, we can observe that there exists a pair of orthogonal vectors if and only if F is satisfiable. Indeed, $u_i[h] \cdot w_j[h] = 0$ if and only if clause c_h is satisfied either by partial truth assignment x_i or by partial truth assignment y_j. Hence, $u_i \cdot w_j = 0$ if and only if every c_h is satisfied by either x_i or y_j, for $1 \leq h \leq d$. Such an observation represents the cardinal mechanism of the whole proof. Indeed, now we can solve our instance of OV and have answered the original CNF-SAT problem. Hence,

we wonder what happens if there exists an algorithm that solves OV in $O(m^{2-\epsilon_1}\text{poly}(d))$ time, for a constant $\epsilon_1 > 0$. Recalling that $m = 2^{\frac{n}{2}}$ and $d = k$, the consequence is that we can solve CNF-SAT in

$$\begin{aligned} O(m^{2-\epsilon_1}\text{poly}(d)) &= O((2^{\frac{n}{2}})^{2-\epsilon_1}\text{poly}(k)) \\ &= O(2^{(1-\frac{\epsilon_1}{2})n}\text{poly}(k)) \\ &= O(2^{(1-\epsilon_2)n}\text{poly}(k)) \end{aligned}$$

where $\epsilon_2 = \frac{\epsilon_1}{2}$, which would contradict SETH. Thus it holds that $\neg\text{OVH} \Rightarrow \neg\text{SETH}$, that is, if there exists a subquadratic time algorithm for OV, then SETH is false, which leads to the conclusion that $\text{SETH} \Rightarrow \text{OVH}$.

Chapter 5

Quadratic Conditional Lower Bound for Exact SMLG

All the attempts at solving SMLG online feature a quadratic term in the time complexity of their algorithms. This term appears regardless of facing the exact or approximate variant of the problem, which may suggest that what we are dealing with is an actual theoretical barrier. Understanding whether this was the case or not was the aim of Paper I, in which we proved that there do exist quadratic conditional lower bounds that apply even to very specific graph structures. We found this result to be surprising if compared to what we know for classical string matching. In fact, exact string matching in plain text can be solved in linear time [30], while approximate string matching (under edit distance) has a conditional quadratic lower bound [10]. Instead, when dealing with graphs, we proved that exact matching already has a quadratic lower bound, and thus the same holds for approximate matching too.

In order to prove our conditional lower bounds, we employ the recent techniques based on OV. At a general level, the strategy is to find a reduction from OV to exact SMLG that can be performed in linear time, implying that a subquadratic-time algorithm for SMLG would provide a subquadratic-time algorithm for OV, contradicting OVH and thus SETH. Depending on how we structure the reduction, it is possible to make the lower bound hold for very specific classes of graphs. The baseline is to define the pattern string on top of one set of vectors, and the graph on top of the other set, forcing the pattern to match the graph if and only if there is a pair of orthogonal vectors. The pattern is just a sequence of characters, a very simple structure with not much room for clever tricks, thus the crucial part of the reduction is always defining the graph in such a way that it will match only the "right" patterns.

We start by presenting the most immediate structure for the graph, which can be seen as a development of an earlier reduction for a problem on regular expressions [11]. Then, we shall highlight some weaknesses of this approach, and shift towards a significantly different graph structure, which we can further refine to make the reduction work even for a very restricted class of DAGs. If we allow the graph to be undirected, then the most basic structure for which we can find a reduction from OV is even simpler, just a chain of nodes. We present all of the announced results assuming a constant alphabet tuned to our needs but, as we proved in our work, the same conditional lower bounds hold even if restricted to a binary alphabet.

5.1 A First Reduction Scheme

Let us start with a first attempt at finding a reduction from OV to exact SMLG. Although we will later point out the limits of this approach and take a different route, this first solution introduces useful graph gadgets that are the base ingredients also for more refined reductions. Already in this preliminary reduction, we use directed edges and we do not introduce cycles in the graph, allowing the result to hold for DAGs.

Let X and Y be the two sets of n binary vectors $\{x_1, \ldots, x_n\}$ and $\{y_1, \ldots, y_n\}$ of length d in the OV problem. We want to define pattern string P on top of set X and graph $G = (V, E, \ell)$ on top of set Y, and we also want P to have a match in G if and only if there exists a pair $(x_i, y_j) \in X \times Y$ of orthogonal vectors. To give a better intuition on how we proceed, let us consider the following example of an OV instance, which we will carry on throughout the whole explanation:

$$X = \begin{Bmatrix} x_1 = 010 & 001 = y_1 \\ x_2 = 001 & 011 = y_2 \\ x_3 = 101 & 100 = y_3 \\ x_4 = 110 & 110 = y_4 \end{Bmatrix} = Y$$

The pairs of orthogonal vectors are (x_1, y_1), (x_1, y_3), (x_2, y_3) and (x_4, y_1), which are the solutions to OV. We now transform such instance of OV into an equivalent instance of SMLG, defining first the pattern, and then the graph.

Pattern. For each vector $x_i \in X$, we define subpattern $\mathtt{b}P_{x_i}\mathtt{e}$ such that $P_{x_i} = x_i$ (to be precise, P_{x_i} contains characters, while x_i contains natural

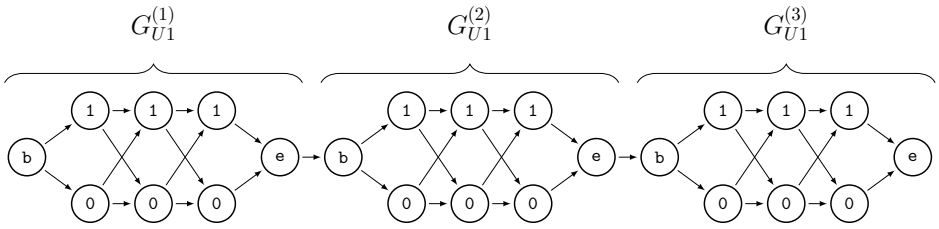

Figure 5.1: Subgraph G_{U1} for our example. Gadgets $G_{U1}^{(j)}$ in general are $n-1$, where n is the number of vectors in the OV instance. In our case $n-1=3$. Subgraph G_{U2} is identical.

numbers); the final pattern is $P = \texttt{bb}P_{x_1}\texttt{eb}P_{x_2}\texttt{e}, \ldots, \texttt{b}P_{x_{n-1}}\texttt{eb}P_{x_n}\texttt{e}$. Thus, set X in our example becomes

$$P = \texttt{b b010e b011e b101e b110e e}.$$

Graph. The idea for building the graph is to have some gadgets encoding the vectors in Y disposed in a column-like manner, and then attach a row of "overflow" gadgets to the left and one to the right of such a column. In this way, we can make subpattern P_{x_i} align with any of the gadgets in the column, and let the rest of the pattern match the overflow rows. If we devise the gadgets in the column properly, P_{x_i} will match the j-th gadget if and only if vectors $x_i \in X$ and $y_j \in Y$ are orthogonal. Moreover, to account for the rest of the pattern, the overflow rows should be able to match any number of subpatterns of any type.

Let us first see how to define the subgraphs representing the overflow rows. We call the gadgets in these subgraphs *universal gadgets*, because of their ability of matching any subpattern P_{x_i}, and we use the notation $G_{U_1}^{(j)}$ for those on the left row, and $G_{U_2}^{(j)}$ for those on the right row. The entire left row is subgraph G_{U1}, while the right row is subgraph G_{U2}. The structure of the gadgets, of which Figure 5.1 shows an example, is identical for all gadgets, but we use different indexes to refer to their different position in the final graph. To build gadget $G_{U_1}^{(j)}$, we start by placing a node labelled with `b`. Then, we place two parallel lines of d nodes, where recalling that d is the length of each vector in Y. All the nodes of the first line are labelled with `1`, while all the nodes of the second line are labelled with `0`. Finally, we place a node labelled with `e`, and we connect these nodes as shown in Figure 5.1. As is clear in the illustration, for any possible binary string of length d there is a path from `b` to `e` spelling that string, which means that such a gadget can match any subpattern $\texttt{b}P_{x_i}\texttt{e}$. Gadgets $G_{U_2}^{(j)}$ are identical

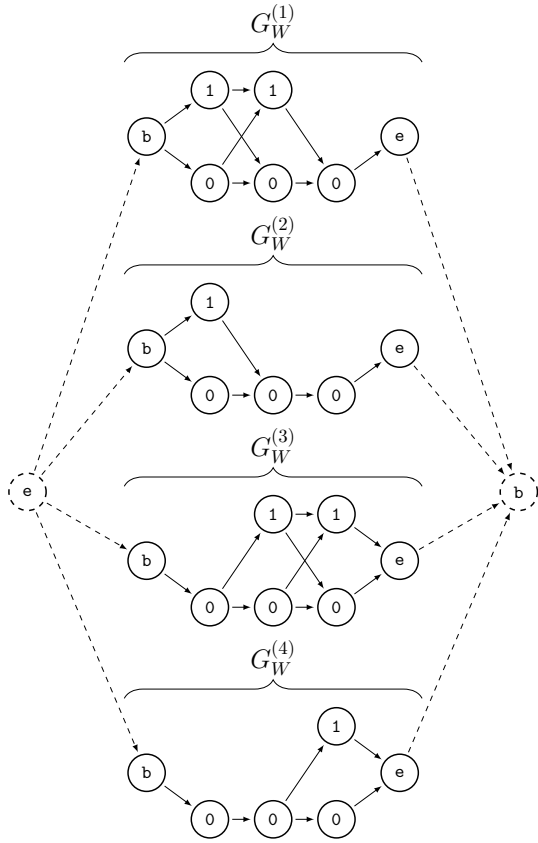

Figure 5.2: Subgraph G_W for our example. There is one gadget $G_W^{(j)}$ for each vector $y_j \in V$. The dashed `e`-node and `b`-node belong to G_{U1}^{n-1} and G_{U2}^{1}, respectively, and the dashed edges show the connection with such gadgets and thus with the universal subgraphs on the left and on the right.

to gadgets $G_{U_1}^{(j)}$. We construct G_{U1} (G_{U2}) by connecting the `e`-nodes with the `b`-nodes of $n-1$ universal gadgets $G_{U1}^{(j)}$ ($G_{U2}^{(j)}$) one after the other, in a row-like manner as in Figure 5.1.

Now we define the gadgets that encode each vector $y_j \in Y$, which we will denote $G_W^{(j)}$ and call *vector* gadgets. These gadgets need to have the property of matching subpattern P_{x_i} if and only if vectors x_i and y_j are orthogonal. To this end, we place the nodes in gadget $G_W^{(j)}$ as we did for gadget $G_{U_1}^{(j)}$, but with the difference that we drop the `1`-nodes for those positions h, $1 \leq h \leq d$, such that $y_j[h] = 1$. Figure 5.2 provides an example. The logic here is that the nodes in $G_W^{(j)}$ describe which characters

are allowed to be present in subpattern P_{x_i} when x_i is orthogonal to y_j. For instance, if we have a 0 in vector y_j at position h, that is, $y_j[h] = 0$, then $x_i[h]$ can be either 0 or 1 and still respect orthogonality; if $y_j[h] = 1$, $x_i[h]$ is forced to be 0 to respect orthogonality. Analogously, by dropping the 1-node for position h if $y_j[h] = 1$, we force pattern P_{x_i} to have its own 0 at that position to be able to match, which formally result in the following lemma.

Lemma 5.1 *Subpattern* $\mathtt{b}P_{x_i}\mathtt{e}$ *has a match in* G_W *if and only if there exists* $y_j \in Y$ *such that* $x_i \cdot y_j = 0$.

In subgraph G_W, we dispose the n vector gadgets $G_W^{(j)}$ one on top of the other, in a column-like manner as in Figure 5.2. We are now ready to put together all the gadgets to compose the final graph. As anticipated earlier, graph G consists of subgraphs G_{U1}, G_W and G_{U2}. Universal subgraphs G_{U1} and G_{U2} are identical and serve the purpose of matching up to $n-1$ subpatterns of P, while G_W implements the logic of the reduction. To build final graph G, we connect the e-node of $G_{U1}^{(}n-1)$ to the b-nodes of every $G_W^{(j)}$, and the e-node of every $G_W^{(j)}$ to the b-node of $G_{U2}^{(1)}$, as Figure 5.2 suggests. Notice that the b- and e-nodes of graph G forces pattern P to properly align each subpattern P_{x_i} against one single graph gadget, with no possibility to overlap two or more gadgets. Moreover, there are at most $n-1$ universal subgadgets in G_{U1} and G_{U2}, which means that one subpattern P_{x_i} is always forced to match in G_W.

