

**FACULTY
OF MATHEMATICS
AND PHYSICS**
Charles University

DOCTORAL THESIS

Sudatta Bhattacharya

**String Similarity Measures: Metric
Embeddings and Applications**

Computer Science Institute of Charles University

Supervisor of the doctoral thesis: prof. Mgr. Michal Koucký, Ph.D.

Co-advisor: Mgr. Martin Koutecký, Ph.D.

Study programme: Computer Science

Study branch: Theory of Computing, Discrete
Models and Optimization

Prague 2025

I declare that I carried out this doctoral thesis on my own, and only with the cited sources, literature and other professional sources. I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In date

Author's signature

*To the Dumbledores, Hermiones, and Weasleys in my life —
who taught, guided, and stood by me,
through every dark forest and forbidden corridor.*

Title: String Similarity Measures: Metric Embeddings and Applications

Author: Sudatta Bhattacharya

Department: Computer Science Institute of Charles University

Supervisor: prof. Mgr. Michal Koucký, Ph.D., Computer Science Institute of Charles University

Co-advisor: Mgr. Martin Koutecký, Ph.D., Computer Science Institute of Charles University

Abstract: This thesis investigates metric embeddings between fundamental string similarity measures, focusing on edit distance, Hamming distance, and Indel distance. These measures are central to computational tasks in bioinformatics, natural language processing, data compression, and error correction. While Hamming distance is easy to compute, it captures only simple differences. In contrast, edit and Indel distances are more expressive but computationally intensive.

Metric embeddings enable approximating complex metrics by mapping them into simpler spaces, such as embedding edit distance into Hamming distance, with bounded distortion. This approach allows us to leverage efficient algorithms for simpler metrics to be used for more complex ones.

We introduce novel embeddings from edit to Hamming distance that preserve additional structural properties and demonstrate their algorithmic applications. We also explore reverse embeddings from Hamming to edit distance and show how optimal embeddings inform lower bound results. Additionally, we study relationships between edit and Indel distances and investigate alphabet reduction as a form of internal metric embedding.

Together, these contributions yield new theoretical insights and practical tools for handling large-scale string data, significantly advancing the algorithmic understanding of string similarity measures.

Keywords: Edit distance, Hamming distance, Metric Embedding, Algorithms, Complexity

Acknowledgements

Thanks to those who gave me my Hogwarts, and to the ones who believed in me - even when I was still waiting for my letter....

There are many people I would like to thank, and this thesis would not have been possible without each of them in my life. Let me begin from the very start of my journey in Prague.

Back during my master's in India, when I had just decided to pursue research in theoretical computer science, a lab-mate told me about my current PhD supervisor, **Michal Koucký**. I emailed him, honestly not expecting a reply, as he was already a highly reputed researcher. But, true to the kind and approachable person he is, he replied. We had a Skype interview soon after, and then, to my amazement, he invited me to visit Prague for a week and present my master's thesis work. That was such a huge opportunity for me. The moment I landed in Prague, I fell in love with the city, I have not seen a more beautiful place. I really liked the department here at Charles University, and Michal, as a host, was simply amazing. He treated me to every lunch and dinner we had together and didn't let me pay. It was such a generous gesture.

When I received the email from Michal saying he would accept me as his student, I was overjoyed. I had found the opportunity I had hoped for. It truly felt like I had received my Hogwarts letter. And thus began my PhD journey, under the command and mentorship of my supervisor, **Michal Koucký**.

To thank Michal is not easy, because words feel inadequate to express my gratitude. First and foremost, I thank him for accepting me into the PhD program, and for giving me the chance to do my PhD in this beautiful city of Prague. He introduced me to fascinating and challenging problems and encouraged me through every phase, including the hard ones. I began my PhD during the Covid-19 pandemic, a time full of uncertainty, and I am profoundly grateful for his steady guidance and unwavering support. I thank him for all the brainstorming sessions, for his immense patience, his honest feedback and communication, and for giving me the freedom to explore problems independently, while always knowing exactly when to step in and help. All the work in this thesis is a joint collaboration with him, and I am grateful for the detailed feedback he gave me throughout the writing process. From him, I have learned not only about research and problem-solving, but also about time management, effective communication, and what truly matters in academic life. I feel deeply honoured to have been his student, and I aspire to become the kind of researcher and mentor that he is.

My heartfelt thanks go to my co-advisor, **Martin Koutecký**, with whom I worked during the early years of my PhD. His encouraging words and thoughtful advice came at moments when I needed them the most, and helped me keep going. I

would also like to thank Martin for letting me be part of the Compot workshop. I would also like to thank **Mike Saks**, with whom I had the privilege of meeting and collaborating. I cannot overstate how much I have learned from him. His dedication to research, his patience, and his almost magical ability to distill complex ideas into simple, elegant thoughts have had a profound impact on me. It is a rare and inspiring talent, to make complicated problems feel approachable, and I feel truly fortunate to have experienced it first-hand. Beyond his intellectual brilliance, I was equally moved by his humility and warmth. It genuinely surprised me how someone of his stature could be so down to earth, curious, and approachable. I am not exaggerating when I say that when you bring a problem to him, he asks exactly the right questions - thoughtful, precise, and illuminating, that gently guide you toward a revelation. It leaves you with the empowering feeling that you solved the problem yourself. This approach is nothing short of magical. Every discussion I have had with him has been eye-opening, deeply satisfying, and a boost to my confidence. It is truly an honour to work with him. He helped me refine several proofs in this thesis and provided valuable feedback throughout our collaboration.

My sincere thanks go to **Elazar Goldenberg**, whom I met during HALG 2023 and who quickly became both a close friend and a valued collaborator. I am grateful for the stimulating problems he constantly introduces (Elazar truly never runs out of interesting problems!), and for being a consistent source of academic insight and personal support throughout my PhD. Several results in this thesis are joint work with him, and he has also given me countless suggestions to help refine my writing. Before meeting Elazar, I was going through a particularly difficult time. But after our first discussions - spending time thinking through problems together, I felt a renewed sense of motivation and confidence. Looking back, I truly feel fortunate to have met him at exactly that moment. Thank you, Elazar, for being so humble, so approachable, and so full of curiosity. I genuinely enjoy working with him, and I sincerely hope our collaboration and friendship continue for a lifetime.

I would also like to thank **Diptarka Chakraborty** and **Debarati Das** for insightful discussions and valuable advice regarding both research and my future plans. I look forward to further collaborations with them.

A special thanks to our department's very own superwoman, **Petra Milštainová**, for handling all the bureaucratic and administrative matters with incredible efficiency and grace. Since the very beginning of my time in Prague, she has been an indispensable support system. She made my life significantly easier by taking care of all the paperwork and logistics. Many many thanks to Petra, for her patience, warmth, and ever-present smile.

I also want to thank **Ben Moore**, whom I met at Charles University while he was a postdoc. Ben has inspired me in numerous ways. We became close friends, and I am grateful for all our discussions on life, research, and the future. Thank you for tolerating my endless questions, letting me vent, and answering with patience and wisdom. From him, I have learned the value of staying calm in tough situations and how not to get easily triggered. A true Ravenclaw spirit.

I am also thankful to **Zdeněk Dvořák**, with whom I have a publication - although it is not part of this thesis, I learned a great deal from working on that single paper. The techniques of computer-assisted proofs that I learned from him have

greatly influenced my understanding and will be invaluable in my future research. I am also grateful to him for setting up the committee for my PhD defence. Additionally, he was responsible for organizing my state exam—coordinating the dates and gathering all the examiners, which I know was no small task. It was a long and stressful day, and I truly appreciate the effort that went into making it run smoothly. I would also like to thank my state examiners: **Pavel Pudlák**, **Jiří Sgall**, **Jiří Fiala**, and **Martin Tancer**, for their time, thoughtful questions, and support.

I am deeply grateful to those who supported me both academically and emotionally throughout my PhD. **Rahul Gangopadhyay** helped me during my master's and continued to guide me at various points during my PhD. **Pankaj Pundir**, my first friend in Prague, helped me right from the beginning, I truly enjoyed all our musical sessions together. **Manvi Grover**, whom I met in Prague, became a very close friend, our shopping trips and get-togethers became essential and joyful parts of my life here. **Fariba**, another friend I made in Prague, always encouraged me to step outside my comfort zone, though it was scary at first, I ended up enjoying every outing. **Sanjana Dey**, whom I met during HALG 2023, became a dear friend. I thoroughly enjoyed our “fun lunches” with her and Elazar, thanks for being the sister I never had. I would also like to thank all my officemates for many useful and fun discussions over the years.

Let me now go further back to my master's days. Before joining IIIT-Delhi, I had minimal exposure to algorithms and complexity theory, just some basic competitive coding. I never imagined I would one day be doing serious research in theoretical computer science.

The journey at IIITD began nearly seven years ago with the orientation and refresher modules. One of the first subjects I took was taught by **Syamantak Das**. From that moment, my interest in theoretical computer science soared. In my first semester, I enrolled in two TCS courses, one of which was “Introduction to Graduate Algorithms”, taught by my master's advisor **Debajyoti Bera**. Before this course, I didn't even know that designing an algorithm wasn't enough, you also had to prove it correct! After just two weeks of classes, I knew I wanted to do my thesis in TCS under him.

The topic he gave me was challenging and new, yet deeply fascinating. I am grateful to him for introducing me to it and for all the discussions and meetings. Beyond that, he taught me how to write proper emails, structure technical papers, and communicate ideas clearly. He taught me when to stop thinking and start writing, how to construct formal proofs, and most of all, what it means to be a dedicated teacher. If I ever become a professor, I hope to follow his example.

Meanwhile, I was also attending “Modern Algorithm Design” taught by **Syamantak Das**, who encouraged me to pursue a thesis in TCS and a career in research. He taught me how to approach problems formally and guided me through numerous concepts in algorithm design. I also worked as a TA in some of his courses, which further enriched my learning. I will always be grateful for his mentorship and hope to work with him again in the future.

Even further back, I want to thank all my teachers and professors who shaped me along the way. From school - **Saha Sir**, **Banik Sir**, **Ganga Madam**, **Giri Sir**, **Majhee Sir** - to university, especially **Mousumi Ma'am**, with whom I did my bachelor's project, I owe you all my gratitude.

Although I am not religious, I do believe in some form of a higher power. So I thank the almighty - Generator, Operator, and Destroyer - for everything I have. Once again, words cannot truly express how thankful I am to my parents. I would like to thank my parents, who let me board the Hogwarts Express long before I knew where it would take me. They have always placed me and my education above everything else. In a world where very few people are fortunate enough to have someone who says, “Don’t worry about anything else, just follow your dream, I will take care of the rest”, I consider myself deeply blessed. Thank you for always believing in me and being there for me. I owe all of my success to my parents.

I would also like to thank my aunt (*choto pisi*) and uncle (*choto pisu*) for their constant support and encouragement, especially during my time in Delhi. I thank my grandparents, maternal uncles, and all my relatives who believed in me and gave me their time, patience, and love.

Last but by no means least, I want to thank my childhood friends: **Sneha, Sanchary** (thanks Sancha for always letting me vent, calming me down, and offering solid emotional support), and **Aishi**, for always being there for me. Thank you for picking up the phone no matter the hour, you kept me sane.

During my PhD, I was supported by the following grants: EXPRO (grant agreement no. 19-27871X), GAČR (grant no. 22-22997S), GAČR (grant no. 24-10306S), Charles University project UNCE 24/SCI/008, and GAUK project GAUK125424 of the Charles University Grant Agency. I would also like to thank **Karthik C.S.** for hosting me at Rutgers, and Rutgers and DIMACS for partially funding my stay there.

I would also like to thank our department at Charles University for organizing such wonderful workshops like KAMAK, HOMONOLO, the Spring School, and the unforgettable retreats. I am grateful to have been part of the EXPRO workshops. I will truly miss those experiences.

My five-year journey in my own Hogwarts has been truly magical!!!

I solemnly swear that I am up to no good.
- Marauder's Map

Contents

1	Introduction	13
1.1	Preliminaries and Notation	13
1.1.1	String Distance Measures	14
1.1.2	Alignment	15
1.2	Metric Embeddings	16
1.2.1	Motivations for Embeddings	16
1.3	Thesis Roadmap	17
2	Edit to Hamming Embedding	18
2.1	CGK Embedding	18
2.2	Edit to Hamming Embedding via Locally Consistent Decomposition	19
2.2.1	Related work	20
2.2.2	Decomposition Technique	20
2.2.3	Encoding a grammar	21
2.2.4	Our Embedding	22
2.3	Our Decomposition technique in detail	23
2.3.1	Notations and preliminaries	24
2.3.2	Decomposition algorithm	26
2.3.3	Algorithm description	27
2.3.4	Correctness of the decomposition algorithm	29
2.4	Table of parameters	38
2.5	Summary	38
3	Applications of the Edit to Hamming Embedding	40
3.1	Edit distance sketch	40
3.1.1	Hamming distance sketch	42
3.1.2	Edit distance sketch using locally consistent decomposition	42
3.1.3	Rolling sketch for edit distance	44
3.2	Streaming k -edit approximate pattern matching via string decomposition	61
3.2.1	Related work	63
3.2.2	Notations and preliminaries	64
3.2.3	Encoding a grammar	65
3.2.4	k -mismatch approximate pattern matching	66
3.2.5	Algorithm overview	66
3.2.6	Description of the algorithm	67
3.2.7	Correctness of the algorithm	70
3.2.8	Time complexity of the algorithm	73

3.2.9	Space complexity of the algorithm	73
3.3	Summary	74
4	Hamming to Edit Embedding	76
4.1	Introduction	76
4.1.1	Summary of Results	77
4.2	Motivation and Implications	77
4.3	Easy Embedding with Rate $1/\Theta(\log n)$	79
4.3.1	Alphabet Reduction to Binary	80
4.4	Recursive Embedding with Rate $1/\Theta(\log n)$	82
4.4.1	Size Analysis	83
4.4.2	Correctness: Proof of Isometry	83
4.5	Embedding with Rate $1/\mathcal{O}(\log^* n \times \log \log^* n)$: Multi-Level Recursion	85
4.5.1	Multi-Level Recursive Embedding Definition	85
4.5.2	Length Computation of the Embedding	87
4.5.3	Isometry Theorem	87
4.5.4	Final Embedding After Alphabet Reduction	91
4.6	Constant-Rate Embedding	91
4.6.1	Embedding Construction	92
4.6.2	Analysis of the Edit Alignment	93
4.7	Upper Bound on Rate of Embedding	94
4.7.1	Straight Alignment Necessity	96
4.7.2	Characterization as Projective Embeddings	97
4.7.3	Rate Upper Bound	99
4.8	Summary	101
5	Edit and Indel distance	103
5.1	Introduction	103
5.1.1	Motivation	104
5.2	Embeddings between Indel distance metric and Edit distance metric	105
5.2.1	Tiskin's embedding from Edit to Indel metric	105
5.2.2	Indel to Edit Embeddings	105
5.2.3	Obtaining Scaling Isometric Embedding	106
5.2.4	Obtaining Approximate Scaling Isometric Embedding	109
5.3	Summary	114
6	Alphabet Reduction	116
6.1	Introduction	116
6.1.1	Our Results	117
6.1.2	Preliminaries and Notations	119
6.2	Alphabet Reduction - Succinct Embedding	119
6.2.1	Normalized Scaled Isometric Embedding - Our Finding	120
6.2.2	Upper Bounds	121
6.2.3	Lower Bounds	129
6.3	Alphabet Reduction - Binary Alphabets	130
6.4	Summary	136

7 Conclusion	137
7.1 Future directions	138
Bibliography	140
List of Figures	147
List of Tables	149
List of Abbreviations	150
List of Publications	151

1

Introduction

“We’ve all got both light and dark inside us. What matters is the part we choose to act on. That’s who we really are.”
- Sirius Black

The study of *string similarity measures* lies at the core of many computational problems across diverse domains, including **bioinformatics**, **natural language processing**, **data compression**, **plagiarism detection**, and **error correction**. Given two strings, quantifying how “similar” or “different” they are is often the crucial first step in comparing genomes, classifying documents, detecting patterns, or ensuring data integrity in communication systems.

String similarity is typically captured using well-defined distance metrics, each emphasizing different aspects of the transformations required to convert one string into another. Among these, *Edit Distance*, *Hamming Distance*, and *Indel Distance* are among the most widely studied and applied. Each metric provides a unique computational perspective on reasoning about strings, characterized by distinct algorithmic strengths and limitations.

This thesis explores fundamental aspects of string similarity measures, emphasizing metric embeddings that transform complex metrics like edit distance into simpler metrics such as Hamming distance (and vice-versa). Metric embeddings are mappings that approximately (or exactly) preserve distances up to bounded distortion, enabling efficient algorithmic solutions by leveraging simpler geometry and well-understood computational frameworks. These embeddings have profound applications in approximation algorithms, sketching, streaming algorithms, and establishing conditional lower bounds.

1.1 Preliminaries and Notation

Let Σ be a finite alphabet, and denote by Σ^* the set of all finite strings over Σ . The length of a string x is denoted by $|x|$. Typically, we denote strings by lowercase letters x, y or uppercase letters X, Y . The set of all strings of length n over alphabet Σ is denoted by Σ^n . For a string X , the i^{th} character is denoted by either x_i or $X[i]$. A substring from position p to q is denoted by $x[p, q]$, with $x[p, q] = x[p, q - 1]$ and $x[p, \dots] = x[p, |x|]$. If $q < p$, then $x[p, q]$ is the empty string ε . Concatenation of two strings x and y is denoted by $x \cdot y$. All logarithms used in this thesis are base 2 unless stated otherwise.

Given strings $x \in \Sigma^n$ and $y \in \Sigma^m$, we define several important distance measures as follows.

1.1.1 String Distance Measures

Hamming Distance

The *Hamming distance* (denoted by Δ_{Ham}) measures the number of differing characters at corresponding positions in two strings of equal length:

$$\Delta_{Ham}(x, y) = |\{i : x_i \neq y_i\}|.$$

This metric is central in **coding theory** and **error detection**, specifically handling substitution errors. Hamming distance can be computed optimally in $\mathcal{O}(n)$ time, simply by comparing corresponding symbols.

Edit Distance (Levenshtein Distance)

The *edit distance* (denoted by Δ_{edit}) between two strings is defined as the minimum number of operations: **insertions**, **deletions**, and **substitutions**, required to transform one string into another. It satisfies the properties of a metric, including the triangle inequality.

The classic dynamic programming algorithm by Wagner and Fischer [1] computes $\Delta_{edit}(x, y)$ in $\mathcal{O}(n^2)$ time, where n is the length of the strings. While there have been improvements with polylogarithmic speedups [2, 3], the best known exact algorithm runs in $\mathcal{O}(n + k^2)$ time [4], where $k = \Delta_{edit}(x, y)$.

Under the Strong Exponential Time Hypothesis (SETH), it is conjectured that no algorithm can compute edit distance in truly subquadratic time in the worst case [5]. Despite this, recent advances have led to approximation algorithms that achieve constant-factor approximations of edit distance in almost-linear time [6, 7, 8, 9], marking significant progress in practice.

Indel Distance

The *Indel distance* considers only insertions and deletions and directly relates to the longest common subsequence (LCS). It is defined as:

$$\text{Indel}(x, y) = |x| + |y| - 2 \cdot \text{LCS}(x, y).$$

Indel distance inherits computational properties similar to edit distance, including the quadratic lower bound under SETH and similar approximation algorithmic approaches.

Longest Common Subsequence (LCS)

The *LCS* measures similarity by the length of the longest sequence present in both strings in identical order. While useful for similarity assessments, LCS does not constitute a metric as it violates the triangle inequality. Computationally (exact computation), it shares complexity characteristics with Indel distance, including known lower bounds.

1.1.2 Alignment

We now define the important notion of string alignment, which underlies edit and Indel distances.

Definition 1.1 (Alignment). *Given two strings $X = x_1x_2\dots x_n$ and $Y = y_1y_2\dots y_m$, an alignment is a partial function $\mathcal{A} : [1, n] \rightarrow [1, m]$ such that for all $i < j$, we have $\mathcal{A}(i) < \mathcal{A}(j)$. This defines a monotonic mapping between positions in X and Y .*

The alignment can be visualized as a bipartite graph between the characters of X and Y , where there is an edge between x_i and y_j if and only if $\mathcal{A}(i) = j$. Each character may participate in at most one alignment edge, and the edges must not cross.

The cost of an alignment is the number of unmatched characters (insertions and deletions) plus the number of mismatched aligned characters (substitutions), i.e., positions where $x_i \neq y_j$ but there is an edge between x_i, y_j . The minimum alignment cost over all valid alignments equals the edit distance $\Delta_{edit}(X, Y)$.

For Δ_{indel} , alignments are further restricted to match only identical characters ($x_i = y_j$), i.e., if there is an edge between x_i and y_j then $x_i = y_j$. The cost then consists solely of unmatched characters (insertions and deletions), and the minimum such cost equals $\Delta_{indel}(X, Y)$.

Definition 1.2 (Straight Alignment / Identity Alignment). *A straight alignment (also called an identity alignment) between two strings x and y of equal length is defined by aligning each character at position i in x with the character at the same position i in y .*

Formally, for $x, y \in \Sigma^n$, the alignment $\mathcal{A}(i) = i$ for all $i \in [1, n]$ is a straight alignment. This alignment performs no edits if $x = y$, and its cost is exactly the Hamming distance between x and y otherwise.

When $\mathcal{A}(i) = j$, we say that character x_i is *aligned* to character y_j . Conceptually, the alignment \mathcal{A} describes a transformation from x to y via insertions, deletions, and substitutions. An insertion in x may equivalently be viewed as a deletion in y , so it is often convenient to consider only deletions and substitutions, but deletions performed on both strings.

Given an alignment \mathcal{A} , we can also measure its effect on a substring of x . We say that “ \mathcal{A} performs k edits on the substring $x[p, q]$ ” if there are k edit operations (insertions, deletions, or substitutions) that occur within the span of $x[p, q]$. If edits affect characters outside the substring, such as insertions just before x_p or just after x_q , we explicitly note that those edits occur at the beginning or end of the substring.

Below are several facts about edit distance and alignments that will be used repeatedly throughout this thesis:

Fact 1.3. *Let x and y be two strings, and let s be another string. Then,*

$$\Delta_{edit}(x, y) = \Delta_{edit}(s \cdot x, s \cdot y) = \Delta_{edit}(x \cdot s, y \cdot s).$$

That is, adding a common prefix or suffix to both strings does not change the edit distance.

Fact 1.4. *Let x and y be two strings of the same length. Suppose alignment \mathcal{A} aligns character $x[i]$ to $y[j]$. Then the cost of \mathcal{A} satisfies:*

$$\text{cost}(\mathcal{A}) \geq 2|i - j|.$$

Fact 1.5. *Let x and y be two strings and let \mathcal{A} be an alignment from x to y . If \mathcal{A} performs no edits on the substring $x[p, q]$, then $x[p, q]$ must occur as a contiguous substring of y .*

1.2 Metric Embeddings

An *embedding* from a metric space $(\mathcal{X}, d_{\mathcal{X}})$ into another metric space $(\mathcal{Y}, d_{\mathcal{Y}})$ is a function $f : \mathcal{X} \rightarrow \mathcal{Y}$ such that for all $x_1, x_2 \in \mathcal{X}$,

$$d_{\mathcal{X}}(x_1, x_2) \leq d_{\mathcal{Y}}(f(x_1), f(x_2)) \leq D \cdot d_{\mathcal{X}}(x_1, x_2),$$

for some distortion factor $D \geq 1$. Such an embedding approximately preserves pairwise distances and enables the simulation of computations in $(\mathcal{X}, d_{\mathcal{X}})$ using algorithms designed for the simpler space $(\mathcal{Y}, d_{\mathcal{Y}})$.

If $D = 1$ and the distance is preserved exactly for all pairs, i.e., $d_{\mathcal{Y}}(f(x_1), f(x_2)) = d_{\mathcal{X}}(x_1, x_2)$, the embedding is called an *isometric embedding*. More generally, if $d_{\mathcal{Y}}(f(x_1), f(x_2)) = c \cdot d_{\mathcal{X}}(x_1, x_2)$ for some fixed scaling constant $c > 0$, the embedding is referred to as a *scaled isometric embedding*.

There are several key criteria for evaluating the quality of an embedding:

1. **Distortion:** This measures the multiplicative error in preserving distances. Lower distortion implies higher fidelity.
2. **Rate:** Defined as the ratio of the dimension of the input space to that of the output space. Lower rates may be undesirable in applications like sketching or compression. The closer the rate is to 1, the better.
3. **Computational Efficiency:** The time complexity required to compute $f(x)$ for any given $x \in \mathcal{X}$ is also critical, especially in large-scale or streaming contexts.

1.2.1 Motivations for Embeddings

Metric embeddings are powerful algorithmic tools that enable the approximation of complex metric spaces using simpler, more computationally tractable ones. In the context of string similarity, embedding a computationally hard metric such as *edit distance* into an easier metric like *Hamming distance* has wide-ranging applications.

Many algorithmic problems that rely on Hamming distance, such as sketching, streaming, clustering, and nearest-neighbor search, admit highly efficient or even optimal algorithms. In contrast, the same problems under edit distance often lack such efficient solutions due to its higher computational complexity. Therefore, a distortion-bounded embedding from edit distance into Hamming distance could allow us to apply fast Hamming-based algorithms to approximate solutions under

edit distance. Naturally, the usefulness of such reductions depends on the quality of the embedding, and some applications demand additional structural properties from the embedding, as discussed in Chapter 3.

Interestingly, as we explore in Chapter 4, embeddings in the reverse direction—from Hamming into edit distance—are also of significant theoretical interest. Proving tight conditional lower bounds for problems under Hamming distance is often more tractable. An optimal embedding from Hamming to edit distance would effectively reduce these problems to their edit-distance counterparts, transferring the lower bounds as well.

Beyond these algorithmic motivations, embeddings can help us investigate finer structural questions: How similar are edit and Indel distances? How does the structure of the space Σ^n differ from that of $\{0, 1\}^m$, even under the same distance function? Understanding such relationships deepens our grasp of the geometric and algorithmic landscape of string metrics. These questions are explored in Chapters 5 and 6.

1.3 Thesis Roadmap

This thesis comprises six chapters derived from four research papers:

Chapter 2: Embedding edit distance into Hamming space.

Chapter 3: Applications of these embeddings.

Chapter 4: Reverse embeddings—Hamming into edit distance.

Chapter 5: Embeddings between edit distance and Indel distance.

Chapter 6: Alphabet reduction techniques as embeddings within metrics.

These contributions collectively advance our understanding of string similarity measures, providing practical tools and theoretical insights for algorithm design and complexity analysis.

2

Edit to Hamming Embedding

“The truth. It is a beautiful and terrible thing, and should therefore be treated with great caution.”
- Dumbledore

In this chapter, we explore techniques for embedding strings from the edit distance metric into the Hamming metric with provably bounded distortion.

Note. Trivial embedding such as the identity mapping can have large distortion. For example, consider the strings: $x = \{01\}^{n/2}$ and $y = \{10\}^{n/2}$. Then:

$$\Delta_{Ham}(x, y) = n \quad \text{and} \quad \Delta_{edit}(x, y) = 2,$$

which gives the inequality:

$$\Delta_{edit}(x, y) \leq \Delta_{Ham}(x, y) \leq (n/2) \cdot \Delta_{edit}(x, y).$$

In Section 2.1, we begin with an accessible introduction to a simple yet powerful embedding technique known as the CGK embedding. In the subsequent sections, we present our new embedding, which achieves similar distortion guarantees but offers additional structural and algorithmic advantages.

- The results presented in this chapter form a significant part of the work published in [10].

2.1 CGK Embedding

The CGK embedding, introduced by Chakraborty, Goldenberg, and Koucký [11], is a randomized mapping from strings under the edit distance metric into strings under the Hamming distance metric. For any two binary strings $x, y \in \{0, 1\}^n$ with edit distance $\Delta_{edit}(x, y) = k$, the embedding ensures that the Hamming distance between their images $f(x, r)$ and $f(y, r)$ is at most quadratic in k , i.e., $\Delta_{edit}(x, y)/2 \leq \Delta_{Ham}(f(x, r), f(y, r)) \leq \mathcal{O}(k^2)$, with high probability over the choice of random string r . A crucial property of this embedding is its efficient computation in a streaming fashion, requiring only a single pass over the input and using linear time and small space. This embedding significantly simplifies

algorithmic problems by reducing the complexity of edit distance computations to the more tractable Hamming distance.

Embedding Algorithm. The embedding algorithm proceeds as follows:

1. Initialize a pointer $i = 1$ to the start of the string x and prepare an empty output string.
2. For each position $j = 1, \dots, 3n$, perform:
 - (a) If $i \leq n$, append the current bit x_i to the output string, and increment the pointer i by a random bit $h_j(x_i) \in \{0, 1\}$.
 - (b) If $i > n$, append a zero to the output string.

Here, h_j are independent random functions mapping $\{0, 1\} \rightarrow \{0, 1\}$.

Theorem 2.1 (CGK Embedding Distance Guarantee). *For any strings $x, y \in \{0, 1\}^n$ with $\Delta_{edit}(x, y) = k$, the randomized embedding f satisfies:*

$$\frac{1}{2} \cdot \Delta_{edit}(x, y) \leq \Delta_{Ham}(f(x, r), f(y, r)) \leq \mathcal{O}(k^2)$$

with probability at least $2/3$ over the choice of random bits r .

Proof Idea. The proof relies on analyzing the embedding algorithm as a random walk on the integer line. Initially, both embeddings of x and y are identical until the first difference is encountered. At this point, the embeddings become unsynchronized, and the random increments ensure independent progress. The crucial observation is that the embeddings synchronize quickly again, and the total number of unsynchronized steps is bounded by the time it takes for a random walk to reach a particular target, resulting in the quadratic bound. This random walk analogy provides a clear probabilistic interpretation and leads to the formal guarantee stated above.

2.2 Edit to Hamming Embedding via Locally Consistent Decomposition

We introduce a new randomized embedding from the edit distance metric to the Hamming metric that achieves distortion guarantees similar to the CGK embedding. However, our construction provides a key additional advantage: *locality*.

This locality property ensures that when a symbol is appended to the end of a string, or a symbol is removed from its beginning, the embedding can be updated efficiently using the embedding of the original string. This makes our approach particularly well-suited for dynamic and streaming environments, where strings are processed incrementally. Informally, our main theorem is stated as follows:

Theorem 2.2 (Informal Main Theorem). *For any strings $x, y \in \Sigma^n$ with $\Delta_{edit}(x, y) \leq k$, there exists a randomized embedding $f : \Sigma^n \rightarrow \Sigma^{m'}$ satisfying:*

$$\Delta_{edit}(x, y) \leq \Delta_{Ham}(f(x), f(y)) \leq \tilde{O}(k^2)$$

with probability at least $3/4$.

We prove the theorem by using a novel string decomposition algorithms combined with a simple encoding.

2.2.1 Related work

The problem of embedding edit distance to other distance measures, like Hamming distance, ℓ_1 , etc. has been studied extensively. In [11], the authors have given a randomized embedding from edit distance to Hamming distance, where any string $x \in \{0, 1\}^n$ can be mapped to a string $f(x) \in \{0, 1\}^{3n}$, given a random string $r \in \{0, 1\}^{\log^2 n}$, such that, $\Delta_{edit}(x, y)/2 \leq \Delta_{Ham}(f(x), f(y)) \leq \mathcal{O}(\Delta_{edit}(x, y)^2)$ with probability at least $2/3$. Batu, Ergun and Sahinalp [12] have introduced a dimensionality reduction technique, where any string x of length n can be mapped to a string $f(x)$ of length at most n/r , for any parameter r , with a distortion of $\tilde{O}(r)$. They used the locally consistent parsing technique for their embedding. Ostrovsky and Rabani [13] gave an embedding from edit distance to ℓ_1 distance with a distortion of $2^{\mathcal{O}(\sqrt{\log n \log \log n})}$. Jowhari [14] also gave a randomized embedding from edit distance to ℓ_1 distance with a distortion of $\mathcal{O}(\log n \log^* n)$. He used the embedding given by Cormode and Muthukrishnan [15] who showed that any string x of length n can be mapped to a vector $f(x)$ of length $m = \mathcal{O}(2^{n \log n})$, such that for any pair of strings x, y of length n each, $\Delta_{edit}(x, y)/2 \leq \|f(x) - f(y)\|_{\ell_1} \leq \mathcal{O}(\log n \log^* n) \cdot \Delta_{edit}(x, y)$. Since the size of the vector was too large, [14] used random hashing to get his final embedding.

2.2.2 Decomposition Technique

To construct this embedding we provide a new technique to align two strings x and y in oblivious manner. In nutshell, we provide a decomposition procedure that breaks x and y into the same number of “short” blocks so that at most k pairs of blocks in the decomposition of x and y differ, and all other pairs of blocks are matching in an optimal alignment. So the edit distance of x and y is the sum of edit distances of the differing blocks. To be more specific our blocks are not short in their length but they are short in the sense that each of them can be described by a context-free grammar of size $\tilde{O}(k)$. Our decomposition algorithm constructs the grammars. Our decomposition is based on *locally consistent parsing* of strings a technique similar to the one used in [16, 12, 14, 17]. Our main technical result is:

Theorem 2.3 (String decomposition). *There is an algorithm running in time $\tilde{O}(nk)$ that for each string x of length at most n produces grammars G_1^x, \dots, G_s^x such that with probability at least $1 - \mathcal{O}(1/\sqrt{n})$, $x = \text{eval}(G_1^x) \cdots \text{eval}(G_s^x)$ and each of the grammars is of size $\tilde{O}(k)$. Furthermore, for any two strings x and y of edit distance at most k with grammars G_1^x, \dots, G_s^x and $G_1^y, \dots, G_{s'}^y$, resp., that*

are produced by the algorithm using the same randomness, the following is true simultaneously with probability at least $4/5$:

1. $s = s'$,
2. $G_i^x = G_i^y$, for all $i \in \{1, \dots, s\}$ except for at most k indices i , and
3. $\Delta_{edit}(x, y) = \sum_i \Delta_{edit}(\text{eval}(G_i^x), \text{eval}(G_i^y))$.

Here, for a grammar G , $\text{eval}(G)$ denotes its evaluation. Our decomposition can be used immediately to give an embedding of edit distance into Hamming distance with distortion $\mathcal{O}(k)$.

Another distinguishing feature of our decomposition procedure compared to the technique of CGK random walks is its parallelizability. CGK random walk seems inherently sequential whereas our decomposition procedure can be easily parallelized. We believe that our decomposition will allow for further applications beyond our simple embedding, and sketches and streaming algorithm that we show in the next chapter.

2.2.3 Encoding a grammar

We will encode the grammars obtained from the above decomposition algorithm using a simple encoding.

We will set a parameter $N \geq n^3$ to be a suitable integer: Let $F_{\text{KR}} : \{0, 1\}^* \rightarrow \{1, \dots, N\}$ be a hash function picked at random, such as Karp-Rabin fingerprint [18], so for any two strings $u, v \in \{0, 1\}^*$, if $u \neq v$ then $\Pr_{F_{\text{KR}}}[F_{\text{KR}}(u) = F_{\text{KR}}(v)] \leq (|u| + |v|)/N$.

Set $M = 3S \cdot \lceil 1 + \log |\Gamma| \rceil$. We will encode a grammar G over Γ of length at most S given by our decomposition algorithm by a string $\text{Enc}(G)$ over alphabet $\{1, \dots, 2N\}$ of length M . The encoding is obtained as follows: First, order the rules of the grammar G lexicographically. Then encode the rules in binary one by one using $3 \cdot \lceil 1 + \log |\Gamma| \rceil$ bits for each rule. (The extra bit allows to mark unused symbols.) This gives a binary string of length at most M , which we pad by zeros to the length precisely M . We call the resulting binary string $\text{Bin}(G)$. Compute $h_G = F_{\text{KR}}(\text{Bin}(G))$. We replace each 0 in $\text{Bin}(G)$ by h_G , and each 1 in $\text{Bin}(G)$ by $N + h_G$ to obtain the string $\text{Enc}(G)$. Clearly, $\text{Enc}(G)$ is a string over alphabet $\{1, \dots, 2N\}$ of length exactly M . The encoding can be computed in time $\mathcal{O}(M)$. For completeness, we encode any grammar G of length more than S or that uses rules with more than two symbols on the right as $\text{Enc}(G) = 1^M$.

By the property of F_{KR} the following holds.

Lemma 2.4. *Let G, G' be two grammars of size at most S output by our decomposition algorithm. Let F_{KR} be chosen at random.*

1. $\text{Enc}(G) \in \{1, \dots, 2N\}^M$.
2. If $G = G'$ then $\text{Enc}(G) = \text{Enc}(G')$.
3. If $G \neq G'$ then $\text{Enc}(G) = \text{Enc}(G')$ with probability at most $2M/N$.
4. If $\text{Enc}(G) \neq \text{Enc}(G')$ then $\Delta_{Ham}(\text{Enc}(G), \text{Enc}(G')) = M$, that is they differ in every symbol.