Lemma 5.2 *Pattern* P *has a match in* G *if and only if a subpattern* $\mathtt{b}P_{x_i}\mathtt{e}$ *of* P *has a match in subgraph* G_W.

The correctness of the reduction follows from the combination of Lemma 5.1 and Lemma 5.2. With this reduction, the result that we achieved so far is a conditional lower bound stating that, given graph $G = (V, E, \ell)$ and pattern string P, exact SMLG cannot be solved in time $O(|E|^{1-\epsilon}|P|)$ or $O(|E||P|^{1-\epsilon})$, unless OVH, and thus SETH, fail. Can we tighten this lower bound? The answer is yes, but before fine-tuning the details, our reduction scheme needs a major restructuring, as we point out in the next section.

5.2 Weaknesses of the First Reduction Scheme

The structure of gadget G_W follows the same idea of a reduction provided in the work of Backurs and Indyk [11], where they study which types of regular expressions are hard to match. Among many others, they analyse

regular expressions that are a composition of *or* operators $|\cdot|$, as for instance $[(a|b)(b|c)]|[(a|c)b]$. Given a regular expression r of type $|\cdot|$ and a text t, they found a reduction from OV to the problem of determining whether a substring of t can be generated by r. The idea for the reduction is to encode the binary vectors $x_1, \ldots, x_n$ of X as $t = x_1 2 x_2 2 \ldots 2 x_n$, where 2 is a separator character; then, define regular expression $r = R_W^{(1)} \mid R_W^{(2)} \mid \cdots \mid R_W^{(n)}$, where $R_W^{(j)}$ can generate x_i if and only if $x_i \in X$ is orthogonal to $y_j \in Y$. Their gadgets $R_W^{(j)}$ work in the same way as our gadgets $G_W^{(j)}$, and indeed the NFA accepting the same language as r has the same structure of subgraph G_W.

Although for regular expressions this reduction scheme might be good enough, for graphs it turns out to be quite limiting. The major problem is the big fan-out needed to connect $G_{U1}^{(n-1)}$ with every $G_W^{(j)}$. Indeed, one valuable improvement to the lower bound would be to make it hold at least for DAGs of constant degree. This and other improvements are possible, but to achieve them we have to change the disposition of the $G_W^{(j)}$ gadgets in the graph and their connections with the other gadgets.

5.3 Refined Reduction

We are now ready to see how the structure of the reduction can be improved, and we will do this in two steps: first, we heavily reshape the graph; then, we fine tune the new graph to achieve even tighter lower bounds. For the refined reduction, the definition of universal gadgets and vector gadgets will remain the same, but their disposition and number will change. In the current graph we have G_{U1}, G_W and G_{U2} placed on the left, in the center and on the right, respectively, and by limiting the number of universal gadgets in G_{U1} and G_{U2} we force the pattern to use G_W in order to find a match. In the new graph, we avoid the big fan-out between G_W and the universal subgraphs by shaping G_W in a row-wise manner as well. The idea is to have G_{U1}, G_W and G_{U2} disposed on three rows, with G_{U1} at the top, G_W in the middle and G_{U2} at the bottom. By using special character sequences, we can make the pattern always start a match in G_{U1} or G_W and finish it in G_W or G_{U2}, if there is any.

The final graph with a possible pattern match for our example is depicted in Figure 5.3. We now describe this new structure and comment on the benefits. We have again two universal subgraphs, G_{U1} and G_{U2}, and they are constructed in the same way as before, with the only difference that now they feature $2(n-1) = 2n-2$ universal gadgets instead of $n-1$. Here, we apply the first modification which, as depicted in Figure 5.3, con-

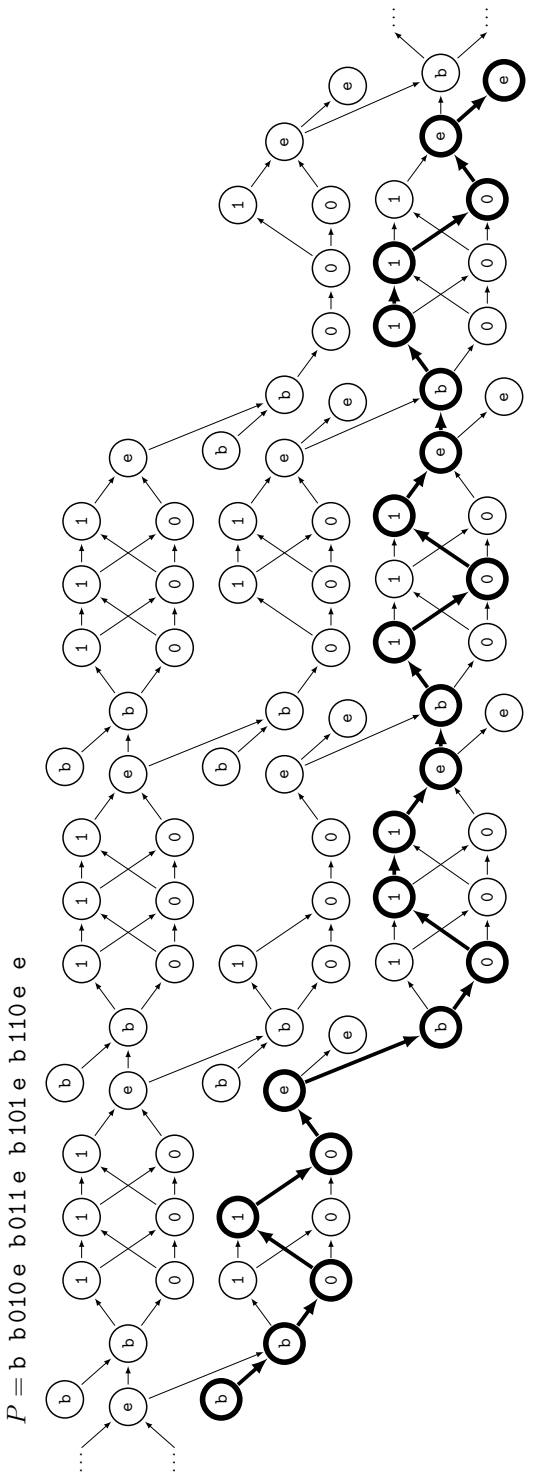

Figure 5.3: Final graph G with a match for pattern P in bold for our example. The dots hide three gadgets for G_{U1} on the left, and three gadgets for G_{U2} on the right.

sists in connecting a new b-node to each already-existing b-node in G_{U1}, and also in connecting each e-node in G_{U2} to a new e-node. Formally, for each b-node $b^{(j)}$ in G_{U1}, we define a new b-node $b^{(j)}_{new}$ and we connect them with the edge $(b^{(j)}_{new}, b^{(j)})$. Similarly, for each e-node $e^{(j)}$ in G_{U2} we define e-node $e^{(j)}_{new}$, and we connect the previous e-node with the new one with edge $(e^{(j)}, e^{(j)}_{new})$. In other words, we are placing a new node for each universal subgadget in both G_{U1} and G_{U2}. These additional nodes mark where a match for a pattern could begin and where it could end. Since there are no consecutive e-nodes in G_{U1}, no match can end there, and no match can begin in G_{U2}, since it has no consecutive b-nodes.

Vector subgraph G_W consists again of the n vectors gadgets $G^{(j)}_W$, but they are now disposed horizontally on a row, and they are not connected between each other. For each one of these gadgets, we also place both a new b-node and a new e node, and we connect them as we did for G_{U1} and G_{U2}. The major difference to the previous graph is that now each one of the last $n-1$ universal gadgets in G_{U1} is connected to a single vector gadget in G_W, which is in turn connected to a single universal gadget in G_{U2}, among the first $n-1$. The consecutive b-nodes and e-nodes are the key for correctness. Since the pattern starts with bb... and ends with ...ee, a match can start only in G_{U1} or G_W, and it can end only in G_W or G_{U1}, because these special substrings cannot be matched anywhere else. This forces the pattern to always use at least one vector gadget in G_W to have a proper match in the graph, and we know that that can happen if and only if at least one P_{x_i} matches at least one $G^{(j)}_W$, that is $x_i \cdot y_j = 0$.

Before improving on this reduction, let us first analyze which are the advantages with regard to the previous reduction. The main observation is that now every node has a constant number of incoming and outgoing edges, thus we are avoiding the big fan-out. This is already a better lower bound by itself: no algorithm can solve exact SMLG in subquadratic time on DAGs of constant degree. Note also that the alphabet that we are using is constant, but we are nonetheless using 4 characters. In the next section, we will see how to further push the restrictions on the graph and the alphabet.

5.4 Determinism

The reduced degree in the new graph leads the way to an important feature: determinism. In our current graph, each node has at most two out-neighbours, and they almost always have different labels. The only case in which this does not happen is for the e-nodes of gadgets $G^{(j)}_{U1}$, for

$n-1 \leq j \leq 2n-2$. Thus, we are close to having a deterministic graph, meaning a graph where all of the out-neighbours of a node have a different label, and it is tempting to try to extend our reduction to cover such a case. This might seem unlikely, because we know that we can check in linear time whether or not a deterministic finite automaton (DFA) accepts a given string. With such a reduction, the main difference between DFA acceptance and exact SMLG would be only the fact that in a deterministic DAG we do not know where a match for the pattern could start, while in a DFA we have a single start state. This is actually the case, because the graph in the reduction can be made deterministic. Let us see how.

As observed before, the only non-deterministic nodes in the graph are the e-nodes of gadgets $G_{U1}^{(j)}$, because they need to give the choice to either continue matching G_{U1} or to jump to G_W. However, consider $G_W^{(1)}$ in our example. This gadget encodes vector $y_1 = 010$, and up to the first pair of 0- and 1-nodes it does not differ from a $G_{U1}^{(j)}$ gadget. They shape differently when we encounter the 1 character in y_j, because $G_W^{(1)}$ lacks a 1-node, which is instead present in $G_{U1}^{(j)}$. Hence, our idea is to merge every $G_W^{(j)}$ with the corresponding $G_{U1}^{(n-1+j)}$, starting from the left, for as long as they have the same structure, and divide them only when they start to shape differently. As depicted by Figure 5.4, we leave the entire $G_W^{(j)}$ unchanged, but when we encounter the first 1 in y_j to encode in $G_W^{(j)}$ at position h, we also place a partial version of gadget $G_{U1}^{(n-1+j)}$, starting at position h and continuing until the end of the gadget, that is, until reaching the e-node. Then, we connect the 0- and 1-nodes of $G_W^{(j)}$ in position $h-1$ to the 1-node in position h in $G_{U1}^{(n-1+j)}$. We do so also for every subsequent position h for which $y_j[h] = 1$, that is, we have no 1-node in $G_W^{(j)}$. If $h = 1$, then we connect the b-node of $G_W^{(j)}$ to the 1-node in $G_{U1}^{(n-1+j)}$ instead. Finally, we connect the e-node in $G_W^{(j)}$ to the b-node in $G_{U2}^{(j)}$, and the e-node in $G_{U1}^{(n-1+j)}$ to the b-node in $G_w^{(j+1)}$

The intuition behind this construction is that gadgets $G_W^{(j)}$ and $G_{U1}^{(j)}$ are separated only when the pattern has to make a meaningful decision on how to continue a match, and thus when choosing one path or another determines whether we are matching $G_W^{(j)}$ or $G_{U1}^{(n-1+j)}$. Indeed, if vector y_j has a series of 0s at the beginning, it does not make a difference whether we are matching $G_W^{(j)}$ or $G_{U1}^{(n-1+j)}$. The first meaningful decision point is when we encounter the first 1 in y_j. If the pattern keeps matching $G_W^{(j)}$, it faces the same choice for any subsequent 1 in vector y_j. Therefore, there

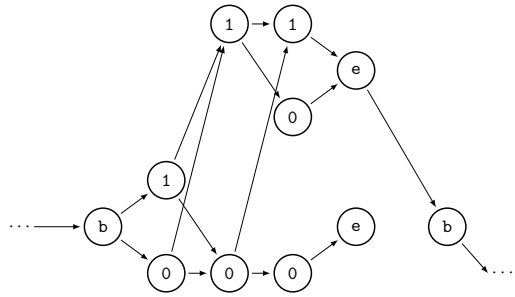

Figure 5.4: Gadgets $G_W^{(5)}$ and $G_{U1}^{8)}$ of our example, merged together. The $G_W^{(5)}$ substructure encodes vector $x_2 = 011$.

is no non-determinism anymore, because the path to choose is always only one: if $y_j[h] = \texttt{0}$, then we match $G_W^{(j)}$; if $y_j[h] = \texttt{1}$ and we have a `0` in the pattern, then we match $G_W^{(j)}$; if $y_j[h] = \texttt{1}$ and we have a `1` in the pattern, then we match $G_{U1}^{(n-1+j)}$ from this position onwards.

5.5 Lower Degree and Binary Alphabet

Making the graph deterministic required significant changes to the structure of the graph. The other features that we include require smaller changes, and we now list the additional results that we achieved, leaving the more technical details to the actual paper. In our current graph, the sum of indegree and outdegree of every node is at most 4, and the alphabet is of size 4. We would like to see how much we can reduce the degree and the alphabet size, and still obtain a quadratic problem.

We start by reducing the degree. Let us first observe that if every node of a DAG has in-degree at most 1 and out-degree at most 2, then it is a tree or an arborescence (a set of trees whose roots are connected in a chain of nodes, possibly forming a cycle). The same is true for the reverse, that is in-degree at most 2 and out-degree at most 1. We know that for trees the problem is solvable in linear time [4]. A future extended version of Paper I will also present a linear-time algorithm for arborescences. However, if the two aforementioned structures coexist in the same graph, then we have a DAG where the sum of in-degree and out-degree is at most 3[1]. In Paper I,

[1]To be precise, this definition includes also nodes with in-degree 3 and out-degree 0, or vice versa. For simplicity, we did not discuss these cases explicitly, but it is easy to see that they are not necessary for the reduction, so they could be omitted and still we would achieve the same results.

we show that the quadratic conditional lower bound holds also for such DAGs, which we call 3-DDAGs.

The last improvement to our lower bound concerns the alphabet size. We apply the following encoding to both the pattern and the graph, achieving a binary alphabet.

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

Given 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 $\alpha(\sigma)$. We also need to make the pattern start with characters `ebb` (instead of just `bb`), end with characters `eeb` (instead of just `ee`), and modify the graph accordingly, to exploit the properties of sequence `eb`. Then, we observer the following.

Lemma 5.3 *For any string $x \in \Sigma^+$, its binary encoding $\alpha(x)$ contains* `0110` *if and only if x contains* `eb`.

Thus, this transformation of the pattern and the graph is correct because the encoding of the string `eb` in the pattern is aligned with the encoding of the `eb`-edges in the graph, and thus the encoding of all the characters and node labels properly align as well. The final result is the core finding of our work.

Theorem 5.1 *For any constant $\epsilon > 0$, the String Matching in Labeled Graphs (SMLG) problem for a binary alphabet and a labeled* deterministic *directed acyclic graph (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 it is restricted to graphs in which the sum of outdegree and indegree of any node is at most three (i.e, 3-DDAGs).*

5.6 Undirected Graphs: Zig-zag Matching

This section is original content for this thesis yet unpublished, thus we present all the needed formal proofs. We show the results for the special case of undirected graphs by proving the following theorem.

Theorem 5.2 *The conditional lower bound stated in Theorem 5.1 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.*

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.

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

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

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

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

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. Here follows an example:

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

$$\begin{array}{lccccccccc} & & & \mathtt{1} & & \mathtt{0} & & \mathtt{0} & & \\ P' = \mathtt{b} & \mathtt{t\ s} & \mathtt{ABA} & \mathtt{s} & \mathtt{ABABABA} & \mathtt{s} & \mathtt{ABABABA} & \mathtt{s} & & \\ & & \mathtt{1} & & \mathtt{0} & & \mathtt{1} & & & \\ & \mathtt{t\ s} & \mathtt{ABA} & \mathtt{s} & \mathtt{ABABABA} & \mathtt{s} & \mathtt{ABA} & \mathtt{s} & \mathtt{t} & \mathtt{e} \end{array}$$

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 $\mathtt{0} = \mathtt{ABABABA}$ and $\mathtt{1} = \mathtt{ABA}$. Positions h with only a `0`-node and no `1`-node are encoded instead with a path that can be matched only by $\mathtt{0} = \mathtt{ABABABA}$ (see Figure 5.5). 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 5.6 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

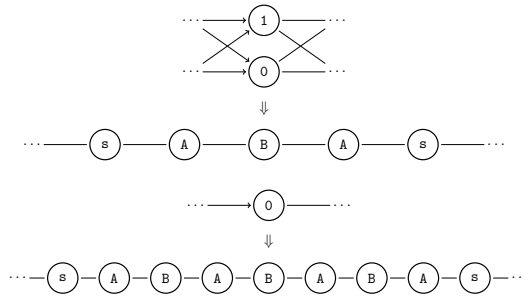

Figure 5.5: New substructures. (a) The old substructure is replaced by an undirected path that can match either `sABAs` (which represents `1`) by going forward only, or `sABABABAs` (which represents `0`), by going forward, backward, and forward again. (b) An undirected path replacing a `0`-node can match only the string `sABABABAs`.

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 5.7. In such a 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, backwards and forwards 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 positions i to match inside $LG_W^{(j)}$. We will address this minor issue at the end (see page 42).

In order to construct the final graph LG we concatenate all $LG^{(1)}$, $LG^{(2)}$, ..., $LG^{(n)}$ into a single undirected path. Figure 5.8 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 $O(nd)$.

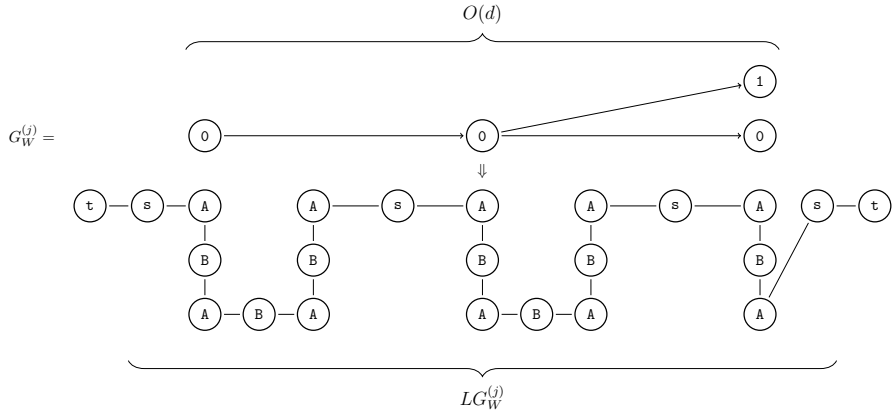

Figure 5.6: A subgraph $G_W^{(j)}$ is converted into a linear structure $LG_W^{(j)}$ using `s` as separator.

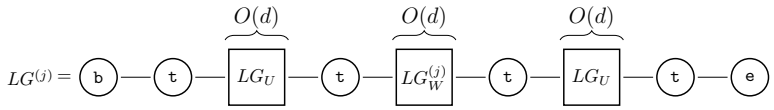

Figure 5.7: The $LG_W^{(j)}$ structure surrounded by two instances of LG_U. The `t`-nodes establish the beginning and the end of a match for a subpattern $\texttt{t}P'_{x_i}\texttt{t}$ while the `b`- and `e`-nodes are the starting and ending point for a match of the whole pattern P'.

In order 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 5.4 *If subpattern* $\texttt{t}P'_{x_i}\texttt{t}$ *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 $\texttt{t}P'_{x_i}\texttt{t}$. 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

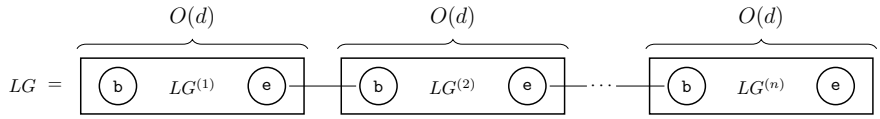

Figure 5.8: The final graph LG.

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, i.e 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 $s_k^{\left(LG_W^{(j)}\right)}$ 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, i.e., $s_1^{\left(LG_W^{(j)}\right)}$.

Inductive Case $k > 1$. The inductive hypothesis tells us that all the nodes up to $s_k^{\left(LG_W^{(j)}\right)}$ 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 $s_k^{\left(LG_W^{(j)}\right)}$ the next s-nodes that can be matched by $\mathtt{s}_{k+1}^{(P'_{x_i})}$ are $s_{k-1}^{\left(LG_W^{(j)}\right)}$ and $s_{k+1}^{\left(LG_W^{(j)}\right)}$. Character $\mathtt{s}_{k+1}^{(P'_{x_i})}$ cannot match node $s_{k-1}^{\left(LG_W^{(j)}\right)}$ 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 $s_{k+1}^{\left(LG_W^{(j)}\right)}$. □

Lemma 5.5 *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. We have already proved this property for gadget G_W in Lemma 5.1, 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, while we placed an encoded `0` if we had only a `0`-node. Lemma 5.4 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. On the other hand, 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 forwards, backwards and forwards 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 5.1 to complete the proof. □

The main difference from 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 5.6 *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 5.5 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 $\mathtt{t}P'_{x_i}\mathtt{t}$ which 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 5.6 gives us a property which holds only if a subpattern is in 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:

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

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 a case we can simply add a dummy subpattern $\bar{P} = \texttt{s ABA s ABA s} \ldots \texttt{s ABA s}$ (with d repetitions of `ABA`) at the end of P as if 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 $LG^{(j)}$, while 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[2]. Now we are ready to present the end result.

Lemma 5.7 *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 5.6 this means that there exists a subpattern $P'^{(q)}_{x_i}$, $q \in \{1, 2\}$ which has a match in $LG^{(j)}_W$, for some j. Lemma 5.5 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 exists two orthogonal vectors $x_i \in X$ and $y_j \in Y$. Thanks to Lemma 5.5 we find a subpattern P'_{x_i} matching $LG^{(j)}_W$. By construction, P'_{x_i} has to be in even position either in $P'^{(1)}$ or in $P'^{(2)}$. By Lemma 5.6 this means that either $P'^{(1)}$ or $P'^{(2)}$ has a match in LG. □

Theorem 5.2 follows directly from the correctness of these constructions, except for the alphabet size reduction to binary, which we leave for the extended version of Paper I (of which a preprint version is available [24, 23]).

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

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

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 $LG^{(j)}_W$, for some j, since every such subpattern occurs in an even position.

Chapter 6

Indexing Conditional Lower Bound for Exact SMLG

In the previous chapter we proved that the quadratic algorithms proposed more than 20 years ago for exact SMLG were indeed optimal for solving the problem online, under OVH and SETH. Nevertheless, an indexing scheme can help improving the situation. As discussed in Chapter 3, there exist indexing schemes that provide subquadratic time queries, but all of them present some critical issues, because either the queries are quadratic in some bad instances, or the time for building the index is exponential in the worst case. Therefore, one question arises: can we have both subexponential indexing and subquadratic time queries? We addressed this problem in Paper II, proving that any polynomial indexing scheme cannot provide subquadratic time queries, unless OVH and SETH fail. This result can be achieved by devising a reduction from OV to many instances of SMLG, each one addressing a different OV subproblem, obtaining the right complexity by properly setting the size of these instances. Nevertheless, we decided to follow a slightly different approach partially known as folklore knowledge, in order to achieve a much more general result. Instead of reducing OV to SMLG, we reduce OV to smaller instances of OV itself, proving that if indexing OV in polynomial time would provide subquadratic time queries, then OVH would fail. This allows us to prove that not only SMLG, but any problem that has a reduction from OV cannot be indexed in polynomial time for subquadratic time queries, unless OVH is false. Of course, the reduction from OV should have a few key properties to make this mechanism work, and thus we formally define them so that our result could be used as a black box. The idea is to provide indexing lower bounds basically for free: if a reduction from OV to a certain problem is found and it respects some properties, then both the online and indexing lower bounds hold.