2.2.4 Our Embedding

Given our string decomposition theorem 2.8 and the encoding procedure, we define our embedding as follows: Let n and $k \leq n$ be two parameters, and let $p \geq 2N + 1$ be a prime such that $p \geq (nM)^3$. For a string $x \in \Sigma^*$ of length at most n , we compute its embedding by first running the decomposition algorithm of Theorem 2.8 to obtain grammars G_1, G_2, \dots, G_s . Each grammar G_i is encoded using $\text{Enc}(G_i)$ from Section 3.2.3, with the same randomly chosen hash function F_{KR} . The final embedding is the concatenation:

$$f(x) = \text{Enc}(G_1) \cdot \text{Enc}(G_2) \cdots \text{Enc}(G_s).$$

Now, we can prove the following formal theorem for our embedding, proving its distance guarantees.

Theorem 2.5. *Let $x, y \in \Sigma^*$ be strings of length at most n such that $\Delta_{\text{edit}}(x, y) \leq k$. Let $f(x)$ and $f(y)$ be obtained using the same randomness for the decomposition algorithm and the same choice of F_{KR} . Then, with probability at least $3/4$, we have:*

$$\Delta_{\text{edit}}(x, y) \leq \Delta_{\text{Ham}}(f(x), f(y)) \leq \tilde{\mathcal{O}}(k^2).$$

Proof. Assume the decomposition algorithm on x and y satisfies all conclusions of Theorem 2.3:

1. For x we get $\text{eval}(G_1^x) \cdot \text{eval}(G_2^x) \cdots \text{eval}(G_s^x)$,
2. For y , we get $\text{eval}(G_1^y) \cdots \text{eval}(G_s^y)$, with $s \leq n$,
3. Each grammar is of size at most S ,
4. $\Delta_{\text{edit}}(x, y) = \sum_i \Delta_{\text{edit}}(\text{eval}(G_i^x), \text{eval}(G_i^y))$, and
5. At most k indices i satisfy $G_i^x \neq G_i^y$.

Assume F_{KR} is chosen such that $\text{Enc}(G_i^x) \neq \text{Enc}(G_i^y)$ for each differing pair. Then:

1. $\Delta_{\text{Ham}}(f(x), f(y)) \geq M \geq \Delta_{\text{edit}}(x, y)$, since if $\Delta_{\text{edit}}(x, y) > 0$, at least one pair of grammars differ.
2. $\Delta_{\text{Ham}}(f(x), f(y)) \leq M \cdot k = \tilde{\mathcal{O}}(k^2)$, since at most k pairs differ.

Failure probabilities:

1. Decomposition failure probability is at most $1/5$ for large n .
2. Probability that $\text{Enc}(G_i^x) = \text{Enc}(G_i^y)$ for any differing pair is at most $2kM/N < 1/n$, by choice of N .

Thus, total failure probability $\leq 1/5 + 1/n < 1/4$, and the theorem follows. \square

2.3 Our Decomposition technique in detail

We first provide the intuition for our technique. We would like to break a string x into small blocks *obliviously* so that when a string y is broken by the same procedure, the difference between x and y caused by the edit operations is confined within the corresponding blocks of x and y , and the overall decomposition is not affected by them. For random binary strings x and y this could be done fairly easily: look on all the (overlapping) windows of $\log n$ consecutive bits in each of the strings and for each window decide at random whether to make a break at that window or not. To make it consistent between x and y use some random hash function $H : \{0, 1\}^{\log n} \rightarrow \{0, \dots, D - 1\}$ so that if the hash function evaluates to 0 on a given window then start a next block of the decomposition. If we chose D suitably, say $D \geq 10k \log n$, then we are unlikely to start a new block in any window which is affected by the at most k edit operations on x and y . In that case we obtain the desired decomposition. Hence, decomposing random strings x and y is easy.

The issue is what to do with non-random strings. Consider for example strings x and y that are very sparse, so they contain \sqrt{n} ones sprinkled within the vast ocean of zeros. The hash function H will see mostly windows of 0's and occasionally a window of the form $0^i 1 0^{\log(n)-i-1}$. The decomposition will have no effect on such strings despite the fact that the string might contain $\Omega(\sqrt{n})$ bits of entropy.

However, we can compress such sparse strings: replace stretches of zeros by some binary encoded information about their length, and try to break the strings again. Still, this will fail if in our example the stretches of zeros are replaced by stretches of some repeated pattern such as $(01)^*$. So we need slightly more general compression which will compress any $\log n$ bits into $\log(n)/2$ bits. By repeating the sequence of steps: split and compress, we will eventually get the desired decomposition of each string.

Our actual algorithm mimics the above intuition. It is technically easier to work with a larger alphabet, so we extend the input alphabet Σ by adding special compression symbols into the work alphabet Γ . (Without loss of generalization we can assume that Σ is of size $\mathcal{O}(n^3)$ otherwise we can hash each symbol of our input strings using some perfect hash function into an alphabet of size $\mathcal{O}(n^3)$ without affecting the edit distance of a given pair of strings.) To split a string we will use a random $\tilde{\mathcal{O}}(k)$ -wise independent hash function $H : \Gamma^2 \rightarrow \{0, \dots, D - 1\}$, for $D = \Theta(k \log n)$. If the hash function is zero on a pair of consecutive symbols in a string, we start a new block of the decomposition on the first symbol in the pair.

Then in each resulting block we replace stretches of repeated symbols by a special compression symbol from Γ representing the block, and we use a pair-wise independent hash function $C : \Gamma^2 \rightarrow (\Gamma \setminus \Sigma)$ to compress non-overlapping pairs of symbols into one symbol. This latter step requires some care as we have to make sure that we select non-overlapping pairs in the same way in x and y . For the selection of non-overlapping pairs we use the locally consistent coloring of Cole and Vishkin [19, 20, 21] where the selection of pairs depends only on the context of $\mathcal{O}(\log^* n)$ symbols. The compression reduces the size of each block by a factor of $2/3$. We repeat the compress and split process for $\mathcal{O}(\log n)$ iterations until

each compressed block of x is of size at most 2. *Decompression* of each block then gives us the desired decomposition of x . (See Fig. 2.1 for an illustration.)

It is natural and convenient to represent each of the blocks by a context-free grammar which corresponds to the compression process. We can argue that the grammars will be of size $\mathcal{O}(D \log n)$ with high probability. So we can represent each string by a sequence of small grammars so that if x and y are at edit distance at most k then at most k pairs of their grammars will differ, and the sum of the edit distances of differing pairs is the edit distance of x and y . Note, that edit distance of two strings represented by context-free grammars can be computed efficiently [22]. These are the main ideas behind our decomposition algorithm, and we provide more details in Section 3.2.2

Building a sketch from the string decomposition is straightforward: We encode each grammar in binary using fixed number of bits, and we use off-the-shelf sketch for Hamming distance to sketch the sequence of grammars. As the Hamming distance sketch does not recover identical bits but only the mismatched bits we make sure that if two grammars differ then their binary encoding differ in every bit. Over binary alphabet this might be impossible but over large alphabets one could use error-correcting codes to achieve the desired effect of recovering the differing grammars; for simplicity we use the Karp-Rabin fingerprint of the whole grammar to encode the binary 0 and 1 distinctly. See Section 3.2.3 for the details of our encoding and Section 3.1.2 for details of the sketch for edit distance.

To design a rolling sketch for edit distance where we can extend the represented string by a new symbol or repeatedly remove the first symbol of the represented string we will employ our decomposition technique together with the rolling sketch for Hamming distance of Clifford, Kociumaka, and Porat [23]. We will argue that appending a new symbol to a string affects only some fixed number of grammars in the decomposition of a string. There is a certain threshold T so that except for the last T grammars the decomposition of a string stays the same regardless of how many other symbols are appended. Hence, we will keep a buffer of at most T *active* grammars corresponding to the recently added symbols, and upon addition of a new symbol we will only update those grammars. We are guaranteed that the grammars before this threshold will stay the same forever, so we can *commit* them into the rolling Hamming sketch (in the form of their binary encoding.) Similarly, we will keep a buffer of up-to T *active* grammars that capture the symbols that were deleted from the sketch most recently. Once they become “mature” enough we can commit them by removing their binary encoding from the rolling Hamming sketch. (See Fig. 3.1 for an illustration.) This allows to maintain a rolling sketch for edit distance.

Evaluation of an edit distance query on two rolling sketches will use their Hamming sketch to recover differing committed grammars. Together with the active grammars of inserted and deleted symbols this provides enough information for evaluating the edit distance query. Technical details are explained in Section 3.1.3. In Section 2.4 we give a table of parameters used throughout the chapter.

2.3.1 Notations and preliminaries

$\text{Dict}(x) = \{x[i, i + 1], i \in [n - 1]\}$, is the dictionary of string x , which stores all pairs of consecutive symbols that appear in x .

Grammars Let $\Sigma \subseteq \Gamma$ be two alphabets and $\# \notin \Gamma$. A *grammar* G is a set of *rules* of the type $c \rightarrow ab$ or $c \rightarrow a^r$, where $c \in (\Gamma \cup \{\#\}) \setminus \Sigma$, $a, b \in \Gamma$ and $r \in \mathbb{N}$. c is the *left hand side* of the rule, and ab or a^r is the *right hand side* of the rule. $\#$ is the starting symbol. The size $|G|$ of the grammar is the number of rules in G . We only consider grammars where each $a \in \Gamma \cup \{\#\}$ appears on the left hand side of at most one rule of G , we call such grammars *deterministic*. (We assume that rules of the form $c \rightarrow a^r$ are stored in implicit (compressed) form.) The $\text{eval}(G)$ is the string from Σ^* obtained from $\#$ by iterative rewriting of the intermediate results by the rules from G . If the rewriting process never stops or stops with a string not from Σ^* , $\text{eval}(G)$ is undefined. Observe, that we can replace each rule of the type $c \rightarrow a^r$ by a collection of at most $2\lceil \log r \rceil$ new rules of the other type using some auxiliary symbols. Hence, for each grammar G there is another grammar G' using only the first type of the rules such that $\text{eval}(G) = \text{eval}(G')$ and $|G'| \leq |G| \cdot 2\lceil \log |\text{eval}(G)| \rceil$. Using a depth-first traversal of a deterministic grammar G we can calculate its *evaluation size* $|\text{eval}(G)|$ in time $\mathcal{O}(|G|)$. Given a deterministic grammar G and an integer m less or equal to its evaluation size, we can construct in time $\mathcal{O}(|G|)$ another grammar G' of size $\mathcal{O}(|G|)$ such that $\text{eval}(G') = \text{eval}(G)[m, \dots]$. G' will use some new auxiliary symbols. Given a deterministic grammar G , using a depth-first traversal on symbols reachable from the starting symbol $\#$ we can identify in time $\mathcal{O}(|G|)$ the smallest sub-grammar $G' \subseteq G$ with the same evaluation.

We will use the following observation of Ganesh, Kociumaka, Lincoln and Saha [22]:

Proposition 2.6 ([22]). *There is an algorithm that on input of two grammars G_x and G_y of size at most m computes the edit distance k of $\text{eval}(G_x)$ and $\text{eval}(G_y)$ in time $\mathcal{O}((m + k^2) \cdot \text{poly}(\log m + n))$, where $n = |\text{eval}(G_x)| + |\text{eval}(G_y)|$.*

Locally consistent coloring The following color reduction procedure allows for locally consistent parsing of strings. The technique was originally proposed by Cole and Vishkin [19] and further studied by Linial [20, 21].

Proposition 2.7 ([19, 20, 21]). *There exists a function $F_{\text{CVL}} : \Gamma^* \rightarrow \{1, 2, 3\}^*$ with the following properties. Let $R = \log^* |\Gamma| + 20$. For each string $x \in \Gamma^*$ in which no two consecutive symbols are the same:*

1. $|F_{\text{CVL}}(x)| = |x|$ and $F_{\text{CVL}}(x)$ can be computed in time $\mathcal{O}(R \cdot |x|)$.
2. For $i \in \{1, \dots, |x|\}$, the i -th symbol of $F_{\text{CVL}}(x)$ is a function of symbols of x only in positions $\{i - R, i - R + 1, \dots, i + R\}$.
3. No two consecutive symbols of $F_{\text{CVL}}(x)$ are the same.
4. Out of every three consecutive symbols of $F_{\text{CVL}}(x)$ at least one of them is 1.
5. If $|x| = 1$ then $F_{\text{CVL}}(x) = 3$, and otherwise $F_{\text{CVL}}(x)$ starts by 1 and ends by either 2 or 3.

The first three items are standard for $R = \log^* |\Gamma| + 10$. The other two can be obtained by a simple modification of the output of the standard function. In the output, replace first in parallel each sequence 232 by 212, and then each

sequence 323 by 313. This guarantees the fourth condition. To satisfy the fifth condition, if $|x| = 1$, set $F_{\text{CVL}}(x) = 3$, if $|x| = 2$, set $F_{\text{CVL}}(x) = 12$, if $|x| = 3$, set $F_{\text{CVL}}(x) = 123$, and if $|x| = 4$, set $F_{\text{CVL}}(x) = 1212$. If $|x| > 4$ then replace the sequence at the beginning of the output as follows: if it starts by a word from $\{2, 3\}\{2, 3\}1$ replace it by 121, if it starts by $\{2, 3\}1\{2, 3\}\{2, 3\}$ replace it by 1212, if it starts by $\{2, 3\}1\{2, 3\}1$ replace it by 1231. Then at the end of the sequence, replace $1\{2, 3\}1$ by 123, and $1\{2, 3\}\{2, 3\}1$ by 1212. This will increase the local dependency to at most $R = \log^* |\Gamma| + 20$.

2.3.2 Decomposition algorithm

In this section we describe our main technical tool that we have developed. It is a randomized procedure that splits a string x into blocks $B_1^x, B_2^x, \dots, B_s^x$ and for each block it produces a grammar of size at most $S = \tilde{\mathcal{O}}(k)$. Furthermore, if $B_1^x, B_2^x, \dots, B_s^x$ is the decomposition for a string x and $B_1^y, B_2^y, \dots, B_{s'}^y$ is the decomposition for a string y , obtained using the same randomness, where $\Delta_{\text{edit}}(x, y) \leq k$ then with good probability, $s = s'$ and $B_i^x = B_i^y$ for all but k indices i . The edit distance of x and y can be calculated as $\Delta_{\text{edit}}(x, y) = \sum_i \Delta_{\text{edit}}(B_i^x, B_i^y)$ where i ranges over the differing blocks.

First we provide an overview of the algorithm, specific details are given in the next sub-section. The decomposition procedure proceeds in $\mathcal{O}(\log n)$ rounds. In each round, the algorithm maintains a decomposition of x into *compressed* blocks. In each round each block of size at least two is first *compressed* and then *split*. The compression is done by compressing pairs of consecutive symbols into one using a randomly chosen pair-wise independent hash function $C_\ell : \Gamma^2 \rightarrow \Gamma$, where ℓ is the round number (*level*). Non-overlapping pairs of symbols are chosen for compression using a *locally consistent coloring* so that every three symbols shrink to at most two. Prior to the compression of pairs we replace each repeated sequence a^r of a symbol a , $r \geq 2$, by a special character $\mathbf{r}_{a,r}$.

The splitting procedure uses a $\tilde{\mathcal{O}}(k)$ -wise independent hash function $H_\ell : \Gamma^2 \rightarrow \{0, \dots, D-1\}$ to select places where to subdivide each block into sub-blocks, where $D = \tilde{\mathcal{O}}(k)$ is a suitable parameter. We start a new block at each consecutive pair of symbols ab , where $H_\ell(ab) = 0$.

After $\mathcal{O}(\log n)$ rounds, each block is compressed into at most two symbols and we output a grammar that can generate the block.

For the correctness of the algorithm we will need to establish several properties of the algorithm. Some of these properties are related to behaviour on a single string x , others analyze the behaviour of the procedure on a pair of strings x and y of edit distance at most k .

The properties we want from the algorithm when it runs on x are the following: In each round, each block should be compressed by factor at least $2/3$ while the size of the required grammar capturing the compression should be $\tilde{\mathcal{O}}(k)$. The former is achieved by the design of the compression procedure. The latter goal is provided by the property of the splitting procedure which makes sure that each block $B = b_1 b_2 \cdot b_m$ resulting from a split has small dictionary $\text{Dict}(B) = \{b_i b_{i+1}, i = 1, \dots, m-1\}$. In particular, we require $|\text{Dict}(B)| = \tilde{\mathcal{O}}(k)$. The grammar size will be proportional to this dictionary.

For the compression procedure we require that it preserves information so the

function C_ℓ is one-to-one on each $\text{Dict}(B)$. Since the total size of all dictionaries is bounded by $\tilde{\mathcal{O}}(n)$ this can be easily achieved by picking C_ℓ at random provided that its range size is $\Omega(n^3)$.

Additionally, we need the following property to hold on a pair of strings x and y of edit distance at most k with good probability: The splitting procedure should never split x or y in a *region* which is affected by edit operations that transform x to y (for some canonical choice of those operations.) The total size of those regions will be again $\tilde{\mathcal{O}}(k)$ so we can satisfy this property if each pair of symbols has probability at most $1/\tilde{\mathcal{O}}(k)$ to start a new block. This constrains the choice of the range size for the splitting function H_ℓ .

In the next section we describe the decomposition algorithm fully, and then we establish its properties.

2.3.3 Algorithm description

Let n be an upper bound on the length of the input string and $k \leq n$ be given. Set $L = \lceil \log_{3/2} n \rceil + 3$ to be an upper bound on the decomposition depth. Let Σ be an input alphabet of size at most n^3 , $\Sigma_c = \{c_1, c_2, \dots, c_{Ln^3}\}$ and $\Sigma_r = \{r_{a,r}, a \in \Sigma \cup \Sigma_c, r \in \{2, 3, \dots, n\}\}$ be auxiliary pair-wise disjoint alphabets. Let $\Gamma = \Sigma \cup \Sigma_c \cup \Sigma_r$ be the working alphabet, and $\#$ be a symbol not in Γ . Notice $|\Gamma| = \mathcal{O}(n^5 + |\Sigma|)$. We call symbols from $\Sigma_c^0 = \Sigma$ *level-0 compression symbols*, and for $\ell \geq 1$, symbols from $\Sigma_c^\ell = \{c_i, (\ell - 1)n^3 < i \leq \ell n^3\}$ are *level- ℓ compression symbols*. Additionally, symbols from $\Sigma_r^\ell = \{r_{a,r} \in \Sigma_r, a \text{ is a level-}(\ell - 1) \text{ compression symbol}\}$ are also *level- ℓ compression symbols*.

Let $R = \log^* |\Gamma| + 20$, $D = 110R(L+1)k$ and $S = 30DL \log n + 6$ be parameters. The algorithm is a recursive algorithm of depth at most L . It starts by selecting at random several hash functions: For $\ell = 1, \dots, L$, it selects at random a compression hash function $C_\ell : \Gamma^2 \rightarrow \Sigma_c^\ell$ from a pair-wise independent hash family, and for $\ell = 0, \dots, L$, it selects at random a splitting function $H_\ell : \Gamma^2 \rightarrow \{0, \dots, D - 1\}$ from a $(5D \log n)$ -wise independent hash family.

Main building blocks of the algorithm are two functions, Compress and Split. The first one compresses strings by a factor of $2/3$, and the other splits strings at random points. Their pseudo-code is provided as Algorithm 1 and 2. We describe them next.

Compress. The function $\text{Compress}(B, \ell)$ takes as input a string B over alphabet Γ of length at least two, and an integer $\ell \geq 1$, which denotes the level number. Divide B into minimum number of blocks B_1, \dots, B_m , $B = B_1 B_2 B_3 \dots B_m$, so that in each B_i either all the characters are the same, i.e. $B_i = a^r$ for some $a \in \Gamma$ and $r \geq 2$, or no two adjacent characters are the same. The first step is to compress the B_i 's which contain repeated characters by simply replacing the whole B_i with the symbol $r_{a,|B_i|}$, where a is the repeated character. Then for the remaining blocks, the following compression is applied: Let B_i be an uncompressed block. Each character of B_i is colored by applying $F_{\text{CVL}}(B_i)$. Divide B_i into blocks $B_i = B'_1 B'_2 \dots B'_s$, such that for each B'_j only the first character is colored 1. Now, according to Proposition 2.7, length of each B'_j is either 2 or 3. If $B'_j = ab$, replace it with $C_\ell(ab)$ else if $B'_j = abc$, replace it with $C_\ell(ab) \cdot c$, where $a, b, c \in \Gamma$. The actual pseudo-code given below performs the compression of blocks of repeats in two stages, where in the first stage we replace the repeated sequence a^r by

$r_{a,r} \cdot \#$, and then in the next stage we remove the extra symbol $\#$. This simplifies analysis in Lemma 2.17. Assuming that C_ℓ can be evaluated in time $\mathcal{O}(1)$, the running time of $\text{Compress}(B, \ell)$ is dominated by the time needed to compute F_{CVL} -coloring of blocks which is $\mathcal{O}(R \cdot |B|)$ in total.

Algorithm 1 $\text{Compress}(B, \ell)$

Input: String B over alphabet Γ of length at least two, and level number ℓ .

Output: String B'' over alphabet Γ .

- 1 Divide $B = B_1 B_2 B_3 \dots B_m$ into minimum number of blocks so that each maximal subword a^r of B , for $a \in \Gamma$ and $r \geq 2$, is one of the blocks.
 - 2 **for** each $i \in \{1, \dots, m\}$ **do**
 - 3 **if** $B_i = a^r$, where $r \geq 2$ **then** Set $B'_i = r_{a,r} \cdot \#$ and color $r_{a,r}$ by 1 and $\#$ by 2.¹;
 - 4 **else** Set $B'_i = B_i$ and color each symbol of B'_i according to $F_{\text{CVL}}(B_i)$;
 - 5 **end**
 - 6 Set $B' = B'_1 B'_2 \dots B'_m$, $B'' = \varepsilon$, and $i = 1$.
 - 7 **while** $i < |B'|$ **do**
 - 8 **if** $B'[i+1] = \#$ **then** $B'' = B'' \cdot B'[i]$;
 - 9 **else** $B'' = B'' \cdot C_\ell(B'[i, i+1])$;
 - 10 $i = i + 2$.
 - 11 **if** $i \leq |B'|$ and $B'[i]$ is not colored 1 **then** $B'' = B'' \cdot B'[i]$, $i = i + 1$;
 - 12 **end**
 - 13 Return B'' .
-

Split. The function takes as input a string B over alphabet Γ of length at least two, and an integer $\ell \geq 1$. The function splits the string B into smaller blocks. The algorithm works as follows: For each $i \in \{2, \dots, |B| - 1\}$, if $H_\ell(B[i, i+1]) = 0$, start a new block at position i . The running time of $\text{Split}(B, \ell)$ is dominated by the time to evaluate H_ℓ at $|B| - 2$ points.

Algorithm 2 $\text{Split}(B, \ell)$

Input: String B over alphabet Γ of length at least two, and level number ℓ .

Output: A sequence of strings (B_0, B_1, \dots, B_s) over alphabet Γ .

- 14 Let $i_1 < \dots < i_s$ be all $i \in \{2, \dots, |B| - 1\}$ where $H_\ell(B[i, i+1]) = 0$. Set $s = 0$ if no such i exists.
 - 15 Let $i_0 = 1$ and $i_{s+1} = |B| + 1$.
 - 16 For $j = 0, \dots, s$, set $B_j = B[i_j, i_{j+1})$.
 - 17 Return (B_0, B_1, \dots, B_s) .
-

The main recursive step of the algorithm is encompassed in function Process . The function gets a block $B \in \Gamma^*$ as its input. The block might have already been compressed previously, so the function also gets dictionaries that allow decompression of the block. If the block is already of length at most two, then the

¹If $a = r_{b,s}$ for some $b \in \Gamma$ and $s \in \mathbb{N}$, then set $B'_i = r_{b,rs} \cdot \#$. However, such a situation should never happen during the execution of the algorithm as level- ℓ compression symbol can be introduced only at level ℓ .

function outputs the block. Otherwise it compresses the block B using Compress, then it subdivides the compressed block using Split, and invokes itself recursively on each sub-block. For the output, each block is represented by a grammar. The grammar is reconstructed from the compressed block and its dictionaries by a simple bread-first search algorithm provided in the function Grammar.

Algorithm 3 Process($B, (D_1, D_2, \dots, D_{\ell-1}), \ell$)

Input: String $B \in \Gamma^*$, a sequence of dictionaries $D_i \subseteq \Gamma^2$ for decompressing B , and level number ℓ .

Output: A sequence of blocks of B each encoded by a grammar.

```

18 if  $|B| \leq 2$  then Output Grammar( $B, (D_1, D_2, \dots, D_{\ell-1}), \ell - 1$ ) and return ;
19  $A = \text{Compress}(B, \ell)$ .
20  $(B_0, B_1, \dots, B_s) = \text{Split}(A, \ell)$ .
21 For  $i = 0, \dots, s$ , Process( $B_i, (D_1, \dots, D_{\ell-1}, \text{Dict}(B)), \ell + 1$ ).

```

To decompose an input string x into blocks, we first apply function Split($x, 0$) to x and then invoke Process($B, (), 1$) on each of the obtained blocks B . Breaking the string x into sub-blocks guarantees that each block passed to Process has small dictionary whereas the dictionary of x could have been arbitrarily large.

Algorithm 4 Grammar($B, (D_1, D_2, \dots, D_\ell), \ell$)

Input: String $B \in \Gamma^*$, a sequence of dictionaries $D_i \subseteq \Gamma^2$ for decompressing B .

Output: The smallest grammar G for B based on the dictionaries D_i and hash functions C_1, \dots, C_ℓ .

```

22 Let  $C = \{c \in \Sigma_c : c \text{ appears in } B \text{ or } \mathbf{r}_{c,r} \text{ appears in } B \text{ for some } r\}$ . // Symbols
    needed to decompress  $B$ 
23  $G = \{\# \rightarrow B\}$ .
24 for  $j = \ell, \dots, 1$  do
25     for each  $ab \in D_j$  do
26         if  $C_j(ab) \in C$  then  $G = G \cup \{C_j(ab) \rightarrow ab\}$ ,
27          $C = C \cup \{c \in \Sigma_c; c \in \{a, b\} \text{ or } \mathbf{r}_{c,r} \in \{a, b\} \text{ for some } r\}$ ;
28     end
29 end
30 For each  $\mathbf{r}_{a,r}$  appearing in any of the rules in  $G$ , add  $\mathbf{r}_{a,r} \rightarrow a^r$  to  $G$ .
31 Return  $G$ .

```

2.3.4 Correctness of the decomposition algorithm

Our goal is to establish the following theorem which is a stronger version of Theorem 2.3:

Theorem 2.8. *Let x and y be a pair of strings of length at most n with $\Delta_{edit}(x, y) \leq k$. Let G_1^x, \dots, G_s^x and $G_1^y, \dots, G_{s'}^y$ be the sequence of grammars output by the decomposition algorithm on input x and y respectively, using the same choice of random functions C_1, \dots, C_L and H_0, \dots, H_L . The following is true for n large enough:*

1. With probability at least $1 - 2/n$, $x = \text{eval}(G_1^x) \cdots \text{eval}(G_s^x)$ and $y = \text{eval}(G_1^y) \cdots \text{eval}(G_{s'}^y)$.
2. With probability at least $1 - 2/\sqrt{n}$, for all $i \in \{1, \dots, s\}$ and $j \in \{1, \dots, s'\}$, $|G_i^x|, |G_j^y| \leq S$.
3. With probability at least $9/10$, $s = s'$, $G_i^x = G_i^y$, for all $i \in \{1, \dots, s\}$ except for at most k indices i , and $\Delta_{\text{edit}}(x, y) = \sum_i \Delta_{\text{edit}}(\text{eval}(G_i^x), \text{eval}(G_i^y))$.

By union bound, all three parts happen simultaneously with probability at least $9/10 - 2/n - 1/\sqrt{n}$ which is $\geq 4/5$ for n large enough.

To prove the theorem we make some simple observations about the algorithm, first.

Lemma 2.9. *For any string B of length at least two, and $\ell \geq 1$, $|\text{Compress}(B, \ell)| \leq \frac{2}{3}|B| + 1$ and $|\text{Compress}(B, \ell)| < |B|$.*

Proof. Let $B = B_1 B_2 B_3 \dots B_m$ be as in the procedure. Every block B_i that equals to a^r , for some a and $r \geq 2$, is reduced to one symbol by the compression. The other blocks are colored using $F_{\text{CVL}}(\cdot)$ and compressed. Unless a block B_i is of size one, the coloring induces division of the block B_i into subwords of size two or three, where the former is compressed into one symbol and the latter into two symbols. Hence, each such a block is compressed to at most $2/3$ of its size. So the only blocks B_i that do not shrink are of size one, and are sandwiched between blocks of repeated symbols (that shrink by a factor of at least two). The worst-case situation is when m is odd, blocks B_i are of size one for odd i , and of size two for even i . In that case the original string B shrinks to size $\lfloor \frac{2}{3}|B| \rfloor + 1$. This proves the first inequality. The second inequality is also clear from the analysis above: The only time the string does not shrink is if it is of size one. \square

Corollary 2.10. *On a string B of length at most n , the depth of the recursive calls of Process is at most L .*

Indeed, from the previous lemma it follows that each block after ℓ compressions and splits is of size at most $(2/3)^\ell |B| + 3$. Hence, after $L = \lceil \log_{3/2} n \rceil + 3$ recursive calls Process must stop the recursion.

Lemma 2.11. *Let $B \in \Gamma^*$ be of length at most n , and $\ell \in \{0, \dots, L\}$. Let $(B_0, B_1, \dots, B_s) = \text{Split}(B, \ell)$ where $H_\ell : \Gamma^2 \rightarrow \{0, \dots, D-1\}$ is chosen at random from $(5D \log n)$ -wise independent hash family. Then with probability at least $1 - 1/n^3$, for all $j \in \{0, \dots, s\}$, $|\text{Dict}(B_j)| \leq 5D \log n$.*

Proof. If for some $j \in \{0, \dots, s\}$, $|\text{Dict}(B_j)| > 5D \log n$, then there exists $1 < r < t \leq |B|$ such that $|\text{Dict}(B[r, t])| = 5D \log n$ and for all $i \in \{r, \dots, t-1\}$, $H_\ell(B[i, i+1]) \neq 0$. (Pick r to be the position in B of the second symbol of B_j and r some later position in B_j .) For a fixed r and t with $|\text{Dict}(B[r, t])| = 5D \log n$, $\Pr_{H_\ell}[\forall i \in \{r, \dots, t-1\}, H_\ell(B[i, i+1]) \neq 0] \leq \left(1 - \frac{1}{D}\right)^{5D \log n}$ by the $(5D \log n)$ -wise independence of H_ℓ . Hence, $\Pr_{H_\ell}[\exists 1 < r < t \leq |B|, |\text{Dict}(B[r, t])| = 5D \log n \text{ and } \forall i \in \{r, \dots, t-1\}, H_\ell(B[i, i+1]) \neq 0] \leq |B|^2 \left(1 - \frac{1}{D}\right)^{5D \log n} \leq n^2 e^{-5 \log n} \leq 1/n^3$. \square

Lemma 2.12. For $B \in \Gamma^*$, $\ell \leq L$, $D_1, D_2, \dots, D_\ell \subseteq \Gamma^2$, $\text{Grammar}(B, (D_1, \dots, D_\ell), \ell)$ outputs a grammar G of size at most $3|B| + 6 \sum_i |D_i|$, and runs in time $\tilde{\mathcal{O}}(|B| + \sum_i |D_i|)$.

Proof. The main loop of the algorithm iterates over all the pairs from D_j . In each iteration we can add a rule of the type $c \rightarrow ab$ to G . Hence, the number of such rules in G is at most $|B| + 2 \sum_i |D_i|$. Last, we add to G rules for symbols from Σ_r that appear on right hand sides of rules in G . This increases the size of G by at most factor of 3. If C is stored using some efficient data structure such as binary search trees or hash tables, each iteration takes $\tilde{\mathcal{O}}(1)$ time. (We assume that evaluation of $C_j(\cdot)$ takes $\mathcal{O}(1)$.) Hence, the total running time is bounded by claimed bound. \square

During processing of a string x , there are at most Ln calls to the function Split . (The actual number of calls is $\mathcal{O}(n)$ as the strings shrink exponentially but our simple upper bound suffices.) The probability that any one of them would produce a block with dictionary larger than $5D \log n$ is at most Ln/n^3 . We can conclude the next corollary which implies the second item of Theorem 2.8.

Corollary 2.13. For n large enough, on a string x of length at most n , processing the string x produces a sequence of grammars each of size at most $S = 30DL \log n + 6$ with probability at least $1 - 1/n$.

For the grammars produced by the algorithm to be deterministic, we need that each C_ℓ is one-to-one on $\text{Dict}(B)$ for each block B on which $\text{Compress}(B, \ell)$ is invoked. That will happen with high probability by a standard argument:

Lemma 2.14. Let $B \in \Gamma^*$ be of length at most n and $\ell \in \{1, \dots, L\}$. Let $C_\ell : \Gamma^2 \rightarrow \{c_i, (\ell - 1)n^3 < i \leq \ell n^3\}$ be chosen at random from a pair-wise independent family of hash functions. Then with probability at least $1 - |B|/n^2$, C_ℓ is one-to-one on $\text{Dict}(B)$.

Proof. For two distinct elements from $\text{Dict}(B)$, the probability of a collision for randomly chosen C_ℓ is at most $1/n^3$. By the union bound, the probability that C_ℓ is not one-to-one on $\text{Dict}(B_j)$ is at most $|\text{Dict}(B)|^2/n^3 \leq |B|/n^2$ as $|\text{Dict}(B)| \leq |B| \leq n$. \square

During processing of a string x , there are at most Ln calls to the function Compress . For a fixed level $\ell \in \{1, \dots, L\}$, the total size of blocks B for which $\text{Compress}(B, \ell)$ is invoked is at most n . By the previous lemma and the union bound, the probability that during any of those calls $\text{Compress}(B, \ell)$ uses a function C_ℓ that is not one-to-one on $\text{Dict}(B)$ is at most $1/n$. If all the hash functions C_1, C_2, \dots, C_L that are used to compress blocks of x are one-to-one on their respective blocks then the grammars that Grammar produces will be deterministic, and they will evaluate to their respective blocks of x . (We can actually conclude a stronger statement that each C_ℓ will be one-to-one on the union of all blocks at level ℓ with high probability.) We can conclude the next corollary which implies the first item of Theorem 2.8.

Corollary 2.15. For n large enough, on a string x of length at most n , with probability at least $1 - L/n$, processing the string x produces a sequence of grammars G_1, G_2, \dots, G_s such that $x = \text{eval}(G_1) \cdots \text{eval}(G_s)$.

At this point we can estimate the running time of the decomposition algorithm. We can let the algorithm fail, and produce some trivial decomposition of x , whenever Split produces a block with dictionary larger than $5D \log n$. If it does not fail, then all grammars are of size at most S which is $\tilde{\mathcal{O}}(k)$. There are at most n of them so time spent in Grammar(...) is bounded by $\tilde{\mathcal{O}}(nk)$. The total time spent in Compress(...) is proportional to the sum of sizes of all non-trivial blocks over all levels of recursion which is $\mathcal{O}(nL) = \tilde{\mathcal{O}}(n)$. (A more accurate estimate on the total size of blocks is $\mathcal{O}(n)$ since the blocks are shrinking geometrically in each iteration.) This means that the time to execute all calls to Compress is $\mathcal{O}(nLR) = \tilde{\mathcal{O}}(n)$. The time spent in Split(...) is dominated by the time needed to evaluate H_ℓ . The number of evaluation points at a given level ℓ is proportional to the total size of all blocks at that level. Since H_ℓ can be evaluated at a single point in time $\mathcal{O}(D \log n) = \tilde{\mathcal{O}}(k)$, we get a trivial upper bound $\mathcal{O}(nLD \log n) = \tilde{\mathcal{O}}(nk)$ on time spent in Split. Hence, in total the decomposition procedure runs in time $\tilde{\mathcal{O}}(nk)$. (We believe that the total running time can be improved to $\tilde{\mathcal{O}}(n)$ on average. One could argue that in expectation the number of grammars the procedure produces is $\tilde{\mathcal{O}}(n/k)$ as the average block size a string x is decomposed into should be at least $\Omega(D/\log n)$. So we believe that the total running time of calls to Grammar is $\tilde{\mathcal{O}}(n)$. Using multi-point evaluation of $(5D \log n)$ -wise independent hash functions we could reduce the time for evaluation of H_ℓ on a given level to $\tilde{\mathcal{O}}(n)$.)