6.1 Linear Independent-Components Reduction

One of the main goals of Paper II is to give the tools for using our result as a black-box, so that it can be easily used to extend lower bounds for new and old problems, allowing to cover also the indexing case. To make this possible, we have to properly set up a formal framework of definitions, of which the two main pieces are the *linear independent-components* (*lic*) reduction and the property stated by Lemma 6.1 below.

As we mentioned, the strategy for obtaining the lower bound is to partition an OV instance into smaller OV sub-instances of the right size, and then to show that an index for these sub-instances built in polynomial time providing subquadratic time queries would lead to a contradiction with OVH. The way to extend this result to SMLG is to use the reduction that we already know exists from OV to SMLG, which allows us to convert the OV sub-instances into SMLG instances. Then, if we assume to have a polynomial index providing subquadratic time queries for SMLG, the same holds for those OV sub-instances, contradicting OVH. If this scheme works for SMLG, it should work also for all of those problems that have a reduction from OV. This make sense intuitively, but we have to be careful, because not all types of reductions are compatible with our reasoning.

The idea of the *lic*-reduction is to properly formalize which properties a reduction from OV to problem P must have in order to make the indexing lower bound hold for P automatically, without further adjustments. Intuitively, we enforce that the two vectors sets in OV must be addressed separately, that is the structures built on top of them must not depend on both sets. Moreover, we want to give the tools to properly address the parameter d referring to the length of the vectors, and that is, why we provide a parametrized definition of *lic*-reduction.

Definition 6.1 (Linear Independent-Components (*lic*) Reduction) *Problem A has a* linear independent-components *(lic) reduction with parameter k to problem B, indicated as $A \leq^k_{lic} B$, if the following two properties hold:*

i) ***Correctness****: There exists a reduction from A to B modeled by functions r_x, r_y and s. That is, for any input (a_x, a_y) for A, we have $r_x(a_x) = b_x$, $r_y(a_y) = b_y$, (b_x, b_y) is a valid input for B, and s solves A given the output $B(b_x, b_y)$ of an oracle to B, namely $s(B(r(a_x), r(a_y))) = A(a_x, a_y)$.*

ii) ***Parameterized linearity****: Functions r_x, r_y and s can be computed in linear time in the size of their input, multiplied by $k^{O(1)}$.*

The usefulness of a *lic*-reduction resides in its ability to implicitly link the indexability of two problems. In other words, if we have problems A and B, and we have an index for B, then a *lic*-reduction allows us to use the same index also for A. We present this property with the following lemma, in which a problem is called (α, δ, β)-*polynomially indexable with parameter* k if its input can be divided in two parts p_x and p_y, and an index built in time $O(k^{O(1)}|p_x|^{\alpha})$ can answer queries in time $O(k^{O(1)}(|p_y| + |p_x|^{\delta}|p_y|^{\beta}))$. If this is true only when $k = O(1)$, then we just say that the problem is (α, δ, β)-*polynomially indexable*.

Lemma 6.1 *Given problems A and B and constants $\alpha, \beta, \delta > 0$, if it holds that $A \leq^{k}_{lic} B$, and B is (α, δ, β)-polynomially indexable, then A is (α, δ, β)-polynomially indexable with parameter k.*

In Paper II we give a formal proof of this lemma, but it is indeed a natural result. The idea is that every time we are presented a query for problem A, we first use the reduction to convert it into a query for problem B, and then we use the index built for B to answer such a query.

The definition of *lic*-reduction, combined with Lemma 6.1, can be used as a black box to transfer the indexing lower bound for OV to many problems that have a reduction from it. What we have to show now is how to prove this indexing lower for OV in the first place. There was already a preliminary folklore result in this sense, but we extended it to provide the tools for proving even tighter lower bounds. Since the focus of this thesis is on SMLG, we shall see how to use this technique to provide an indexing lower bound for this problem. Nonetheless, we also discuss how to apply the technique to edit distance, and which indexing lower bound we can obtain for it.

6.2 Indexing Lower Bound for OV

To make our technique work, we need to prove that if OV is (α, δ, β)-polynomially indexable with parameter d, where d is the length of the vectors, then OVH is false. A first version of this result with a specific condition on parameters δ and β is known as folklore knowledge.

Theorem 6.1 (Folklore) *If OV is (α, δ, β)-polynomially indexable with parameter d, and $\beta + \delta < 2$, then OVH fails.*

We extended the condition on parameters δ and β to prove tighter lower bounds, but for this we first need to introduce a more general version of OV, that we call (N, M)-OV.

Problem 6.1 (**(N, M)-OV**)

Input: *Two sets $X, Y \subseteq \{0,1\}^d$, such that $|X| = N$ and $|Y| = M$.*

Output: *True if and only if there exists $(x, y) \in X \times Y$ such that $x \cdot y = 0$.*

The difference from standard OV is that in (N, M)-OV the two vector sets can have different sizes. This feature is necessary for softening the condition on parameters δ and β, raising up the lower bound.

Theorem 6.2 *If (N, M)-OV is (α, δ, β)-polynomially indexable with parameter d, and either $\delta < 1$ or $\beta < 1$, then OVH fails. That is, under OVH, we cannot support $O(N^\delta M^\beta)$-time queries for (N, M)-OV, for either $\delta < 1$ or $\beta < 1$, even after polynomial-time preprocessing.*

Here, performing a query means telling whether a set of M vectors has a vector orthogonal to one of the N vectors on which we built the index. As we anticipated earlier, the idea behind proving this lower bound is reducing an instance of OV, with n vectors of length d, to many instances of (N, M)-OV. Thus, vector set X is divided into $\lceil \frac{n}{N} \rceil$ sub-sets of vectors, each one containing N vectors; analogously, vector set Y is divided into $\lceil \frac{n}{M} \rceil$ subsets, each consisting of M vectors. Figure 6.1 gives an intuition of this reasoning. Assuming that we can index (N, M)-OV in time $O(N^\alpha)$ to answer queries in time $O(N^\delta M^\beta)$, we can now achieve a subquadratic-time algorithm for the original OV instance. To give an intuition of why this works, we present the following example. Suppose that sets X and Y of n vectors each are an instance of OV, and assume that we can index (N, M)-OV in time $O(N^4)$ and answer queries in time $O(N^{\frac{1}{2}} M^{\frac{1}{2}})$. Our goal is to solve the original OV instance of n vectors in time $O(n^{2-\epsilon})$, for some $\epsilon > 0$. We apply the reduction scheme that we discussed above, splitting set X into $\lceil \frac{n}{N} \rceil$ subsets X_i, and set Y into $\lceil \frac{n}{M} \rceil$ subsets Y_j. Now, each pair (X_i, Y_j) is an instance of (N, M)-OV, and if we solve all of them we get an answer for the original OV instance. Thus, we first build an index on each X_i, and then we perform one query for each pair (X_i, Y_j). We have $\lceil \frac{n}{N} \rceil$ indexes to build, so this takes time $O(N^4 \lceil \frac{n}{M} \rceil)$, and we have $\lceil \frac{n}{N} \rceil \lceil \frac{n}{M} \rceil$ queries to answer, which takes time $O(\lceil \frac{n}{N} \rceil \lceil \frac{n}{M} \rceil N^{\frac{1}{2}} M^{\frac{1}{2}})$. The key is to make the vector subsets of the right size, so that the overall time complexity results will be subquadratic in n. For this specific example a good choice is $N = n^{\frac{1}{4}}$ and $M = n^{\frac{1}{2}}$ (not the only one, for instance $M = n$ works as well). If we substitute this values for N and M in the time complexities for indexing and querying we obtain $O(n^{\frac{7}{4}} + n^{\frac{7}{8}}) = O(n^{\frac{7}{4}}) = O(n^{2-\frac{1}{4}})$, that is, $\epsilon = \frac{1}{4}$. In Paper II, we show that given parameters α, δ and β we can always choose N and M such that the overall time complexity for the original OV instance results will be subquadratic, and we provide a formula for doing so.

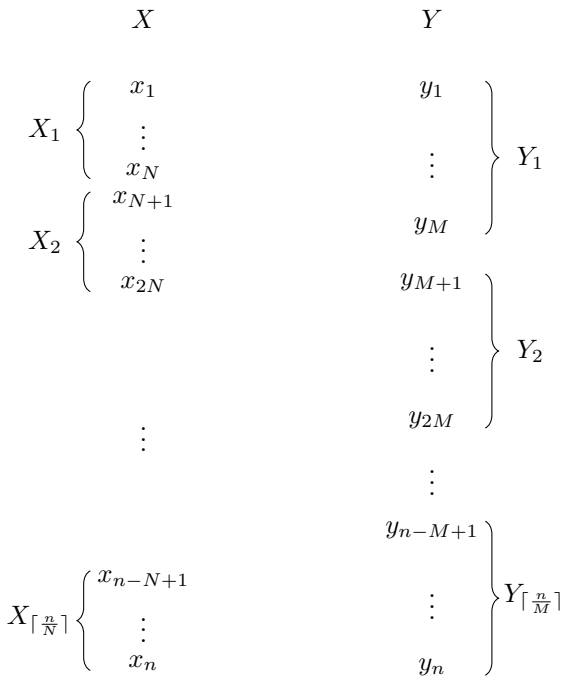

Figure 6.1: The splitting of an instance of OV into many sub-instances of (N, M)-OV.

6.3 Indexing Lower Bound for SMLG and EDIT

Combining the definition of *lic*-reduction, Lemma 6.1 and Theorem 6.1, we have the technique that we wanted for proving lower bound. Now, given any problem with a reduction from OV, it suffices to prove that that is a *lic*-reduction to obtain the indexing lower bound. If we also want the tighter lower bound, then we have to make the *lic*-reduction work from (N, M)-OV as well. For some problems, all of these steps are very simple or even not needed, as their original reductions already respect all of the constraints. For other problems, few adjustments are needed. Let us see how to use this technique on SMLG, for which we have to add some edges in the reduction graph to tighten the lower bound for cyclic graphs. Then, to see how the technique generalises to other problems, we also apply it to EDIT.

Looking at our reduction from OV to SMLG presented in Chapter 5, we can easily verify that it is indeed a *lic*-reduction. Starting from vectors sets X and Y, we defined pattern string P using solely X, thus this definition implements function $r_x(X) = P$. Similarly, the construction of graph G, performed using only Y, realises function $r_y(Y) = G$. Finally, function $s(\mathsf{SMLG}(P, G))$ returns $True$ when $\mathsf{SMLG}(P, G) = True$, that is, when P has a match in G, which we know to be a correct answer also for OV thanks to the correctness of our reduction. Parameter k in this case is the length d of the vectors, pattern P contains n substrings of length $O(d)$, and graph G consists of $5n - 4 = O(n)$ gadgets each of size d. Thus, our reduction is performed in time $O(nd)$, which is linear in n times $k = d$, and we achieve the following result.

Theorem 6.3 *For any $\alpha, \beta, \delta > 0$ such that $\beta+\delta < 2$, there is no algorithm preprocessing a labelled graph $G = (V, E, \ell)$ in time $O(|E|^\alpha)$ such that for any pattern string P we can solve the SMLG problem on G and P in time $O(|P| + |E|^\delta |P|^\beta)$, unless OVH is false. This holds even if restricted to a binary alphabet, and to deterministic DAGs in which the sum of out-degree and in-degree of any node is at most three.*

Difficulties arise when trying to extend this reduction to (N, M)-OV. In this case we have to address vector sets X and Y of variable length N and M, respectively. If $N \geq M$, the reduction still works with no modifications, because the graph can always accommodate for shorter patterns, but the case $N < M$ poses a problem. The reduction cannot work when the pattern has more characters than the graph has nodes, thus a restructuring is needed. In particular, allowing for cycles to be present in the graph seemed to us an unavoidable consequence. This can be done by adding two

back-edges in the reduction, creating a cycle in the universal gadgets able to account for any number of overflowing subpatterns. We thus achieve the stronger lower bound.

Theorem 6.4 *For any $\alpha, \beta, \delta > 0$, with either $\beta < 1$ or $\delta < 1$, there is no algorithm preprocessing a labelled graph $G = (V, E, \ell)$ in time $O(|E|^\alpha)$ such that for any pattern string P we can solve the SMLG problem on G and P in time $O(|P| + |E|^\delta |P|^\beta)$, unless OVH is false.*

It remains open whether this tighter lower bound can be achieved also for DAGs.