Proposition 2.16. *Given $k \leq n$, the running time of the decomposition algorithm on a string x of length at most n is $\tilde{\mathcal{O}}(nk)$ with probability at least $1 - 1/n$.*

It remains to address the properties of the algorithm run on a pair of strings x and y of edit distance at most k to establish Theorem 2.8. For the pair of strings x and y we fix a *canonical decomposition of x and y* to be a sequence of words $w_0, w_1, \dots, w_k, u_i, \dots, u_k, v_1, \dots, v_k \in \Gamma^*$ such that $x = w_0 u_1 w_1 u_2 w_2 \dots u_k w_k$, $y = w_0 v_1 w_1 v_2 w_2 \dots v_k w_k$ and $|u_i|, |v_i| \leq 1$ for all i . By the definition of edit distance such a decomposition exists: each pair (u_i, v_i) represents one edit operation, and we fix one such decomposition to be *canonical*. Observe, if we now partition x into blocks B_1^x, \dots, B_s^x so that each B_i^x starts within one of the w_j 's, and we partition y into blocks B_1^y, \dots, B_s^y so that each block B_i^y starts at the corresponding location in w_j as B_i^x , then $\Delta_{edit}(x, y) = \sum_i \Delta_{edit}(B_i^x, B_i^y)$.

We need to understand what happens with the decomposition of x and y when we apply the Compress function. Let $x = uvw$ and $x' = \text{Compress}(x, \ell) = u'w'v'$, for some $u, w, v, u'w'v' \in \Gamma^*$. We say that a symbol c in w' *comes from the compression of w* if either it is directly copied from w by Compress, or it is the image $c = C_\ell(ab)$ of a pair of symbols ab where a belongs to w , or $c = \mathbf{r}_{a,r}$ replaced a block a^r where the first symbol of a^r belongs to w . w' *is the compression of w* if it consists precisely of the symbols that come from the compression of w . Furthermore, we say a symbol c in w' *comes weakly from the compression of w* if either it is directly copied from w by Compress, or it is the image $c = C_\ell(ab)$ of a pair of symbols ab where a or b belong to w , or $c = \mathbf{r}_{a,r}$ replaced a block a^r where some symbol of a^r belongs to w . w' *is the weak compression of w* if it consists precisely of the symbols that come weakly from the compression of w . Notice, a weak compression of w might contain an extra symbol at the beginning compared to the compression of w .

The following lemma captures what compression does to the canonical decomposition of x and y . (See Fig. 2.2 for illustration.)

Lemma 2.17. *Let x and y be strings over Γ , and let $x' = \text{Compress}(x, \ell)$ and $y' = \text{Compress}(y, \ell)$. Let $x = w_0 u_1 w_1 u_2 w_2 \cdots u_q w_q$ and $y = w_0 v_1 w_1 v_2 w_2 \cdots v_q w_q$ for some strings w_i, u_i and v_i where for $i \in \{1, \dots, q\}$, $|u_i|, |v_i| \leq 4R + 24$. Then there are $w'_0, w'_1, \dots, w'_q, u'_1, \dots, u'_q, v'_1, \dots, v'_q \in \Gamma^*$ such that for $i \in \{1, \dots, q\}$, $|u'_i|, |v'_i| \leq 4R + 24$, $x' = w'_0 u'_1 w'_1 u'_2 w'_2 \cdots u'_q w'_q$ and $y' = w'_0 v'_1 w'_1 v'_2 w'_2 \cdots v'_q w'_q$. Moreover, each w'_i is the compression of the same subword of w_i in both x and y .*

For each $x = w_0 u_1 w_1 u_2 w_2 \cdots u_q w_q$, $y = w_0 v_1 w_1 v_2 w_2 \cdots v_q w_q$ and ℓ we fix one choice of $w'_0, \dots, w'_q, u'_0, \dots, u'_q, v'_0, \dots, v'_q$ satisfying the lemma. We will refer to it as the *canonical decomposition* of x' and y' induced by the decomposition of x and y as given by the lemma.

Proof. The first stage of Compress replaces maximal blocks of repeated symbols by shortcuts. To simplify our analysis first we will reassign blocks of repeated symbols among neighboring blocks of w_i, u_i and v_i , resp., so each maximal block of symbols in x and y is fully contained in one of the words w_i, u_i or v_i .

For $i = 1, \dots, q-1$ we define words $w_i^{(1)}$ and parameters $a_i, b_i \in \Gamma$ and $k_i, k'_i \in \mathbb{N}$ as follows: If w_i contains at least two distinct symbols let $w_i = a_i^{k_i} w_i^{(1)} b_i^{k'_i}$ so that k_i and k'_i are maximum possible, otherwise $w_i = a_i^{k_i}$ for some a_i and k_i (k_i might be zero), and we set $w_i^{(1)} = \varepsilon$, $b_i = a_i$ and $k'_i = 0$. Let $w_0 = w_0^{(1)} b_0^{k'_0}$ for maximum possible k'_0 and some symbol b_0 . Let $w_q = a_q^{k_q} w_q^{(1)}$ for maximum possible k_q and some symbol a_q . For $i = 1, \dots, q$, we let $u_i^{(1)} = b_{i-1}^{k'_{i-1}} u_i a_i^{k_i}$. Similarly, $v_i^{(1)} = b_{i-1}^{k'_{i-1}} v_i a_i^{k_i}$. Hence, $x = w_0^{(1)} u_1^{(1)} w_1^{(1)} \cdots u_q^{(1)} w_q^{(1)}$ and $y = w_0^{(1)} v_1^{(1)} w_1^{(1)} \cdots v_q^{(1)} w_q^{(1)}$.

Next, if there is a maximal block of symbols a^r contained in $u_s^{(1)} w_s^{(1)} \cdots u_t^{(1)}$ starting in $u_s^{(1)}$ and ending in $u_t^{(1)}$, $s \neq t$, we add all the symbols of the a^r to the end of $u_s^{(1)}$ and remove them from the other $u_i^{(1)}$, $i = s+1, \dots, t$. (Notice, $w_i^{(1)} = \varepsilon$ for $s < i < t$ because of the definition of $w_i^{(1)}$, and $u_i^{(1)}$ will become empty for $s < i < t$.) We do this for all maximal blocks of repeated symbols that span multiple $u_i^{(1)}$. We perform similar moves on $v_i^{(1)}$'s. After all of those moves we denote the resulting subwords by $w_i^{(2)}, u_i^{(2)}$, and $v_i^{(2)}$. (Notice, $w_i^{(2)} = w_i^{(1)}$ for all i .) We have: $x = w_0^{(2)} u_1^{(2)} w_1^{(2)} \cdots u_q^{(2)} w_q^{(2)}$ and $y = w_0^{(2)} v_1^{(2)} w_1^{(2)} \cdots v_q^{(2)} w_q^{(2)}$. At this stage, each maximal block of repeated symbols in x or y is contained in one of the subwords $w_i^{(2)}, u_i^{(2)}$, and $v_i^{(2)}$.

The first stage of Compress replaces each maximal block a^r , $r \geq 2$, by a sequence $\mathbf{r}_{a,r}\#$, and we apply this procedure on each subword $w_i^{(2)}, u_i^{(2)}$, and $v_i^{(2)}$ to obtain corresponding subwords $w_i^{(3)}, u_i^{(3)}$, and $v_i^{(3)}$. Observe, for $i = 1, \dots, q$, $|u_i^{(3)}|, |v_i^{(3)}| \leq 4R + 28$. This is because every u_i is transformed into $u_i^{(3)}$ by appending or prepending possibly empty block of repeated symbols, i.e., $u_i^{(3)} = a^r u_i b^{r'}$ for some a, b, r, r' , or removing its content entirely. Each block of repeats is reduced to two symbols so each $u_i^{(3)}$ is longer than the original by at most 4 symbols. Similarly for $v_i^{(3)}$.

Next, coloring function F_{CVL} is used on parts of x and y that are not obtained from repeated symbols; the two symbols replacing each repeated block are colored by 1 and 2, resp. We refer to this as $\{1, 2, 3\}$ -coloring. At most R first and last

symbols of each $w_i^{(3)}$ might be colored differently in x and y as the color of each symbol depends on the context of at most R symbols on either side of the symbol, and that context might differ in x and y . Hence, only symbols near the border of $w_i^{(3)}$ that are in vicinity of $u_i^{(3)}$'s and $v_i^{(3)}$'s, resp., might get different colors. All the other symbols of $w_i^{(3)}$ are colored the same in both x and y . The coloring is then used to make decisions on which pairs of symbols are compressed into one.

We will let u'_i be the symbols that come from the compression of symbols in $w_i^{(3)}$, the first up-to $R+2$ symbols of $w_i^{(3)}$, and the last up-to $R+3$ symbols of $w_{i-1}^{(3)}$. Next we specify precisely which symbols of $w_i^{(3)}$ and $w_{i-1}^{(3)}$ are considered to be compressed into symbols belonging to u'_i . For $i = 0, \dots, q$, if $|w_i^{(3)}| \geq R+3$, let s_i^x be the position of the first symbol in $w_i^{(3)}$ among positions $R+1, R+2, R+3$ which is colored 1 in x by the $\{1, 2, 3\}$ -coloring. If $|w_i^{(3)}| < R+3$, let $s_i^x = 1$. Next, if $|w_i^{(3)}| \geq 2R+3$ set t_i^x to be the first position from left colored 1 among the symbols of $w_i^{(3)}$ at positions $R+1, R+2, R+3$ counting from right. If $|w_i^{(3)}| < 2R+3$, set t_i^x to be equal to s_i^x . For $i = 0$, if $|w_0^{(3)}| \geq R+3$ then redefine $s_0^x = 1$. For $i = q$, redefine $t_q^x = |w_q^{(3)}| + 1$ and if $|w_q^{(3)}| < R+3$ then redefine s_q^x to t_q^x . Similarly, define s_i^y and t_i^y based on the $\{1, 2, 3\}$ -coloring of y .

Notice, $s_i^x \neq t_i^x$ iff $s_i^y \neq t_i^y$. Furthermore, if $s_i^x \neq t_i^x$ then either $i \in \{q, 0\}$ or $|w_i^{(3)}| \geq 2R+3$ so $s_i^x = s_i^y$ and $t_i^x = t_i^y$ as the symbols R -away from either end of $w_i^{(3)}$ are colored the same in x and y . We let u'_i to be the compression of $w_{i-1}^{(3)}[t_{i-1}^x, |w_{i-1}^{(3)}|] \cdot u_i^{(3)} \cdot w_i^{(3)}[1, s_i^x]$ and similarly, v'_i to be the compression of $w_{i-1}^{(3)}[t_{i-1}^y, |w_{i-1}^{(3)}|] \cdot v_i^{(3)} \cdot w_i^{(3)}[1, s_i^y]$. We let w'_i be the compression of $w_i^{(3)}[s_i^y, t_i^y]$.

Hence, u'_i comes from the compression of at most $|u_i^{(3)}| + 2R + 5 \leq 6R + 33$ symbols. Since each symbol after a symbol colored 1 is *removed* by the compression, and each consecutive triple of symbols contains at least one symbol colored by 1, the at most $6R+27$ symbols are compressed into at most $(6R+33) \cdot 2/3 + 2 = 4R+24$ symbols. So u'_i is of length at most $4R + 24$. Similarly for v'_i . \square

The following generalization of the previous lemma will be useful to design a rolling sketch. It considers situation where x and y are prefixed by some strings u and v , resp., that we want to ignore from the analysis. The proof of the lemma is a straightforward modification of the above proof.

Lemma 2.18. *Let $x, y, u, v \in \Gamma^*$, and let $u'x' = \text{Compress}(ux, \ell)$ and $v'y' = \text{Compress}(vy, \ell)$, where x' is the weak compression of x , and y' is the weak compression of y . Let $x = u_0w_0u_1w_1u_2w_2 \cdots u_qw_q$ and $y = v_0w_0v_1w_1v_2w_2 \cdots v_qw_q$ for some strings w_i , u_i and v_i where for $i \in \{0, \dots, q\}$, $|u_i|, |v_i| \leq 4R+24$. Then there are $w'_0, w'_1, \dots, w'_q, u'_0, u'_1, \dots, u'_q, v'_0, v'_1, \dots, v'_q \in \Gamma^*$ such that for $i \in \{0, \dots, q\}$, $|u'_i|, |v'_i| \leq 4R + 24$, $x' = u'_0w'_0u'_1w'_1u'_2w'_2 \cdots u'_qw'_q$ and $y' = v'_0w'_0v'_1w'_1v'_2w'_2 \cdots v'_qw'_q$. Moreover, each w'_i is the compression of the same subword of w_i in both x and y .*

Let $x \in \Sigma^*$. Let $H_0, H_1, \dots, H_L, C_1, C_2, \dots, C_L$ be chosen. We define inductively the *trace* of the algorithm on x at level $\ell \geq 0$ to consist of sequences $B^x(\ell, 1), \dots, B^x(\ell, s_\ell^x) \in \Gamma^*$, of auxiliary sequences $A^x(\ell, 1), \dots, A^x(\ell, s_\ell^x) \in \Gamma^*$ and $t_{\ell, 1}^x, \dots, t_{\ell, s_\ell^x+1}^x \in \mathbb{N}$. Their meaning is: $B^x(\ell, i)$ is compressed into $A^x(\ell, i)$ and that is split into blocks $B^x(\ell+1, j)$ for $t_{\ell+1, i}^x \leq j < t_{\ell+1, i+1}^x$. (See Fig. 2.1 for

illustration.)²

Set

$$B^x(0, 1), \dots, B^x(0, s_0^x) = \text{Split}(x, 0).$$

For $\ell = 1, \dots, L$ we define $B^x(\ell, 1), \dots, B^x(\ell, s_\ell^x)$ inductively. Set $t_{\ell,1}^x = 1$. For $i = 1, \dots, s_{\ell-1}^x$, if $|B^x(\ell - 1, i)| \geq 2$, then

$$A^x(\ell - 1, i) = \text{Compress}(B^x(\ell - 1, i), \ell),$$

and for $(B_0, B_1, \dots, B_s) = \text{Split}(A^x(\ell - 1, i), \ell)$ set

$$B^x(\ell, t_{\ell,i}^x) = B_0, \quad B^x(\ell, t_{\ell,i}^x + 1) = B_1, \quad \dots, \quad B^x(\ell, t_{\ell,i}^x + s) = B_s$$

and $t_{\ell,i+1}^x = t_{\ell,i}^x + s + 1$. If $|B^x(\ell - 1, i)| < 3$, then set $B^x(\ell, t_{\ell,i}^x)$ and $A^x(\ell - 1, i)$ to $B^x(\ell - 1, i)$, and $t_{\ell,i+1}^x = t_{\ell,i}^x + 1$. For $j = s_{\ell-1}^x$, set $s_\ell^x = t_{\ell,j+1}^x$.

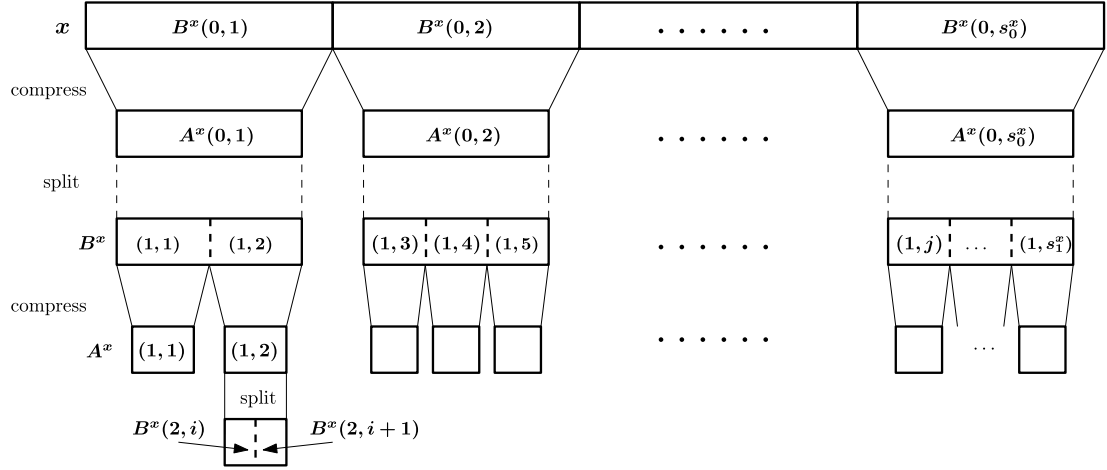


Figure 2.1 The hierachical decomposition of x .

Furthermore, for x and $y \in \Sigma^*$, $\ell, i \geq 0$, define a canonical decomposition of blocks $A^x(\ell, i)$, $B^x(\ell, i)$, $A^y(\ell, i)$, $B^y(\ell, i)$ inductively as follows. Let $A^x(-1, 1) = x$ and $A^y(-1, 1) = y$. Let $t_{-1,1}^x = 1, t_{-1,2}^x = 2, s_{-1}^x = 1, t_{-1,1}^y = 1, t_{-1,2}^y = 2$, and $s_{-1}^y = 1$. Let

$$A^x(-1, 1) = w_0 u_1 w_1 u_2 w_2 \cdots u_k w_k \quad \& \quad A^y(-1, 1) = w_0 v_1 w_1 v_2 w_2 \cdots v_k w_k$$

be the canonical decomposition of the pair x and y .

For $\ell \geq 0$ and $j \in \{1, \dots, s_\ell^x\}$, let i be such that $t_{\ell-1,i}^x \leq j < t_{\ell-1,i+1}^x$ and $m = j - t_{\ell-1,i}^x$. Then $B^x(\ell, j)$ is the m -th block of $\text{Split}(A^x(\ell - 1, i), \ell)$. If the decomposition of $A^x(\ell - 1, i)$ is defined and is equal to $w_0 u_1 w_1 u_2 w_2 \cdots u_q w_q$, for some $u_i, w_i \in \Gamma^*$, then the decomposition of $B^x(\ell, j)$ is the restriction of the decomposition of $A^x(\ell - 1, i)$ to symbols of the m -th block of $\text{Split}(A^x(\ell - 1, i), \ell)$. Otherwise the decomposition of $B^x(\ell, j)$ is undefined. Similarly for $B^y(\ell, j)$. (See Fig. 2.2.)

For $\ell \geq 0$ and $j \in \{1, \dots, s_\ell^x\}$, if $B^x(\ell, j)$ and $B^y(\ell, j)$ have defined decompositions $B^x(\ell, j) = w_0 u_1 w_1 u_2 w_2 \cdots u_q w_q$ and $B^y(\ell, j) = w_0 v_1 w_1 v_2 w_2 \cdots v_q w_q$

²To avoid double and triple indexes we use our notation $B^x(\ell, i)$ and $A^x(\ell, i)$ instead of the usual $B_{\ell,i}^x$ and $A_{\ell,i}^x$.

for some $u_i, v_i, w_i \in \Gamma^*$, then we let $A^x(\ell, j) = w'_0 u'_1 w'_1 u'_2 \cdots w'_q$ and $A^y(\ell, j) = w'_0 v'_1 w'_1 v'_2 \cdots w'_q$ be their canonical decomposition induced by $B^x(\ell, j)$ and $B^y(\ell, j)$ as given by Lemma 2.17.

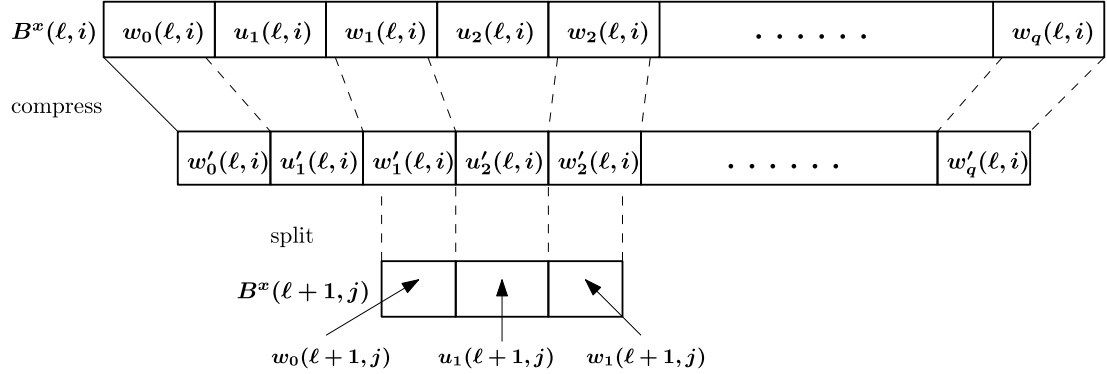


Figure 2.2 Decomposition of $B^x(\ell, i)$ after compression and split.

To conclude item 3 of Theorem 2.8 we want to argue that x and y are recursively split into sub-blocks that respect their canonical decomposition. So we want all splits of blocks to occur in matching parts of x and y . For $A^x(\ell - 1, i)$ with canonical decomposition $w_0 u_1 w_1 u_2 w_2 \cdots u_q w_q$ we say that $\text{Split}(A^x(\ell - 1, i), \ell)$ makes *undesirable split* if it starts a new block at a position j that either belongs to one of the u_1, u_2, \dots, u_q or is the first or last symbol of one of the w_0, w_1, \dots, w_q . Recall, $\text{Split}(A^x(\ell - 1, i), \ell)$ starts a new block at each position j such that $H_\ell(A^x(\ell - 1, i)[j, j + 1]) = 0$. Since H_ℓ is chosen at random a given position starts a new block with probability $1/D$.

Similarly, for $A^y(\ell - 1, i)$ with canonical decomposition $w'_0 v_1 w'_1 v_2 \cdots v_{q'} w'_{q'}$ we say that $\text{Split}(A^y(\ell - 1, i), \ell)$ makes *undesirable split* if it starts a new block at a position j that either belongs to one of the $v_1, v_2, \dots, v_{q'}$ or is the first or last symbol of one of the $w'_0, w'_1, \dots, w'_{q'}$. If $A^x(\ell - 1, i)$ and $A^y(\ell - 1, i)$ have *matching* canonical decomposition (that is $q = q'$ and each $w_j = w'_j$) and both $\text{Split}(A^x(\ell - 1, i), \ell)$ and $\text{Split}(A^y(\ell - 1, i), \ell)$ make no undesirable split then $A^x(\ell - 1, i)$ and $A^y(\ell - 1, i)$ are split in the same number of blocks with matching canonical decomposition as they are split at the same positions in the corresponding w_j 's.

For given $\ell \in \{0, \dots, L\}$, if no undesirable split happens during $\text{Split}(A^x(\ell' - 1, i), \ell')$ and $\text{Split}(A^y(\ell' - 1, i), \ell')$, for any $\ell' < \ell$ and i , then for each $\ell' < \ell$, the number of blocks $B^x(\ell', i)$ and $B^y(\ell', i)$ will be the same, i.e., $s_{\ell'}^x = s_{\ell'}^y$, and blocks $B^x(\ell', i)$ and $B^y(\ell', i)$ will have matching canonical decomposition. The total number of u_j 's in canonical decomposition of all $B^x(\ell', i)$, $i = 1, \dots, t_{\ell'}^x$, will be at most k , and similarly for v_j 's. Thus, there will be at most $(4R + 24 + 2)k + 2$ positions where an undesirable split can happen in $\text{Split}(A^x(\ell - 1, i), \ell)$ for any i . Similarly, there are at most $(4R + 26)k + 2$ positions where an undesirable split can happen in $\text{Split}(A^y(\ell - 1, i), \ell)$. By union bound, the probability that an undesirable split happens in some $\text{Split}(A^y(\ell - 1, i), \ell)$ or $\text{Split}(A^x(\ell - 1, i), \ell)$, for some ℓ and i , is at most $2(4R + 28)k(L + 1)/D \leq 11Rk(L + 1)/D \leq 1/10$.

Thus, if no undesirable split happens there are at most k indices i for which the canonical decomposition of $B^x(\ell, i)$ contains some u_j . All other blocks $B^x(\ell, i)$

have a canonical decomposition consisting of a single block w_0 , for various w_0 depending on ℓ and i . Similarly, the canonical decomposition of $B^y(\ell, i)$ contains v_j if and only if $B^x(\ell, i)$ contains u_j . Blocks $B^y(\ell, i)$ that do not contain v_j are identical to $B^x(\ell, i)$ so they have the same grammar.

Hence, if no undesirable split happens, item 3 of Theorem 2.8 will be satisfied.

The following theorem generalizes item 3 of Theorem 2.8 and it will be useful to construct the rolling sketch in Section 3.1.3.

Theorem 2.19. *Let $u, v, x, y \in \Sigma^*$ be strings such that $|ux|, |vy| \leq n$ and $\Delta_{edit}(x, y) \leq k$. Let G_1^x, \dots, G_s^x and $G_1^y, \dots, G_{s'}^y$ be the sequence of grammars output by the decomposition algorithm on input ux and vy respectively, using the same choice of random functions C_1, \dots, C_L and H_0, \dots, H_L . With probability at least $1 - 1/5$ the following is true: There exist integers r, r', t, t' such that $s - t = s' - t'$,*

$$\begin{aligned} x &= \text{eval}(G_t^x)[r, \dots] \cdot \text{eval}(G_{t+1}^x) \cdots \text{eval}(G_s^x) \quad \& \\ y &= \text{eval}(G_{t'}^y)[r', \dots] \cdot \text{eval}(G_{t'+1}^y) \cdots \text{eval}(G_{s'}^y), \quad \& \\ \Delta_{edit}(x, y) &= \Delta_{edit}(\text{eval}(G_t^x)[r, \dots], \text{eval}(G_{t'}^y)[r', \dots]) \\ &\quad + \sum_{i>0} \Delta_{edit}(\text{eval}(G_{t+i}^x), \text{eval}(G_{t'+i}^y)). \end{aligned}$$

Its proof is a minor modification of the proof above. We start with the canonical decomposition of $x = w_0 u_1 w_1 \cdots u_k w_k$ and $y = w_0 v_1 w_1 \cdots v_k w_k$, form the decomposition $ux = uu_0 w_0 u_1 w_1 \cdots u_k w_k$ and $vy = vv_0 w_0 v_1 w_1 \cdots v_k w_k$ where $u_0 = v_0 = \varepsilon$, and follow the compression and split procedures. We want to argue that during each split operation, all splits occur either in w_j 's and are the same on ux and vy , or they occur in u or v where we do not care for them. Again we define a split to be *undesirable* if it starts a new block at a position j that belongs to one of the $u_0, u_1, \dots, u_k, v_0, v_1, \dots, v_k$ or it is the position of the first or last symbol of w_0, w_1, \dots or w_k . Inductively we maintain that whenever a block $B^{ux}(\ell, i)$ contains a descendant of the compression of u_j , its corresponding block $B^{vy}(\ell, i')$ contains a descendant of the compression of v_j . (Here, the correspondence is counting from the highest index i to the lowest and similarly for i' , so $B^{ux}(\ell, i)$ corresponds to $B^{vy}(\ell, i')$ if $i - i' = s_\ell^{ux} - s_\ell^{vy}$.) If the blocks contain a descendant of u_0 and v_0 , resp., then we apply Lemma 2.18 to construct a descendant decomposition after their compression. For all other blocks that contain some w_j, u_j or v_j we use Lemma 2.17 to construct its descendant decomposition. We do not care for decomposition of blocks $B^{ux}(\ell, i)$ that are descendants of u but do not contain u_0 , and similarly we do not care for decomposition of blocks $B^{vy}(\ell, i)$ that are descendants of v but do not contain v_0 . (They might be decomposed arbitrarily so the number of blocks that are descendants of u might differ from the number of blocks that are descendants of v .) Inductively, there are at most $2(4R + 28)(k + 1)$ positions where an undesirable split can happen in blocks $B^{ux}(\ell, i)$ and $B^{vy}(\ell, i)$ for given level ℓ . In total there are at most $2(4R + 28)(k + 1)(L + 1)$ positions where an undesirable split can happen. Thus, the probability of making an undesirable split during a run of the algorithm is bounded by $2(4R + 28)(k + 1)(L + 1)/D \leq 22Rk(L + 1)/D \leq 1/5$. If no undesirable split ever happens then the symbols that are weak compression of symbols from x and y are contained within the corresponding blocks $B^{ux}(\ell, i)$ and

$B^{yv}(\ell, i')$. For the blocks $B^{ux}(\ell, i)$ and $B^{vy}(\ell, i')$ that contain descendants of u_0 and v_0 it is fine if their prefixes that descend from u and v , resp., which are to the left of the descendants of u_0 and v_0 , are split differently in $B^{ux}(\ell, i)$ and $B^{vy}(\ell, i')$. This does not affect the correspondence between blocks $B^{ux}(\ell, i)$ and $B^{vy}(\ell, i')$ that weakly come from x and y . This concludes the proof of Theorem 2.19.

2.4 Table of parameters

Definition	Asymptotics	Meaning	Reference
$R = \log^* \Gamma + 20$	$\log^* n$	compression locality	Section 2.3.1
$L = \lceil \log_{3/2} n \rceil + 3$	$\log n$	recursion depth	Section 3.2.2, Corollary 2.10
$D = 110c - R(L + 1)k$	$k \log n \log^* n$	1/splitting probability	Section 3.2.2, Lemma 2.11
$S = 30DL \log n + 6$	$k \log^3 n \log^* n$	maximum grammar size	Section 3.2.2, Theorem 2.8
$M = 3S \cdot \lceil 1 + \log \Gamma \rceil$	$k \log^4 n \log^* n$	grammar encoding size	Section 3.2.3
$N \geq n^3$	n^3	F_{KR} range size	Section 3.2.3

Table 2.1 Table of parameters

2.5 Summary

This chapter investigated the problem of embedding strings from the edit distance metric into the Hamming metric with bounded distortion. A naive approach such as the identity mapping can lead to distortion as large as $\Theta(n)$, highlighting the necessity of more refined techniques.

We began by introducing the CGK embedding [11], a randomized construction that maps strings of length n into strings of length $3n$ over the Hamming space. The embedding ensures that the Hamming distance between the images of two strings with edit distance k is between $\Omega(k)$ and $\mathcal{O}(k^2)$, with high probability. This construction is notable for its simplicity, efficient one-pass implementation, and its interpretability through a random walk model that controls the divergence and resynchronization of two strings.

We then presented a new embedding scheme that achieves similar distortion guarantees while offering additional structural benefits. The central idea is a randomized decomposition of a string into a sequence of blocks, each of which admits a succinct description by a context-free grammar of size $\tilde{\mathcal{O}}(k)$. For two strings differing by at most k edits, this decomposition aligns their structure such that only $\mathcal{O}(k)$ blocks differ, and the edit distance is preserved as the sum of the distances across corresponding blocks.

A distinguishing feature of our approach is its *locality*: appending a symbol to the end of a string or removing one from the beginning affects only a bounded number of blocks, and the embedding can be updated incrementally without recomputing from scratch. This makes our construction particularly suited for streaming and dynamic environments. In addition, our decomposition procedure admits efficient parallelization, in contrast to the inherently sequential nature of the CGK embedding.

To represent the blocks in Hamming space, we defined a simple encoding based on Karp-Rabin hashing, ensuring that any two distinct grammars differ

in all positions of their encodings with high probability. This property yields an embedding into the Hamming space that satisfies

$$\Delta_{edit}(x, y) \leq \Delta_{Ham}(f(x), f(y)) \leq \tilde{O}(k^2)$$

with high probability, where f is our embedding function.

The decomposition algorithm runs in near-linear time $\tilde{O}(nk)$ and ensures grammars of small size with high probability. We also established that our embedding admits extensions to sketching and rolling embeddings, which are developed in the next chapter. These results suggest that our decomposition framework may have applications beyond the embedding problem itself.

3

Applications of the Edit to Hamming Embedding

“Understanding is the first step to acceptance, and only with acceptance can there be recovery.”
- Dumbledore

In this chapter we will present two applications of our embedding from previous chapter. These applications are specific for the type of decomposable embedding that we presented in the previous chapter.

- The results presented in this chapter are part of the work published in [10] and the entire paper [24].

3.1 Edit distance sketch

A problem for edit distance that saw a major progress in recent years is *sketching*. In sketching we want to map a string x to a short sketch $\text{sk}_{n,k}^{\text{ED}}(x)$ so that from sketches $\text{sk}_{n,k}^{\text{ED}}(x)$ and $\text{sk}_{n,k}^{\text{ED}}(y)$ of two strings x and y we can compute their edit distance, either exactly or approximately. Apriori it is not even obvious that short sketches for edit distance exist. In a surprising construction, Belazzougui and Zhang [25] gave an exact edit distance sketch of size $O(k^8 \log^5 n)$ bits. The sketch size was then improved to $O(k^3 \log^2(\frac{n}{\delta}) \log n)$ bits by Jin, Nelson and Wu [26], where the $\Delta_{\text{edit}}(x, y)$ was computed exactly from the sketches with probability at least $1 - \delta$, if $\Delta_{\text{edit}}(x, y) \leq k$. The current best sketch is of size $O(k^2 \log^3 n)$ bits and was given by Kociumaka, Porat and Starikovskaya [27]. [26] gives a lower bound $\Omega(k)$ on the size of a sketch for exact edit distance.

The major problem in edit distance computation as well as in sketching is how to align the matching parts of two strings x and y . Finding an optimal alignment of two strings is the crux in the computation of edit distance and its sketching. In sketching finding a good alignment is even more challenging as we do not have both strings in our hands simultaneously to look for the matching. To the best of our knowledge, to resolve this issue all edit distance sketches use *CGK random walk* on strings [11] which allows to embed the edit distance metrics into Hamming distance metrics with distortion $O(k)$. The walk implicitly fixes some reasonably good matching between the two strings. Going from the CGK random walk to a sketch is non-trivial undertaking and all three sketch results rely on sophisticated machinery to achieve it.

In this section we show how to use our decomposition technique from the previous chapter, to align two strings x and y in oblivious manner, along with already available good Hamming distance sketches to give good edit distance sketches.

The decomposition technique readily yields a sketch for exact edit distance of size $\mathcal{O}(k^2)$:

Theorem 3.1 (Sketch for edit distance). *There is a randomized sketching algorithm $\text{sk}_{n,k}^{\text{ED}}$ that on an input string x of length at most n produces a sketch $\text{sk}_{n,k}^{\text{ED}}(x)$ of size $\mathcal{O}(k^2)$ in time $\mathcal{O}(nk)$, and a comparison algorithm running in time $\mathcal{O}(k^2)$ such that given two sketches $\text{sk}_{n,k}^{\text{ED}}(x)$ and $\text{sk}_{n,k}^{\text{ED}}(y)$ for two strings x and y of length at most n obtained using the same randomness of the sketching algorithm outputs with probability at least $1 - 1/n$ (over the randomness of the sketching and comparison algorithms) the edit distance of x and y if it is less than k and ∞ otherwise.*

Furthermore, we can also provide a *rolling sketch*, a sketch in which we can update the stored string by appending a symbol or removing its first symbol.

Theorem 3.2 (Rolling sketch for edit distance). *There are algorithms $\text{Append}(sk_x, a)$, $\text{Remove}(sk_{ax}, a)$, and $\text{Compare}(sk_x, sk_y)$ such that for integer parameters $k \leq m$:*

1. *Given a sketch sk_x representing a string x and a symbol a , $\text{Append}(sk_x, a)$ outputs a sketch sk_{xa} for the string xa in time $\mathcal{O}(k^2)$.*
2. *Given a sketch sk_{ax} representing a string ax for a symbol a , $\text{Remove}(sk_{ax}, a)$ outputs a sketch sk_x for the string x in time $\mathcal{O}(k^2)$.*
3. *Given two sketches sk_x and sk_y representing strings x and y obtained from the same random sketch for empty string using two sequences of at most m operations Append and Remove , $\text{Compare}(sk_x, sk_y)$ calculates the edit distance of x and y if it is less than k , and outputs ∞ otherwise. The algorithm $\text{Compare}(sk_x, sk_y)$ runs in time $\mathcal{O}(k^2)$.*

All the sketches are of size $\mathcal{O}(k^2)$. The probability that any of the algorithms fails or produces incorrect output is at most $1/m$ over the initial randomness of the sketch for empty string and internal randomness of the algorithms.

We remark that we did not attempt to optimize the running time of either of our algorithms, or poly-log factors in the sketch sizes, and we believe that both parameters can be readily improved by usual amortization techniques of processing symbols in batches of size $\mathcal{O}(k)$. We believe that the update time in the last theorem can be improved to $\mathcal{O}(1)$ by buffering $\mathcal{O}(k)$ symbols that shall be inserted or removed without affecting the other parameters of the algorithm.

To design our edit distance sketches, we use existing optimal Hamming distance sketches together with our decomposition.

3.1.1 Hamming distance sketch

For two strings x and y of the same length, we define their *mismatch information* $\text{MIS}(x, y) = \{(i, x[i], y[i]); i \in \{1, \dots, |x|\} \text{ and } x[i] \neq y[i]\}$. The Hamming distance of x and y is $\Delta_{\text{Ham}}(x, y) = |\text{MIS}(x, y)|$.

There exist various sketches for Hamming distance, which allow to compute Hamming distance with low error probability [28, 29]. Moreover, [30, 23] also allow to retrieve the mismatch information. For our purposes we will use the sketch given by Clifford, Kociumaka, and Porat [23].