To show how to generalize our technique, we would like to provide a lower bound also for EDIT, but we have to make some considerations first. A reduction from OV to EDIT has been provided by Backurs and Indyk [10], but in that reduction the construction of one of the two edit-distance strings depends on both vector sets of OV. Therefore, that is not a *lic*-reduction. This construction choice seems made from convenience and not from necessity, hence we could try to modify the reduction to make it *lic*. However, in an indexing context, it makes more sense to solve a slightly different version of EDIT, where we build the index on a long text and then answer queries for shorter strings. In this setting, a query for pattern string P in text T asks to find substring x of T at minimum edit distance with P, namely $edit(x, P)$ is minimized. Thus, we consider it more useful to find a reduction to this version of EDIT. In their work [10], Backurs and Indyk refer to this version of the problem as PATTERN, and they use it as an intermediate step towards their final result. If we stop the reduction at this step, the construction of the text string depends only on one vector set, while the pattern string depends only on the other vector set; plus, PATTERN is proven to have a solution under a certain threshold if and only if OV has a pair of orthogonal vectors. That is, we have a *lic*-reduction from OV to PATTERN that can be performed in linear time, multiplied by $d^{O(1)}$, and thus we achieve an indexing lower bound.

Theorem 6.5 *For any $\alpha, \beta, \delta > 0$ such that $\beta + \delta < 2$, there is no algorithm preprocessing a string T in time $O(|T|^\alpha)$, such that for any pattern string P we can find a substring of T at minimum edit distance with P, in time $O(|P| + |T|^\delta |P|^\beta)$, unless OVH is false.*

Chapter 7

Founder Block Graphs

In Chapters 5 and 6 we showed that it is hard to solve SMLG with a subquadratic time complexity, no matter if we are considering the exact or approximate variant, and regardless of the data structures that we are using for indexing. Thus, if we want to have efficient solutions, we must make stronger assumptions and look at special cases of the problem. In this spirit, we focus on graphs that are meant to represent a specific type of collection of strings, with the goal of identifying a class of graphs for which SMLG is easier to solve. This type of graphs is heavily used in *pangenomics*, an area of bioinformatics that aims to represent multiple similar genome sequences as compactly as possible, while still retaining information on their individual variations. Moreover, solving SMLG for these graphs can, more in general, provide additional knowledge on how to handle collections of similar strings, for the type of queries specific to SMLG.

To understand the nature of the graphs we mentioned, let us start from the concept of *multiple sequence alignment* (MSA). An MSA is a collection of strings, or sequences, of the same length, usually represented by placing the strings on top of each other, forming a matrix of characters (see Figure 7.1). Normally, we say that a string has a match in an MSA when it equals a

```
A G C G A C T A G A T A C
A G C T A C T A G A T A G
A G C G A T T A G T T A C
A G C T A C T A G T T A C
```

Figure 7.1: A multiple sequence alignment MSA$[4, 13]$ of 4 strings, with 13 columns.

substring of one of the strings in the MSA. However, a feature that we are interested to allow is *recombination*, that is the possibility for a string to "jump" from a row to another at specific locations to complete a match. Moreover, MSAs tend to be redundant structures, where the same substring can appear in multiple rows at the same column position. Thus, we have reasons to look for alternative ways of representing MSAs, possibly in a more compact manner. One initial solution is *founder sequences*.

Given an MSA, its founder sequences are a set of strings, of the same length of the sequences in the MSA, and a set of column positions called *discontinuities*. The key property is that we must be able to reconstruct every sequence of the MSA by scanning a founder sequence from left to right, possibly switching to another sequence at a discontinuity. Founder sequences are typically much fewer in number than the strings in the original MSA [43], and even though finding an optimal set of founders is NP-*hard* [37], reasonable approximations are possible [18, 36]. When looking for a match, a query string can jump from one sequence to another when crossing a discontinuity. In this way, the set of virtually represented sequences is much larger than the founder sequences themselves, and includes all the sequences of the MSA. Thus, we both reduced the space needs and achieved recombination.

Another possible solution to obtain this goal is to represent the MSA with a *variation graph*, where different substrings are represented as different paths in a graph. Variation graphs offer a better way to control recombination than founder sequences. This is because discontinuities always allow switching from one sequence to any other sequence, allowing even for too much recombination, while in variation graphs we could simply drop the edges that we do not want. Unfortunately, general variation graphs fall into those categories of graphs for which our lower bounds hold.

Given this situation, our work proposes a new type of graphs called *founder block graphs*, which are meant to solve SMLG in subquadratic time, while offering controlled recombination. To define founder block graphs, we first need to introduce *block graphs*. A block graph is a labelled directed acyclic graph consisting of consecutive blocks, where a block represents a set of sequences of the same length as parallel (unconnected) nodes. There are edges only from nodes of one block to the nodes of the next block. A founder block graph is a block graph with each block representing the segments of founder sequences in between discontinuities. Let us fix all of these concepts in more formal definitions.

Let $\mathcal{P}$ be a *partitioning* of $[1..n]$, that is, a sequence of subintervals $\mathcal{P} = [x_1..y_1], [x_2..y_2], \ldots, [x_b..y_b]$, where $x_1 = 1$, $y_b = n$, and for all $j > 2$,

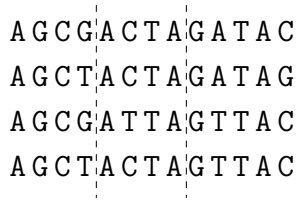

Figure 7.2: A possible segmentation for the MSA in Figure 7.1.

$x_j = y_{j-1} + 1$. A *segmentation* S of $\mathsf{MSA}[1 \dots m, 1 \dots n]$ based on partitioning $\mathcal{P}$ is a sequence of b sets $S^k = \{\mathsf{spell}(\mathsf{MSA}[i, x_k..y_k]) \mid 1 \leq i \leq m\}$ for $1 \leq k \leq b$; in addition, we require for a (proper) segmentation that $\mathsf{spell}(\mathsf{MSA}[i, x_k..y_k])$ is not an empty string for any i and k. We call set S^k a *block*, while $\mathsf{MSA}[1..m, x_k..y_k]$ or just $[x_k..y_k]$ is called a *segment*. The *length* of block S^k is $L(S^k) = y_k - x_k + 1$ and the *width* of block S^k is $W(S^k) = |S^k|$.

Definition 7.1 (Block Graph) *A* block graph *is a graph $G = (V, E, \ell)$ where $\ell : V \to \Sigma^+$ is a function that assigns a string label to every node and for which the following properties hold.*

1. *Set V can be partitioned into a sequence of b* blocks *$V^1, V^2, \dots, V^b$, that is, $V = V^1 \cup V^2 \cup \dots \cup V^b$ and $V^i \cap V^j = \emptyset$ for all $i \neq j$;*
2. *If $(v, w) \in E$ then $v \in V^i$ and $w \in V^{i+1}$ for some $1 \leq i \leq b - 1$; and*
3. *if $v, w \in V^i$ then $|\ell(v)| = |\ell(w)|$ for each $1 \leq i \leq b$ and if $v \neq w$, $\ell(v) \neq \ell(w)$.*

With *gapless* MSAs, that is MSAs with no gap characters, block S^k equals segment $\mathsf{MSA}[1..m, x_k..y_k]$, and in that case *founder graph* $G(S)$ is a block graph induced by segmentation S. The idea is to have a graph in which the nodes represent the strings in S while the edges retain the information of how such strings can be recombined to spell any sequence in the original MSA. Figures 7.2 and 7.3 show an example of this. In our work, we show that for a special subclass of founder block graphs it is indeed possible to solve SMLG in linear time, and we also provide a linear time algorithm to construct such graphs starting from an MSA.

7.1 Repeat-Free Founder Block Graph

We now introduce a concept that constitutes the core of efficient indexability for founder block graphs. To make our notation consistent across the third and fourth paper, we already use the notation of the latter here.

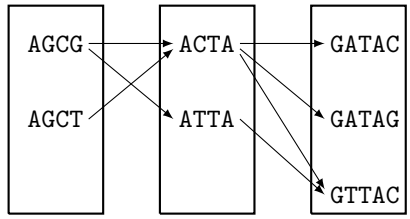

Figure 7.3: The founder block graph induced by the segmentation in Figure 7.2.

Definition 7.2 *Founder block graph $G(S)$ is* repeat-free *if each $\ell(v)$ for $v \in V$ occurs in $G(S)$ only as prefix of paths starting with v.*

This is the definition of what we call *repeat-free* property, which states that each node label can appear only once in the entire founder block graph. The primary function of this property is to tailor the matching algorithm to which node a match could cross. As our reductions have shown, lacking this information is the main problem in solving SMLG, even for deterministic DAGs. From an automata-theory point of view, we could say that the absence of a start state is a source of non-determinism. The repeat-free property handles this inconvenience by making each node label a unique identifier. Let us now see how to use this fact to our advantage for developing a matching algorithm.

7.2 Indexing Repeat-Free Founder Block Graphs

The first matching algorithm we propose consists of two steps, employing an Aho-Corasick automaton [3] for the first and tries for the second. In the first step, we build an Aho-Corasick automaton of all node labels $\ell(v)$, and then we use it to scan query string Q. Thus, we now know which labels $l(v)$ appear as a substring of Q, and we can keep a reference to nodes v to locate them in the graph. The nodes that we found in this way are potential matches, and we now have to verify which ones extend to actual matches. This is where step two starts. For each node v, we consider all labels $\ell(w)$ such that $(v, w) \in E$, and we build trie $F(v)$ on top of them. Similarly, we consider all labels $\ell(u)^{-1}$, that is, the reverse of $\ell(u)$, such that $(u, v) \in E$, and we build trie $R(v)$ on top of them. Assume that substring $Q[i..j]$ matches label $\ell(v)$. To verify if the remaining right part $Q[i..j]$ can extend to a full match, we can visit the tries on the left and on the right of the block containing $\ell(v)$. If we can match $Q[1..i]$ in the tries on the left and $Q[j..|Q|]$ in the tries on the right, then we have a full match for Q;

if we cannot, then $Q[i..j]$ cannot extend to a full match, and we have to choose another candidate.

Unfortunately, following this strategy, the time complexity remains quadratic in the query string length. This is because visiting all the tries for a candidate $\ell(v)$ takes $O(|Q| \log \sigma)$ time, where σ is the size of the alphabet, but we have to repeat this process for each candidate $\ell(v)$, which at worst can be as many as $O(|Q|)$. A query time complexity of $O(|Q|^2 \log \sigma)$ is not exactly what we are looking for, but the situation is better than it seems. In Paper III, we show how BWT-based techniques can yield linear time matching algorithms without the need of building the tries, but before that we explain that even the current approach works. Indeed, the missing piece for achieving linear time queries is observing that matches in repeat-free founder block graphs are always unique.

Property 7.1 *Let $G(S) = (V, E, \ell)$ be a repeat-free founder block graph. If string Q has a match in $G(S)$ spanning at least one entire node label, Q matches nowhere elses in the graph.*

Proof. By contradiction, assume that Q matches at least two different path labels in the graph, and let us call such paths *matching paths*. Consider one matching path and substring $Q[i..j]$ matching $\ell(v)$, for some $v \in V$ in that path. Now consider any other matching path, in which $Q[i..j]$ matches in another position in the graph. We were assuming a repeat-free founder block graph, but we have found that $\ell(v)$ appears more than once in the graph, a contradiction. □

Thanks to this property, we can conclude that it is enough to try to extend one arbitrary candidate $Q[i..j] = \ell(v)$ to a full match since, if it exists, that match is unique.

7.3 Construction of Repeat-Free FBGs from Gapless MSA

Now that we know that it is possible to index a repeat-free founder block graph for linear time queries, we design a method for building such a graph starting from a segmentation. In order to do so, we need some guarantees on the graph that we build based on the segmentation from which we start.