Let $k \leq n$ be integers and $p \geq n^3$ be a prime. [23] give a randomized sketch for Hamming distance $\text{sk}_{n,k,p}^{\text{Ham}} : \{1, \dots, p-1\}^* \rightarrow \{0, \dots, p-1\}^{k+4}$ computable in time $\mathcal{O}(n)$ with the following properties.¹

Proposition 3.3 ([23]). *There is a randomized algorithm working in time $\mathcal{O}(k \log^3 p)$ that given sketches $\text{sk}_{n,k,p}^{\text{Ham}}(x)$ and $\text{sk}_{n,k,p}^{\text{Ham}}(y)$ of two strings x and y of length $\ell \leq n$ constructed using the same randomness decides whether $\Delta_{\text{Ham}}(x, y) \leq k$, and if so returns $\text{MIS}(x, y)$, with probability of error at most $1/n$ over the randomness of the sketches and the internal randomness of the algorithm.*

They also construct the following update procedures for their sketch. We will use them to construct a rolling sketch for edit distance.

Proposition 3.4 (Lemma 2.3 of [23]). *For $x \in \{1, \dots, p\}^*$ of length less than n and $a \in \{1, \dots, p\}$, in time $\mathcal{O}(k \log p)$ we can compute:*

1. $\text{sk}_{n,k,p}^{\text{Ham}}(xa)$ and $\text{sk}_{n,k,p}^{\text{Ham}}(ax)$, given $\text{sk}_{n,k,p}^{\text{Ham}}(x)$ and a .
2. $\text{sk}_{n,k,p}^{\text{Ham}}(x)$ given $\text{sk}_{n,k,p}^{\text{Ham}}(xa)$ or $\text{sk}_{n,k,p}^{\text{Ham}}(ax)$, and a .

Corollary 2.5 of [23] states that appending a character to a sketch of x can be done even faster namely in amortized time $\mathcal{O}(\log p)$.

3.1.2 Edit distance sketch using locally consistent decomposition

Let n and $k \leq n$ be two parameters, and $p \geq 2N + 1$ be a prime such that $p \geq (nM)^3$. For a string $x \in \Sigma^*$ of length at most n , we compute its sketch by running first the decomposition algorithm of Theorem 2.8 to get grammars G_1, G_2, \dots, G_s . Encode each grammar G_i by encoding $\text{Enc}(G_i)$ from Section 3.2.3 using the same F_{KR} picked at random. Concatenate the encoding to get a string $w = \text{Enc}(G_1) \cdot \text{Enc}(G_2) \cdots \text{Enc}(G_s)$. Calculate the Hamming sketch $\text{sk}_{n',m',p}^{\text{Ham}}(w)$ on w for strings of length $n' = nM$ and Hamming distance at most $k' = kM$ from Section 3.1.1. Set the sketch $\text{sk}_{n,k}^{\text{ED}}(x) = \text{sk}_{n',k',p}^{\text{Ham}}(w)$. The calculation of $\text{sk}_{n,k}^{\text{ED}}(x)$ can be done in time $\tilde{\mathcal{O}}(nk)$ as the number of grammars is at most n and each grammar requires $\tilde{\mathcal{O}}(k)$ time to be encoded into binary. The Hamming sketch can be constructed in time $\tilde{\mathcal{O}}(nk)$. (We believe that on average we expect only $\tilde{\mathcal{O}}(n/k)$ grammars to be produced for a given string x so the actual running time should be $\tilde{\mathcal{O}}(n)$ on average.)

¹Clifford, Kociumaka and Porat have the sketch size only $k + 3$ elements but we include as an extra item the randomness of the sketch, which is a single element from $\{0, \dots, p-1\}$ used to compute Karp-Rabin fingerprint.

Theorem 3.5. *Let $x, y \in \Sigma^*$ be strings of length at most n such that $\Delta_{edit}(x, y) \leq k$. Let $\text{sk}_{n,k}^{\text{ED}}(x)$ and $\text{sk}_{n,k}^{\text{ED}}(y)$ be obtained using the same randomness for the decomposition algorithm and the same choice of F_{KR} . With probability at least $2/3$, we can calculate $\Delta_{edit}(x, y)$ from $\text{sk}_{n,k}^{\text{ED}}(x)$ and $\text{sk}_{n,k}^{\text{ED}}(y)$.*

Assume that the output of the decomposition algorithm on x and y satisfies all the conclusions of Theorem 2.19. In particular, for x we get $\text{eval}(G_1^x) \cdot \text{eval}(G_2^x) \cdots \text{eval}(G_s^x)$ and for y we get $\text{eval}(G_1^y) \cdots \text{eval}(G_s^y)$, for some $s \leq n$, each of the grammars is of size at most S , $\Delta_{edit}(x, y) = \sum_i \Delta_{edit}(\text{eval}(G_i^x), \text{eval}(G_i^y))$, and the number of pairs G_i^x and G_i^y where $G_i^x \neq G_i^y$ is at most k . Assume that F_{KR} is chosen so that $\text{Enc}(G_i^x) \neq \text{Enc}(G_i^y)$ for each of the pairs where G_i^x and G_i^y differ.

In order to determine $\Delta_{edit}(x, y)$, we recover the (Hamming) mismatch information between $\text{Enc}(G_1^x) \cdot \text{Enc}(G_2^x) \cdots \text{Enc}(G_s^x)$ and $\text{Enc}(G_1^y) \cdot \text{Enc}(G_2^y) \cdots \text{Enc}(G_s^y)$ from $\text{sk}_{n,k}^{\text{ED}}(x)$ and $\text{sk}_{n,k}^{\text{ED}}(y)$. That gives grammars G_i^x and G_i^y , for all i where $G_i^x \neq G_i^y$. (Whenever the two grammars differ, their encoding differ in every symbol by Lemma 3.15 so we can recover them from the Hamming mismatch information.) Calculating the edit distance of each of the pair of differing grammars using the algorithm from Proposition 2.6 we recover $\Delta_{edit}(x, y)$ as the sum of their edit distances.

The sum is correct unless some of the assumptions fail: The probability that the grammar decomposition fails (does not have properties from Theorem 2.8) for the pair x and y is at most $1/5$ for n large enough. The probability that the choice of F_{KR} fails (two distinct grammars have the same encoding) is at most $2kM/N < 1/n$ by the choice of N . The probability that the Hamming distance sketch fails to recover the mismatch information between all the grammars is at most $1/n$. So in total, the probability that the output of the algorithm is incorrect is at most $1/3$.

The running time of the comparison algorithm is $\tilde{O}(k^2)$: The Hamming mismatch information can be recovered in time $\tilde{O}(kM) = \tilde{O}(k^2)$ (Proposition 3.3), then we build the $\leq k$ mismatched grammars in time $\tilde{O}(k^2)$, and run the edit distance computation on the pairs of grammars in time $\sum_{i < k} \tilde{O}(k + k_i^2) \leq \tilde{O}(k^2)$, where k_i is the edit distance of the i -th pair of mismatched grammars. (We interrupt the edit distance computation if it takes more time than $\tilde{O}(k^2)$ which would indicate $\Delta_{edit}(x, y) > k$.)

To decide whether $\Delta_{edit}(x, y) > k$ we note that on input x and y , the Hamming sketch either outputs the correct mismatched places if their number is $\leq k'$ or it outputs ∞ if there are more mismatches than that or the sequences sketched by the Hamming sketch are of different length. (We assume that the Hamming sketch knows the number of symbols it is sketching.) In the ∞ -case we know that there are more than k different pairs of grammars or the decomposition of x and y failed, and we can report $\Delta_{edit}(x, y) > k$. In the other case we try to calculate the edit distance of the differing pairs of grammars. If we spend more than $\tilde{O}(k^2)$ time on it or we get a number larger than k then we report $\Delta_{edit}(x, y) > k$. This correctly decides whether $\Delta_{edit}(x, y) > k$ with probability at least $2/3$.

To prove Theorem 3.1 we build a more robust sketch by taking $c \log n$ independent copies of the sketch $\text{sk}_{n,k}^{\text{ED}}$. To calculate the edit distance of two sketched strings we run the edit distance calculation on each of the corresponding pairs of copies, and output the majority answer. A usual application of Chernoff bound

shows that the probability of correct answer is at least $1 - 1/n$ for suitable constant $c > 0$.

3.1.3 Rolling sketch for edit distance

In this section we will construct the rolling sketch of Theorem 3.2. We will use two claims that will be proved in Section 3.1.3. The first one addresses how much a compression of a string w might change depending on what is appended to it.

Lemma 3.6. *Let $\ell \in \{0, \dots, L\}$ and $v, u, w \in \Gamma^*$. Let $w'u' = \text{Compress}(wu, \ell)$ and let $w''v' = \text{Compress}(wv, \ell)$, where w' is the compression of w when compressing wu and w'' is the compression of w when compressing wv . Let $t = |w'| - 3(R + 1)$ or $t = |w'u'| - |u| - 3(R + 1)$. Then $w'[1, t] = w''[1, t]$.*

The next lemma addresses how much the overall decomposition of a string x might change if we append a suffix z to it.

Lemma 3.7. *Let $x, z \in \Sigma^*$, $|xz| \leq n$. Let $H_0, \dots, H_L, C_1, \dots, C_L$ be given. Let $G_1^x, G_2^x, \dots, G_s^x$ be the output of the decomposition algorithm on input x , and $G_1^{xz}, G_2^{xz}, \dots, G_{s'}^{xz}$ be the output of the decomposition algorithm on input xz using the given hash functions. Let $T = L(3R + 6)$.*

1. $G_i^x = G_i^{xz}$ for all $i = 1 \dots, s - T$.
2. $|x| \leq \sum_{i=1}^{\min(s+T, s')} |\text{eval}(G_i^{xz})|$.

The second part says that if x is decomposed into s grammars by itself, then it can be recovered from the first $s + T$ grammars for xz . Hence, appending extra symbols to x cannot increase the number of grammars that cover x by more than T .

Let $m \geq k$ and $n \geq 10m^3$ be integers. A rolling sketch for a string obtained by up-to m insertions (to the right end) and m deletions (from the left end) from an empty word consists of three data structures: *insertion buffer*, *deletion buffer* and a Hamming distance sketch $\text{sk}_{n', k', p}^{\text{Ham}}$, where $k' = (4T + 1)(k + 2)M$, $n' = nM$ and $p \geq n'^3$ is a chosen prime.

The insertion buffer maintains a buffer of *committed grammars* $G_{s-4T+1}, G_{s-4T+2}, \dots, G_s$ and a buffer of *active grammars* G_1^i, \dots, G_t^i , $t \leq T$. The deletion buffer is similar, it maintains a buffer of *committed grammars* $G_{r-4T+1}, G_{r-4T+2}, \dots, G_r$ and a buffer of *active grammars* $G_1^d, \dots, G_{t'}^d$, $t' \leq T$. The Hamming sketch is a sketch of grammars $G_{r-2T+1}, G_{r-2T+2}, \dots, G_{s-2T}$, each encoded as a string of length M over the alphabet $\{1, \dots, 2N\}$.

In addition to that, the sketch keeps track of the current value of r and s , and remembers a collection of pair-wise independent hash functions C_1, \dots, C_L , a collection of $(5D \log n)$ -wise independent hash functions H_0, \dots, H_L , and randomness for Karp-Rabin fingerprint to compute binary encoding of grammars. The hash functions and the randomness of Karp-Rabin fingerprint are chosen at random when creating the sketch for empty string. This extra information requires $\tilde{O}(k)$ bits to specify.

Initially, the committed grammars in the insertion and deletion buffers are all treated as empty sets, there are no active grammars in the insertion or deletion buffers so $t = t' = 0$ and $s = r = 0$.

For $u, x \in \Sigma^*$, if in total a string ux was inserted into the sketch then $G_1, \dots, G_s, G_1^i, \dots, G_t^i$ represents ux , that is ux is the concatenation of the evaluation of the grammars. If in total the string u was deleted from the sketch, then $G_1, \dots, G_r, G_1^d, \dots, G_{t^d}^d$ represents u . (See Fig. 3.1 for an illustration.)

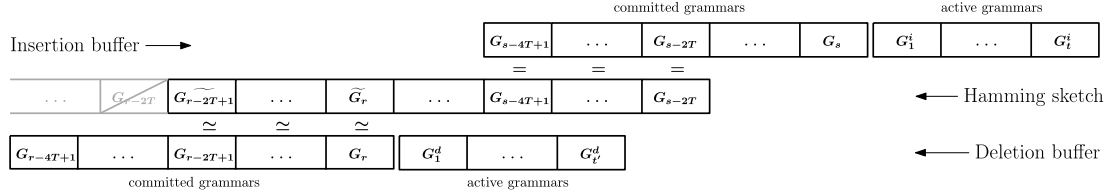


Figure 3.1 Rolling sketch.

Appending a symbol. When we append additional symbol a to the sketch we modify input buffers as follows: We update the active grammars G_1^i, \dots, G_t^i by appending a as explained further below. Say the update produces grammars $G_1^{i'}, \dots, G_{t'}^{i'}$. If $t' \leq T$ then the produced grammars will become the active grammars, and no more changes are done to the sketch. Otherwise we commit the first $t' - T$ grammars $G_1^{i'}, \dots, G_{t'-T}^{i'}$ one-by-one into the committed buffer as grammars $G_{s+1}, \dots, G_{s+t'-T}$ and we keep the remaining grammars as the active grammars.

Committing a grammar G_{s+1} into the committed buffer will trigger addition of G_{s-2T+1} into the Hamming sketch at the end of the represented sequence of grammars (if $s - 2T + 1 > 0$), and removing the grammar G_{s-4T+1} from the committed buffer. For insertion into the Hamming sketch, the grammar G_{s-2T+1} is encoded into binary as in Section 3.2.3 and then the binary string is encoded using the Karp-Rabin fingerprint F_{KR} of *all* the grammars $G_{s-4T+1}, \dots, G_{s+1}$, instead of only the grammar G_{s-2T+1} . (Thus, a change in any of the neighboring grammars will trigger a recovery of also the grammar G_{s-2T+1} when calculating a mismatch information from the Hamming sketch.) We repeat this process for each grammar being committed.

By the second part of Lemma 3.7 $t' \leq t + T \leq 2T$ so we will commit at most $T = \tilde{O}(1)$ grammars. It takes time $\mathcal{O}(MT) = \tilde{O}(k)$ to prepare the binary encoding of each of the committed grammars, and $\tilde{O}(k^2)$ to insert it into the Hamming sketch. The update of the active grammars takes $\tilde{O}(k)$ time as described below. So in total this step takes $\tilde{O}(k^2)$ time.

Removing a symbol. Deletion buffer works in manner similar to insertion buffer, we add the removed symbol a to the active grammars, but when committing the grammar G_{r+1} , we use F_{KR} -fingerprint of all the grammars $G_{r-4T+1}, \dots, G_{r+1}$ to encode grammar G_{r-2T+1} which is then *removed* from the beginning of the sequence of grammars represented by the Hamming sketch (if $r - 2T + 1 > 0$), i.e., we update the Hamming sketch to reflect this removal. Similarly to appending a symbol, this step takes time $\tilde{O}(k^2)$.

Active grammar update. The update of active grammars G_1^i, \dots, G_t^i when appending a is done as follows. $G_1, \dots, G_s, G_1^i, \dots, G_t^i$ represents ux so we need to calculate the grammars for uxa . We claim that only the active grammars might change: At some point, G_s became committed so at that time there was T active grammars following it. If at that point the grammars together represented a string

z , by appending more symbols to z we cannot change grammars G_1, G_1, \dots, G_s according to the first part of Lemma 3.7. So appending a to ux will affect only the active grammars.

From the analysis in the proof of Lemma 3.7 it follows that for $\ell \in \{0, \dots, 1\}$ if $B^{ux}(\ell, 1), \dots, B^{ux}(\ell, s_\ell^{xy})$ is the trace of the decomposition algorithm on ux at level ℓ , and $B^{uxa}(\ell, 1), \dots, B^{uxa}(\ell, s_\ell^{xya})$ is the trace on uxa , then their difference spans at most $\ell(3R + 6)$ last symbols of $B^{ux}(\ell, 1) \dots B^{ux}(\ell, s_\ell^{xy})$.

So instead of decompressing the active grammars completely, adding a and recompressing them back, we only decompress the necessary part of each trace $B^{ux}(\ell, 1) \dots B^{ux}(\ell, s_\ell^{xy})$. Let $\# \rightarrow v_i$ be the starting rule of the active grammar G_i . Starting from the string $v_1 \cdot v_2 \dots v_t$, for each $\ell = L, \dots, 1$, we iteratively rewrite all level- ℓ symbols in the string using the appropriate grammars while only maintaining at most T last symbols of the resulting string. (Care has to be taken to maintain information about any sequence a^r stretching from those T last symbols to the left.)

We add a to the resulting string and re-apply compress and split procedures for levels $0, 1, \dots, \ell - 1$ to recompress only the part of the trace affected by modifications. As we perform the compression of symbols we maintain a set G of all grammar rules needed for decompression. (We initialize G with the union of all rules from the active grammars G_1^i, \dots, G_t^i minus the starting rules, and we iteratively add new rules coming from the recompression.) For the recompression we need to know the context of up-to $R + 1$ symbols preceding the modified part of the trace. On the other hand, the modification can affect the recompression of up-to $R + 1$ symbols to the left from the left-most modified symbol in the trace. Those $R + 1$ symbols all happen to be within the decompressed suffix of the trace of size at most T .

Eventually, we get a new level- L trace $B^{uxa}(L, s_L^{xya} - t' + 1), \dots, B^{uxa}(L, s_L^{xya})$, for some t' . Each new grammar G_j^i is obtained by taking the grammar $G \cup \{\# \rightarrow B^{uxa}(L, s_L^{xya} - t' + j)\}$ and removing from it all useless rules. This can be done in time $\mathcal{O}(|G|)$. (See Section 2.3.1).

Overall the update of active grammars on insertion of a single symbol will require $\mathcal{O}(LT) = \tilde{\mathcal{O}}(1)$ evaluations of split hash functions H_0, \dots, H_L , $\mathcal{O}(LT) = \tilde{\mathcal{O}}(1)$ evaluations of compress hash functions C_1, \dots, C_L , and $\mathcal{O}(T(LT + \sum_{j=1}^t |G_j^i|))$ time to produce the new grammars. As the total size of the grammars is $\tilde{\mathcal{O}}(k)$ and the time to evaluate H_ℓ at a single point is also $\tilde{\mathcal{O}}(k)$, the overall time for the update of active grammars is $\tilde{\mathcal{O}}(k)$. We provide a more detailed description of the update procedure in Section 3.1.3.

Edit distance evaluation. Consider strings x and y of length at most m and edit distance at most k . Consider the rolling sketch $\text{sk}_{m,k}^{\text{Rolling}}(x)$ for x obtained by inserting symbols ux and removing symbols u , for some $u \in \Sigma^*$ where $|ux| \leq m$. Consider also the rolling sketch for y obtained by inserting symbols vy and removing symbols v , for some $v \in \Sigma^*$ where $|vy| \leq m$. Both sketches should use the same randomness that is to start from the same sketch for empty string.

The rolling sketch for x consists of the insertion buffer with committed grammars $G_{s^x-4T+1}^x, G_{s^x-4T+2}^x, \dots, G_{s^x}^x$ and with active grammars $G_1^{ix}, \dots, G_{t^x}^{ix}$, and the deletion buffer with committed grammars $G_{r^x-4T+1}^x, G_{r^x-4T+2}^x, \dots, G_{r^x}^x$ and active grammars $G_1^{dx}, \dots, G_{t'^x}^{dx}$, $t'^x \leq T$. Its Hamming sketch sketches the sequence of grammars $G_{r^x-2T+1}^x, G_{r^x-2T+2}^x, \dots, G_{s^x-2T}^x$. Similarly for y , we have the

committed insertion grammars $G_{s^y-4T+1}^y, G_{s^y-4T+2}^y, \dots, G_{s^y}^y$, etc.

We extend the notation so for $j \in \{1, \dots, t^x\}$, we let $G_{s^x+j}^x$ denote the active grammar G_j^{ix} , and similarly for y . Let $d^x = s^x + t^x - r^x$ and $d^y = s^y + t^y - r^y$. We assume that the hash functions used to decompose ux and vy into grammars satisfy the probabilistic conclusion of Theorem 2.19. That means that grammars G_r^x, \dots and G_r^y, \dots can be aligned from the right so G_j^x corresponds to $G_{j-d^x+d^y}^y$, for $j \geq r^x$ (they might not be identical because of the edit operations). Without loss of generality we assume that $d^x \geq d^y$.

Before proceeding with the algorithm we first observe that $d^x - d^y < 2T$. Let $p^x \geq r^x + 1$ be the index of the grammar $G_{p^x}^x$ which produces the first symbol of x when we evaluate all the grammars. Similarly, $p^y \geq r^y + 1$ is the index of $G_{p^y}^y$ which produces the first symbol of y . By Lemma 3.7 applied on $x \leftarrow u$ and $z \leftarrow x$ we get that $p^x \leq r^x + t^x + T \leq r^x + 2T$, and similarly $p^y \leq r^y + 2T$. By our assumption on success of Theorem 2.19, $s^x + t^x - p^x = s^y - t^y - p^y$. Hence, $s^x + t^x - s^y - t^y = p^x - p^y \leq r^x + 2T - r^y - 1 \leq r^x - r^y + (2T - 1)$. Thus $d^x - d^y = s^x + t^x - r^x - s^y - t^y + r^y \leq r^x - r^y + (2T - 1) - r^x + r^y \leq 2T - 1$.

If $d^x < 10T$ then we can recover all the grammars $G_{r^x-2T+1}^x, G_{r^x-2T+2}^x, \dots, G_{s^x-2T}^x$ from their Hamming sketch by constructing an auxiliary *dummy* Hamming sketch sk' for a sequence of 1's of length $(s^x - r^x)M$ and comparing the two sketches. (M is the length of the encoding of each grammar.) Their mismatch information reveals all the grammars $G_{r^x-2T+1}^x, \dots, G_{s^x-2T}^x$. Since $d^y \leq d^x$, we can similarly recover all the grammars $G_{r^y-2T+1}^y, \dots, G_{s^y-2T}^y$ from their Hamming sketch.

Thus we know all grammars $G_{r^x+1}^x, G_{r^x+2}^x, \dots, G_{s^x+t^x}^x$ and $G_{r^y+1}^y, G_{r^y+2}^y, \dots, G_{s^y+t^y}^y$. We also know grammars $G_1^{dx}, \dots, G_{t^x}^{dx}$ and $G_1^{dy}, \dots, G_{t^y}^{dy}$ that need to be *subtracted* from our grammars. As noted in Section 2.3.1, for each of the grammars we can calculate its evaluation size. From that information we can easily identify p^x and p^y , and shorten the grammars $G_{p^x}^x$ and $G_{p^y}^y$ to produce only symbols of x and y , respectively. We can combine all the grammars of x into one grammar G^x , and all the grammars of y into G^y , and run the algorithm of Ganesh, Kociumaka, Lincoln and Saha [22] to calculate the edit distance of x and y . Since $T = \tilde{O}(1)$, that will take time $\tilde{O}(|G^x| + |G^y| + k^2) = \tilde{O}(k^2)$.

If $d^x \geq 10T$ then we proceed as follows. Clearly, $d^y \geq 8T$, so $s^y - r^y \geq 7T$ and $s^x - r^x \geq 9T$. Thus $G_{r^x-2T+1}^x, G_{r^x-2T+2}^x, \dots, G_{s^x-2T}^x$ and $G_{r^y-2T+1}^y, G_{r^y-2T+2}^y, \dots, G_{s^y-2T}^y$ consist of at least $7T$ grammars each, and those grammars are sketched by their Hamming sketches. Although we assume that there is a correspondence between the grammar G_j^x , for $j \geq r^x$, and $G_{j-d^x+d^y}^y$ the sequences $G_{r^x-2T+1}^x, \dots, G_{s^x-2T}^x$ and $G_{r^y-2T+1}^y, \dots, G_{s^y-2T}^y$ are misaligned in their Hamming sketches by $d^x - d^y$ grammars. To rectify this misalignment, we prepend $(d^x - d^y)M$ copies of symbol 1 into the sketch for $G_{r^y-2T+1}^y, \dots, G_{s^y-2T}^y$. Furthermore, if $t^x < t^y$ then we append $(t^y - t^x)M$ ones into the sketch for $G_{r^y-2T+1}^y, \dots, G_{s^y-2T}^y$, to rectify the difference in the number of sketched grammars. Otherwise if $t^x > t^y$ then we append $(t^x - t^y)M$ ones into the sketch for $G_{r^x-2T+1}^x, \dots, G_{s^x-2T}^x$.

Now we can calculate the mismatch information from the Hamming sketches to find out the pairs of grammars G_j^x and $G_{j-d^x+d^y}^y$, $j \geq r^x + 1$, that are different.

If for some $j \in \{r^x + 1, \dots, r^x + 2T\}$, G_j^x and $G_{j-d^x+d^y}^y$ differ then because we use the Karp-Rabin fingerprint of the two grammars to encode also the neighboring grammars up-to distance $2T$, we recover from the sketch all the grammars G_j^x and

$G_{j-d^x+d^y}^y$, for $j = r^x + 1, \dots, r^x + 2T$. By counting the evaluation size of each of those grammars and comparing it with the evaluation size of active grammars in deletion buffers of x and y , resp., we identify p^x and p^y , and how much the grammars $G_{p^x}^x$ and $G_{p^y}^y$ should be shortened to produce only symbols of x and y . After shortening $G_{p^x}^x$ and $G_{p^y}^y$ we calculate the edit distance of their evaluation. We sum it up with the edit distance of evaluation of each pair of grammars G_j^x and $G_{j-d^x+d^y}^y$, for $j > p^x$, that was identified as mismatch by the Hamming distance sketch or that belongs among the active grammars in insertion buffers of either x or y . There will be at most T mismatched pairs involving the active grammars, and $(4T + 1)k$ pairs identified by the Hamming sketch.

In the remaining case when G_j^x and $G_{j-d^x+d^y}^y$ are identical for all $j \in \{r^x + 1, \dots, r^x + 2T\}$, we might not be able to recover all those grammars from the Hamming sketches, and we might not be able to identify p^x and p^y . However, since $G_{p^x}^x = G_{p^y}^y$, we know that the part of x produced by $G_{p^x}^x$ is either a prefix or suffix of the part of y produced by $G_{p^y}^y$. The difference in the size of the two parts is the edit distance of the two parts. The difference is given by the difference between the total evaluation size of active grammars in the deletion buffer of x , and the total evaluation size of active grammars in the deletion buffer of y together with grammars $G_{r^y-j}^y$, for $j = 0, \dots, d^x - d^y - 1$. The latter grammars are in the committed deletion buffer of y and they agree with $G_{r^x+1}^x, \dots, G_{r^x+d^x-d^y}^x$. Hence, the edit distance of the parts of x and y coming from $G_{p^x}^x$ and $G_{p^y}^y$ can be determined. All other mismatching pairs of grammars are identified by the Hamming sketch or are among active grammars of the insertion buffers. So we proceed as in the previous case to calculate their contribution to the edit distance of x and y . The edit distance of x and y is the sum of those edit distances.

We see that in both the cases we need the Hamming sketch to be able to recover at least T mismatched grammars at the very end caused by the dummy padding, $4T$ grammars at the beginning corresponding to $G_{r^x-2T+1}^x, G_{r^x-2T+2}^x, \dots, G_{r^x+2T}^x$, $2T$ neighbors of $G_{r^x+2T}^x$ to the right, and at most $(4T + 1)k$ mismatched grammars caused by the edit operations between x and y . This is less than $M(4T + 1)(k + 2)$ which is the number of mismatches our Hamming sketch can recover.

The time needed to compare the sketched strings can be bounded as follows: In total the procedure generates at most $\mathcal{O}(Tk)$ pairs of grammars of total size $\tilde{\mathcal{O}}(k^2)$ on which it runs edit distance computation from Proposition 2.6. If those edit distance computations take total time more than $\tilde{\mathcal{O}}(k^2)$ we can terminate them as we know the overall edit distance is larger than k . Recovering differing grammars from the Hamming distance sketch takes time $\tilde{\mathcal{O}}(k') = \tilde{\mathcal{O}}(k^2)$. Their follow-up processing such as counting their evaluation size and shortening them is proportional to their total size which is $\tilde{\mathcal{O}}(k^2)$. Hence, the time for comparing strings is $\tilde{\mathcal{O}}(k^2)$.

Failure probability. The update operations can fail if the grammar decomposition produces large grammars or the grammars are not deterministic (because of a collision caused by compression hash functions). This happens with probability at most $1/n$ for each update. Since we perform at most m updates, the failure probability of any update operations is at most $m/n \leq 1/10m^2$ by our choice of m and n .

When comparing two sketches for strings x and y of edit distance at most k , Theorem 2.19 might fail to align their grammar decomposition. This happens

with probability at most $1/5$. With probability at most $2/n$ the Hamming sketches might fail to recover the differing pairs of grammars. There is no other source of failure for strings of edit distance at most k so the probability of the compare operation failing is at most $1/3$. To boost the success probability of comparison from $2/3$ to $1 - 1/2m$, we again form a more robust sketch by taking $c \log m$ independent copies of the rolling edit distance sketch and operate on them simultaneously. For comparison we output the most frequent answer from the individual sketches. This multiplies the failure probability of update operations by $c \log m$, so it is still at most $1/2m$ for m large enough. The comparison will fail with probability at most $1/2m$.

For strings of edit distance more than k the comparison of an individual edit sketch will fail either because the Hamming sketch would need to recover more than k pairs of differing grammars or because the total edit distance of the differing grammars is more than k . In both these failure cases we can always output ∞ to be on the safe side.

Proofs of Lemma 3.6 and 3.7

Here we prove the remaining two lemmas.

Proof of Lemma 3.6. For the simplicity of case analysis we first compare the compression of wu and w . Consider the division of $w = B_1 \dots B_m$ when calling $\text{Compress}(w, \ell)$, and the division $wu = B'_1 \dots B'_{m'}$ when calling $\text{Compress}(wu, \ell)$. Let $w''' = \text{Compress}(w, \ell)$, from line. Let a be the last symbol of w . We consider three cases.

If $B_m = a^r$, $r \geq 1$, then $B_i = B'_i$ for all $i = 1 \dots, m - 1$, and B_m is a prefix of $B'_{m'}$, not necessarily proper. In this case, the compression of each B_i and B'_i , $i = 1, \dots, m - 1$, is the same, so w''' equals to w' in all but possibly the last symbol.

Otherwise, B_m consists of at least two singleton symbols. If the first symbol of u is a , then $B'_m = B_m[1, |B_m| - 1]$, and $B'_{m+1} = a^r$, for some $r \geq 2$. F_{CVL} will color the same all symbols of B_m and B'_m except for at most the last R symbols. Hence, B_m and B'_m will be compressed the same except for at most the last R symbols. The last R symbols are compressed into at most R symbols in w''' , and B'_{m+1} will be compressed into a single symbol. In this case we conclude that $w'[1, |w'| - R - 1] = w'''[1, |w'| - R - 1]$.

If the first symbol of u is not a then B_m is a prefix of $B'_{m'}$, and the compression of B_m and $B'_m[1, |B_m|]$ will differ in at most R last symbols. So $w'[1, |w'| - R] = w'''[1, |w'| - R]$.

Hence, in all three cases $w'[1, |w'| - R - 1] = w'''[1, |w'| - R - 1]$. Moreover, $||w'| - |w'''|| \leq R + 1$.

A similar argument gives $w''[1, |w''| - R - 1] = w'''[1, |w''| - R - 1]$, and $||w''| - |w'''|| \leq R + 1$. By the triangle inequality, $||w'| - |w''|| \leq 2(R + 1)$. Hence, $w'[1, |w'| - 3(R + 1)] = w''[1, |w'| - 3(R + 1)]$. Since $|u'| \leq |u|$, $|w'u'| - |u| \leq |w'|$. The claim follows. \square

Proof of Lemma 3.7. Part 1. Consider strings $B^x(\ell, i)$ from the trace of the algorithm on x given the hash functions $H_0, \dots, H_L, C_1, \dots, C_L$. (See Section 2.3.4) Similarly for $B^{xz}(\ell, i)$.

For $\ell = 0, \dots, L$ we will define integers i_ℓ and Δ_ℓ satisfying:

1. For all $i < i_\ell$, $B^x(\ell, i) = B^{xz}(\ell, i)$,
2. $B^x(\ell, i_\ell)[1, |B^x(\ell, i_\ell)| - \Delta_\ell] = B^{xz}(\ell, i_\ell)[1, |B^x(\ell, i_\ell)| - \Delta_\ell]$,
3. $\Delta_\ell + \sum_{i=i_\ell+1}^{s_\ell^x} |B^x(\ell, i)| \leq \ell(3R+3) + 1$.

For $\ell = 0$, $B^x(0, 1), B^x(0, 2), \dots, B^x(0, s_0^x) = \text{Split}(x, 0)$, so we set $i_0 = s_0^x$ and $\Delta_0 = 1$. Since $B^{xz}(0, 1), B^{xz}(0, 2), \dots, B^{xz}(0, s_0^{xz}) = \text{Split}(xz, 0)$, $s_0^x \leq s_0^{xz}$ and $B^x(0, s_0^x)$ might differ from $B^{xz}(0, s_0^x)$ by containing the last symbol of x which might be the first symbol of $B^{xz}(0, s_0^x + 1)$. Otherwise, $B^x(0, s_0^x)$ is the prefix of $B^{xz}(0, s_0^x)$ so the properties of i_0 and Δ_0 are satisfied.

For $\ell = 1, \dots, L$, having defined $i_{\ell-1}$ and $\Delta_{\ell-1}$ we will define i_ℓ and Δ_ℓ : Define

$$A_{\ell-1}^x = \text{Compress}(B^x(\ell-1, i_{\ell-1}), \ell) \quad \& \quad A_{\ell-1}^{xz} = \text{Compress}(B^{xz}(\ell-1, i_{\ell-1}), \ell),$$

$$(B_0, B_1, \dots, B_m) = \text{Split}(A_{\ell-1}^x, \ell) \quad \& \quad (B'_0, B'_1, \dots, B'_{m'}) = \text{Split}(A_{\ell-1}^{xz}, \ell).$$

For simplicity of exposition in this proof we assume that for any $B \in \Gamma^*$ of size at most 2, $\text{Compress}(B, \ell) = B$ and $\text{Split}(B, \ell) = (B)$, so they both perform no action on B of size at most 2.

Let

$$w = B^x(\ell-1, i_{\ell-1})[1, |B^x(\ell-1, i_{\ell-1})| - \Delta_{\ell-1}],$$

$$u = B^x(\ell-1, i_{\ell-1})[1 + |B^x(\ell-1, i_{\ell-1})| - \Delta_{\ell-1}, \dots],$$

$$v = B^{xz}(\ell-1, i_{\ell-1})[1 + |B^x(\ell-1, i_{\ell-1})| - \Delta_{\ell-1}, \dots].$$

By Lemma 3.6, $A_{\ell-1}^x$ and $A_{\ell-1}^{xz}$ agree on at least the first $|A_{\ell-1}^x| - (3R+3) - |u| = |A_{\ell-1}^x| - (3R+3) - \Delta_{\ell-1}$ symbols. (This is trivial when $|w| \leq 2$, in particular, when $|B^x(\ell-1, i_{\ell-1})| \leq 2$ or $|B^{xz}(\ell-1, i_{\ell-1})| \leq 2$.)

Let $i \in \{0, \dots, m\}$ be the largest i such that for all $j < i$, $B_j = B'_j$. Let i_ℓ be the index of block B_i among blocks $B^x(\ell, 1), B^x(\ell, 2), \dots, B^x(\ell, s_\ell^x)$. Notice, $B^x(\ell, j) = B^{xz}(\ell, j)$, for all $j < i_\ell$. Let $\Delta_\ell \geq 0$ be the smallest integer such that $B_i[1, |B_i| - \Delta_\ell] = B'_i[1, |B_i| - \Delta_\ell]$. (So the second property holds for ℓ , as $B^x(\ell, i_\ell) = B_i$ and $B^{xz}(\ell, i_\ell) = B'_i$.) Since $B_i[|B_i| - \Delta_\ell + 1, \dots] \cdot B_{i+1} \dots B_m$ forms a part of a suffix of $A_{\ell-1}^x$ on which $A_{\ell-1}^x$ and $A_{\ell-1}^{xz}$ differ, $\Delta_\ell + \sum_{j=i+1}^m |B_j| \leq (3R+3) + \Delta_{\ell-1}$.

Notice, $\sum_{j=i_\ell+1+s-i}^{s_\ell^x} |B^x(\ell, j)| \leq \sum_{j=i_{\ell-1}+1}^{s_{\ell-1}^x} |B^x(\ell-1, j)|$ as each $B^x(\ell, j)$ on the left is a part of a compression of some $B^x(\ell-1, j)$ on the right. Hence,

$$\begin{aligned} & \sum_{j=i_\ell+1}^{s_\ell^x} |B^x(\ell, j)| + \Delta_\ell \\ & \leq (3R+3) + \Delta_{\ell-1} + \sum_{j=i_{\ell-1}+1}^{s_{\ell-1}^x} |B^x(\ell-1, j)| \\ & \leq (3R+3) + (\ell-1)(3R+3) + 1 \leq \ell(3R+3) + 1 \end{aligned}$$

Eventually, $\Delta_L + \sum_{i=i_L+1}^{s_L^x} |B^x(L, i)| \leq L(3R+3) + 1$. Also $B^x(L, j) = B^{xz}(L, j)$ for $j = 1, \dots, i_L - 1$. Hence, $G_j^x = G_j^{xz}$ for $j = 1, \dots, i_L - 1$. Since each $|B^x(\ell, j)| \geq$

1, $s_L^x - i_L \leq L(3R + 3) + 1$, which implies $s_L^x - (L(3R + 3) + 2) \leq s_L^x - T \leq i_L - 1$ and the claim follows.