Lemma 7.1 (Characterization) *Let $\mathcal{P} = [x_1..y_1], [x_2..y_2], \ldots, [x_b..y_b]$ be the partitioning corresponding to a segmentation S inducing a block graph $G = (V, E)$. The segmentation S is valid if and only if, for all blocks $V^i \subseteq V$, $1 \leq t \leq m$ and $j \neq x_i$, if $v \in V^i$ then $\mathsf{MSA}[t, j \ldots j + |\ell(v)| - 1] \neq \ell(v)$.*

Intuitively, we are saying that making the segmentation repeat-free is enough to make the induced founder graph repeat-free. To check this fact, we need to make sure that each string in a segment appears nowhere else as a substring in the MSA.

The next algorithm that we present outputs a valid (repeat-free) optimal segmentation given an MSA. In doing so, we have to define what we mean by optimal segmentation. One can use many different objective functions to fix this concept, and we start with minimizing the maximal segment length. The main tool that we employ are values $v(j)$, computed for $1 \leq j \leq n$. Value $v(j)$ is the greatest integer such that segment $\mathsf{MSA}[1..m, v(j)+1..j]$ is valid (thus, it might remain undefined for small j). If such $v(j)$ does not exist for some j, we set $v(j) = 0$. That is, given position j, value $v(j)$ tells which is the smallest valid segment that ends at j.

We could look at this from the reverse point of view, and define a value that tells us which is the smallest valid segment starting at position j. These values are called $f(j)$, and we employ them to develop efficient solutions for building segmentations of MSA with gaps. In this chapter, we present preliminary techniques for the gapless case using values $v(j)$, and we limit ourselves to explain how to preprocess these values in time $O(mns_{\mathtt{max}} \log \sigma)$, and how to use them to compute an optimal segmentation in time $O(ns_{\mathtt{max}})$, where $s_{\mathtt{max}}$ is the greatest score among the the scores computed for $\mathsf{MSA}[1..m, 1..j]$, $1 \leq j \leq n$. In the next chapter, we improve our results changing the approach and developing segmentation algorithms for the gapped case using values $f(j)$. The next chapter also covers the case of the maximum number of blocks as objective function.

We first assume to have already computed values $v(j)$, and we use them to compute an optimal segmentation. Then, we explain how to compute them. From the following recursion, we can derive a dynamic programming algorithm for computing the optimal segmentation.

$$s(j) = \min_{j':1 \leq j' \leq v(j)} \max(j - j', s(j'))$$

To compute the score $s(n)$ of the optimal segmentation of the MSA, we start comparing $\max(j - j', s(j'))$ from $j' = v(j)$, decreasing j' by one. Observe that function $s(j')$ is monotonically increasing, which actually is the reason why we can use dynamic programming. Instead, value $j - j'$ is monotonically decreasing, thus there must be position j^* and corresponding value s^* such that $j - j' > s(j')$ for $j' < j^*$, $j - j' = s(j') = s^*$ for $j' = j^*$, and $j - j' < s(j')$ for $j' > j^*$. Notice that $j - j' > s^*$ for $j' < j^*$, and $s(j') > s^*$ for $j' > j^*$, thus

$$s(j) = \min_{j':1 \leq j' \leq v(j)} \max(j - j', s(j')) = s^*.$$

This means that, starting from $j' = v(j)$ and decreasing j', we can stop as soon as we find that $j - j' > s(j')$, because we are sure to have computed $s(j) = s^*$ at the previous step. Since $j - j'$ starts at $j - v(j)$ and increases by one at each step until it reaches $s(j) = s^*$, this procedure takes $O(s(j))$ time, and the overall time complexity is then $O(ns_{\max})$.

For preprocessing values $v(j)$, in Paper III we use the bidirectional BWT index [13] of the MSA. Here, we describe a more conceptual algorithm that uses standard BWT. At column j, consider the trie containing the reverse of the rows of $\mathsf{MSA}[1..m, 1..j]$. If, in this trie, we follow a path with label p until reaching some node u, the number of leaves in the subtree rooted at u equals the number of times string p appears as a row of segment $\mathsf{MSA}[1..m, j - |p| + 1..j]$. Let this number be k. If the BWT interval relative to string p is of length k, then p appears exactly k times in the entire MSA as rows of $\mathsf{MSA}[1..m, j - |p| + 1..j]$, making the segment valid. If the BWT interval is larger than k, then string p appears also somewhere else in the MSA, and the segment is not valid. Thus, we choose as $v(j)$ the closest column to j such that the number of leaves in each trie subtree equals the length of the corresponding BWT interval. We can search in the BWT and partially construct the tries in parallel so that we do not have to actually reach the leaves, but anyway $O(m(j - v(j)) \log \sigma)$ time has to be spent for each column. As $j - v(j) \leq s(j)$, the overall time complexity is $O(mns_{\max} \log \sigma)$. Paper III improves over this approach so that both preprocessing and the final evaluation of the score take (randomised) $O(mn)$ time.

Chapter 8

Elastic Founder Graphs

Paper III is first of all a proof of concept: we wanted to find at least one class of graphs admitting linear time queries, and in doing so we studied graphs with meaningful applications in bioinformatics. Nevertheless, there are limitation to the approaches that we proposed that we would like to surpass.

The main issue that we have to handle are gaps. Assuming to have a gapless MSA is definitely too restrictive a requirement, from both a theoretical and practical point of view. This model can manage only collections of strings of the same length, where their similarity is measured only on the basis of single-position mismatches, rather than by using more popular metrics as the (multiple) longest common subsequence. In bioinformatics applications, insertions and deletions are common features among genomes of different individuals of the same species. In this chapter, we improve on this aspect by allowing gaps in the MSA, which in turn leads to the definition of a different and more general type of graph, called *Elastic Founder Graph* (EFG). As we will see, we need to develop new techniques to achieve linear time queries in EFGs.

Another feature that might seem too demanding is the repeat-free property. Anyway, some additional assumption on the structure of the graph has to be made, as for exact SMLG in generic EFGs we prove a quadratic lower bound, conditioned on OVH and SETH. Nonetheless, we also try to relax this constraint as much as possible, introducing the notion of *semi-repeat freeness*. To set the basis for this chapter, we formally define the concept of EFG, exemplified in Figure 8.1.

Definition 8.1 (Elastic block and founder graphs) *Recall the definition of* block graph *in Chapter 2. We call a block graph* elastic *if its third condition is relaxed in the sense that each V^i can contain non-empty*

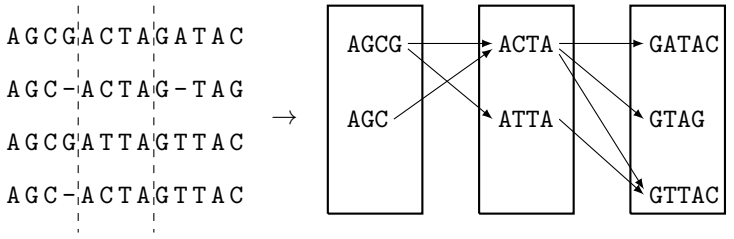

Figure 8.1: An EFG obtained from the segmentation of an MSA with gaps.

variable-length strings. An elastic founder graph *(EFG) is an elastic block graph* $G(S) = (V, E, \ell)$ induced *by a segmentation* S *as follows: For each* $1 \leq k \leq b$ *we have* $S^k = \{\mathsf{spell}(\mathsf{MSA}[i, x_k..y_k]) \mid 1 \leq i \leq m\} = \{\ell(v) : v \in V^k\}$. *It holds* $(v, w) \in E$ *if and only if there exists* $k \in [1..b-1]$ *and* $t \in [1..m]$ *such that* $v \in V^k$, $w \in V^{k+1}$ *and* $\mathsf{spell}(\mathsf{MSA}[t, x_k..y_{k+1}]) = \ell(v)\ell(w)$.

8.1 Conditional Hardness of EFGs

To justify the use of the (semi)-repeat-free property for EFGs, we prove that, without it, exact SMLG is a quadratic problem under OVH, and thus SETH. Of course, this involves finding a reduction from OV to exact SMLG on EFGs, but the overall idea and structure of the graph are not too different from what we already saw in Chapter 5. Thus, we describe the reduction only at a high level, leaving the details to the full paper, and we concentrate instead on highlighting the fundamental feature that makes the graph construction possible.

Consider the graph that we employed in the reduction in Chapter 5 structured on three rows; we want to transform it into an EFG that still retains the same properties. The important features are the three-rows structure and the special labels that force the beginning and the end of a pattern match in specific positions (top and middle row for the beginning, middle and bottom row for the end). What we need to do is to structure the graph in blocks, and the first idea is to divide the nodes of the original graph in logical one-character columns, and place all the node labels in a column in the same block as one-character nodes. The problem with this approach is that, when there are two nodes with the same label in the same logical column, they "collapse" together into the same node when placed into the same block.

To avoid this, we exploit the fact that, in an EFG, strings of different length can live together in the same block as different strings. For example,

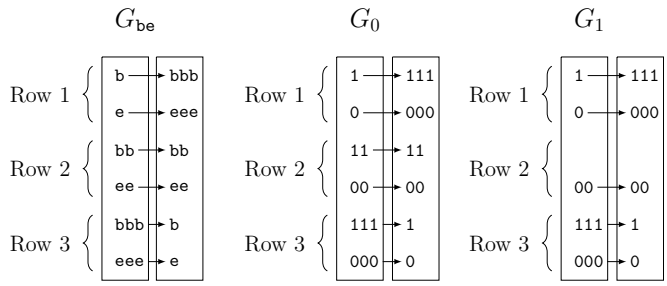

Figure 8.2: The structure of the gadgets used in the reduction from OV to exact SMLG. Notice how we can form the same strings following different paths in different rows.

we can have b and bb as node labels in the same block. If we look at the logical columns in the original graph, we notice that the same character never appears more than three times in the same column. For instance, character b appears at the beginning of $G_{U1}^{(n-1+j)}$, of $G_W^{(j)}$ and of $G_{U2}^{(j-1)}$. The trick here, depicted in Figure 8.2, is to use two blocks to encode one of the logical columns. Looking at character b, in the first block we place strings $\mathtt{b}^{(1)}$, $\mathtt{bb}^{(1)}$ and $\mathtt{bbb}^{(1)}$, from top to bottom; in the second block we place strings $\mathtt{bbb}^{(2)}$, $\mathtt{bb}^{(2)}$ and $\mathtt{b}^{(2)}$, from top to bottom (the number in the exponent refers to the first or second block). Then, we connect these strings in a row-wise manner, meaning that we connect $\mathtt{b}^{(1)}$ with $\mathtt{bbb}^{(2)}$, $\mathtt{bb}^{(1)}$ with $\mathtt{bb}^{(2)}$ and $\mathtt{bbb}^{(1)}$ with $\mathtt{b}^{(2)}$. We now have three logical rows such that each pair of strings spells bbbb, and we could say that we prevented the representations of character b from collapsing together. It is possible to apply the same trick to all characters in all logical columns, and thus obtain an EFG with the same properties of the original graph. Of course, this works because we change the pattern accordingly, that is, we repeat each character four times. Since this reduction mimics our original reduction which is a *lic*-reduction, both online and indexing hardness results for exact SMLG on EFGs follow.

Theorem 8.1 *For any constant $\epsilon > 0$, it is not possible to find a match for a query string Q into an EFG $G = (V, E, \ell)$ in either $O(|E|^{1-\epsilon}\,|Q|)$ or $O(|E|\,|Q|^{1-\epsilon})$ time, unless OVH fails. This holds even if restricted to an alphabet of size 4.*

Theorem 8.2 *For any $\alpha, \beta, \delta > 0$ such that $\beta+\delta < 2$, there is no algorithm preprocessing an EFG $G = (V, E, \ell)$ in time $O(|E|^{\alpha})$ such that for any query string Q we can find a match for Q in G in time $O(|Q| + |E|^{\delta}|Q|^{\beta})$, unless OVH is false. This holds even if restricted to an alphabet of size 4.*

8.2 Indexing (Semi-)Repeat-Free EFGs

Given that EFGs are hard to query and index in the general case, let us impose an additional property so that we can take advantage of it in designing our algorithms. We saw in Chapter 7 that we can use the repeat-free property for this purpose, but we would also like to see if we can relax this requirement and still achieve the same results. For this reason, we present the semi-repeat-free property.