Part 2. The proof of this part proceeds similarly to the first part. Let $B^x(\ell, i)$ and $B^{xz}(\ell, i)$ be as above. For $\ell = 0, \dots, L$, we will define a sequence of integers $i_\ell, t_\ell, p_\ell, r_\ell$ satisfying:

1. t_ℓ is the last index i such that $B^{xz}(\ell, i)$ contains some symbol that comes from the compression of x , and r_ℓ is the length of the prefix of $B^{xz}(\ell, t_\ell)$ that comes from the compression of x .
2. $i_\ell \leq t_\ell$ and for all $i < i_\ell$, $B^x(\ell, i) = B^{xz}(\ell, i)$,
3. $B^x(\ell, i_\ell)[1, p_\ell] = B^{xz}(\ell, i_\ell)[1, p_\ell]$,
4. $r_\ell - p_\ell + \sum_{j=i_\ell}^{t_\ell-1} |B^{xz}(\ell, j)| \leq \ell(3R + 3) + 1$.

For $\ell = 0$, $B^x(0, 1), B^x(0, 2), \dots, B^x(0, s_0^x) = \text{Split}(x, 0)$ and $B^{xz}(0, 1), B^{xz}(0, 2), \dots, B^{xz}(0, s_0^{xz}) = \text{Split}(xz, 0)$,

so we set $i_0 = s_0^x$. If $|B^x(0, s_0^x)| \leq |B^{xz}(0, s_0^x)|$ we set $t_0 = i_0$, and $r_0 = p_0 = |B^x(0, i_0)|$, otherwise the last symbol of x starts the block $B^{xz}(0, s_0^x + 1)$ so we set $t_0 = i_0 + 1$, $p_0 = |B^x(0, s_0^x)|$ and $r_0 = 1$. (For completeness we set $i_{-1} = t_{-1} = 1$.) Clearly, the four properties are satisfied by this choice.

For $\ell = 1, \dots, L$, having defined $i_{\ell-1}, t_{\ell-1}, p_{\ell-1}, r_{\ell-1}$ we will define i_ℓ, t_ℓ, p_ℓ , and r_ℓ . As before, let

$$A_{\ell-1}^x = \text{Compress}(B^x(\ell - 1, i_{\ell-1}), \ell) \quad \& \quad A_{\ell-1}^{xz} = \text{Compress}(B^{xz}(\ell - 1, i_{\ell-1}), \ell).$$

Case 1. Consider the case when $i_{\ell-1} = t_{\ell-1}$. Let

$$\begin{aligned} w &= B^{xz}(\ell - 1, i_{\ell-1})[1, p_{\ell-1}], \\ u_x &= B^{xz}(\ell - 1, i_{\ell-1})[1 + p_{\ell-1}, r_\ell], \\ u_z &= B^{xz}(\ell - 1, i_{\ell-1})[1 + r_{\ell-1}, \dots], \\ v &= B^x(\ell - 1, i_{\ell-1})[1 + p_{\ell-1}, \dots]. \end{aligned}$$

Let $A_{\ell-1}^{xz} = w'u'_x u'_z$ where w' comes from the compression of w , u'_x comes from the compression of u_x , and u'_z comes from the compression of u_z . Let $A_{\ell-1}^x = w''v'$ where w'' comes from the compression of w , and v' comes from the compression of v . Set $r'_\ell = |w'u'_x|$, let $p'_\ell \leq r'_\ell$ be the largest integer so that $A_{\ell-1}^{xz}[1, p'_\ell] = A_{\ell-1}^x[1, p'_\ell]$. By Lemma 3.6, $p'_\ell \geq |w'| - 3(R + 1)$ so $r'_\ell - p'_\ell \leq |w'u'_x| - |w'| + 3(R + 1) \leq |u'_x| + 3(R + 1)$. Furthermore, $|u'_x| \leq |u_x| \leq r_{\ell-1} - p_{\ell-1} \leq (\ell - 1) \cdot (3R + 3) + 1$, which follows by properties of $p_{\ell-1}$ and $r_{\ell-1}$, so $r'_\ell - p'_\ell \leq \ell(3R + 3) + 1$.

Let $(B_0, B_1, \dots, B_m) = \text{Split}(A_{\ell-1}^{xz}, \ell)$. Let $i \geq 0$ be the smallest integer such that $p'_\ell \leq \sum_{j=0}^i |B_j|$ and let $t \geq 0$ be the smallest such that $r'_\ell \leq \sum_{j=0}^t |B_j|$. Set $p_\ell = p'_\ell - \sum_{j=0}^{i-1} |B_j|$ and $r_\ell = r'_\ell - \sum_{j=0}^{t-1} |B_j|$. Let i_ℓ be the index of block B_i among blocks $B^{xz}(\ell, 1), B^{xz}(\ell, 2), \dots, B^{xz}(\ell, s_\ell^{xz})$, and let t_ℓ be the index of B_t among those blocks. Notice, $B^x(\ell, j) = B^{xz}(\ell, j)$, for all $j < i_\ell$. We conclude the case by observing that $r_\ell - p_\ell + \sum_{j=i}^{t-1} |B_j| = \sum_{j=i}^{t-1} |B_j| + (r'_\ell - \sum_{j=0}^{t-1} |B_j|) - (p'_\ell - \sum_{j=0}^{i-1} |B_j|) = p'_\ell - r'_\ell \leq \ell(3R + 3) + 1$.

Case 2. The case $i_{\ell-1} < t_{\ell-1}$ is similar. In this case we let w and v to be as in the previous case and $u = B^{xz}(\ell-1, i_{\ell-1})[1 + p_{\ell-1}, \dots]$. We let $p'_\ell \leq |A_{\ell-1}^{xy}|$ be the largest integer so that $A_{\ell-1}^{xz}[1, p'_\ell] = A_{\ell-1}^x[1, p'_\ell]$. By Lemma 3.6, $p'_\ell \geq |A_{\ell-1}^{xy}| - 3(R+1) - |u| = |A_{\ell-1}^{xy}| - 3(R+1) - |B^{xz}(\ell-1, i_{\ell-1})| + p_{\ell-1}$. Rearranging terms: $-p'_\ell + |A_{\ell-1}^{xy}| \leq -p_{\ell-1} + |B^{xz}(\ell-1, i_{\ell-1})| + 3(R+1)$.

Let $i \geq 0$ be the smallest integer such that $p'_\ell \leq \sum_{j=0}^i |B_j|$. Let $p_\ell = p'_\ell - \sum_{j=0}^{i-1} |B_j|$, and i_ℓ be the index of the block B_i within $B^{xz}(\ell, 1), B^{xz}(\ell, 2), \dots, B^{xz}(\ell, s_\ell^{xz})$. Hence, $-p_\ell + \sum_{j=i_\ell}^{i_\ell+m-i} |B^{xz}(\ell, j)| = -p_\ell + \sum_{j=i}^m |B_j| = -p'_\ell + \sum_{j=0}^m |B_j| = -p'_\ell + |A_{\ell-1}^{xy}| \leq -p_{\ell-1} + |B^{xz}(\ell-1, i_{\ell-1})| + 3(R+1)$.

Let $C_{\ell-1}^{xz} = \text{Compress}(B^{xz}(\ell-1, t_{\ell-1}), \ell)$ and $(B'_0, B'_1, \dots, B'_{m'}) = \text{Split}(C_{\ell-1}^{xz}, \ell)$. Let r'_ℓ be the largest position in $C_{\ell-1}^{xz}$ of a symbol coming from compression of x , and $t \geq 0$ be the smallest integer such that $r'_\ell \leq \sum_{j=0}^t |B_j|$, and set $r_\ell = r'_\ell - \sum_{j=0}^{t-1} |B'_j|$. Let t_ℓ be the index of the block B'_t within $B^{xz}(\ell, 1), B^{xz}(\ell, 2), \dots, B^{xz}(\ell, s_\ell^{xz})$. Clearly, $r'_\ell \leq r_{\ell-1}$ so $r_\ell + \sum_{j=t_\ell-t}^{t_\ell-1} |B^{xz}(\ell, j)| = r_\ell + \sum_{j=0}^{t-1} |B'_j| \leq r'_\ell \leq r_{\ell-1}$.

Notice, $\sum_{j=i_\ell+m-i+1}^{t_\ell-t-1} |B^{xz}(\ell, j)| \leq \sum_{j=i_{\ell-1}+1}^{t_{\ell-1}-1} |B^{xz}(\ell-1, j)|$. By partitioning the sum, rearranging the terms and using the upper bounds derived so far we have: $r_\ell - p_\ell + \sum_{j=i_\ell}^{t_\ell-1} |B^{xz}(\ell, j)| = -p_\ell + \sum_{j=i_\ell}^{i_\ell+m-i} |B^{xz}(\ell, j)| + \sum_{j=i_\ell+m-i+1}^{t_\ell-t-1} |B^{xz}(\ell, j)| + \sum_{j=t_\ell-t}^{t_\ell-1} |B^{xz}(\ell, j)| + r_\ell \leq -p_{\ell-1} + |B^{xz}(\ell-1, i_{\ell-1})| + 3(R+1) + \sum_{j=i_{\ell-1}+1}^{t_{\ell-1}-1} |B^{xz}(\ell-1, j)| + r_{\ell-1} = r_{\ell-1} - p_{\ell-1} + \sum_{j=i_{\ell-1}}^{t_{\ell-1}-1} |B^{xz}(\ell-1, j)| + 3(R+1) \leq (\ell-1) \cdot (3R+3) + 1 + 3(R+1) \leq \ell(3R+3) + 1$, where the second to last inequality follows by the properties of our numbers for $\ell-1$.

For $\ell = L$ we get: $r_L - p_L + \sum_{j=i_L}^{t_L-1} |B^{xz}(L, j)| \leq L(3R+3) + 1$. Since $p_L \leq |B^{xz}(L, i_L)|$, and $r_L \geq 0$ we get: $\sum_{j=i_L+1}^{t_L-1} |B^{xz}(L, j)| \leq L(3R+3) + 2$. Since each $B^{xz}(L, j)$ is of non-zero size, $t_L - i_L - 1 \leq L(3R+3) + 2$. Thus $t_L \leq i_L + L(3R+3) + 3 \leq s_L^x + L(3R+3) + 3$, as $i_L \leq s_L^x$. The claim follows. \square

Appending a symbol to a grammar decomposition

In this section we provide a detailed description of the process of updating the active grammars of a string x when appending a new symbol a . By Lemma 3.7 only the last T grammars of x might change when adding a new symbol a . As observed already previously, Lemma 3.7 also implies that once a grammar becomes more than $(T+1)$ -th grammar from the end it will never change, despite the fact that the number of grammars that follow it might shrink after adding more symbols. (Adding more symbols might create periodicity that will be exploited by the compression.) Our rolling sketch algorithm keeps at most T active grammars that might still change after adding more symbols. It is convenient for our implementation of the update function to have access also to the previous at most T committed grammars (to have the proper context for re-compression). Our rolling sketch algorithm has those committed grammars available in appropriate buffers. Thus we will assume that the update function is always invoked with exactly $T+1$ grammars, unless x is decomposed into less than $T+1$ grammars. Some of the first few grammars from the output of the update procedure should be discarded as they correspond to grammars that should stay the same. In particular, if there are t active grammars and s committed grammars then we should discard the first $\min(s, T+1-t)$ grammars from its output. The following statement encapsulates the properties of our update procedure `UpdateActiveGrammars()`.

Theorem 3.8. *Let integers $k \leq n$ and functions C_1, \dots, C_L and H_0, \dots, H_L be given. For any $a \in \Sigma$ and $x \in \Sigma^*$ of length at most n with G_1, \dots, G_s being the grammars output by the decomposition algorithm on input x using functions $C_1, \dots, C_L, H_0, \dots, H_L$, $\text{UpdateActiveGrammars}(G_{s-\min(s,T+1)+1}, \dots, G_s, a)$ outputs a sequence of grammars $G'_1, \dots, G'_{t'}$ such that $G_1, \dots, G_{s-\min(s,T+1)}$, $G'_1, \dots, G'_{t'}$ is the sequence that would be output by the decomposition algorithm on $x \cdot a$ using the functions $C_1, \dots, C_L, H_0, \dots, H_L$. The update algorithm runs in time $\tilde{O}(kLT) = \tilde{O}(k)$ and outputs $t' \leq 4TL$ grammars.*

Here we assume that the decomposition algorithm does not fail neither on x nor on $x \cdot a$ with respect to producing correct deterministic grammars so the first two parts of Theorem 2.8 are satisfied for x and $y = x \cdot a$, and the choice of functions C_1, \dots, C_L and H_0, \dots, H_L . For the simplicity of our implementation, we assume a stronger property of C_1, \dots, C_L , that each C_ℓ is one-to-one on the union of all blocks of x and $x \cdot a$ at level ℓ . (See remark after Lemma 2.14.)

Auxiliary functions Our update algorithm uses several simple and straightforward auxiliary functions we describe next. Function $\text{DecompressSymbol}(c, G, \ell, t)$ takes a symbol $c \in \Gamma$ and if it is a level- ℓ symbol compressed by the grammar G then it returns its decompression truncated to the length of at most t symbols. Otherwise it returns the original symbols c .

Algorithm 5 $\text{DecompressSymbol}(c, G, \ell, t)$

Input: A symbol c , a grammar G , a level ℓ , maximum output size $t \geq 2$.

Output: Decompresses c if it was compressed at level ℓ . Returns at most t symbols of the decompression.

- 32 **if** $c \in \Sigma_c^\ell$ **then** let $a, b \in \Gamma$ be such that $c \rightarrow ab \in G$. Return ab .
33 **if** $c \in \Sigma_r^\ell$ **then** let $a \in \Gamma, r \in \mathbb{N}$ be such that $c = \mathbf{r}_{a,r}$. Return $a^{\min(t,r)}$.
34 Return c .
-

Function $\text{DecompressString}(Z, G, \ell)$ decompresses all level- ℓ compression symbols in a string $Z \in \Gamma^*$ using the grammar G , and returns the resulting decompressed string.

Algorithm 6 $\text{DecompressString}(Z, G, \ell)$

Input: A string Z , a grammar G , and level ℓ .

Output: Decompresses z at level ℓ .

- 35 $Y = \varepsilon$.
36 **for** $i = 1$ **to** $|Z|$ **do** $Y = Y \cdot \text{DecompressSymbol}(Z[i], G, \ell, \infty)$.
37 Return Y .
-

Function $\text{DecompressSymbolLength}(c, \ell)$ returns the length of the decompression of a symbol c at level ℓ .

Algorithm 7 DecompressSymbolLength(c, ℓ)

Input: A symbol c , a level ℓ .**Output:** Returns the length of decompression of c at level ℓ .

```
38 if  $c \in \Sigma_c^\ell$  then return 2.
39 if  $c \in \Sigma_r^\ell$  then let  $a \in \Gamma, r \in \mathbb{N}$  be such that  $c = \mathbf{r}_{a,r}$ . Return  $r$ .
40 Return 1.
```

Algorithm 8 CompressWithGrammar(B, ℓ)

Input: String B over alphabet Γ , and level number ℓ .**Output:** String B'' over alphabet Γ , and set of applied rules G' .

```
41 if  $|B| \leq 1$  then return  $B, \emptyset$ .
42 Set  $G' = \emptyset$ .
43 Divide  $B = B_1 B_2 B_3 \dots B_m$  into minimum number of blocks so that each maximal
   subword  $a^r$  of  $B$ , for  $a \in \Gamma$  and  $r \geq 2$ , is one of the blocks.
44 for each  $i \in \{1, \dots, m\}$  do
45   | if  $B_i = a^r$ , where  $r \geq 2$  then
46   |   | Set  $B'_i = \mathbf{r}_{a,r} \cdot \#$  and color  $\mathbf{r}_{a,r}$  by 1 and  $\#$  by 2.
47   |   |  $G' = G' \cup \{\mathbf{r}_{a,r} \rightarrow a^r\}$ ;
48   |   end
49   | else Set  $B'_i = B_i$  and color each symbol of  $B'_i$  according to  $F_{\text{CVL}}(B_i)$ 
50   end
51 Set  $B' = B'_1 B'_2 \dots B'_m$ ,  $B'' = \varepsilon$ , and  $i = 1$ .
52 while  $i < |B'|$  do
53   | if  $B'[i+1] = \#$  then  $B'' = B'' \cdot B'[i]$ 
54   | else
55   |   |  $B'' = B'' \cdot C_\ell(B'[i, i+1])$ ;
56   |   |  $G' = G' \cup \{C_\ell(B'[i, i+1]) \rightarrow B'[i, i+1]\}$ ;
57   |   end
58   |  $i = i + 2$ .
59   | if  $i \leq |B'|$  and  $B'[i]$  is not colored 1 then  $B'' = B'' \cdot B'[i]$ ,  $i = i + 1$ 
60   end
61 Return  $B'', G'$ .
```

Function $\text{CompressWithGrammar}(B, \ell)$ is an extension of $\text{Compress}(B, \ell)$ that in addition to compressed block B at level ℓ returns the set of grammar rules used for the compression of B at this level.

Finally, function $\text{FindCompressedPrefix}(Z, p, \ell)$ returns the length of the smallest prefix of a string Z that decompresses into at least p symbols at level ℓ .

Algorithm 9 FindCompressedPrefix(Z, p, ℓ)

Input: String Z , an integer p , level ℓ .

Output: Smallest index j such that level ℓ decompression of $Z[1, j]$ has length $\geq p$.

```
62  $q = 0$  and  $j = 0$ .
63 while  $q < p$  do
64    $j = j + 1$ ;
65    $p = p + \text{DecompressSymbolLength}(Z[j], \ell)$ ;
66 end
67 Return  $j$ .
```

Main functions The core of the update function

UpdateActiveGrammars($(G_1, \dots, G_t), a$) is build around the functions we describe next. The functions use globally accessible set of grammar rules G that contains all the rules from G_1, \dots, G_t except for the starting rules. (This set of rules is deterministic assuming the remark after Theorem 3.8.)

The functions will build a sequence of strings Z_L, Z_{L-1}, \dots, Z_0 each of length at most $2T$. Z_L is the concatenation of the right-hand-sides of starting rules of G_1, \dots, G_t . For $\ell = L, \dots, 1$, $Z_{\ell-1}$ is then build inductively by decompressing a (suitable) largest suffix of Z_ℓ so that $Z_{\ell-1}$ would be of length at most $T + 4 \leq 2T$. The decompression is provided by function PartiallyDecompress(Z, F, ℓ) which returns tuple Z', F', u, r' . In the case that the first symbol of the decompressed suffix of Z_ℓ is the level- ℓ repeat symbol $\mathbf{r}_{a,r}$ that would expand $Z_{\ell-1}$ beyond the limit of $T + 4$ symbols, we truncate the expansion of that symbol to the length $r_\ell = r'$. The return value u indicates how many symbols of Z were left uncompressed (which would include the partially decompressed symbol $\mathbf{r}_{a,r}$). It satisfies that if $u \neq 0$ then $|Z'| \geq T$. Strings Z_L, \dots, Z_0 satisfy that for $\ell = L, \dots, 1$, if $|Z_\ell| \geq T$ then $|Z_{\ell-1}| \geq T$. (In particular, if UpdateActiveGrammars() is invoked with at least $T + 1$ grammars, then all Z_ℓ are of length at least T . The compression of the first grammar might depend on unseen grammars in that case so we cannot re-compress it at will.)

Strings Z_L, \dots, Z_0 are accompanied by strings of integers F_L, \dots, F_0 over the alphabet $\{0, \dots, L + 1\}$. The value of $F_\ell[i]$ indicates at which level the symbol $Z_\ell[i]$ becomes the first symbol in its block. In particular, $F_\ell[i] < \ell$ indicates that a block starts at position i of Z_ℓ . This value is relevant for re-compression of updated strings Z_0, \dots, Z_L . The initial values of F_L are computed using SplittingDepth(G). Function SplittingDepth(G) is fairly straightforward: For a grammar G , it inductively decompresses the first two symbols of the evaluation of G . It finds the lowest level ℓ , at which the first two symbols of the decompression give zero when function H_ℓ is applied on them.

After obtaining Z_L, \dots, Z_0 , UpdateActiveGrammars($(G_1, \dots, G_t), a$) appends a to Z_0 , and then re-compresses Z_0, \dots, Z_{L-1} using a function Recompress(B, Z, F, u, r, ℓ). We provide more details on function Recompress(B, Z, F, u, r, ℓ) further below.

Invoking UpdateActiveGrammars($(G_1, \dots, G_t), a$) returns a sequence of updated grammars.

Algorithm 10 PartiallyDecompress(Z, F, ℓ)

Input: String Z , splitting depth string F , and level ℓ .**Output:** Decompressed string Z' , splitting depth string F' , unused count u , repeat count r' .

```
68 Set  $Z' = \varepsilon$  and  $F' = \varepsilon$ .
69 for  $u = |Z|$  to 1 do
70   if  $Z[u] = r_{a,r}$ , where  $r_{a,r} \in \Sigma_r^\ell$  then
71     if  $|Z'| + r \leq T + 3$  then  $Z' = a^r \cdot Z'$  and  $F' = F[u] \cdot (L + 1)^{r-1} \cdot F'$ 
72     else
73        $r' = T - |Z'| + 1$ ;
74        $Z' = a^{r'} \cdot Z'$  and  $F' = (L + 1)^{r'} \cdot F'$ ;
75       Return  $Z', F', u, r'$ .
76   end
77 end
78 else if  $Z[u] = a$ , where  $a \in \Sigma_c^\ell$  then
79    $Z' = b \cdot c \cdot Z'$ , where  $a \rightarrow b \cdot c$  is in  $G$ ;
80    $F' = F[u] \cdot (L + 1) \cdot F'$ ;
81 end
82 else  $Z' = Z[u] \cdot Z'$  and  $F' = F[u] \cdot F'$ 
83 if  $|Z'| \geq T$  then return  $Z', F', u - 1, 0$ .
84 end
85 Return  $Z', F', 0, 0$ .
```

Algorithm 11 SplittingDepth(G)

Input: Non-empty grammar G .**Output:** The first level ℓ where G would be separated as a new block.

```
86 Let  $v$  be such that  $\# \rightarrow v \in G$ . //  $v$  are the first two symbols of eval( $G$ ).
87  $d = L + 1$ .
88 for  $\ell = L, \dots, 0$  do
89   if  $|v| \geq 2$  and  $H_\ell(v[1, 2]) = 0$  then  $d = \ell$ .
90    $u = \text{DecompressSymbol}(v[1], G, \ell, 2)$ 
91   if  $|v| \geq 2$  then  $u = u \cdot \text{DecompressSymbol}(v[2], G, \ell, 2)$ .
92    $v = u$ .
93 end
94 Return  $d$ .
```

Algorithm 12 UpdateActiveGrammars(AG, a)

Input: List of grammars $AG = (G_1, \dots, G_t)$ representing a string x , and a symbol a .

Output: Updated list of grammars AG' representing string $x \cdot a$.

```
95 // Construct a set of rules  $G$ , initial compressed string  $Z_L$  and splitting depth
    string  $F_L$ .
96 For  $i = 1, \dots, t$ , let  $\# \rightarrow v_i$  be the starting rule in  $G_i$ .
97 Set  $G = \bigcup_{i=1}^t G_i \setminus \{\# \rightarrow v_i\}$ .
98 Set  $Z_L = v_1$  and  $F_L = 0 \cdot (L + 1)^{|v_1|-1}$ .
99 For  $i = 2, \dots, t$ , set  $Z_L = Z_L \cdot v_i$  and  $F_L = F_L \cdot \text{SplittingDepth}(G_i) \cdot (L + 1)^{|v_i|-1}$ .
100 // Perform partial decomposition
101 for  $\ell = L$  to 1 do
102 |  $Z_{\ell-1}, F_{\ell-1}, u_\ell, r_\ell = \text{PartiallyDecompress}(Z_\ell, F_\ell, \ell)$ .
103 end
104 // Perform re-compression
105  $Z_0 = Z_0 \cdot a$ ;  $B = \text{Split}(Z_0, 0)$ ;
106 for  $\ell = 1$  to  $L$  do
107 |  $B', G' = \text{Recompress}(B, Z_\ell, F_\ell, u_\ell, r_\ell, |Z_{\ell-1}|, \ell)$ 
108 |  $G = G \cup G'$ 
109 |  $B = B'$ 
110 end
111 Let  $B = (B_1, \dots, B_{t'})$ .
112  $AG' = ()$ .
113 for  $i = 1$  to  $t'$  do
114 |  $G' = G \cup \{\# \rightarrow B_i\}$ .
115 | Remove from  $G'$  unnecessary rules to get  $G'_i$  (as in Section 2.3.1).
116 | Append  $G'_i$  to  $AG'$ .
117 end
118 Return  $AG'$ .
```

Function $\text{Recompress}(B, Z, F, u, r, \ell)$ gets a sequence $B = (B_0, \dots, B_s)$ of blocks that represent compression of the updated $Z_{\ell-1}$ (after adding a) up-to level $\ell - 1$. It also gets the original Z_ℓ , the splitting depth string F_ℓ , the number of symbols u_ℓ that were decompressed from Z_ℓ to get the original $Z_{\ell-1}$ and the parameter r_ℓ that indicates that the first r_ℓ symbols of $Z_{\ell-1}$ are a partial decomposition of the repeat symbol $Z_\ell[u]$. It outputs a sequence of blocks B' that represent the updated block Z_ℓ compressed up-to level ℓ , and a set of rules G' that were used for compression at level ℓ .

Blocks B_1, \dots, B_s can be independently compressed and split at level ℓ . The block B_0 needs a special treatment though as it needs to be combined with its possible remainder in Z_ℓ .

This is done in function $\text{RecompressFirstBlock}(B_0, Z, F, u, r, \ell)$. Remaining blocks for the output $\text{Recompress}()$ are obtained from Z_ℓ by splitting it into blocks according to F_ℓ .

Algorithm 13 Recompress(B, Z, F, u, r, z, ℓ)

Input: $B = (B_0, \dots, B_s)$ sequence of blocks, original uncompressed string Z , splitting depth string F of Z , u number of uncompressed symbols in Z , repeat count r , $z = |Z_{\ell-1}|$, and level ℓ .
Output: B' a new sequence of blocks representing B together with $Z[1, u]$, and set of newly added rules G' .

```
119 if  $z < T$  then  $B' = ()$ ,  $G' = \emptyset$ ,  $u' = 0$ ,  $j = 0$ . // No symbols precede  $B_0$ .
120 else
121    $B', G', u' = \text{RecompressFirstBlock}(B_0, Z, F, u, r, \ell)$ . // Compress block  $B_0$ .
122    $j = 1$ .
123 end
124 // Compress blocks  $B_j, \dots, B_s$ .
125 for  $i = j$  to  $s$  do
126   if  $|B_i| \leq 2$  then  $B'' = (B_i)$ ;  $G'' = \emptyset$ 
127   else
128      $B'_i, G'' = \text{CompressWithGrammar}(B_i, \ell)$ .
129      $B'' = \text{Split}(B'_i, \ell)$ .
130   end
131   Append  $B''$  to  $B'$ .
132    $G' = G' \cup G''$ .
133 end
134  $i = u'$ . // Separate remaining blocks in  $Z$ .
135 while  $i > 0$  do
136   while  $i > 1$  and  $F[i] > \ell$  do  $i = i - 1$ .
137   Add  $Z[i, u']$  as the first item of  $B'$ .
138    $i = i - 1$ ;  $u' = i$ .
139 end
140 Return  $B', G'$ .
```

Function $\text{RecompressFirstBlock}(B_0, Z, F, u, r, \ell)$ is the most complicated function of the whole re-compression process. The function is invoked only if $|Z_{\ell-1}| \geq T$. The function gets the first level $\ell - 1$ block B_0 that needs to be combined with its remainder in $Z = Z_\ell$. The remainder is a suffix of $Z_\ell[1, u]$, where r indicates that the first r symbols of the original $Z_{\ell-1}$ were obtained by the partial decompression of $Z_\ell[u]$. If $r \neq 0$ then the compression of the part of B_0 that follows its leading a 's ($Z_\ell[u] = \mathbf{r}_{a,r'}$) is independent of the compression of the part of Z belonging to B_0 and preceding $Z_\ell[u]$, as $r' - r \geq 2$. Thus we can compress that part of B_0 , combine it with an appropriate repetition symbol $\mathbf{r}_{a,r''}$ and append it to the appropriate suffix of $Z_\ell[1, u - 1]$ (which is already compressed at level ℓ .) If $r = 0$ then we invoke a function $\text{CrossOverBlock}(B_0, Z[u', \dots], u - u' + 1, \ell)$, where u' is the first symbol in Z_ℓ that belongs to the block of B_0 . Eventually, we split the compressed block B_0 using $\text{Split}()$.

Algorithm 14 RecompressFirstBlock(B_0, Z, F, u, r, ℓ)

Input: Block B_0 , an original uncompressed string Z , splitting depth string F of Z , u number of uncompressed symbols in Z , repeat count r , and level ℓ .

Output: B' a new sequence of blocks representing B_0 together with $Z[1, u]$, and set of newly added rules G' , number u' of unused symbols in Z .

```
141 if  $r \neq 0$  then  $u = u - 1$ .
142  $u' = u + 1$ . // Find the beginning of block  $B_0$  in the uncompressed part  $Z$ .
143 while  $u' > 1$  and  $d[u'] \geq \ell$  do  $u' = u' - 1$ .
144 if  $r \neq 0$  then
145     // Block  $B_0$  starts by partially decompressed symbol  $\mathbf{r}_{a,r}$ .
146     Let  $a \in \Gamma$  and  $r' \in \mathbb{N}$  be such that  $Z[u + 1] = \mathbf{r}_{a,r'}$ .
147     for  $i = 1$  to  $|B_0|$  do if  $B_0[i] \neq a$  then break;
148
149     if  $B_0[i] = a$  then  $B' = \varepsilon, G'' = \emptyset, i = i + 1$ .
150     else  $B', G'' = \text{CompressWithGrammar}(B_0[i, \dots], \ell)$ .
151      $B' = Z[u', u] \cdot \mathbf{r}_{a,r'-r+i-1} \cdot B'$ .
152 end
153 else
154      $B', G'' = \text{CrossOverBlock}(B_0, Z[u', \dots], u - u' + 1, \ell)$ .
155 end
156  $B'' = \text{Split}(B', \ell)$ .
157 Return  $B'', G'', u' - 1$ .
```

Function $\text{CrossOverBlock}(B, Z, u, \ell)$ gets a block B that was compressed up-to level $\ell - 1$ and needs to be combined with its remainder $Z[1, u]$ that is compressed up-to level ℓ . (The resulting block should correspond to “ $Z[1, u] \cdot B$ ”.) We know that $|Z_{\ell-1}| \geq T \geq L(3R + 3)$ otherwise $\text{RecompressFirstBlock}()$ and $\text{CrossOverBlock}()$ would not be called. By the three properties of $\Delta_{\ell-1}$ and $i_{\ell-1}$ defined in the proof of Part 1 of Lemma 3.7 we know that the first $3(R + 1)$ symbols of $Z_{\ell-1}$ were not modified as a result of appending the new symbol to x . Hence the first $\min(3(R + 1), |B|)$ symbols of B correspond to the decompression of $Z[u + 1, \dots]$.

In this part of B we look for any repeated symbol. If we find a repeated symbol there, we combine the compression of the part of B starting at the repeated symbol with the original part of $Z[u + 1, \dots]$ that produced the symbols of B preceding the repeated symbol (and also with $Z[1, u]$). By the properties of compression, repeated symbols break dependence between compressed symbols.

If $|B| \leq 2R + 20$ then at least $3(R + 1) - 2R - 20 > 2$ unchanged symbols follow B . Thus B ends at its original location as it was split at some level $< \ell$ and the first two symbols of the next block at all levels $< \ell$ are the same as originally.

Finally, if $|B| > 2R + 20$ and there is no repeated symbol in the first up-to $3(R + 1)$ symbols of B then we can compress B to get B' , strip from B' the compression of the first $R + 10$ symbols and combine it with the original compression of those $R + 10$ symbols from Z . (The first up-to $3(R + 1)$ symbols of B consist of singletons. The compression of a singleton depends on the context of at most $R + 3$ symbols on either side.)

Algorithm 15 CrossOverBlock(B, Z, u, ℓ)

Input: Block B , an original uncompressed string Z , number u of unused symbols in Z , and level ℓ .

Output: B' and set of newly added rules G' .

```
158           // Try to find a repeated symbol in unmodified B.
159  $i = 1$ .
160 while  $i < |B|$  and  $i < 3(R + 1)$  and  $B[i] \neq B[i + 1]$  do  $i = i + 1$ .
161 if  $i < |B|$  and  $B[i] = B[i + 1]$  then
162     |           //  $B[i]$  is a repeated symbol.
163     |    $B', G' = \text{CompressWithGrammar}(B[i, \dots], \ell)$ .
164     |    $j = \text{FindCompressedPrefix}(Z[u + 1, \dots], i - 1, \ell)$ .
165     |    $B' = Z[1, u + j] \cdot B'$ .
166 end
167 else if  $|B| \leq 2R + 20$  then
168     |    $j = \text{FindCompressedPrefix}(Z[u + 1, \dots], |B|, \ell)$ . // At least two unchanged
169     |   |   symbols follow B.
170     |    $B' = Z[1, u + j], G' = \emptyset$ .
171 end
172 else
173     |    $B', G' = \text{CompressWithGrammar}(B, \ell)$ .
174     |    $p = \text{FindCompressedPrefix}(B', R + 10, \ell)$ .
175     |    $j = \text{FindCompressedPrefix}(Z[u + 1, \dots], R + 10, \ell)$ .
176     |    $B' = Z[1, u + j - 1] \cdot B'[p, \dots]$ .
177 end
178 Return  $B', G'$ .
```

The correctness of the update algorithm follows from its description.

Time analysis We assume that strings are represented efficiently (e.g. by balanced trees) so we can extract a sub-string, concatenate strings, etc. in time $\tilde{\mathcal{O}}(1)$. All strings that we will operate on will be of length $\mathcal{O}(T)$. Similarly, we assume that grammars are represented efficiently so that we can look-up a rule with a given left-hand symbol, append two grammars, etc. in time $\tilde{\mathcal{O}}(1)$.

Then DecompressSymbol() and DecompressSymbolLength() takes time $\tilde{\mathcal{O}}(1)$. The time complexity of each of the functions CompressWithGrammar(), DecompressString(), FindCompressedPrefix(), PartiallyDecompress() and CrossOverBlock() is proportional to the length of strings on which it operates so it is $\tilde{\mathcal{O}}(T)$. Time of SplittingDepth() is proportional to the depth of the grammar, which in our case is at most $\tilde{\mathcal{O}}(L)$. Each RecompressFirstBlock() executes $\mathcal{O}(T)$ operations on strings and grammars, and $\mathcal{O}(T)$ evaluations of H_ℓ (inside the calls to Split()). Since H_ℓ is $\tilde{\mathcal{O}}(k)$ -wise independent, its evaluation takes time $\tilde{\mathcal{O}}(k)$. So RecompressFirstBlock() takes time $\tilde{\mathcal{O}}(kT)$.

Similarly, each Recompress() executes upto one call to RecompressFirstBlock(), $\mathcal{O}(T)$ operations on strings and grammars, and $\mathcal{O}(T)$ evaluations of H_ℓ to split blocks. Again, its total time complexity is $\tilde{\mathcal{O}}(kT)$.

Eventually, UpdateActiveGrammars() executes up-to T SplittingDepth(), $\mathcal{O}(T)$ string operations, L calls to PartiallyDecompress() and Recompress(), and

then up-to $\mathcal{O}(LT)$ invocations of grammar minimization procedure costing $\tilde{\mathcal{O}}(k)$ time each. Thus, the total time for `UpdateActiveGrammars()` is $\tilde{\mathcal{O}}(LTk)$.

The number of grammars the algorithm outputs is at most $\sum_{\ell=0}^L |Z_\ell| \leq 2T(L+1) \leq 4TL$.

3.2 Streaming k -edit approximate pattern matching via string decomposition

In this section we give an algorithm for streaming k -edit approximate pattern matching which uses space $\tilde{\mathcal{O}}(k^2)$ and time $\tilde{\mathcal{O}}(k^2)$ per arriving symbol. This improves substantially on the recent algorithm of Kociumaka, Porat and Starikovskaya [31] which uses space $\tilde{\mathcal{O}}(k^5)$ and time $\tilde{\mathcal{O}}(k^8)$ per arriving symbol. In the k -edit approximate pattern matching problem we get a pattern P and text T and we want to identify all substrings of the text T that are at edit distance at most k from P . In the streaming version of this problem both the pattern and the text arrive in a streaming fashion symbol by symbol and after each symbol of the text we need to report whether there is a current suffix of the text with edit distance at most k from P . We measure the total space needed by the algorithm and time needed per arriving symbol.

Pattern matching is a classical problem of finding occurrences of a given pattern P in text T . It can be solved in time linear in the size of the pattern and text [32, 33, 18]. The classical algorithms use space that is proportional to the pattern size. In a surprising work [34], Porat and Porat were the first to design a pattern matching algorithm that uses less space. They designed an *on-line* algorithm that pre-processes the pattern P into a small data structure, and then it receives the text symbol by symbol. After receiving each symbol of the text, the algorithm is able to report whether the pattern matches the current suffix of the text. The algorithm uses poly-logarithmic amount of memory for storing the data structure and processing the text. This represents a considerable achievement in the design of pattern matching algorithms.

Porat and Porat also gave a small-space online algorithm that solves approximate pattern matching up-to Hamming distance k , *k -mismatch approximate pattern matching*. In this problem we are given the pattern P and a parameter k , and we should find all substrings of the text T that are at Hamming distance at most k from P . Their algorithm uses $\tilde{\mathcal{O}}(k^3)$ space, and requires $\tilde{\mathcal{O}}(k^2)$ time per arriving symbol of the text. Subsequently this was improved to space $\tilde{\mathcal{O}}(k)$ and time $\tilde{\mathcal{O}}(\sqrt{k})$ [35]. There has been a series of works [36, 37, 35, 38, 39, 40, 41, 42, 43] on online and streaming pattern matching, and the line of work culminated in the work of Clifford, Kociumaka and Porat [23] who gave a fully *streaming* algorithm with similar parameters as [35].

In the streaming setting, also the pattern arrives symbol by symbol and we do not have the space to store all of it at once. An important feature of the algorithm of Clifford, Kociumaka and Porat is that their algorithm not only reports the k -mismatch occurrences of the pattern but for each k -mismatch occurrence of P it can also output the full information about the difference between P and the current suffix of the text, so called *mismatch information*.

Beside approximate pattern matching with respect to Hamming distance,

researchers also consider approximate pattern matching with respect to other similarity measures such as edit distance. Edit distance $\Delta_{edit}(x, y)$ of two strings x and y is the minimum number of insertions, deletions and substitutions needed to transform x into y . The *k-edit approximate pattern matching problem* is a variant of the approximate pattern matching where we should find all substrings of T that are at edit distance at most k from P . Since there could be quadratically many such substrings, we usually only require to report for each position in T whether there is a substring of T ending at that position that has edit distance at most k from P . In the streaming version of the problem we want to output the minimal distance of P to a current suffix of the text after receiving each symbol of T . Again we assume that the text as well as the pattern arrive symbol by symbol, and we are interested in how much space the algorithm uses, and how much time it takes to process each symbol.

Starikovskaya [41] proposed a streaming algorithm for the k -edit pattern matching problem, which uses $\tilde{O}(k^8\sqrt{m})$ space and takes $\tilde{O}(k^2\sqrt{m} + k^{13})$ time per arriving symbol. Here, we denote $m = |P|$ and $n = |T|$. Recently, using a very different technique Kociumaka, Porat and Starikovskaya [31] constructed a streaming algorithm, which uses $\tilde{O}(k^5)$ space and $\tilde{O}(k^8)$ amortized time per arriving symbol of the text.

In this work we substantially improve on the result of Kociumaka, Porat and Starikovskaya. We give a streaming algorithm for k -edit approximate pattern matching that uses $\tilde{O}(k^2)$ space and $\tilde{O}(k^2)$ time per arriving symbol.

Theorem 3.9. *Given integer $k \geq 0$, there exists a randomized streaming algorithm for the k -edit approximate pattern matching problem that uses $\tilde{O}(k^2)$ bits of space and takes $\tilde{O}(k^2)$ time per arriving symbol of the text.*

We speculate that some amortization techniques could bring the time complexity of our k -edit approximate pattern matching algorithm further down. However, it seems unlikely to achieve complexity below $\tilde{O}(k)$ per arriving symbol as one could then solve the plain edit distance problem in sub-quadratic time contradicting the Strong Exponential Time Hypothesis (SETH) [44]. It is an interesting open question to achieve smaller space complexity than $\tilde{O}(k^2)$. Currently, all known sketching techniques for edit distance that people use for k -edit approximate pattern matching give sketches of size $\Omega(k^2)$.

The technique of Kociumaka, Porat and Starikovskaya [31] for edit distance pattern matching to large extent emulates the inner working of Hamming approximate pattern matching algorithms. To that effect Kociumaka, Porat and Starikovskaya had to design a *rolling* sketch for edit distance where multiple sketches can be “homomorphically” combined into one. This requires sophisticated machinery. Here we use a somewhat different approach. **We use the locally consistent decomposition of strings which preserves edit distance of Bhattacharya and Koucký [10]**, which was presented in the previous chapter. The decomposition in essence translates edit distance to Hamming distance. Hence, we apply the k -mismatch approximate pattern matching algorithm of Clifford, Kociumaka and Porat [23] on the stream of symbols coming from the decomposition as a black box. Bhattacharya and Koucký [10] also constructed a rolling sketch with limited update abilities, namely adding and deleting a symbol. We do not use that sketch here.

3.2.1 Related work

Landau and Vishkin [45] gave the first algorithm for the k -mismatch approximate pattern matching problem which runs in time $\mathcal{O}(k(m \log m + n))$ and takes $\mathcal{O}(k(m + n))$ amount of space. This was then improved to $\mathcal{O}(m \log m + kn)$ time and $\mathcal{O}(m)$ space by Galil and Giancarlo [46]. Later, Amir, Lewenstein and Porat [47] proposed two algorithms running in time $\mathcal{O}(n\sqrt{k \log k})$ and $\tilde{\mathcal{O}}(n + k^3(n/m))$. The latter was improved by Clifford, Fontaine, Porat, Sach and Starikovskaya [35] who gave an $\tilde{\mathcal{O}}(n + k^2(n/m))$ time algorithm. Charalampopoulos, Kociumaka and Wellnitz, in their FOCS'20 paper [48], also proposed an $\tilde{\mathcal{O}}(n + k^2(n/m))$ time algorithm with slightly better *polylog* factors. An $\tilde{\mathcal{O}}(n + kn/\sqrt{m})$ time algorithm was given by Gawrychowski and Uznański [49], which showed a nice tradeoff between the $\mathcal{O}(n\sqrt{k \log k})$ and $\tilde{\mathcal{O}}(n + k^2(n/m))$ running times. Not only that, they also showed that their algorithm is essentially optimal upto *polylog* factors, by proving a matching conditional lower bound. The *polylog* factors in the running time were then improved further by a randomized algorithm by Chan, Golan, Kociumaka, Kopelowitz and Porat [50], with running time $\mathcal{O}(n + kn(\sqrt{\log m/m}))$. This problem is thus quite well studied.

For the edit distance counterpart of the problem however, there is still a significant gap between the best upper bound and the known conditional lower bound. Landau and Vishkin [51] proposed an $\mathcal{O}(nk)$ time algorithm for the problem. This algorithm is still the state of the art for larger values of k . Cole and Hariharan [52] gave an algorithm running in time $\mathcal{O}(n + m + k^4(n/m))$ (this runs faster if $m \geq k^3$). In their unified approach paper [48], Charalampopoulos, Kociumaka and Wellnitz also proposed an algorithm running in time $\mathcal{O}(n + m + k^4(n/m))$. The same authors in their FOCS'22 paper [53] gave an algorithm running in time $\mathcal{O}(n + k^{3.5}\sqrt{\log m \log kn/m})$, finally improving the bound after 20 years. For the lower bound, Backurs and Indyk [44] proved that a truly subquadratic time algorithm for computing edit distance would falsify SETH. This would imply that an algorithm for the k -edit approximate pattern matching which is significantly faster than $\mathcal{O}(n + k^2(n/m))$ is highly unlikely.

Online k -mismatch approximate pattern matching problem was first solved by Benny Porat and Ely Porat in 2009 [34]. They gave an online algorithm with running time $\tilde{\mathcal{O}}(k^2)$ and space $\tilde{\mathcal{O}}(k^3)$ per arriving symbol of the text. Clifford, Fontaine, Porat, Sach and Starikovskaya in their SODA'16 paper [35], improved it to $\tilde{\mathcal{O}}(k^2)$ space and $\mathcal{O}(\sqrt{k} \log k + \text{poly}(\log(n)))$ time per arriving symbol of the text. Clifford, Kociumaka and Porat [23] proposed a randomized streaming algorithm which uses $\mathcal{O}(k \log(m/k))$ space and $\mathcal{O}(\log(m/k)(\sqrt{k \log k} + \log^3 m))$ time per arriving symbol. The space upper bound is optimal up-to logarithmic factors, matching the communication complexity lower bound. All these algorithms use some form of rolling sketch.

In the streaming model, Starikovskaya proposed a randomized algorithm [41] for the k -edit approximate pattern matching problem, which takes $\mathcal{O}(k^8 \sqrt{m} \log^6 m)$ space and $\mathcal{O}((k^2 \sqrt{m} + k^{13}) \log^4 m)$ time per arriving symbol. Kociumaka, Porat and Starikovskaya [31] proposed an improved randomized streaming algorithm, which takes $\tilde{\mathcal{O}}(k^5)$ space and $\tilde{\mathcal{O}}(k^8)$ amortized time per arriving symbol of the text.

3.2.2 Notations and preliminaries

We will use definitions and notations from the previous chapter. We will also use the following proposition, which can be obtained from Landau-Vishkin algorithm [45] see e.g. a combination of Lemma 6.2 and Theorem 7.13 in [48]:

Corollary 3.10. *For every pair of grammars G_x and G_y representing strings x and y , respectively, and given a parameter k we can find in time $\mathcal{O}((m + k^2) \cdot \text{poly}(\log(m + n)))$, where $n = |x| + |y|$ and $m = |G_x| + |G_y|$, the length of a suffix of x with the minimum edit distance to y among all the suffixes of x , provided that the edit distance of the suffix and y is at most k . If the edit distance of all the suffixes of x to y is more than k then the algorithm stops in the given time and reports that no suffix was found.*

Decomposition algorithm

Let us revisit the decomposition from previous chapter, and a specific theorem from previous section. Bhattacharya and Koucký [10] (from Chapter 2) give a string decomposition algorithm (*BK-decomposition algorithm*) that splits its input string into blocks, each block represented by a small grammar. With high probability over the choice of randomness of the algorithm, two strings of length at most n and edit distance at most k are decomposed so that the number of blocks is the same and at most k corresponding pairs of blocks differ. The edit distance between the two strings corresponds to the sum of edit distances of differing pairs of blocks.

More specifically, the BK-decomposition algorithm gets two parameters n and k , $k \leq n$, and an input x . It selects at random pair-wise independent functions C_1, \dots, C_L and S -wise independent functions H_0, \dots, H_L from certain hash families, and using those hash functions it decomposes x into blocks, and outputs a grammar for each of the block. We call the sequence of the produced grammars the *BK-decomposition of x* . Here, parameters $L = \lceil \log_{3/2} n \rceil + 3$ and $S = \mathcal{O}(k \log^3 n \log^* n)$. As shown in [10], the algorithm satisfies the following property.

Proposition 3.11 (Theorem 3.1 [10], Theorem 2.8 from Chapter 2). *Let x be a string of length at most n . The BK-decomposition algorithm outputs a sequence of grammars G_1, \dots, G_s such that for n large enough:*

1. *With probability at least $1 - 2/n$, $x = \text{eval}(G_1, \dots, G_s)$.*
2. *With probability at least $1 - 2/\sqrt{n}$, for all $i \in \{1, \dots, s\}$, $|G_i| \leq S$.*

The randomness of the algorithm is over the random choice of functions C_1, \dots, C_L and H_0, \dots, H_L .

The functions C_1, \dots, C_L can be described using $\mathcal{O}(\log^2 n)$ bits in total and the S -wise independent functions H_0, \dots, H_L can be described using $\mathcal{O}(S \log^2 n)$ bits in total. We also need the following special case of Theorem 3.12 [10].

Proposition 3.12 (Theorem 3.12 [10], Theorem 2.19 from Chapter 2). *Let $u, x, y \in \Gamma^*$ be strings such that $|ux|, |y| \leq n$ and $\Delta_{\text{edit}}(x, y) \leq k$. Let G_1^x, \dots, G_s^x*

and $G_1^y, \dots, G_{s'}^y$ be the sequence of grammars output by the BK-decomposition algorithm on input x and y respectively, using the same choice of random functions C_1, \dots, C_L and H_0, \dots, H_L . With probability at least $1 - 1/5$ the following is true: There exist an integer $r \geq 1$, such that

$$x = \text{eval}(G_{s-s'+1}^x)[r, \dots] \cdot \text{eval}(G_{s-s'+2}^x, \dots, G_s^x) \quad \& \quad y = \text{eval}(G_1^y, \dots, G_{s'}^y),$$

and

$$\begin{aligned} \Delta_{\text{edit}}(x, y) &= \Delta_{\text{edit}}(\text{eval}(G_{s-s'+1}^x)[r, \dots], \text{eval}(G_1^y)) \\ &\quad + \sum_{i=2}^{s'} \Delta_{\text{edit}}(\text{eval}(G_{s-s'+i}^x), \text{eval}(G_i^y)). \end{aligned}$$

The grammars for x can be built incrementally. For a fixed choice of functions C_i, H_i , and a string x we say that grammars G_1^x, \dots, G_t^x are *definite* in its BK-decomposition G_1^x, \dots, G_s^x if for any string z and the BK-decomposition $G_1^{xz}, \dots, G_{s'}^{xz}$ of xz obtained using the same functions C_i, H_i , $G_1^x = G_1^{xz}, \dots, G_t^x = G_t^{xz}$. It turns out that all, but $\tilde{O}(1)$ last grammars in the BK-decomposition of x are always definite. The following claim appears in [10]:

Proposition 3.13 (Lemma 4.2 [10], Lemma 3.7 from this chapter). *Let n and k be given and $R = \mathcal{O}(\log n \log^* n)$ be a suitably chosen parameter. Let $x, z \in \Gamma^*$, $|xz| \leq n$. Let $H_0, \dots, H_L, C_1, \dots, C_L$ be given. Let $G_1^x, G_2^x, \dots, G_s^x$ be the output of the BK-decomposition algorithm on input x , and $G_1^{xz}, G_2^{xz}, \dots, G_{s'}^{xz}$ be the output of the decomposition algorithm on input xz using the given hash functions.*

1. $G_i^x = G_i^{xz}$ for all $i = 1 \dots, s - R$.
2. $|x| \leq \sum_{i=1}^{\min(s+R, s')} |\text{eval}(G_i^{xz})|$.

The following claim bounds the resources needed to update BK-decomposition of x when we append a symbol a to it.

Proposition 3.14 (Theorem 5.1 [10], Theorem 3.8 from this chapter). *Let $k \leq n$ be given and $R = \mathcal{O}(\log n \log^* n)$ be a suitably chosen parameter. Let functions C_1, \dots, C_L and H_0, \dots, H_L be given. Let $a \in \Sigma$ and $x \in \Sigma^*$ be of length at most n , and let G_1^x, \dots, G_s^x be the grammars output by the BK-decomposition algorithm on input x using functions $C_1, \dots, C_L, H_0, \dots, H_L$.*

UpdateActiveGrammars($G_{s-\min(s, R+1)+1}^x, \dots, G_s^x, a$) outputs a sequence of grammars $G'_1, \dots, G'_{t'}$ such that $G_1^x, \dots, G_{s-\min(s, R+1)}^x, G'_1, \dots, G'_{t'}$ is the sequence that would be output by the BK-decomposition algorithm on $x \cdot a$ using the same functions $C_1, \dots, C_L, H_0, \dots, H_L$. The update algorithm runs in time $\tilde{O}(k)$ and outputs $t' \leq 4RL$ grammars.

3.2.3 Encoding a grammar

Let S and $M = \mathcal{O}(S \log n) = \mathcal{O}(k \log^4 n \log^* n)$ be parameters determined by the BK-decomposition algorithm. [10] shows that each grammar of size at most S can be encoded as a string of size M over some polynomial-size alphabet $\{1, \dots, 2\alpha\}$,

where the integer α can be chosen so that $2M/\alpha \leq 1/n$. The encoding Enc satisfies that if two grammars differ, their encodings differ in every coordinate. The encoding is randomized, and one needs $\mathcal{O}(\log n)$ random bits to select the encoding function. The encoding can be calculated in time linear in M , and given $\text{Enc}(G)$ we can decode G in time $\mathcal{O}(M)$. The encoding satisfies:

Proposition 3.15. *Let G, G' be two grammars of size at most S output by BK-decomposition algorithm. Let encoding Enc be chosen at random.*

1. $\text{Enc}(G) \in \{1, \dots, 2\alpha\}^M$.
2. If $G = G'$ then $\text{Enc}(G) = \text{Enc}(G')$.
3. If $G \neq G'$ then with probability at least $1 - (2M/\alpha)$, $\Delta_{\text{Ham}}(\text{Enc}(G), \text{Enc}(G')) = M$, that is they differ in every symbol.

3.2.4 k -mismatch approximate pattern matching

Clifford, Kociumaka and Porat [23] design a streaming algorithm for k -mismatch approximate pattern matching with the following properties. The algorithm first reads a pattern P symbol by symbol, and then it reads a text T symbol by symbol. Upon reading each symbol of the text it reports whether the word formed by the last received $|P|$ symbols of the text are within Hamming distance at most k from the pattern. If they are within Hamming distance at most k we can request the algorithm to report the mismatch information between the current suffix of the text and the pattern. The parameters k and n are given to the algorithm at the beginning, where n is an upper bound on the total length of the pattern and the text. By *mismatch information* between two strings x and y of the same length we understand $\text{MIS}(x, y) = \{(i, x[i], y[i]); i \in \{1, \dots, |x|\} \text{ and } x[i] \neq y[i]\}$. So the Hamming distance of x and y is $\Delta_{\text{Ham}}(x, y) = |\text{MIS}(x, y)|$. Clifford, Kociumaka and Porat [23] give the following main theorem.

Proposition 3.16 ([23]). *There exists a streaming k -mismatch approximate pattern matching algorithm which uses $\mathcal{O}(k \log n \log(n/k))$ bits of space and takes $\mathcal{O}((\sqrt{k \log k} + \log^3 n) \log(n/k))$ time per arriving symbol. The algorithm is randomized and its answers are correct with high probability, that is it errs with probability inverse polynomial in n . For each reported occurrence, the mismatch information can be reported on demand in $\mathcal{O}(k)$ time.*

3.2.5 Algorithm overview

Now we provide the high-level view of how we proceed. We will take the pattern P and apply on it the BK-decomposition algorithm. That will give us grammars $G_1^P, G_2^P, \dots, G_r^P$ encoding the pattern. This has to be done incrementally as the symbols of P arrive. Then we will incrementally apply the BK-decomposition algorithm on the text T .

We will not store all the grammars in memory, instead we will use the K -mismatch approximate pattern matching algorithm of Clifford, Kociumaka and Porat [23] (*CKP-match algorithm*) on the grammars. Here $K = k \cdot M$, where M is the encoding size of each grammar. For a suitable parameter $R = \tilde{\mathcal{O}}(1)$, we

will feed the grammars G_1^P, \dots, G_{r-R}^P to the CKP-match algorithm as a pattern. In particular, we will encode each grammar by the encoding function Enc from Section 3.2.3, and we will feed the encoding into the CKP-match algorithm symbol by symbol.

Then as the symbols of the text T will arrive, we will incrementally build the grammars for T while maintaining only a small set of *active* grammars. Grammars that become *definite* will be fed into the CKP-match algorithm as its input text. (Again each one of the grammars encoded by Enc.) The CKP-match algorithm will report K -mismatch occurrences of our pattern in the text. Each K -mismatch occurrence corresponds to a match of the pattern grammars to the text grammars, with up-to k differing pairs of grammars. We will recover the differing pairs of grammars and calculate their overall edit distance. We will combine this edit distance with the edit distance of the last R grammars of the pattern from the last R grammars of the text. (The last R grammars of the text contain the active grammars which were not fed into the CKP-match algorithm, yet.) If the total edit distance of the match does not exceed the threshold k , we report it as an k -edit occurrence of P in T . If required we can also output the edit operations that transform the pattern into a suffix of T . (Among the current suffixes of T we pick the one which gives the smallest edit distance from P .)

The success probability of our scheme in reporting a particular occurrence of P in T is some constant $\geq 1/2$. Thus, we run the processes in parallel $\mathcal{O}(\log n)$ times with independently chosen randomness to achieve small error-probability.

We describe our algorithm in more details next.

3.2.6 Description of the algorithm

Now we describe one run of our algorithm. The algorithm receives parameters n and k , based on them it sets parameters $L = \mathcal{O}(\log n)$, $R = \mathcal{O}(\log n \log^* n)$, $S = \mathcal{O}(k \log^3 n \log^* n)$, $M = \mathcal{O}(k \log^4 n \log^* n)$, $K = k \cdot M = \mathcal{O}(k^2 \log^4 n \log^* n)$. Then it chooses at random pair-wise independent functions C_1, \dots, C_L and S -wise independent functions H_0, \dots, H_L needed by the BK-decomposition algorithm. It also selects the required randomness for the encoding function Enc. It initializes the CKP-match algorithm for K -mismatch approximate pattern matching on strings of length at most $n \cdot M$.

There are two phases of the algorithm. In the first phase the algorithm receives a pattern P symbol by symbol and incrementally builds a sequence of grammars G_1^P, \dots, G_r^P representing the pattern P . All but the last R grammars are encoded using Enc and sent to our instance of CKP-match algorithm as its pattern (symbol by symbol of each encoding). In the second phase our algorithm receives an input text T symbol by symbol. It will incrementally build a sequences of grammars G_1^T, G_2^T, \dots representing the received text. Whenever one of the grammars becomes *definite* it is encoded by Enc and sent to our instance of CKP-match algorithm as the next part of its input text (symbol by symbol).

In the first phase, our algorithm uses the procedure given by Proposition 3.14 to construct the grammars G_1^P, \dots, G_r^P incrementally by adding symbols of P . The algorithm maintains a buffer of $2R$ *active* grammars which are updated by the addition of each symbol. Whenever the number of active grammars exceeds $2R$ we encode the *oldest* (left-most) grammars that are definite and pass them to our

instance of CKP-match algorithm as the continuation of its pattern. The precise details of updating the grammars of the pattern are similar to that of updating them for text which we will elaborate on more. After the input pattern ends, we keep only R grammars $G_{r-R+1}^P, \dots, G_r^P$, and we send all the other grammars to the CKP-match algorithm. Then we announce to the CKP-match algorithm the end of its input pattern. So the CKP-match algorithm received as its pattern encoding of grammars G_1^P, \dots, G_{r-R}^P in this order. (In the case we end up with fewer than $R + 1$ grammars representing P ($r \leq R$), we apply a *naïve* pattern matching algorithm without need for the CKP-match algorithm. We leave this simple case as an exercise to the reader.) For the rest of this description we assume that $r > R$.

In the second phase, the algorithm will receive the input text T symbol by symbol. It will incrementally build a sequence of grammars representing the text using the algorithm from Proposition 3.14. We will keep at most R *active* grammars G_1^a, \dots, G_t^a on which the algorithm from Proposition 3.14 will be applied. The active grammars represent a current suffix of T . The prefix of T up-to that suffix is represented by grammars G_1^T, \dots, G_s^T which are definite. Out of those definite grammars we will explicitly store only the last R in a buffer, the other grammars will not be stored explicitly. (They will be used to calculate the current edit distance and to run the update algorithm from Proposition 3.14.) The encoding of all the definite grammars will be fed into the CKP-match algorithm as its input text whenever we detect that a grammar is definite.

As the algorithm proceeds over the text it calculates a sequence of integers m_1, m_2, \dots, m_s , where the algorithm stores only the last R of them in a buffer. Each value m_i is the minimal edit distance of $\text{eval}(G_1^P, \dots, G_{r-R}^P)$ (a prefix of the pattern) to any suffix of $\text{eval}(G_1^T, \dots, G_i^T)$ (a suffix of a prefix of the text) if the edit distance is less than k . m_i is considered infinite otherwise. (Values m_1, \dots, m_{r-R-1} are all considered to be infinite.) The value m_i will be calculated after G_i^T becomes definite and we send the grammar to our CKP-match algorithm. (The CKP-match algorithm will facilitate its calculation.) Values m_i will be used to calculate the edit distance of the current suffix of the input text received by the algorithm. See Fig. 3.2 for an illustration.

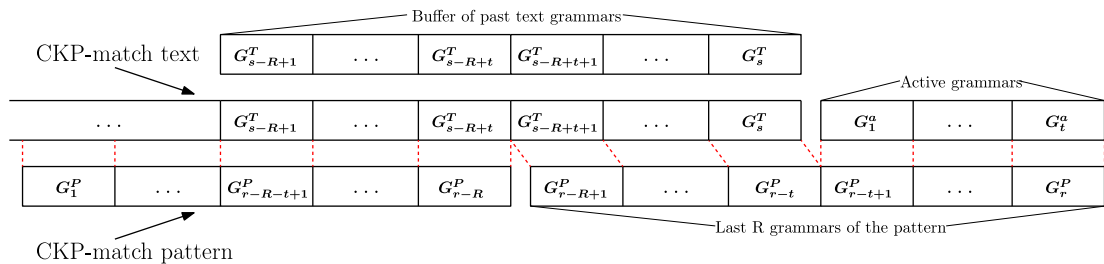


Figure 3.2 The alignment of text and pattern grammars after arrival of some text symbol. The pattern P is represented by grammars G_1^P, \dots, G_r^P . Grammars G_1^P, \dots, G_{r-R}^P are encoded by Enc and sent to the CKP-match algorithm as its pattern. The current text T is represented by the sequence of grammars $G_1^T, \dots, G_s^T, G_1^a, \dots, G_t^a$. Grammars G_1^T, \dots, G_s^T are encoded and committed to the CKP-match algorithm as its text. Grammars G_1^a, \dots, G_t^a are active grammars of the text, and might change as more symbols are added to the text.

We are ready to describe the basic procedures performed by the algorithm.

Symbol arrival. Upon receiving the next symbol a of the input text, our algorithm invokes the algorithm from Proposition 3.14 on the $R + 1$ grammars $G_{s-R+t}^T, \dots, G_s^T, G_1^a, \dots, G_t^a$ to append the symbol a . From the algorithm we receive back grammars $G_{s-R+t}^T, \dots, G_s^T, G_1^{a'}, \dots, G_{t'}^{a'}$, where $t' < 4RL$. (Here, $\text{eval}(G_1^{a'}, \dots, G_{t'}^{a'}) = \text{eval}(G_1^a, \dots, G_t^a) \cdot a$. The grammars $G_{s-R+t}^T, \dots, G_s^T$ received from the algorithm are discarded as they are definite and should not change. The update algorithm needs them to have the proper context for compression.) If $t' > R$ then grammars $G_1^{a'}, \dots, G_{t'-R}^{a'}$ become definite and we will *commit* each of them to the CKP-match algorithm as explained further. We will commit them in order $G_1^{a'}, \dots, G_{t'-R}^{a'}$. The remaining grammars $G_{t'-R+1}^{a'}, \dots, G_{t'}^{a'}$ are relabelled as G_1^a, \dots, G_t^a and become the active grammars for the addition of the next symbol.

At this point our algorithm can output the minimal possible edit distance of the pattern to any suffix of the text received up-to this point. We explain below how such query is calculated.

Committing a grammar. When a grammar G becomes definite the algorithm commits the grammar as follows. Thus far, grammars G_1, \dots, G_s were committed and the sequence of values m_1, \dots, m_s was calculated. We set $G_{s+1} = G$, calculate encoding $\text{Enc}(G_{s+1})$ and send the encoding symbol by symbol to our CKP-match algorithm. At this point we can calculate m_{s+1} using the mismatch information provided by our CKP-match algorithm. If $s + 1 < r - R$ then we set m_{s+1} to ∞ otherwise we continue as follows to calculate m_{s+1} .

We query our CKP-match algorithm for the Hamming distance between encoding of G_1^P, \dots, G_{r-R}^P (the pattern to the CKP-match algorithm) and the encoding of $G_{s-r+R+2}^T, G_{s-r+R+3}^T, \dots, G_{s+1}^T$ (the current suffix of the text of the CKP-match algorithm). If the Hamming distance is less than $K = k \cdot M$, then we let the CKP-match algorithm to recover the mismatch information. By the design of the encoding function, if two grammars differ then their encodings differ in all M positions (unless the encoding function Enc fails which happens only with negligible probability.) Hence, the mismatch information consists of encoding of up-to k pairs of grammars, with their indexes relative to the pattern. Thus, from the mismatch information we recover pairs of grammars $(G_1, G'_1), \dots, (G_{k'}, G'_{k'})$, for some $k' \leq k$ where G_i come from the text and G'_i come from the pattern.

If (G_1, G'_1) is not the very first grammar pair $(G_{s-r+R+2}^T, G_1^P)$ (which we recognize by their index in the mismatch information) then we compute the edit distance for each pair of strings $\text{eval}(G_i)$ and $\text{eval}(G'_i)$, $i = 1, \dots, k'$. We set m_{s+1} to be the sum of those distances.

If (G_1, G'_1) is the pair (G_{s-R+2}^T, G_1^P) then we apply the algorithm from Corollary 3.10 to calculate the minimal edit distance between any suffix of $\text{eval}(G_1)$ and the string $\text{eval}(G'_1)$. For $i = 2, \dots, k'$, we compute the edit distance of $\text{eval}(G_i)$ and $\text{eval}(G'_i)$. We set m_{s+1} to be the sum of the k' calculated values.

However, if the CKP-match algorithm declares that the Hamming distance of its pattern to its current suffix is more than K , we set $m_{s+1} = \infty$.

Finally, we discard G_{s-r+R} from the buffer of the last R committed grammars, and we discard m_{s-R+2} from the buffer of values m_i . We set s to be $s + 1$. This finishes the process of committing a single grammar G , and a next grammar might be committed.

Pattern edit distance query. After we process the arrival of a new symbol, update the active grammars as described above and commit grammars as necessary, the

algorithm is ready to answer the edit distance query on the current suffix of the text T and the pattern P . At this point grammars G_1^T, \dots, G_s^T were already committed to the CKP-match algorithm. There are current active grammars G_1^a, \dots, G_t^a which were not committed to the CKP-match algorithm, and there are R grammars $G_{r-R+1}^P, \dots, G_r^P$ of the input pattern that were not committed to the CKP-match algorithm as part of its pattern. To answer the edit distance query we will compare the edit distance of those last R grammars of pattern P with the last grammars of the text, and we will combine this with a certain value m_i , namely m_{s-R+t} .

Let $d = R - t$. If $d > 0$, for $i = 1, \dots, d$ compute the edit distance of each pair $\text{eval}(G_{s-d+i}^T)$ and $\text{eval}(G_{r-R+i}^P)$. (Each grammar G_{s-d+i}^T is available in the buffer of the last R committed grammars.) For $i = d + 1, \dots, R$, compute the edit distance of each pair $\text{eval}(G_{i-d}^a)$ and $\text{eval}(G_{r-R+i}^P)$. Sum those R values together with m_{s-d} . If the sum is less than k output it, otherwise output ∞ .

Since we are running $\mathcal{O}(\log n)$ independent copies of our algorithm, each of the copies produces an estimate on the edit distance and we output the smallest estimate. That is the correct value with high probability.

3.2.7 Correctness of the algorithm

In this section we argue that the algorithm produces a correct output. First we analyze the probability of certain bad events happening when the algorithm fails and then we argue the correctness of the output assuming none of the bad events happens. There are several sources of failure in our algorithm.

1. The BK-decomposition algorithm might produce a decomposition of either the pattern or some suffix of the text with a grammar that is too big or with grammars that do not represent expected strings. (A failure of Proposition 3.11.)
2. The BK-decomposition algorithm produces a correct decomposition of the pattern and all suffixes of the text but grammars of some suffix of the text T and the pattern P do not align well. (A failure of Proposition 3.12.)
3. The encoding function Enc fails for some pair of grammars produced by the BK-decomposition algorithm that the CKP-match algorithm is supposed to compare. (A failure of Proposition 3.15.)
4. BK-decomposition algorithm does not fail but the CKP-match algorithm fails to identify a K -mismatch occurrence of its pattern or fails to produce correct mismatch information. (A failure of Proposition 3.16.)

The failure probability of events 1), 3) and 4) will be each bounded by inverse polynomial in n , where n is the parameter sent to those algorithms as an upper bound on the length of the processed strings. Thus, if we expect our algorithm to process a text and a pattern of size at most N , we can set the parameter n for the BK-decomposition algorithm to be N^4 and for the CKP-algorithm to be $N^4 \cdot M = \tilde{\mathcal{O}}(N^5)$, where M is calculated from $n = N^4$ and k of the BK-decomposition algorithm. (Parameter k for the BK-decomposition algorithm is set to k , and for the CKP-algorithm to $K = k \cdot M = \tilde{\mathcal{O}}(k^2)$.) We will run $2 \log N$

independent copies of our algorithm on the same text and pattern. Next we calculate the probability of failure in case 1), 3) and 4) in a particular copy of the algorithm.

Event 1. There is one pattern P of length at most N , the probability of either of the two conditions in Proposition 3.11 failing on P is at most $4/\sqrt{n} = 4/N^2$. The probability of failure of Proposition 3.11 on any the at most N prefixes of the text T is at most $N \cdot 4/\sqrt{n} = 4/N$. Thus the probability of the bad event 1) happening is at most $4/N + 4/N^2$.

Event 3. There are at most N grammars of the pattern encoded by Enc and there are at most N grammars of the text encoded by Enc and committed. Thus there are at most N^2 pairs of grammars on which Proposition 3.15 could fail by encoding two distinct grammars by strings of Hamming distance less than M (failure in the third part of Proposition 3.15). Given our setting of parameters, the probability of the bad event 3) happening is at most $N^2/n = 1/N^2$.

Event 4. The probability that the CKP-match algorithm fails during its execution is at most $1/n = 1/N^4$.

Thus, the probability of a failure of 1), 3) or 4) is at most $5/N$, for N large enough. We run $2 \log N$ copies of the algorithm so the probability that any of the copies fails because of events 1), 3), or 4) is at most $10 \log N/N$.

If none of the events 1), 3) and 4) occurs during the execution of the algorithm then the pattern and the text are correctly decomposed into grammars by the BK-decomposition, the grammars are properly encoded by Enc, and the CKP-match algorithm correctly identifies all the occurrences of the pattern grammars in the committed text grammars, and for each of the occurrences we correctly recover the differing pairs of pattern and text grammars. Assuming this happens, we want to argue that with a high probability our algorithm will correctly identify k -edit occurrences of the pattern P in the text T .

After receiving a prefix of the text $T[1, \ell]$, $\ell \leq N$, we want to determine whether some suffix of $T[1, \ell]$ has edit distance at most k from the pattern P . Let a be such that $T[a, \ell]$ has the minimal distance from P . Clearly, if the edit distance between $T[a, \ell]$ and P is at most k then $a \in \{\ell - |P| - k + 1, \dots, \ell - |P| + k + 1\}$. By Proposition 3.12 applied on $u = T[1, a - 1]$, $x = T[a, \ell]$ and $y = P$, each of the $2 \log N$ copies of our algorithm has probability at least $4/5$ that the grammars of T are well aligned with grammars of P . Being well aligned means that $T[a, \ell]$ is a suffix of $\text{eval}(G_{s-r+t+1}^T, \dots, G_s^T, G_1^a, \dots, G_t^a)$ and

$$\begin{aligned} \Delta_{edit}(T[a, \ell], P) &= \Delta_{edit}(\text{eval}(G_{s-r+t+1}^T)[b, \dots], \text{eval}(G_1^P)) \\ &+ \sum_{i=2}^{r-t} \Delta_{edit}(\text{eval}(G_{s-r+t+i}^T), \text{eval}(G_i^P)) \\ &+ \sum_{i=r-t+1}^r \Delta_{edit}(\text{eval}(G_{i-r+t}^a), \text{eval}(G_i^P)), \end{aligned}$$

for appropriate b . Moreover, the minimality of a implies that

$$\begin{aligned}\Delta_{edit}(T[a, \ell], P) &= \min_b \Delta_{edit}(\text{eval}(G_{s-r+t+1}^T)[b, \dots], \text{eval}(G_1^P)) \\ &+ \sum_{i=2}^{r-t} \Delta_{edit}(\text{eval}(G_{s-r+t+i}^T), \text{eval}(G_i^P)) \\ &+ \sum_{i=r-t+1}^r \Delta_{edit}(\text{eval}(G_{i-r+t}^a), \text{eval}(G_i^P)).\end{aligned}$$

Notice, regardless of whether Proposition 3.12 fails or not, the right-hand-side of the last equation is always at least $\Delta_{edit}(T[a, \ell], P)$ since it is an upper-bound on the true edit distance of P to some suffix of T . We will argue that each copy of the algorithm outputs the right-hand-side value of that equation if it has value at most k , and ∞ otherwise. Moreover, if at least one of the copies of our algorithm has $T[a, \ell]$ and P well aligned, then the minimum among the values output by the different copies of our algorithm is $\Delta_{edit}(T[a, \ell], P)$.