Definition 8.2 *EFG $G(S)$ is* semi-repeat-free *if each $\ell(v)$ for $v \in V$ occurs in $G(S)$ only as prefix of paths starting with $w \in V$, where w is from the same block as v.*

For example, the EFG in Figure 8.1 is semi-repeat-free. We show indexing and querying algorithms for semi-repeat-free EFGs; clearly, they work for repeat-free EFGs as well.

For indexing EFG $G = (V, E, \ell)$, we first build string

$$D = \Pi_{i \in \{1,2,\ldots,b\}} \Pi_{v \in V^i, (v,w) \in E} \ell^{-1}(w)\ell^{-1}(v)\$,$$

where $\ell^{-1}(v)$ is the reverse of $\ell(v)$, and thus $\ell^{-1}(w)\ell^{-1}(v)$ is the reverse of $\ell(v)\ell(w)$. When constructing D, what we are doing is scanning all the edges $(v, w) \in E$, concatenating $\ell(v)$ and $\ell(w)$, reversing it, and adding a \$ at the end. Finally, we concatenate together all the strings obtained this way into string D. Then, we build the suffix tree of D. This takes linear time in $|D|$ for a constant alphabet [25, 33, 42, 44]. Queries can now be answered by reading the query string backwards, following the corresponding path in the suffix tree, and taking suffix links when we reach a \$.

We explain this procedure in more detail with an example. Consider the EFG of Figure 8.1, and assume that we have built the corresponding string D, and its suffix tree. First, we search every reversed node label in the suffix tree and mark the suffix-tree node that we reach in this way. Then, suppose that we want to find a match for string $Q =$ `CGATTAGT`. We traverse down the suffix tree of Figure 8.3 matching `TGATTA`, then we find a \$, which tells us that this is the beginning of a node label. In fact, there are nodes v and w in the EFG such that $\ell^{-1}(w)\ell^{-1}(v)\$ =$ `CATTGATTA$`, of which `TGATTA$` is a substring, represented by a path in the suffix tree. Looking at the EFG, this means that we started the match in the middle of node label `GTTAC`, and we reached the beginning of label `ATTA` of its in-neighbour. To continue our match, we need to find the suffix path that starts with label $\ell^{-1}(v) =$ `ATTA`. We can do so by following suffix links until we find a marked node. In this example, we follow two suffix links, removing

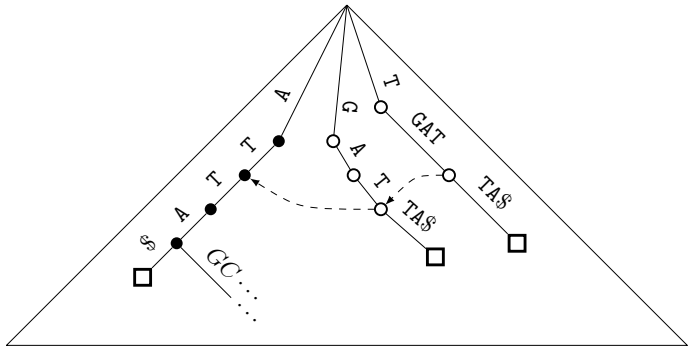

Figure 8.3: The search of string Q = CGATTAGT in the suffix tree of string D for the EFG of Figure 8.1. Inner nodes are round, leaf nodes are squared, the marked node is black. After matching substring TGATTA, we follow suffix links until reaching the marked node that we know represents a node label of the EFG. From here, we descend, completing the match with CG.

characters TG from the beginning of the suffix path and thus discovering the suffix node relative to suffix path AT. This is almost what we wanted, we just have to descend down the path until we find a \$, and then step back up by one node. Now we can be sure that the current node has label ATTA $= \ell^{-1}(v)$. Notice that this is correct because of the semi-repeat-free property. Indeed, all the children of the node with label $\ell^{-1}(v)$ = ATTA are either a \$ branch or branches representing in-neighbours of v in the EFG. This is ensured by the fact that ATTA is unique in the graph, except as a prefix of a node label in the same block, and also because we built the suffix tree with the reverses of node labels, which keeps eventual prefixes in different suffix paths. At this point, we can follow the children whose label starts with G and complete the match finding GC. If the match spanned even more EFG nodes, we would repeat this process of following suffix links until we found a match or we could not continue.

8.3 Constructing Semi-Repeat-Free EFGs from a Gapped MSA

As we did in the previous chapter, we build our EFG in two steps: a preprocessing step and an actual construction step. In the preprocesing step, this time we compute, for each position j, forward value $f(j)$, that is the closest position to j such that segment $[j..f(j)]$ is valid. Clearly, there exists position J after which $f(j)$ for $j > J$ remains undefined.

The choice of using values $f(j)$ over values $v(j)$ is deeply linked with the semi-repeat-free property. If, when indexing and querying EFGs, the semi-repeat-free property is viewed as a constraint relaxation over the repeat-free property, it becomes a necessity when constructing EFGs from MSAs with gaps. To understand why, consider a generic repeat-free segment of an MSA such that rows $\mathsf{MSA}[i_1, j..k]$ and $\mathsf{MSA}[i_2, j..k]$ spell the same string R. If we extend the segment one position to the right, the first row could become $R\mathtt{A}$, while the second could become $R-$, and the segment would no longer be repeat-free. If we adopt the semi-repeat-free property, this situation does not pose a problem anymore.

If we used values $v(j)$ in combination with the semi-repeat-free property, we would have a similar problem, because we would be looking backwards to find semi-repeat-free segments. A segment with rows $\mathtt{A}R$ and $-R$ would not be semi-repeat-free, with rows $-\mathtt{A}R$ and $\mathtt{A}-R$ yes, and with rows $\mathtt{A}-\mathtt{A}R$ and $-\mathtt{A}-R$ no. Instead, using values $f(j)$, and thus looking forwards, we have that if segment with rows $R\mathtt{A}$ and $R-$ is semi-repeat-free, so it is with rows $R\mathtt{A}-$ and $R-\mathtt{A}$, and so they are all of its extensions to the right. In conclusion, every extension to the right of a semi-repeat-free segment $\mathsf{MSA}[1..m, j..f(j)]$ keeps being semi-repeat-free, and this guarantees monotonicity, crucial for correctness.

We give an intuition of how to precompute values $f(j)$, leaving the details to Paper IV. The idea is to do the reverse of what we did in Chapter 7 for values $v(j)$, namely we first build a trie of the rows from column j to n, and then we traverse it upwards starting from the leaves. We stop when the reachable leaves from a node are more than the number of rows, and then we combine this information with searches in the *generalised suffix tree* of the rows to identify the exact $f(j)$ value. We achieve a time complexity of $O(nm \log m)$ for this procedure, which a recent work improved to $O(nm)$ [39].

Once we have values $f(j)$, we can use them to find an optimal segmentation under different objective functions. In Paper IV, we provide segmentation algorithms for maximising the number of blocks and for minimising the maximum block length. The structure of the algorithm is the same for both cases, but the former involves fewer additional details, so that is the one that we chose to present here. In this case, the recursion that we want to compute is

$$s(j) = \max_{\substack{j' : 0 \leq j' < j, \\ \mathsf{MSA}[1..m, j'+1..j] \text{ is} \\ \text{semi-repeat-free segment}}} \max(s(j')+1, s(j-1)).$$

In order to efficiently compute $s(n)$, we utilise precomputed values $f(j)$. First, we sort a list of pairs $(j_1, f(j_1)), (j_2, f(j_2)), \ldots, (j_{n-J}, f(j_{n-J}))$ by second component, where J is such that $f(j_{n-J+1}), f(j_{n-J+2}), \ldots, f(j_n)$ are not defined. Assume that we have computed all the values of the recursion up to $s(j)$, and that now we want to compute $s(j+1)$. Consider the case in which there are k previous positions $j_1, j_2, \ldots, j_k$ such that $j+1 = f(j_1) = f(j_2) = \ldots = f(j_k)$. Any segmentation ending at position $j+1$ must have its last segment starting and ending at one of the positions pairs $(j_1, f(j_1)), (j_2, f(j_2)), \ldots, (j_k, f(j_k))$. This means that values $s(j_1) + 1, s(j_2)+1, \ldots, s(j_k)+1)$ are the scores of all of the possible segmentations ending at $j+1$. There is one last possible segmentation that can end at $j+1$, that is the one that we obtain by extending the block ending at j to include also the column at position $j+1$, and that maintains the same score $s(j)$. Among all of these possible choices, we chose the best scoring segmentation as $s(j+1) = \max_{1 \leq x \leq k} \max(s(j_x)+1, s(j))$. If this means adding a new block to one of the segmentations ending at $j_1, j_2, \ldots, j_k$, then we say that we open a new block; otherwise, we say that we extend the old block. In the case in which $j+1$ is not the $f(j_k)$ value of any position $0 \leq j_k \leq j$, that is, $\forall j_k \in [1, j] \,.\, j+1 \neq f(j_k)$, then we cannot open any new block, but only extend the current one, thus $s(j+1) = s(j)$.

Notice that if $j = f(j_1) = f(j_2) = \cdots = f(j_k)$, clearly $j+1 > j = f(j_k)$, which means that we can process the list of pairs $(j_x, f(j_x))$ from left to right without going back. This implies that the algorithm runs in $O(n)$.

Chapter 9

Discussion

This thesis analysed the problem of matching strings inside graphs, with the goal of giving an all-round view of lower and upper bounds. We explained why the problem is hard to solve in less than quadratic time in general, and we proposed special cases for which it is possible to work around this difficulty. We intentionally focused on the theoretical aspects of the problem, which constitute my personal contribution, while the implementation for practical bioinformatic purposes is a merit of the other authors. This work is published in four original papers.

9.1 Summary

Paper I provided a quadratic conditional lower bound for exact SMLG. This is the first lower bound proposed for the problem, and matches the complexity of the algorithms discovered 20 years earlier, making them optimal unless OVH and SETH are false. In this paper, we wanted to find the simplest graph structures for which the lower bound held, with regard to the degree of the graph and the alphabet size. We found that a binary alphabet is already enough for making the problem quadratic, and that the boundary between linear and quadratic time complexity lies between trees and 3-DDAGs, that is, graphs with a maximum sum of in-degree and out-degree 3.

Paper II provided an indexing lower bound for exact SMLG. We showed that no index built in polynomial time can provide sub-quadratic time queries for exact SMLG. We proved this result as an application of a general framework, which we devised to generalise this indexing lower bound to any problem with a linear independent-components reduction from OV. The definition that we gave of linear independent-components reduction

retroactively applies also to older problems like edit distance, providing indexing lower bounds also for them.

Paper III focused on finding a special class of graphs able to circumvent the lower bounds from Paper I and Paper II. In doing so, we decided to concentrate on graphs able to represent collections of strings. This choice was motivated by theoretical interest in expanding the knowledge on data structures such as elastic degenerate strings, and by the usefulness of such data structures in bioinformatics. We found that repeat-free founder block graphs (FBGs) built on top of gapless multiple sequence alignments (MSA) allow efficient indexing and linear time queries.

Paper IV expanded and significantly improved Paper III, defining the larger class of semi-repeat-free elastic founder graphs (EFGs), that can handle gaps in the MSA while still providing similar performances to repeat-free FBGs. In this paper we provide a complete overview of the problem, showing: a hardness result for general EFGs that motivates the introduction of the semi-repeat-free property; how to index semi-repeat-free EFGs to obtain linear time queries; how to construct semi-repeat-free EFGs from MSAs with gaps.

9.2 Future Works

Using a recent technique [2], our lower bound for exact SMLG has been improved. The new lower bound states that for every algorithm A there exists a constant c^* such that A cannot solve SMLG in time $O(\frac{|E||P|}{\log^c |E|})$ or $O(\frac{|E||P|}{\log^c |P|})$, for any $c \geq c^*$[1]. In other words, there is a limit to the number of log-factors that we can shave from the time complexity of SMLG, a possibility that our lower bound did not rule out. A possible future direction would be to understand if we can extend this result to the indexing case, maybe expanding our framework. Indeed, the new lower bound is obtained with a reduction from *Formula*-SAT, which is a more general problem than CNF-SAT.

In our quest of finding a graph structure that allows efficient indexing, we left the door open for a peculiar type of graphs. Indeed, we do not have any lower bounds for FBGs that are not repeat-free, and this leaves us wondering whether the repeat-free property is necessary when all the node labels in a block have the same length.

When we build FBGs or EFGs, we place the edges using a sort of order-1 criteria: if two node labels of consecutive blocks form a substring of a row

[1] This lower bound is conditioned on technical complexity hypothesis whose specification is out of the scope of this thesis.

in the MSA, then we place the edge. We could consider studying high order models, where we decide to place or not to place k edges at the same time. Nevertheless, we realised that, most probably, any order above the first always leads to the construction of the exact same graph.

One feature of EFGs is that SMLG is quadratic without the semi-repeat-free property, but has close to linear (or actually linear) indexing time and linear query time when adding said property. One future direction can be exploring the possible trade off, if there are any. For example, is there a class of graphs indexable in time $O(|E|^2)$ and providing queries in time $O(|P| \log |E|)$?

Instead of trying to find the best class of graphs for our needs, we could also opt for a drastic change of perspective. If, in general, our model of computation does not allow us to solve SMLG in subquadratic time, we could chose another model of computation. In this spirit, we turn our gaze towards quantum computation. This model of computation allows to represent an exponential number of values in linear time and space, offering astonishing improvements in terms of asymptotic time complexities. In our preliminary work [22], we proposed a quantum algorithm for solving exact SMLG in linear time for a special class of graphs online, without building any index. We are interested in developing this line of research, possibly extending this result to a wider class of graphs.

References

[1] Amir Abboud, Arturs Backurs, and Virginia Vassilevska Williams. Tight hardness results for LCS and other sequence similarity measures. In Venkatesan Guruswami, editor, *IEEE 56th Annual Symposium on Foundations of Computer Science, FOCS 2015, Berkeley, CA, USA, 17-20 October, 2015, Proceedings*, pages 59–78. IEEE Computer Society, 2015.

[2] Amir Abboud, Thomas Dueholm Hansen, Virginia Vassilevska Williams, and Ryan Williams. Simulating branching programs with edit distance and friends: or: a polylog shaved is a lower bound made. In Daniel Wichs and Yishay Mansour, editors, *48th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2016, Cambridge, MA, USA, June 18-21, 2016, Proceedings*, pages 375–388. ACM, 2016.

[3] Alfred V. Aho and Margaret J. Corasick. Efficient string matching: An aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975.

[4] Tatsuya Akutsu. A linear time pattern matching algorithm between a string and a tree. In Alberto Apostolico, Maxime Crochemore, Zvi Galil, and Udi Manber, editors, *Combinatorial Pattern Matching, 4th Annual Symposium, CPM 93, Padova, Italy, June 2-4, 1993, Proceedings*, volume 684 of *Lecture Notes in Computer Science*, pages 1–10. Springer, 1993.

[5] Mai Alzamel, Lorraine A. K. Ayad, Giulia Bernardini, Roberto Grossi, Costas S. Iliopoulos, Nadia Pisanti, Solon P. Pissis, and Giovanna Rosone. Degenerate String Comparison and Applications. In *18th International Workshop on Algorithms in Bioinformatics, WABI 2018, August 20-22, 2018, Helsinki, Finland, Proceedings*, volume 113 of *LIPIcs*, pages 21:1–21:14, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[6] Mai Alzamel, Lorraine A. K. Ayad, Giulia Bernardini, Roberto Grossi, Costas S. Iliopoulos, Nadia Pisanti, Solon P. Pissis, and Giovanna Rosone. Comparing degenerate strings. *Fundamenta Informaticae*, 175(1-4):41–58, 2020.

[7] Amihood Amir, Moshe Lewenstein, and Noa Lewenstein. Pattern matching in hypertext. In *Algorithms and Data Structures, 5th International Workshop, WADS'97, Halifax, LNCS 1272, Proceedings*, pages 160–173, 1997.

[8] Amihood Amir, Moshe Lewenstein, and Noa Lewenstein. Pattern matching in hypertext. *Journal of Algorithms*, 35(1):82–99, 2000.

[9] Kotaro Aoyama, Yuto Nakashima, Tomohiro I, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Faster Online Elastic Degenerate String Matching. In Gonzalo Navarro, David Sankoff, and Binhai Zhu, editors, *Annual Symposium on Combinatorial Pattern Matching (CPM 2018), Proceedings*, volume 105 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 9:1–9:10, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[10] Arturs Backurs and Piotr Indyk. Edit distance cannot be computed in strongly subquadratic time (unless SETH is false). In Rocco A. Servedio and Ronitt Rubinfeld, editors, *47th Annual ACM on Symposium on Theory of Computing, STOC 2015, Portland, OR, USA, June 14-17, 2015, Proceedings*, pages 51–58. ACM, 2015.

[11] Arturs Backurs and Piotr Indyk. Which regular expression patterns are hard to match? In Irit Dinur, editor, *IEEE 57th Annual Symposium on Foundations of Computer Science, FOCS 2016, 9-11 October 2016, Hyatt Regency, New Brunswick, New Jersey, USA, Proceedings*, pages 457–466. IEEE Computer Society, 2016.

[12] Arturs Backurs and Piotr Indyk. Edit distance cannot be computed in strongly subquadratic time (unless SETH is false). *SIAM Journal on Computing*, 47(3):1087–1097, 2018.

[13] Djamal Belazzougui, Fabio Cunial, Juha Kärkkäinen, and Veli Mäkinen. Linear-time string indexing and analysis in small space. *ACM Transaction on Algorithms*, 16(2):17:1–17:54, 2020.

[14] Giulia Bernardini, Pawel Gawrychowski, Nadia Pisanti, Solon P. Pissis, and Giovanna Rosone. Even faster elastic-degenerate string matching via fast matrix multiplication. In Christel Baier, Ioannis Chatzi-

giannakis, Paola Flocchini, and Stefano Leonardi, editors, *46th International Colloquium on Automata, Languages, and Programming, ICALP 2019, July 9-12, 2019, Patras, Greece, Proceedings*, volume 132 of *LIPIcs*, pages 21:1–21:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[15] Rene De La Briandais. File searching using variable length keys. In R. R. Johnson, editor, *Papers presented at the the 1959 Western Joint Computer Conference, IRE-AIEE-ACM 1959 (Western), San Francisco, California, USA, March 3-5, 1959, Proceedings*, pages 295–298. ACM, 1959.

[16] Karl Bringmann and Marvin Künnemann. Quadratic conditional lower bounds for string problems and dynamic time warping. In Venkatesan Guruswami, editor, *IEEE 56th Annual Symposium on Foundations of Computer Science, FOCS 2015, Berkeley, CA, USA, 17-20 October, 2015, Proceedings*, pages 79–97. IEEE Computer Society, 2015.

[17] Michael Burrows and David Wheeler. A block-sorting lossless data compression algorithm. Technical report, Digital SRC Research Report, 1994.

[18] Bastien Cazaux, Dmitry Kosolobov, Veli Mäkinen, and Tuukka Norri. Linear time maximum segmentation problems in column stream model. In Nieves R. Brisaboa and Simon J. Puglisi, editors, *String Processing and Information Retrieval - 26th International Symposium, SPIRE 2019, Segovia, Spain, October 7-9, 2019, Proceedings*, volume 11811 of *Lecture Notes in Computer Science*, pages 322–336. Springer, 2019.

[19] The Computational Pan-Genomics Consortium. Computational pangenomics: status, promises and challenges. *Briefings in Bioinformatics*, 19(1):118–135, 10 2016.

[20] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009.

[21] Maxime Crochemore and Wojciech Rytter. *Text Algorithms*. Oxford University Press, 1994.

[22] Massimo Equi, Arianne Meijer-van de Griend, and Veli Mäkinen. From bit-parallelism to quantum: Breaking the quadratic barrier. *CoRR, Computing Research Repository*, abs/2112.13005, 2021.

[23] Massimo Equi, Roberto Grossi, and Veli Mäkinen. On the complexity of exact pattern matching in graphs: Binary strings and bounded degree. *CoRR, Computing Research Repository*, abs/1901.05264, 2019.

[24] Massimo Equi, Roberto Grossi, Alexandru I. Tomescu, and Veli Mäkinen. On the complexity of exact pattern matching in graphs: Determinism and zig-zag matching. *CoRR, Computing Research Repository*, abs/1902.03560, 2019.

[25] Martin Farach. Optimal suffix tree construction with large alphabets. In *38th Annual Symposium on Foundations of Computer Science, FOCS '97, Miami Beach, Florida, USA, October 19-22, 1997, Proceedings*, pages 137–143. IEEE Computer Society, 1997.

[26] Travis Gagie, Giovanni Manzini, and Jouni Sirén. Wheeler graphs: A framework for bwt-based data structures. *Theoretical Computer Science*, 698:67–78, 2017.

[27] Russell Impagliazzo and Ramamohan Paturi. Complexity of k-sat. In *14th Annual IEEE Conference on Computational Complexity, Atlanta, Georgia, USA, May 4-6, 1999, Proceedings*, pages 237–240. IEEE Computer Society, 1999.

[28] Russell Impagliazzo and Ramamohan Paturi. On the complexity of k-sat. *Journal of Computer and System Sciences*, 62(2):367–375, 2001.

[29] Chirag Jain, Haowen Zhang, Yu Gao, and Srinivas Aluru. On the complexity of sequence-to-graph alignment. *Journal of Computational Biology*, 27(4):640–654, 2020.

[30] Donald E Knuth, James H Morris, Jr, and Vaughan R Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.

[31] U. Manber and S. Wu. Approximate string matching with arbitrary costs for text and hypertext. In *IAPR Workshop on Structural and Syntactic Pattern Recognition, Bern, Switzerland, Proceedings*, pages 22–33, 1992.

[32] Udi Manber and Eugene W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

[33] Edward M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, 1976.

[34] Gonzalo Navarro. Improved approximate pattern matching on hypertext. In Claudio L. Lucchesi and Arnaldo V. Moura, editors, *LATIN '98: Theoretical Informatics, Third Latin American Symposium, Campinas, Brazil, April, 20-24, 1998, Proceedings*, volume 1380 of *Lecture Notes in Computer Science*, pages 352–357. Springer, 1998.

[35] Gonzalo Navarro. Improved approximate pattern matching on hypertext. *Theoretical Computer Science*, 237(1-2):455–463, 2000.

[36] Tuukka Norri, Bastien Cazaux, Dmitry Kosolobov, and Veli Mäkinen. Linear time minimum segmentation enables scalable founder reconstruction. *Algorithms for Molecular Biology*, 14(1):12:1–12:15, 2019.

[37] Pasi Rastas and Esko Ukkonen. Haplotype inference via hierarchical genotype parsing. In Raffaele Giancarlo and Sridhar Hannenhalli, editors, *Algorithms in Bioinformatics, 7th International Workshop, WABI 2007, Philadelphia, PA, USA, September 8-9, 2007, Proceedings*, volume 4645 of *Lecture Notes in Computer Science*, pages 85–97. Springer, 2007.

[38] Mikko Rautiainen and Tobias Marschall. Aligning sequences to general graphs in $O(V + mE)$ time. *bioRxiv*, pages 216–127, 2017.

[39] Nicola Rizzo and Veli Mäkinen. Linear time construction of indexable elastic founder graphs. *CoRR, Computing Research Repository*, abs/2201.06492, 2022.

[40] Jouni Sirén, Niko Välimäki, and Veli Mäkinen. Indexing graphs for path queries with applications in genome research. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 11(2):375–388, March 2014.

[41] Chris Thachuk. Indexing hypertext. *Journal of Discrete Algorithms*, 18:113–122, 2013. Selected papers from the 18th International Symposium on String Processing and Information Retrieval (SPIRE 2011).

[42] Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[43] Esko Ukkonen. Finding founder sequences from a set of recombinants. In Roderic Guigó and Dan Gusfield, editors, *Algorithms in Bioinformatics, Second International Workshop, WABI 2002, Rome, Italy, September 17-21, 2002, Proceedings*, volume 2452 of *Lecture Notes in Computer Science*, pages 277–286. Springer, 2002.

[44] Peter Weiner. Linear pattern matching algorithms. In *14th Annual Symposium on Switching and Automata Theory, Iowa City, Iowa, USA, October 15-17, 1973, Proceedings*, pages 1–11. IEEE Computer Society, 1973.

[45] Ryan Williams. A new algorithm for optimal 2-constraint satisfaction and its implications. *Theoretical Computer Science*, 348(2):357 – 365, 2005.