Since we have $2 \log N$ copies of the algorithm, the probability that none of the decompositions aligns $T[a, \ell]$ and P well is at most $(1/5)^{2 \log N} < 1/N^4$. This upper-bounds the probability of error of outputting a wrong value of $\min_b \Delta_{edit}(T[b, \ell], P)$ after receiving ℓ symbols of the text. As there will be at most N distinct values of ℓ , the probability of outputting a wrong estimate of the edit distance of P to some suffix of T is at most $N \cdot 1/N^4 = 1/N^3$, conditioned on none of the bad events 1), 3) or 4) happening. Overall, the probability of a failure of our algorithm is at most $\mathcal{O}(\log N/N) \leq 1/\sqrt{N}$, for N large enough, and it could be made an arbitrary small polynomial in N by choosing the parameters differently (n vs N).

It remains to argue that the copy of our algorithm which aligns $T[a, \ell]$ and P well, outputs their edit distance. Consider the copy of the algorithm that aligns grammars of $T[a, \ell]$ and P well. After arrival of the symbol $T[\ell]$ and updating the grammars, there are active grammars G_1^a, \dots, G_t^a , committed grammars G_1^T, \dots, G_s^T and the pattern grammars G_1^P, \dots, G_r^P . If $\Delta_{edit}(T[a, \ell], P)$ is at most k then the number of grammars in which P differs from the last r grammars of T is at most k . Thus the CKP-match algorithm can identify the differing grammars when computing the value m_{s-R+t} which is set to

$$\begin{aligned}m_{s-R+t} &= \min_b \Delta_{edit}(\text{eval}(G_{s-r+t+1}^T)[b, \dots], \text{eval}(G_1^P)) \\ &+ \sum_{i=2}^{r-R} \Delta_{edit}(\text{eval}(G_{s-r+t+i}^T), \text{eval}(G_i^P)).\end{aligned}$$

Since, $m_{s-R+t} \leq \Delta_{edit}(T[a, \ell], P) \leq k$, we have the true value of m_{s-R+t} . Thus,

$$\begin{aligned}\Delta_{edit}(T[a, \ell], P) &= m_{s-R+t} \\ &+ \sum_{i=r-R+1}^{r-t} \Delta_{edit}(\text{eval}(G_{s-r+t+i}^T), \text{eval}(G_i^P)) \\ &+ \sum_{i=r-t+1}^r \Delta_{edit}(\text{eval}(G_{i-r+t}^a), \text{eval}(G_i^P)).\end{aligned}$$

That is precisely how we evaluate the edit distance query of our algorithm.

If $\Delta_{edit}(T[a, \ell], P) > k$ then we will output a value $> k$ as we output some upper bound on the edit distance. Any value $> k$ is treated as the infinity.

3.2.8 Time complexity of the algorithm

In the first phase, we incrementally construct the grammars for the pattern P , using the BK-decomposition algorithm from Proposition 3.14 on each symbol of P at a time. Updating the active grammars for each new symbol takes $\tilde{O}(k)$ time, committing each of the possible $\tilde{O}(1)$ definite grammars to the CKP-match algorithm takes $\tilde{O}(M \cdot \sqrt{K}) = \tilde{O}(k^2)$. Thus the time needed per arriving symbol of the pattern is $\tilde{O}(k^2)$.

For each symbol of the text that arrives during the second phase of the algorithm we need to update the active grammars of the text, update m_s , and evaluate the edit distance of the pattern from the current suffix of text. This includes parts *Symbol arrival*, *Committing a grammar* and *Pattern edit distance query* of the algorithm.

Symbol arrival. Appending a symbol using the BK-decomposition algorithm from Proposition 3.14 takes $\tilde{O}(k)$ time.

Committing a grammar. Encoding the grammar takes $\mathcal{O}(M)$ time using the algorithm from Proposition 3.15, and committing it to the CKP-match algorithm takes time $\tilde{O}(k^2)$, as in the pattern case.

Querying the CKP-match algorithm for Hamming distance K takes $\mathcal{O}(K) = \tilde{O}(k^2)$ time. This recovers at most k pairs of distinct grammars (G_i, G'_i) , $1 \leq i \leq k$. Computing edit distance k_i of each pair of strings $\text{eval}(G_i)$ and $\text{eval}(G'_i)$, takes $\tilde{O}(S + k_i^2) = \tilde{O}(k + k_i^2)$ time using Proposition 2.6. If $\sum_i k_i \leq k$, the total time for the edit distance computation is bounded by $\tilde{O}(k^2)$. If the computation runs for longer we can stop it as we know m_s is larger than k . Running the algorithm from Corollary 3.10 on the first pair of distinct grammars to compute the minimum edit distance between any suffix of $\text{eval}(G_1)$ and the string $\text{eval}(G'_1)$ takes $\tilde{O}(S + k^2)$ time. Thus committing a grammar takes time at most $\tilde{O}(k^2)$ where the longest time takes the minimization algorithm on the first pair of grammars.

Pattern edit distance query. This step requires the alignment of the last R grammars of the pattern with the appropriate grammars of the text and computing their edit distances. Using Proposition 2.6, computing edit distances of R pairs of grammars takes $R \times \tilde{O}(k^2) = \tilde{O}(k^2)$ time.

As there are at most $\tilde{O}(1)$ committed grammars after processing each new symbol, the total time of this step is $\tilde{O}(k^2)$ per arriving symbol.

3.2.9 Space complexity of the algorithm

During either phase of the algorithm, we store $\mathcal{O}(RL) = \tilde{O}(1)$ active and updated grammars and buffer last $\mathcal{O}(R)$ committed grammars. This requires space $\tilde{O}(k)$. Furthermore, the CKP-match algorithm requires $\tilde{O}(K) = \tilde{O}(k^2)$ space. The edit distance algorithm of Proposition 2.6 cannot use more space than its running time so each invocation uses at most $\tilde{O}(k^2)$ space. Similarly, Corollary 3.10 uses space $\tilde{O}(k^2)$. Thus our algorithm uses space at most $\tilde{O}(k^2)$ at any point during its computation.

3.3 Summary

In this chapter, we presented algorithmic applications of the decomposable embedding from edit distance to Hamming distance introduced in Chapter 2. Focusing specifically on edit distance sketches and rolling sketches, we showed how this embedding facilitates efficient and succinct solutions to computationally challenging problems in sublinear and streaming models.

Firstly, we described the construction of an edit distance sketch that achieves a sketch size of $\tilde{O}(k^2)$ bits. This approach leverages a novel oblivious alignment technique enabled by our decomposable embedding, significantly simplifying the sketch construction compared to previous methods reliant on complex random walks. This results in a randomized sketching algorithm capable of producing short sketches in near-linear time, and a comparison algorithm that computes the exact edit distance between sketched strings in time $\tilde{O}(k^2)$ with high probability.

Next, we introduced a rolling sketch data structure, enabling efficient updates to the sketch when a string is modified by appending or removing symbols at its ends. Specifically, we demonstrated update procedures that handle insertions and deletions in $\tilde{O}(k^2)$ time per operation. We outlined how to maintain the structural properties of decompositions through updates, ensuring that edit distances between dynamically changing strings can still be computed rapidly and accurately from the sketches.

Both sketch constructions make crucial use of existing optimal Hamming distance sketches combined with our decomposable embedding technique. Detailed analyses provided in this chapter confirm the efficacy and robustness of these approaches, highlighting their potential for further optimization and broader applicability in streaming algorithms and approximate pattern matching.

In second part of this chapter, we also introduced a randomized algorithm for the streaming k -edit approximate pattern matching problem, significantly enhancing the state-of-the-art results. Specifically, our algorithm achieves a substantial reduction in both time and space complexity, requiring $\tilde{O}(k^2)$ space and $\tilde{O}(k^2)$ processing time per symbol, improving notably upon the previous work by Kociumaka, Porat, and Starikovskaya which used $\tilde{O}(k^5)$ space and $\tilde{O}(k^8)$ time.

Our approach leverages the locally consistent string decomposition introduced in Chapter 2, effectively translating the edit distance problem into a Hamming distance scenario. By decomposing the pattern and text into carefully constructed grammars, we reduce the original edit distance problem into manageable subproblems that can be efficiently solved using Hamming distance-based methods.

We incorporated a streaming algorithm for k -mismatch approximate pattern matching by Clifford, Kociumaka, and Porat as a key subroutine. This enabled efficient identification of approximate pattern occurrences within the encoded decompositions. Furthermore, we employed randomized encoding techniques to guarantee the distinctiveness of grammar representations, thereby ensuring reliable detection and alignment of corresponding substrings.

To achieve robustness against random failures, our method involves running multiple independent instances of the algorithm concurrently and selecting the minimum result among them, thereby guaranteeing high probability correctness of the output.

Additionally, we provided detailed correctness proofs and complexity analyses,

confirming that the algorithm efficiently identifies occurrences of the pattern within a specified edit distance threshold while adhering to tight complexity bounds.

Overall, this chapter demonstrates that decomposable embeddings from edit distance to Hamming space serve as a powerful primitive for algorithm design, enabling fast and space-efficient solutions to core problems in sublinear models and approximate string processing.

4

Hamming to Edit Embedding

*“Working hard is important. But there is something that matters even more:
believing in yourself.”
- Harry Potter*

4.1 Introduction

The study of metric embeddings lies at the heart of understanding the intrinsic geometry of computational problems. By mapping one metric space into another while approximately or exactly preserving distances, embeddings can reveal deep connections between otherwise disparate problems. In this chapter, we focus on embeddings between two of the most fundamental string similarity measures: the *Hamming distance* and the *edit distance*.

The **Hamming distance** counts the number of mismatched symbols between two strings of equal length, and is a cornerstone in coding theory and complexity. In contrast, the **edit distance** measures the minimum number of insertions, deletions, and substitutions required to transform one string into another, and plays a central role in sequence alignment, pattern matching, and bioinformatics.

Despite their similarities, these two metrics differ significantly in structure and complexity. This raises a natural and fundamental question:

Can the Hamming space be isometrically embedded into the edit space?

More concretely, does there exist a mapping $E : \{0, 1\}^n \rightarrow \{0, 1\}^*$ such that for all $x, y \in \{0, 1\}^n$, the equality

$$\Delta_{edit}(E(x), E(y)) = \Delta_{Ham}(x, y)$$

holds? And if so, what is the best possible *rate*, that is, the ratio of the input length to the output length, achievable by such an embedding?

This question is not purely of theoretical interest. Embedding one metric into another enables the transfer of algorithmic and hardness results. For instance, if a problem is known to be hard under the Hamming metric, and there exists a distance-preserving embedding into the edit metric, the same hardness result carries over. Prior constructions based on random interleaving achieved isometric

embeddings with a rate of $1/\Theta(\log n)$, allowing such transfers but only with polynomial blow-up in dimension.

In this chapter, we take a comprehensive view of this embedding problem. We begin by revisiting known constructions and progressively build more efficient embeddings, culminating in a **constant-rate isometric embedding** - a striking result enabled by recent advances in synchronization strings [54]. We also investigate the theoretical limits of this approach: What constraints must any such embedding satisfy? How efficient can an isometric embedding be, even in principle?

Our exploration leads to both new positive results - explicit constructions with progressively better rates and new impossibility results - provable upper bounds on how efficient such embeddings can be. Together, these contributions shed light on the fundamental relationship between substitution-only and general edit-based string similarity measures.

4.1.1 Summary of Results

1. **Low-Rate Constructions:** We begin with simple embeddings that achieve isometry by padding each bit with a distinct marker, obtaining a rate of $1/\Theta(\log n)$. We then introduce recursive and multi-level recursive embeddings that improve the rate to $1/\Theta(\log^* n \cdot \log \log^* n)$.
2. **Constant-Rate Embedding (Main Result):** We prove the existence of a constant-rate isometric embedding from the Hamming metric to the edit metric. This construction uses *synchronization strings* to ensure that each input bit is correctly aligned, even in the presence of insertions and deletions.
3. **Hardness Transfers:** Using our constant-rate embedding, we show that hardness results from the Hamming metric transfer directly to the edit metric without dimension blow-up. We obtain fine-grained lower bounds for problems such as bichromatic closest pair, discrete 1-center, and k -clustering, as well as NP-hardness of approximation results.
4. **Impossibility Results:** We prove structural constraints on isometric embeddings. Specifically, any such embedding must be of a *projective type*—padding the input with fixed strings around permuted and XOR-ed bits. Furthermore, we show that no isometric embedding can achieve a rate exceeding $3/7 + o(1)$, even in principle.

This chapter thus offers both constructive techniques and theoretical insights into the geometry of string metrics, positioning the Hamming-to-edit embedding problem as a central question in the algorithmic study of metric spaces.

4.2 Motivation and Implications

In computational complexity, establishing hardness results is often more tractable in the Hamming metric than in the edit metric. Consequently, the existence of an *isometric embedding* from the Hamming metric into the edit metric would allow

such hardness results to be transferred directly. Indeed, embeddings of this nature have been studied precisely for this purpose.

However, the currently known embeddings achieve a rate of only $\frac{1}{\Theta(\log n)}$. This implies that if a problem is hard over strings of length n in the Hamming metric, the corresponding hardness in the edit metric holds only for strings of length $\Theta(n \log n)$. While this is still meaningful, it incurs an undesirable overhead in dimension.

Intuitively, one might expect that solving computational problems in the n -dimensional edit metric should be at least as hard as solving them in the n -dimensional Hamming metric. Yet, without a constant-rate isometric embedding, this intuition lacks formal justification. Addressing this gap is one of the main motivations of this chapter.

Our key contribution is the construction of a constant-rate isometric embedding from the Hamming metric to the edit metric, enabled by a novel application of *synchronization strings*: combinatorial structures originally developed for correcting insertion and deletion errors [55, 56, 57, 58, 59].

Theorem 4.1. *There exists a universal constant $C \geq 1$ such that for every positive integer n , there exists an isometric embedding $E_n : \{0, 1\}^n \rightarrow \{0, 1\}^{Cn}$ from the Hamming metric into the edit metric.*

The construction and analysis of this embedding, which crucially relies on synchronization strings, is presented in detail in Section 4.6.

An immediate consequence of Theorem 4.1 is the following meta-theorem for a broad class of *discrete* optimization problems in the edit metric. Here, by discrete we refer to problems where the input is a finite set of points in a metric space and the solution is a subset of those points.

If a discrete optimization problem defined over the n -dimensional Hamming metric cannot be solved in time $T(n)$ (for some computable function T), then the same problem over the n -dimensional edit metric cannot be solved in time $T(\Theta(n))$.

To illustrate this transfer principle, we establish tight hardness results in two important complexity settings. In all the results below: Theorems 4.2, 4.3, and 4.4, prior lower bounds in the edit metric over binary strings were known only in dimension $d = O(\log n \cdot \log \log n)$, due to the limitations of earlier embeddings with rate $\Theta\left(\frac{1}{\log d}\right)$. Our embedding allows the dimension to be reduced to $d = O(\log n)$, thereby achieving *optimal* dimensional dependence.

Fine-Grained Complexity. Using Theorem 4.1, we derive conditional lower bounds for two classical problems in the edit metric: the closest pair problem and the discrete 1-center problem.

Theorem 4.2 (BCP). *Assuming the Strong Exponential Time Hypothesis, for every $\delta > 0$ there exists $\varepsilon > 0$ such that given two sets $A, B \subseteq \{0, 1\}^d$ of N binary strings (with $d = O_\delta(\log N)$), any algorithm that computes a $(1 + \varepsilon)$ -approximate closest pair in $A \times B$ under the edit metric requires time $\Omega(N^{2-\delta})$.*

This follows immediately by composing Theorem 4.1 with the corresponding Hamming metric result from [60]. Similarly, by starting from the reduction in [61], one obtains a conditional lower bound of $n^{1.5-o(1)}$ for the *monochromatic* closest pair problem in the edit metric, again with optimal dimension dependence.

We next state the analogous result for the discrete 1-center problem.

Theorem 4.3 (1-center). *Assuming the Hitting Set Conjecture, for every $\varepsilon > 0$ there exists a constant $c > 1$ such that given a set $P \subseteq \{0, 1\}^d$ of N points (with $d = c \log N$), computing the point $x \in P$ that minimizes the maximum edit distance to all other points in P requires time $\Omega(N^{2-\varepsilon})$.*

This result is obtained by applying our embedding to the Hamming-based hardness result in [62].

NP Hardness. We also use Theorem 4.1 to lift known NP-hardness of approximation results from the Hamming metric to the edit metric. Specifically, we obtain the following inapproximability bounds in low-dimensional edit space:

Theorem 4.4. *It is NP-hard to approximate:*

1. *The discrete k -means problem (resp. k -center and k -median) on N points in the $\{0, 1\}^{O(\log N)}$ -dimensional edit metric to within a factor better than 1.38 (resp. $3 - o(1)$ and 1.12);*
2. *The discrete Steiner tree problem on N terminals in the $\{0, 1\}^{O(\log N)}$ -dimensional edit metric to within a factor better than 1.004.*

These results follow from known inapproximability bounds in the Hamming metric due to [63, 64], combined with our constant-rate embedding.

Finally, we note that additional results from [65] could also benefit from the constant-rate embedding framework developed in this chapter.¹

- This chapter presents some of the results from our paper [66], and there are a few other results which are not there in the paper. It also contains an improved new result which is not in the paper.

4.3 Easy Embedding with Rate $1/\Theta(\log n)$

Let $\Sigma = \{0, 1\}$ be the binary alphabet and consider a string $x = x_1 x_2 \cdots x_n \in \Sigma^n$. In the easy embedding we extend the alphabet to

$$\Gamma = \{0, 1\} \cup \{\alpha_1, \alpha_2, \dots, \alpha_n\},$$

where each α_i is a symbol not in $\{0, 1\}$. We define the mapping

$$f : \Sigma^n \rightarrow \Gamma^*$$

by

$$f(x) = x_1 \alpha_1 x_2 \alpha_2 \cdots x_n \alpha_n.$$

¹We are grateful to Tatiana Starikovskaya for recently bringing this work to our attention, which we were not aware of during the preparation of our original manuscript.

A brief argument shows that in any optimal alignment between $f(x)$ and $f(y)$ (for $x, y \in \Sigma^n$), each marker α_i can only match with the corresponding marker in the other string. Hence, the edit distance satisfies

$$\Delta_{edit}(f(x), f(y)) = \Delta_{Ham}(x, y).$$

In a later subsection we reduce the extended alphabet back to binary using a padded encoding.

4.3.1 Alphabet Reduction to Binary

Let \mathcal{X} be a set of strings of length n over an alphabet Σ , where $\{0, 1\} \subseteq \Sigma$. Suppose for every pair of strings $X, Y \in \mathcal{X}$, there exists an optimal edit distance alignment that is *straight*, meaning

$$\Delta_{edit}(X, Y) = \Delta_{Ham}(X, Y),$$

and that all edit operations (substitutions) occur only at positions where both $X[i], Y[i] \in \{0, 1\}$.

We construct an explicit alphabet reduction function $\Phi : \Sigma^n \rightarrow \{0, 1\}^N$ preserving edit distances exactly, embedding each string into a binary string of length $N = O(n \log |\Sigma|)$.

1. Numeric Mapping. Each symbol in Σ is mapped uniquely to an integer in $\{0, 1, \dots, |\Sigma| - 1\}$, ensuring symbols $0, 1 \in \Sigma$ map to integers $0, 1$, respectively.

2. Binary Encoding. Each integer c is encoded in binary using $m := \lceil \log |\Sigma| \rceil$ bits, denoted by

$$\text{bin}(c) \in \{0, 1\}^m.$$

3. Padded Encoding for Alignment Enforcement. For each position $i \in [n]$ in string $X = x_1 x_2 \dots x_n$, define

$$\eta_i^X := 0^{2m} \cdot 1 \cdot \text{bin}(x_i) \cdot 0 \cdot 1^{2m},$$

where 0^{2m} and 1^{2m} denote sequences of $2m$ zeros and ones, respectively.

4. Final Binary Embedding. The binary embedding $\Phi(X)$ concatenates these encodings:

$$\Phi(X) = \eta_1^X \eta_2^X \dots \eta_n^X.$$

Thus, the resulting embedding length is

$$N = n \cdot (5m + 2) = O(n \log |\Sigma|).$$

Theorem 4.5 (Correctness of Alphabet Reduction). *The binary embedding Φ preserves edit distance exactly, i.e.,*

$$\Delta_{edit}(\Phi(X), \Phi(Y)) = \Delta_{edit}(X, Y) \quad \text{for all } X, Y \in \mathcal{X}.$$

Proof. Let \mathcal{A}' be an optimal edit distance alignment between $\Phi(X)$ and $\Phi(Y)$.

Note that if \mathcal{A}' is straight, then

$$\text{cost}(\mathcal{A}') = \Delta_{edit}(X, Y) = \Delta_{Ham}(X, Y),$$

because there are no insertions or deletions, and the only substitutions occur in a single bit of η_i^X (specifically within $\text{bin}(x_i)$ and $\text{bin}(y_i)$), exactly when $x_i \neq y_i$ (and $x_i, y_i \in \{0, 1\}$).

Suppose now that \mathcal{A}' is not straight. We will show we can construct an alignment \mathcal{A} of X, Y satisfying

$$\text{cost}(\mathcal{A}') \geq \text{cost}(\mathcal{A}),$$

and since clearly $\text{cost}(\mathcal{A}) \geq \Delta_{\text{edit}}(X, Y)$, this proves the theorem.

We classify each encoded block η_i^X according to alignment \mathcal{A}' :

1. **Good blocks:** A block η_i^X is *good* if it is aligned fully to exactly one block η_j^Y . Formally, every character of η_i^X aligns only with characters from a single block η_j^Y .
2. **Bad blocks:** Otherwise, a block is *bad*.

We first establish a key structural property of good blocks:

Claim 4.6. *If a block η_i^X has at most one edit operation in alignment \mathcal{A}' , then it must be a good block.*

Proof of Claim 4.6. Suppose first that block $\eta_i^X = 0^{2m} \cdot 1 \cdot \text{bin}(x_i) \cdot 0 \cdot 1^{2m}$ has zero edits. Then, the prefix $0^{2m}1$ matches exactly to some substring of $\Phi(Y)$. Such a substring can only occur as the prefix of a block η_j^Y . Thus, the prefix of η_i^X fully matches the prefix of a single block η_j^Y . By symmetry, the suffix $0 \cdot 1^{2m}$ must similarly match exactly the suffix of some block $\eta_{j'}^Y$. If $j \neq j'$, this alignment would require at least $5m + 2$ insertions to bridge the gap, contradicting zero edits. Hence, $j = j'$, implying the middle segment $\text{bin}(x_i)$ matches perfectly as well, making the block good.

Now suppose the block has exactly one edit operation. We proceed by cases, considering the possible location for this single edit operation within the block:

Case 1: Edit in the prefix $0^{2m}1$.

1. *Deletion:* If a single deletion occurs in the prefix, either the character '1' or one of the initial '0' is removed. If the character '1' is deleted, the resulting substring $0^{2m}\text{bin}(x_i)01^{2m}$ would have length $5m + 1$. However, any substring of Y starting with 0^{2m} and ending with 1^{2m} always has length exactly $5m + 2$. Thus, this scenario is impossible. If a '0' is deleted, then all remaining 0's must match precisely zeros in the prefix of some block η_j^Y , otherwise extra insertions or substitutions are required, again exceeding one edit.
2. *Insertion:* An insertion immediately before the prefix aligns the entire block η_i^X completely to some η_j^Y , thus preserving goodness. Any insertion elsewhere within the prefix extends its length beyond $0^{2m}1$, making exact alignment with the prefix of η_j^Y impossible without further edits. Thus, internal prefix insertions cannot occur.
3. *Substitution:* A substitution within the prefix is unnecessary, since perfect matching of suffix and middle segments guarantees identical prefixes, and the prefix would match perfectly without edits. Hence substitutions cannot happen here.

Thus, any single-edit scenario in the prefix maintains the block as good.

Case 2: Edit in the suffix $0 \cdot 1^{2m}$. By symmetry, identical reasoning to Case 1 shows a single edit in the suffix also preserves goodness.

Case 3: Edit in the middle segment $\text{bin}(x_i)$. Given perfect prefix and suffix matches, we must again have the same block η_j^Y aligning prefix and suffix, as otherwise bridging alignment would require multiple insertions. Thus, no character of $\text{bin}(x_i)$ can align outside η_j^Y . Hence, the block remains good even with a single middle-segment edit.

This exhaustively proves that a block with at most one edit is always good. \square

Thus, the alignment cost satisfies:

$$\text{cost}(\mathcal{A}') \geq 2 \cdot (\#\text{bad blocks}) + (\#\text{good blocks with } \geq 1 \text{ edits}).$$

We now construct an alignment \mathcal{A} between original strings X, Y :

1. For each good block, align corresponding symbols $x_i \leftrightarrow y_j$.
2. For remaining symbols (from bad blocks), choose the optimal alignment.

So, in \mathcal{A} , if x_i is deleted, then that would mean η_i^X was a bad block in \mathcal{A}' . And if x_i is substituted, then that would mean η_i^X was either a bad block in \mathcal{A}' or a good block with ≥ 1 edits.

Therefore, the resulting alignment cost satisfies:

$$\begin{aligned} \text{cost}(\mathcal{A}) &= 2 \cdot (\#\text{deletions}) + (\#\text{substitutions}) \\ &\leq 2 \cdot (\#\text{bad blocks}) + (\#\text{good blocks with } \geq 1 \text{ edits}). \end{aligned}$$

Hence:

$$\Delta_{\text{edit}}(X, Y) \leq \text{cost}(\mathcal{A}) \leq \text{cost}(\mathcal{A}') = \Delta_{\text{edit}}(\Phi(X), \Phi(Y)).$$

Thus, we conclude exact preservation of edit distance:

$$\Delta_{\text{edit}}(\Phi(X), \Phi(Y)) = \Delta_{\text{edit}}(X, Y).$$

\square

4.4 Recursive Embedding with Rate $1/\Theta(\log n)$

We now describe a recursive embedding that also achieves a rate of $\mathcal{O}(\log n)$. The idea is to break the string into two halves, recursively embed each half, and insert a separator that enforces straight alignment.

Let $x \in \{0, 1\}^n$ be a binary string. Suppose we split x into two parts:

$$x = x_1 \cdot x_2,$$

where $x_1 \in \{0, 1\}^{\lfloor n/2 \rfloor}$ and $x_2 \in \{0, 1\}^{\lceil n/2 \rceil}$. The recursive embedding f is defined as:

$$f(x) = f(x_1) \cdot 0^n \cdot 1^n \cdot f(x_2).$$

Also, $f(0) = 0$ and $f(1) = 1$. Here, $0^n 1^n$ serves as a separator between the embeddings of x_1 and x_2 .

4.4.1 Size Analysis

At each level we append exactly $2n$ separator symbols $0^n 1^n$. The recursion depth is $\log_2 n$. Hence

$$|f(x)| = n + 2n \log_2 n = \mathcal{O}(n \log n).$$

Thus the *rate* $= n/|f(x)| = 1/\Theta(\log n)$.

4.4.2 Correctness: Proof of Isometry

Theorem 4.7. *For all $n \in \mathbb{N}$ and all $x, y \in \{0, 1\}^n$, the recursive embedding f satisfies:*

$$\Delta_{edit}(f(x), f(y)) = \Delta_{Ham}(x, y).$$

Proof. We proceed by induction on the input length n .

Base case: $n = 1$. Then $x, y \in \{0, 1\}$, and by definition, $f(x) = x$, $f(y) = y$. Therefore,

$$\Delta_{edit}(f(x), f(y)) = \Delta_{edit}(x, y) = \Delta_{Ham}(x, y),$$

so the theorem holds.

Inductive step: Assume the theorem holds for all input lengths up to $\lceil n/2 \rceil$. Let $x, y \in \{0, 1\}^n$, and define:

$$x = x_1 \cdot x_2, \quad y = y_1 \cdot y_2,$$

where $x_1, y_1 \in \{0, 1\}^{\lceil n/2 \rceil}$ and $x_2, y_2 \in \{0, 1\}^{\lceil n/2 \rceil}$.

Let $f(x)$ be constructed as:

$$f(x) = f(x_1) \cdot 0^n \cdot 1^n \cdot f(x_2), \quad f(y) = f(y_1) \cdot 0^n \cdot 1^n \cdot f(y_2).$$

We prove the following claim first:

Claim 4.8. *For any $x, y \in \{0, 1\}^n$, the edit distance satisfies:*

$$\Delta_{edit}(f(x), f(y)) = \Delta_{edit}(f(x_1), f(y_1)) + \Delta_{edit}(f(x_2), f(y_2)).$$

Proof. Let \mathcal{A} be an optimal alignment between $f(x)$ and $f(y)$. Recall that:

$$f(x) = f(x_1) \cdot 0^n \cdot 1^n \cdot f(x_2), \quad f(y) = f(y_1) \cdot 0^n \cdot 1^n \cdot f(y_2).$$

We will show that the alignment cost decomposes exactly into two independent alignments: one involving $(f(x_1), f(y_1))$ and the other $(f(x_2), f(y_2))$.

Step 1 (Eliminating Cross-Half Alignments):

Suppose there is a character from $f(x_1)$ aligned with a character from $1^n f(y_2)$ (or similarly, a character from $f(y_1)$ aligned to $1^n f(x_2)$, a character from $f(x_2)$ aligned to $f(y_1)0^n$, or a character from $f(y_2)$ aligned to $f(x_1)0^n$). Such an alignment necessarily involves at least n insertions or deletions, as at least one

entire separator block (0^n or 1^n) must be skipped or misaligned. This incurs an edit cost of at least n .

However, the straight alignment has a cost at most $\Delta_{Ham}(x, y) \leq n$. Thus, such cross-half alignments cannot occur in any optimal alignment.

Step 2 (Cases Based on Separator Alignments):

Now, the remaining possible cases involve characters from separators aligning outside their respective separators. We analyze these explicitly:

Case A (No Separator Misalignment): Suppose no character from 0^n in $f(x)$ aligns outside 0^n in $f(y)$, (and similarly for 1^n). Then clearly:

$$\Delta_{edit}(f(x), f(y)) = \Delta_{edit}(f(x_1), f(y_1)) + \Delta_{edit}(0^n 1^n f(x_2), 0^n 1^n f(y_2)).$$

Using the fact that adding identical prefixes/suffixes doesn't change edit distance, we have:

$$\Delta_{edit}(0^n 1^n f(x_2), 0^n 1^n f(y_2)) = \Delta_{edit}(f(x_2), f(y_2)),$$

and thus the claim holds trivially in this scenario.

Case B (Separator Misalignment): Suppose characters from separators align outside. Without loss of generality, assume t_1 characters from 0^n in $f(x)$ align with characters from $f(y_1)$. Then:

1. The corresponding 0^n block in $f(y)$ has at most $n - t_1$ matched characters, leaving at least t_1 unmatched, incurring at least t_1 edit operations.

For the 1^n block, we have two sub-cases:

1. Suppose t_2 characters from 1^n of $f(x)$ align with characters from $f(y_2)$. Then at least t_2 characters from the corresponding 1^n block in $f(y)$ are unmatched, incurring at least t_2 additional edit operations.
2. Alternatively, suppose t_2 characters from 1^n of $f(y)$ align with characters from $f(x_2)$. A similar reasoning as above gives at least t_2 additional edit operations.

In total, at least $t_1 + t_2$ edit operations occur due to separator misalignments. However, consider the following alternative alignment strategy:

1. Match the entire separator $0^n 1^n$ of $f(x)$ directly with the entire separator $0^n 1^n$ of $f(y)$, incurring no edit operations within the separator. Therefore, we eliminate at least $t_1 + t_2$ edits that previously occurred.
2. The t_1 characters from $f(y_1)$ (or $f(x_1)$), previously aligned with characters from 0^n of the opposite string's separator, are now left unmatched. These characters must therefore be deleted or substituted or matched, incurring $\leq t_1$ edit operations.
3. Similarly, the t_2 characters from $f(y_2)$ (or $f(x_2)$), previously aligned with characters from 1^n of the opposite string's separator, are now also left unmatched, again incurring $\leq t_2$ edit operations.

In total, we eliminate at least $t_1 + t_2$ edits and incur at most $t_1 + t_2$ new edits. Thus, this realignment does not increase the overall edit cost; it simply redistributes the edit operations from the separator to the segments $f(x_1), f(y_1)$ and $f(x_2), f(y_2)$. This ensures a clean decomposition of the edit distance:

$$\Delta_{edit}(f(x), f(y)) = \Delta_{edit}(f(x_1), f(y_1)) + \Delta_{edit}(f(x_2), f(y_2)).$$

This completes the proof of the claim. \square

By the inductive hypothesis,

$$\Delta_{edit}(f(x_1), f(y_1)) = \Delta_{Ham}(x_1, y_1), \quad \Delta_{edit}(f(x_2), f(y_2)) = \Delta_{Ham}(x_2, y_2),$$

Therefore, the total cost is:

$$\begin{aligned} \Delta_{edit}(f(x), f(y)) &= \Delta_{edit}(f(x_1), f(y_1)) + \Delta_{edit}(f(x_2), f(y_2)) \\ &= \Delta_{Ham}(x_1, y_1) + \Delta_{Ham}(x_2, y_2) \\ &= \Delta_{Ham}(x, y) \end{aligned}$$

This completes the inductive proof. \square

4.5 Embedding with Rate $1/\mathcal{O}(\log^* n \times \log \log^* n)$: Multi-Level Recursion

We now present a recursive, multi-level embedding approach to significantly reduce the embedding overhead, achieving an embedding rate proportional to the iterated logarithm function $\log^* n$. The central concept is recursively embedding blocks of the input string and inserting carefully chosen padding sequences between them, ensuring precise control of alignment.

4.5.1 Multi-Level Recursive Embedding Definition

Let $X \in \{0, 1\}^n$ be the input binary string. For simplicity of calculation, we assume that n is towers of 2, specifically $n = 2 \uparrow \uparrow \ell$, for some ℓ . So, $\ell = \log^* n$. We begin by partitioning X into consecutive blocks of length exactly $\log n$:

$$X = b_1 b_2 \dots b_k, \quad \text{where } k = \frac{n}{\log n}, |b_i| = \log n.$$

$\Sigma = \{\Sigma_0, \Sigma_1, \dots, \Sigma_\ell\}$ be a collection of pairwise disjoint alphabets, where each Σ_i , for $i > 0$ is of constant size greater than 128, and, $\Sigma_0 = \{0, 1\}$ is the binary alphabet. For each level $1 \leq i \leq \ell$ of recursion, the embedding function $f_i(X)$ is defined recursively by:

$$f_i(X) = f_{i-1}(b_1) r_1^i f_{i-1}(b_2) r_2^i \dots f_{i-1}(b_k) r_k^i,$$

where each r_i^i is a padding string of length $\mathcal{O}(\log n)$, chosen from the alphabet Σ_i .
 $f_0(X) = X$.

The key properties of this construction are:

1. The alphabets $\Sigma_0, \Sigma_1, \dots, \Sigma_\ell$ are disjoint, which ensures that symbols from different levels do not interfere with each other in the alignment.
2. The padding strings r_i^ℓ are chosen from a code with large minimum edit distance, so that any alignment that crosses padding boundaries incurs a significant cost. This enforces that optimal edit alignments respect block boundaries.

Base Case: The recursion terminates at level 0, with embedding f_0 as the previously defined embedding (Section 4.4), known to exactly preserve edit distances (isometry).

Padding Sequence: The padding strings r_i^ℓ at each recursion level l are selected from a special error-correcting code with large minimum edit distance between distinct codewords. This ensures that any alignment crossing the padding boundaries incurs high edit cost, thus forcing block-aligned optimal solutions.

The existence of suitable padding sequences is established by the following lemma and corollary (which will be used subsequently):

Lemma 4.9 (Lemma 6.6). *For every $\varepsilon < 1/2$, let Σ be a finite alphabet with $|\Sigma| > 32/\varepsilon^2$. For every $n \in \mathbb{N}$, there exists $k = \mathcal{O}(\log n)$ satisfying:*

1. *There exists a code $C_{n,\varepsilon} \subseteq \Sigma^k$ with $|C_{n,\varepsilon}| = n$.*
2. *For every distinct $c, c' \in C_{n,\varepsilon}$, it holds that*

$$\Delta_{\text{indel}}(c, c') \geq (2 - \varepsilon)k.$$

Using this lemma, we obtain the following practical corollary:

Corollary 4.10. *For every $n \in \mathbb{N}$, there exists a code $C \subseteq \Sigma^k$ with $|C| = n$, $|\Sigma| = \mathcal{O}(1)$, and $k = \mathcal{O}(\log n)$, such that for all distinct $c, c' \in C$,*

$$\Delta_{\text{edit}}(c, c') > 3 \log n.$$

Proof. From Lemma 6.6, for any $\varepsilon < \frac{1}{2}$, there exists a finite alphabet Σ with $|\Sigma| > \frac{32}{\varepsilon^2}$, and a code $C \subseteq \Sigma^k$ with $|C| = n$, $k = \mathcal{O}(\log n)$, such that for all distinct $c, c' \in C$,

$$\Delta_{\text{indel}}(c, c') \geq (2 - \varepsilon)k.$$

It is a standard fact that for any strings x, y , the edit distance satisfies

$$\Delta_{\text{edit}}(x, y) \geq \frac{1}{2} \Delta_{\text{indel}}(x, y).$$

Applying this bound to the codewords yields:

$$\Delta_{\text{edit}}(c, c') \geq \left(1 - \frac{\varepsilon}{2}\right) k.$$

To ensure $|\Sigma| = \mathcal{O}(1)$, we choose a constant $\varepsilon < \frac{1}{2}$ which gives us $|\Sigma| > 128$. From the construction in Lemma 6.6, we also have $k \geq \frac{2}{\varepsilon} \log n$. Substituting this

into the previous inequality, we get:

$$\Delta_{edit}(c, c') \geq \left(1 - \frac{\varepsilon}{2}\right) \cdot \frac{2}{\varepsilon} \log n = \left(\frac{2 - \varepsilon}{\varepsilon}\right) \log n.$$

Since $\varepsilon < \frac{1}{2}$, it follows that $\frac{2 - \varepsilon}{\varepsilon} > 3$, and therefore:

$$\Delta_{edit}(c, c') > 3 \log n.$$

This completes the proof. □

4.5.2 Length Computation of the Embedding

We now formally analyze the length of strings embedded by this recursive construction.

Define $L(n)$ as the length of the embedding at recursion level ℓ for a string of length $n = 2 \uparrow \uparrow \ell$, i.e., length of $f_\ell(X)$. The recursion governing $L(n)$ is:

$$L(n) = \frac{n}{\log n} \cdot L(\log n) + \mathcal{O}(\log n) \cdot \frac{n}{\log n}.$$

Simplifying, we have:

$$L(n) = \frac{n}{\log n} L(\log n) + \mathcal{O}(n).$$

At the base case (level 0), the embedding length for strings of length 1 is also 1:

$$L(1) = 1.$$

Solving this recursive equation across all recursion levels, we get:

$$L(n) = \mathcal{O}(n) \times l = \mathcal{O}(n \log^* n).$$

4.5.3 Isometry Theorem

We now formally state and rigorously prove the central isometry theorem for the multi-level recursive embedding defined above. This result ensures the exact preservation of distances under the embedding.

Theorem 4.11 (Isometry of Multi-Level Recursive Embedding). *For every pair of binary strings $X, Y \in \{0, 1\}^n$, where $n = 2 \uparrow \uparrow \ell$, the multi-level recursive embedding f_ℓ satisfies:*

$$\Delta_{edit}(f_\ell(X), f_\ell(Y)) = \Delta_{Ham}(X, Y).$$

Proof. We proceed by induction on the recursion level ℓ .

Base Case (Level 0): For $\ell = 0$, we have $n = 1$, and by definition, $f_0(x) = x$. Thus:

$$\Delta_{edit}(f_0(x), f_0(y)) = \Delta_{edit}(x, y) = \Delta_{Ham}(x, y).$$

This establishes the base case.

Inductive Hypothesis: Assume the theorem holds at level $\ell - 1$. Explicitly, for every pair of strings U, V of length $m = 2 \uparrow (\ell - 1)$, it holds that:

$$\Delta_{edit}(f_{\ell-1}(U), f_{\ell-1}(V)) = \Delta_{Ham}(U, V).$$

Inductive Step (Level ℓ): At level ℓ , consider $X, Y \in \{0, 1\}^n$, where $n = 2 \uparrow \ell$. Partition these strings into blocks:

$$X = X_1 X_2 \dots X_k, \quad Y = Y_1 Y_2 \dots Y_k,$$

where each $X_i, Y_i \in \{0, 1\}^{\log n}$, and $k = \frac{n}{\log n}$.
The embedding at level ℓ is defined as:

$$f_\ell(X) = f_{\ell-1}(X_1)r_1^\ell f_{\ell-1}(X_2)r_2^\ell \dots f_{\ell-1}(X_k)r_k^\ell,$$

and similarly for $f_\ell(Y)$.

Definition 4.12 (Good and Bad Blocks). *Let \mathcal{A} be an alignment between $f_\ell(X)$ and $f_\ell(Y)$. We categorize each block of $f_\ell(X)$ into two types based on how \mathcal{A} aligns them with substrings of $f_\ell(Y)$:*

1. **Good f-blocks:** An f-block $f_{\ell-1}(X_i)$ is good if:
 - (a) None of its characters align outside of the substring $r_{i-1}^\ell \cdot f_{\ell-1}(Y_i) \cdot r_i^\ell$ of $f_\ell(Y)$.
 - (b) At least one character from $f_{\ell-1}(X_i)$ aligns with a character in $f_{\ell-1}(Y_i)$.
2. **Good r-blocks:** An r-block r_i^ℓ is good if:
 - (a) None of its characters align outside of the substring $f_{\ell-1}(Y_i) \cdot r_i^\ell \cdot f_{\ell-1}(Y_{i+1})$ of $f_\ell(Y)$.
 - (b) At least one character from r_i^ℓ aligns with a character in r_i^ℓ of $f_\ell(Y)$.
3. **Bad blocks:** Any block that does not satisfy the conditions for a good block is called bad.

Claim 4.13. *If an r-block r_i^ℓ is bad, then the number of edit operations performed by \mathcal{A} on this block is strictly greater than $3 \log n$.*

Proof. Since r_i^ℓ belongs to a high-edit-distance code (Corollary 4.10), we have $\Delta_{edit}(r_i^\ell, r_j^\ell) > 3 \log n$ for every $j \neq i$. Moreover, the alphabets of r-blocks and f-blocks are disjoint. Hence, any alignment of r_i^ℓ that does not fully align it with the matching r-block in $f_\ell(Y)$ must incur at least $3 \log n + 1$ edits. \square

Claim 4.14. *If an f-block $f_{\ell-1}(X_i)$ is bad, then at least one of its adjacent r-blocks r_{i-1}^ℓ or r_i^ℓ must also be bad. Consequently, the number of bad r-blocks is at least half the number of bad f-blocks.*

Proof. Suppose the f-block $f_{\ell-1}(X_i)$ is bad. Then there are two possible cases:

1. If a character from $f_{\ell-1}(X_i)$ aligns to a character in $f_{\ell-1}(Y_j)$ or r_j^ℓ with $j > i$, then the r-block r_i^ℓ cannot align correctly with r_i^ℓ in $f_\ell(Y)$, making r_i^ℓ bad.

2. Similarly, if a character from $f_{\ell-1}(X_i)$ aligns to a character in $f_{\ell-1}(Y_j)$ or r_j^ℓ with $j < i$, then the r-block r_{i-1}^ℓ cannot align correctly with r_{i-1}^ℓ in $f_\ell(Y)$, making r_{i-1}^ℓ bad.

Thus, for each bad f-block, at least one adjacent r-block must also be bad. Each r-block is adjacent to at most two f-blocks, so the number of bad r-blocks must be at least half the number of bad f-blocks. \square

Claim 4.15. *If an f-block $f_{\ell-1}(X_i)$ is good, then the number of edits performed by \mathcal{A} on this block is at least $\Delta_{Ham}(X_i, Y_i)$.*

Proof. Since $f_{\ell-1}$ is isometric at level $l - 1$, we have by induction:

$$\Delta_{edit}(f_{\ell-1}(X_i), f_{\ell-1}(Y_i)) = \Delta_{Ham}(X_i, Y_i).$$

Furthermore, the alphabets of the r-blocks adjacent to $f_{\ell-1}(Y_i)$ are disjoint from the alphabet of $f_{\ell-1}(X_i)$. Hence, any character alignment outside $f_{\ell-1}(Y_i)$ incurs substitution operations, adding to the total cost. Thus, the edit cost is at least $\Delta_{Ham}(X_i, Y_i)$. \square

Lemma 4.16. *Let \mathcal{A} be any alignment between $f_\ell(X)$ and $f_\ell(Y)$. The cost of \mathcal{A} satisfies:*

$$\text{cost}(\mathcal{A}) \geq \Delta_{Ham}(X, Y).$$

Proof. We classify each f-block as good, bad, or deleted according to \mathcal{A} . Define:

$$H = \{i : f_{\ell-1}(X_i) \text{ is good}\}.$$

Using previous claims, we can bound the alignment cost as follows:

$$\begin{aligned} \text{cost}(\mathcal{A}) &\geq (\text{edits from bad r-blocks}) + (\text{edits from good f-blocks}) \\ &\quad + (\text{edits from deleted f-blocks}) \\ &\geq (3 \log n) \times \frac{\# \text{ bad f-blocks}}{2} + \sum_{i \in H} \Delta_{Ham}(X_i, Y_i) \\ &\quad + (\text{length of a f-block}) \times (\# \text{ deleted f-blocks}) \\ &\geq \sum_{i \in H} \Delta_{Ham}(X_i, Y_i) + \sum_{i \notin H} \Delta_{Ham}(X_i, Y_i) \\ &= \Delta_{Ham}(X, Y). \end{aligned}$$

This establishes that any alignment incurs a cost of at least $\Delta_{Ham}(X, Y)$, proving the lemma. \square

Next we show that $\Delta_{edit}(f_\ell(X), f_\ell(Y)) = \Delta_{Ham}(X, Y)$ by proving that if \mathcal{A} is straight, then $\Delta_{edit}(f_\ell(X), f_\ell(Y)) = \Delta_{Ham}(X, Y)$. This completes the proof of the theorem.

First, we establish a useful structural lemma:

Lemma 4.17 (Structural Lemma). *Let $X = x_1 \cdot x_2 \cdots x_n \in \{0, 1\}^n$, where $n = 2 \uparrow \uparrow \ell$. Then the embedding at recursion level l can be expressed as:*

$$f_\ell(X) = P_0 \cdot x_1 \cdot P_1 \cdot x_2 \cdot P_2 \cdots P_{n-1} \cdot x_n \cdot P_n,$$

where each $P_i \in \Sigma^*$.

Proof. We proceed by induction on the recursion level ℓ :

Base Case ($\ell = 0$): The base case is immediate from the definition of the embedding. For $\ell = 0$, we have:

$$f_0(X) = X = x_1,$$

where clearly $P_0 = P_1 = \epsilon$, the empty string, satisfying the lemma trivially.

Inductive Hypothesis: Assume the lemma holds for recursion level $\ell - 1$. That is, for any string $X' \in \{0, 1\}^{n'}$, where $n' = 2 \uparrow\uparrow (\ell - 1)$, we have:

$$f_{\ell-1}(X') = P'_0 \cdot x'_1 \cdot P'_1 \cdot x'_2 \cdots P'_{n'-1} \cdot x'_{n'} \cdot P'_{n'}.$$

Inductive Step: For recursion level ℓ , the input string is partitioned as:

$$X = X_1 X_2 \dots X_k, \quad \text{where } k = \frac{n}{\log n} \text{ and } |X_i| = \log n = 2 \uparrow\uparrow (\ell - 1).$$

Applying the inductive hypothesis to each X_i , we have:

$$f_{\ell-1}(X_i) = P_0^i \cdot x_{((i-1)\log n)+1} \cdot P_1^i \cdot x_{((i-1)\log n)+2} \cdot P_2^i \cdots P_{\log n-1}^i \cdot x_{(i\log n)} \cdot P_{\log n}^i.$$

Thus, the recursive construction for level ℓ gives:

$$\begin{aligned} f_\ell(X) &= f_{\ell-1}(X_1) \cdot r_1^\ell \cdot f_{\ell-1}(X_2) \cdot r_2^\ell \cdots f_{\ell-1}(X_k) \cdot r_k^\ell \\ &= (P_0^1 \cdot x_1 \cdot P_1^1 \cdots x_{\log n} \cdot P_{\log n}^1) \cdot r_1^\ell \cdots (P_0^k \cdot x_{(k-1)\log n+1} \cdots x_n \cdot P_{\log n}^k) \cdot r_k^\ell. \end{aligned}$$

Rewriting this more explicitly:

$$f_\ell(X) = P_0 \cdot x_1 \cdot P_1 \cdot x_2 \cdot P_2 \cdots x_n \cdot P_n,$$

where the padding substrings P_j are appropriately assigned as follows:

$$P_j = \begin{cases} P_t^i & \text{if } j = (i-1)\log n + t, \ t < \log n, \ 1 \leq i \leq \frac{n}{\log n}, \\ P_{\log n}^i \cdot r_i^\ell & \text{if } j = i\log n, \ 1 \leq i \leq \frac{n}{\log n}. \end{cases}$$

This completes the inductive step, proving the lemma. \square

Claim 4.18 (Mismatch Positions in Straight Alignment). *If the alignment \mathcal{A} is straight, then every mismatched position i in the alignment between $f_\ell(X)$ and $f_\ell(Y)$ satisfies:*

$$f_\ell(X)[i], f_\ell(Y)[i] \in \{0, 1\}.$$

Proof. This result follows directly from the structural lemma (Lemma 4.17). Specifically, a straight alignment matches each padding substring P_i exactly with the corresponding padding substring from the other embedded string. Hence, mismatches occur exclusively at original bit positions, which are elements of the binary alphabet $\{0, 1\}$. \square

Thus, a straight alignment precisely corresponds to a bitwise comparison between the original strings, confirming that:

$$\Delta_{edit}(f_\ell(X), f_\ell(Y)) = \Delta_{Ham}(X, Y).$$

This completes the proof of the theorem. \square

4.5.4 Final Embedding After Alphabet Reduction

The final embedding of any string $X \in \{0, 1\}^n$ is given by $\Phi(f_\ell(X))$, where $n = 2 \uparrow \uparrow \ell$, i.e., $\ell = \log^* n$, and Φ denotes the alphabet reduction function defined in Section 4.3.1.

We are able to apply the alphabet reduction at the top level because of two key properties:

- a. The isometry of the multi-level recursive embedding f_ℓ , which ensures that $\Delta_{edit}(f_\ell(X), f_\ell(Y)) = \Delta_{Ham}(X, Y)$.
- b. The structural property required by the alphabet reduction (namely, that the edit distance equals the Hamming distance and substitutions are restricted to $\{0, 1\}$) holds by design in the output of f_ℓ .

Therefore, by correctness of the alphabet reduction (Theorem 4.5), we have:

$$\Delta_{edit}(\Phi(f_\ell(X)), \Phi(f_\ell(Y))) = \Delta_{Ham}(X, Y),$$

for all $X, Y \in \{0, 1\}^n$.

Embedding Size and Rate. Since the number of recursion levels is $l = \log^* n$, and each level introduces a constant-size new alphabet $|\Sigma_i| = \mathcal{O}(1)$, the total alphabet size before reduction is:

$$|\Sigma| = \mathcal{O}(\log^* n).$$

After applying the alphabet reduction Φ , the length of the embedded string is scaled by a factor of $\log |\Sigma| = \mathcal{O}(\log \log^* n)$. Recalling that the length of $f_\ell(X)$ is $L(n) = \mathcal{O}(n \log^* n)$, we get:

$$|\Phi(f_\ell(X))| = \mathcal{O}(n \log^* n \cdot \log \log^* n).$$

Conclusion. The final rate of the embedding using the multi-level recursive construction combined with alphabet reduction is:

$$1/\mathcal{O}(\log^* n \cdot \log \log^* n).$$

4.6 Constant-Rate Embedding

In this section we give an explicit isometric embedding from $(\{0, 1\}^n, \Delta_{Ham})$ into $(\{0, 1\}^{N'}, \Delta_{edit})$ with $N' = \mathcal{O}(n)$. The key ingredient is the use of ε -synchronization strings.

Definition 4.19 (ε -synchronization string [55]). *Let $\varepsilon \in (0, 1)$ and $n \in \mathbb{N}$. A string $S \in \Sigma^n$ is an ε -synchronization string if for every triple of indices $1 \leq i < j < k \leq n + 1$,*

$$\Delta_{\text{indel}}(S_{[i,j]}, S_{[j,k]}) > (1 - \varepsilon)(k - i).$$

We will use the following two Theorems from the paper [55].

Theorem 4.20 (Existence [55, Thm. 5.7]). *For every $\varepsilon \in (0, 1)$ and $n \geq 1$, there exists an ε -synchronization string of length n over an alphabet of size $\Theta(\varepsilon^{-4})$.*

We need the following definitions before stating the next Theorem.

Definition 4.21 (Non-self LCS and edit distance). *For any string S :*

1. *self-LCS(S) is the length of the longest common subsequence of S with itself under the constraint that no character may match its own position.*
2. *self- $\Delta_{\text{edit}}(S)$ is the edit distance between S and itself when matches at the same index are forbidden.*

Theorem 4.22 (Self-alignment bound [55, Thm. 6.4a]). *For some $\varepsilon \in (0, 1)$ if S is an ε -synchronization string then for every substring S' of S , self-LCS(S') $\leq \varepsilon |S'|$.*

Combining the two above Theorem, we have our Corollary:

Corollary 4.23. *For any $\varepsilon \in (0, 1)$ and $n \geq 1$, there exists an ε -synchronization string $S \in \Gamma^n$ (with $|\Gamma| = \Theta(\varepsilon^{-4})$) such that for every substring S' of S ,*

$$\text{self-}\Delta_{\text{edit}}(S') \geq (1 - \varepsilon) |S'|.$$

Proof. Immediate from Theorems 4.20 and 4.22, together with the fact that self- $\Delta_{\text{edit}}(S') \geq |S'| - \text{self-LCS}(S')$. \square

4.6.1 Embedding Construction

Fix $\varepsilon \in (0, 1)$ and choose any integer

$$p > \frac{3}{1 - \varepsilon}.$$

By Theorem 4.20, let

$$S = s_1 s_2 \cdots s_{pn} \in \Gamma^{pn} \quad (\Gamma \cap \{0, 1\} = \emptyset)$$

be an ε -synchronization string. Partition S into n consecutive blocks S_1, \dots, S_n each of length p . Define

$$E_\varepsilon : \{0, 1\}^n \longrightarrow (\Gamma \cup \{0, 1\})^N, \quad N = n + pn,$$

by

$$E(x_1 x_2 \cdots x_n) = x_1 S_1 x_2 S_2 \cdots x_n S_n.$$

Finally, apply the binary-alphabet reduction from Section 4.3.1 to obtain

$$E' : \{0, 1\}^n \longrightarrow \{0, 1\}^{N'}, \quad N' = \mathcal{O}(N) = \mathcal{O}(n).$$

From the next sections, we will drop the subscripts ε , and refer to E_ε as simply E .

4.6.2 Analysis of the Edit Alignment

We now show E (and hence E') is isometric.

Theorem 4.24. *Fix any $\varepsilon \in (0, 1)$ and choose $p > 3/(1 - \varepsilon)$. The map E (and hence E') is an isometric embedding:*

$$\Delta_{edit}(E(x), E(y)) = \Delta_{Ham}(x, y) \quad \forall x, y \in \{0, 1\}^n.$$

Since $|E'(x)| = \mathcal{O}(n)$, this gives a constant-rate embedding.

Proof. The inequality $\Delta_{edit}(E(x), E(y)) \leq \Delta_{Ham}(x, y)$ is immediate by the straight alignment which matches each block S_i to itself and substitutes exactly those i with $x_i \neq y_i$. The reverse inequality follows from the following Lemma which shows any optimal alignment must be straight and thus incur cost $\Delta_{Ham}(x, y)$.

Lemma 4.25. *Let $x, y \in \{0, 1\}^n$ and let \mathcal{A} be any optimal edit alignment between $E(x)$ and $E(y)$. Then \mathcal{A} must be entirely straight, and consequently*

$$\Delta_{edit}(E(x), E(y)) = \Delta_{Ham}(x, y).$$

□

Proof of Lemma 4.25. We show that any deviation from the straight alignment incurs strictly greater cost than the straight alignment itself, contradicting optimality.

Straight vs. non-straight intervals. Consider the coordinate set $[1, N]$ of the two embedded strings. An alignment \mathcal{A} naturally partitions $[1, N]$ into disjoint *straight intervals* and *non-straight intervals*:

1. A *straight interval* $[l, r]$ is one in which every position $i \in [l, r]$ is matched by \mathcal{A} to the *same* position in the other string.
2. A *non-straight interval* $[l, r]$ is one in which no position $i \in [l, r]$ is matched to itself.
3. A *maximal non-straight interval* is a non-straight interval $[l, r]$ such that neither $[l - 1, r]$ nor $[l, r + 1]$ is non-straight—i.e. extending it on either side forces at least one index to align to itself.

Original vs. supplementary symbols. Observe that each position of $E(x)$ (and likewise $E(y)$) is either

1. an *original bit* (one of the $x_i \in \{0, 1\}$), or

2. a *supplementary symbol* (one of the $s_j \in S \subset \Gamma$).

On each straight interval, \mathcal{A} agrees with the “straight alignment,” so its cost there equals exactly the number of original-bit substitutions in that segment.

Cost on a maximal non-straight interval. Let $[l, r]$ be a maximal non-straight interval of length $t = r - l + 1$. We compare:

$$(i) \quad c_{\text{straight}} = \left\lfloor \frac{t}{p+1} \right\rfloor + 1 \quad \text{vs.} \quad (ii) \quad c_{\text{nonstraight}} = \text{cost}(\mathcal{A}_{[l,r]}).$$

Here (i) is an upper bound on the straight-alignment cost, since there are at most $\lfloor t/(p+1) \rfloor + 1$ original characters in $E(x)_{[l,r]}$.

To lower-bound (ii), observe that within $E(x)_{[l,r]}$ there are two kinds of symbols:

1. At most $q := \left\lfloor \frac{t}{p+1} \right\rfloor + 1$ *original* bits.
2. The remaining *supplementary* symbols drawn from blocks of S .

Thus the supplementary substring $S' \subseteq S$ involved has length

$$|S'| \geq t - q \geq t - \left(\frac{t}{p+1} + 1 \right).$$

By Corollary 4.23, any alignment of S' to itself that forbids fixed-point matches costs at least $(1 - \varepsilon)|S'|$. Meanwhile each original-to-supplementary match in \mathcal{A} can only reduce cost by at most one, and there are at most $2q$ such matches (originals from $E(x)$ or $E(y)$). Hence

$$c_{\text{nonstraight}} \geq (1 - \varepsilon)|S'| - 2q \geq (1 - \varepsilon) \left(t - \frac{t}{p+1} - 1 \right) - 2 \left(\frac{t}{p+1} + 1 \right).$$

A brief calculation shows that for any $p > 3/(1 - \varepsilon)$,

$$(1 - \varepsilon) \left(t - \frac{t}{p+1} - 1 \right) - 2 \left(\frac{t}{p+1} + 1 \right) > \frac{t}{p+1} + 1 = c_{\text{straight}}.$$

Thus on any maximal non-straight interval the alignment cost strictly exceeds the straight-alignment cost on the same interval.

Since maximal non-straight intervals are disjoint, their costs add, and each individually exceeds the straight-alignment cost on that segment. Therefore any alignment \mathcal{A} with even one non-straight interval would have total cost exceeding that of the pure straight alignment, contradicting optimality. Hence \mathcal{A} must be entirely straight, and so $\Delta_{\text{edit}}(E(x), E(y)) = \Delta_{\text{Ham}}(x, y)$, as claimed. \square

4.7 Upper Bound on Rate of Embedding

In this section, we establish an upper bound on the embedding rate from the Hamming metric into the edit metric over binary alphabet strings. Specifically, we address the following question:

Given the set of 2^n binary strings of length n equipped with the Hamming metric, what is the minimal length N required for embedding

them into a subset of binary strings of length N under the edit metric, preserving all pairwise distances exactly?

Formally, we seek the smallest N such that there exists an embedding

$$E : (\{0, 1\}^n, \Delta_{Ham}) \rightarrow (\{0, 1\}^N, \Delta_{edit})$$

satisfying

$$\Delta_{edit}(E(X), E(Y)) = \Delta_{Ham}(X, Y) \quad \text{for all } X, Y \in \{0, 1\}^n.$$

For small values of n , the minimal length is straightforward to determine:

1. For $n \leq 2$, we trivially have $N = n$, thus the embedding rate is 1.
2. For $n = 3$, a simple verification reveals $N = 5$.

To tackle the general case, we first establish two important structural results:

1. **Straight alignment necessity:** We show that for any isometric embedding from the Hamming metric to the edit metric, there must exist an optimal alignment between embedded strings that is *straight*. In other words, the alignment must match each position i of the embedded string $E(X)$ directly with the position i of $E(Y)$, allowing substitutions only (no insertions or deletions).
2. **Embedding characterization:** Using this straight-alignment property, we characterize the embedding as being of a specific form, which we term a *projective embedding*.

We define the notion of straight alignment and projective embedding precisely:

We begin by formally defining the main concepts used in our structural characterization.

Definition 4.26 (Projective Embedding). *An embedding*

$$E : \{0, 1\}^n \rightarrow \{0, 1\}^N$$

is called a projective embedding if there exist:

1. A fixed permutation π of $\{1, 2, \dots, n\}$.
2. Fixed binary strings $P_0, P_1, \dots, P_n \in \{0, 1\}^*$.
3. A fixed binary vector (shift vector) $g \in \{0, 1\}^n$.

such that for every input string $X = x_1x_2 \cdots x_n \in \{0, 1\}^n$, the embedding has the form:

$$E(X) = P_0 (x_{\pi(1)} \oplus g_{\pi(1)}) P_1 (x_{\pi(2)} \oplus g_{\pi(2)}) P_2 \cdots P_{n-1} (x_{\pi(n)} \oplus g_{\pi(n)}) P_n,$$

where \oplus denotes bitwise XOR operation.

Intuitively, a projective embedding is constructed by permuting the bits of the input string according to a fixed permutation π , applying a global bitwise XOR with a fixed vector g , and inserting fixed padding strings P_i uniformly at specified positions for all strings.

In other words, the embedding must uniformly apply the same permutation, bitwise shifts (XOR operations), and insert identical padding strings for all input strings in order to preserve Hamming distances exactly under the edit metric.

The intuition behind these definitions is that, in order to preserve the Hamming distances exactly under the edit metric, the insertion of P_i must force an optimal (and thus, straight) alignment between embedded strings. Any deviation (or “non-straight” alignment) would incur extra edit operations and hence violate isometry.

This structural characterization, as we will rigorously show in subsequent sections, directly implies an inherent limitation on the efficiency of any isometric embedding from the Hamming space into the edit space, establishing a concrete upper bound on achievable embedding rates.

4.7.1 Straight Alignment Necessity

The first step is to show that, for any isometric embedding E , there exists an optimal edit alignment between any two embedded strings which is *straight*.

Theorem 4.27 (Optimal Straight Alignment Necessity). *Let*

$$E : \{0, 1\}^n \rightarrow \{0, 1\}^N$$

be an isometric embedding from the Hamming space $(\{0, 1\}^n, \Delta_{Ham})$ into the edit space $(\{0, 1\}^, \Delta_{edit})$. Then for all distinct $X, Y \in \{0, 1\}^n$, there exists an optimal edit alignment between $E(X)$ and $E(Y)$ which is straight; that is, the alignment aligns bit i of $E(X)$ to bit i of $E(Y)$ for every position i (in other words, the only edit operations are substitutions).*

Proof. We prove the claim by induction on $k = \Delta_{Ham}(X, Y)$, the Hamming distance.

Base Case ($k = 1$). If $\Delta_{Ham}(X, Y) = 1$, then X and Y differ in exactly one coordinate. By isometry,

$$\Delta_{edit}(E(X), E(Y)) = 1.$$

Since $|E(X)| = |E(Y)|$, the only edit operation of cost 1 is a single substitution. Hence the optimal alignment matches all but one position, and the mismatch occurs precisely at the coordinate where X and Y differ. Thus, this alignment is clearly straight.

Inductive Hypothesis. Assume the theorem holds for all pairs of strings with Hamming distance at most k .

Inductive Step. Let $X, Y \in \{0, 1\}^n$ satisfy $\Delta_{Ham}(X, Y) = k + 1$. Choose an intermediate string Z such that

$$\Delta_{Ham}(X, Z) = k, \quad \Delta_{Ham}(Y, Z) = 1.$$

By the inductive hypothesis, there exist optimal alignments between $E(X)$ and $E(Z)$, and between $E(Y)$ and $E(Z)$, that are both straight. Let i be the unique position where $E(Y)$ and $E(Z)$ differ.

Define the set of positions where $E(X)$ and $E(Z)$ differ as P , with $|P| = k$. We claim that $i \notin P$. To see this, suppose (for contradiction) $i \in P$. Then bit i in both $E(X)$ and $E(Y)$ would match. Consequently, we can align $E(X)$ and $E(Y)$ with cost at most $k - 1$, mismatching only the positions in $P \setminus \{i\}$. This contradicts the assumption that the embedding is isometric, which requires the optimal cost to be exactly $k + 1$.

Hence, bit i differs in $E(X)$ and $E(Y)$ (since it mismatches in $E(Y)$ and $E(Z)$ but matches in $E(X)$ and $E(Z)$), and the positions in P also differ between $E(X)$ and $E(Y)$. Aligning these positions straight gives an edit distance exactly $k + 1$, which is optimal.

This completes the inductive step and thus the proof. \square

4.7.2 Characterization as Projective Embeddings

Once the straight alignment property is established, we can characterize any isometric embedding as a projective embedding.

Theorem 4.28 (Projective Embedding Characterization). *Let*

$$E : \{0, 1\}^n \rightarrow \{0, 1\}^N$$

be an isometric embedding from the Hamming space $(\{0, 1\}^n, \Delta_{Ham})$ into the edit space $(\{0, 1\}^, \Delta_{edit})$. Then there exist a fixed permutation π of $\{1, 2, \dots, n\}$, fixed binary strings P_0, P_1, \dots, P_n , and a fixed shift vector $g \in \{0, 1\}^n$, such that for every input $X = x_1x_2 \dots x_n \in \{0, 1\}^n$, we have*

$$E(X) = P_0 (x_{\pi(1)} \oplus g_{\pi(1)}) P_1 (x_{\pi(2)} \oplus g_{\pi(2)}) P_2 \cdots P_{n-1} (x_{\pi(n)} \oplus g_{\pi(n)}) P_n,$$

where \oplus denotes bitwise XOR.

In particular, the structure of $E(X)$ is exactly that of a projective embedding.

Proof. Let $X_0 = 0^n$ and for $1 \leq i \leq n$ let X_i be the string with a single 1 in position i . Consider the set

$$U = \{X_0, X_1, \dots, X_n\}.$$

Determining π , P_i , and g . By isometry and Theorem 4.27, for each i there is a unique position p_i where $E(X_0)$ and $E(X_i)$ differ (since $\Delta_{edit}(E(X_0), E(X_i)) = 1$). If $p_i = p_j$ for $i \neq j$, then $E(X_i) = E(X_j)$, contradicting $\Delta_{edit}(E(X_i), E(X_j)) = \Delta_{Ham}(X_i, X_j) = 2$. Thus the p_i are distinct.

Order these positions to define a permutation:

$$p_{\pi(1)} < p_{\pi(2)} < \cdots < p_{\pi(n)}.$$

Next, carve $E(X_0)$ into blocks to define the padding strings P_i as follows:

$$\begin{aligned} P_0 &= E(X_0)[1 : p_{\pi(1)} - 1], \\ P_i &= E(X_0)[p_{\pi(i)} + 1 : p_{\pi(i+1)} - 1] \quad (1 \leq i \leq n-1), \\ P_n &= E(X_0)[p_{\pi(n)} + 1 : N]. \end{aligned}$$

Define the shift vector by

$$g_{\pi(i)} = E(X_0)[p_{\pi(i)}] \quad (1 \leq i \leq n).$$

Then

$$\begin{aligned} E(X_0) &= P_0 g_{\pi(1)} P_1 g_{\pi(2)} \cdots P_{n-1} g_{\pi(n)} P_n \\ &= P_0 (0 \oplus g_{\pi(1)}) P_1 \cdots (0 \oplus g_{\pi(n)}) P_n. \end{aligned}$$

Similarly, for each i , the only mismatch between $E(X_i)$ and $E(X_0)$ is at p_i . Since $E(X_0)[p_i] = (0 \oplus g_i)$, we have $E(Y)[p_i] = (1 \oplus g_i)$:

$$E(X_i) = P_0 \cdots P_{\pi^{-1}(i)-1} (1 \oplus g_i) P_{\pi^{-1}(i)} \cdots P_n.$$

Extending to arbitrary binary string. Let $Y \in \{0, 1\}^n$ and $J = \{i : Y_i = 1\}$. Then $\Delta_{Ham}(X_0, Y) = |J|$, so $\Delta_{edit}(E(X_0), E(Y)) = |J|$. By Theorem 4.27 there is an optimal straight alignment of cost $|J|$. We claim the mismatches occur exactly at $\{p_i : i \in J\}$.

Claim 4.29. *In the optimal straight alignment between $E(X_0)$ and $E(Y)$, the set of mismatch positions is exactly $\{p_i : i \in J\}$.*

Proof. Since $\Delta_{edit}(E(X_0), E(Y)) = |J|$, any optimal alignment has precisely $|J|$ mismatches. Suppose, for contradiction, that for some $q \in J$, the position p_q is matched. Then to still have $|J|$ mismatches, there must be some position $j \notin \{p_i : i \in J\}$ and $j \neq p_q$ at which $E(X_0)$ and $E(Y)$ differ.

Now consider the alignment of $E(X_q)$ with $E(Y)$. By construction, $E(X_0)$ and $E(X_q)$ differ only at position p_q . Hence every mismatch between $E(X_0)$ and $E(Y)$ at any $j \neq p_q$ remains a mismatch between $E(X_q)$ and $E(Y)$. Since there are $|J|$ such positions, it would follow that

$$\Delta_{edit}(E(X_q), E(Y)) \geq |J|,$$

contradicting

$$\Delta_{edit}(E(X_q), E(Y)) = \Delta_{Ham}(X_q, Y) = |J| - 1.$$

Therefore each p_i for $i \in J$ must be a mismatch, and no other positions can mismatch. \square

It follows that for each $i \in J$, the bit at position p_i in $E(X_0)$ (namely $0 \oplus g_i$) flips in $E(Y)$ to $1 \oplus g_i$. Hence

$$E(Y) = P_0 (y_{\pi(1)} \oplus g_{\pi(1)}) P_1 \cdots P_{n-1} (y_{\pi(n)} \oplus g_{\pi(n)}) P_n,$$

completing the proof of Theorem 4.28. □

4.7.3 Rate Upper Bound

We now state and prove the main result concerning the embedding rate:

Theorem 4.30 (Upper Bound on Embedding Rate). *Let $E : \{0, 1\}^n \rightarrow \{0, 1\}^N$ be any isometric embedding from the Hamming metric to the edit metric. Then the embedding rate satisfies:*

$$\frac{n}{N} \leq \frac{3}{7} + o(1).$$

To prove this theorem, we first establish some helpful observations.

In this subsection, whenever we mention an *isometric embedding*, we specifically refer to an isometric embedding from the Hamming metric to the edit metric. Additionally, we will frequently use the characterization provided by Theorem 4.28, which states that every such embedding is a *projective embedding* characterized by a permutation π , a shift vector g , and padding strings P_0, P_1, \dots, P_n . Throughout this subsection, the symbols π , g , and P_i will specifically refer to these elements from Theorem 4.28.

Observation 4.31. *If there exists an isometric embedding E of rate R with permutation π not equal to the identity, then there also exists an isometric embedding of rate R where π is the identity permutation.*

Proof. Given any embedding E with permutation π , reorder the input bits according to π^{-1} . This reordering preserves all Hamming distances exactly and produces an embedding with the identity permutation and identical rate. □

Observation 4.32. *If there exists an isometric embedding E of rate R with shift vector g not equal to the zero vector, then there exists an embedding of rate R with shift vector g equal to the zero vector.*

Proof. Apply a global XOR operation to all inputs using shift vector g . This transformation preserves all Hamming distances and yields an equivalent embedding with a zero shift vector and the same embedding rate. □

We now define a useful quantity:

Definition 4.33 (Padding-length vector). *Given a projective embedding E , we define the padding-length vector as:*

$$L = (|P_0|, |P_1|, \dots, |P_n|),$$

where $|P_i|$ denotes the length of the padding string P_i .

Definition 4.34 (Padding-length vector). *Given a projective embedding E , we define the padding-length vector as:*

$$L = (|P_0|, |P_1|, \dots, |P_n|),$$

where $|P_i|$ denotes the length of the padding string P_i . The total padding length m is defined as the sum of all elements of L :

$$m = \sum_{i=0}^n |P_i|.$$

Observation 4.35. *If there exists an isometric embedding E with rate n/N and either $|P_0| \neq 0$ or $|P_n| \neq 0$, then there exists an embedding with strictly better rate where $|P_0| = |P_n| = 0$.*

Proof. Adding or removing identical strings at the beginning or end of every embedded string does not alter the edit distances between pairs of embedded strings. Thus, removing P_0 and P_n reduces the total embedding length N , strictly improving the embedding rate. \square

Based on these observations, computational experiments give minimal padding requirements for small values of n .

Claim 4.36 (Minimal Padding-Lengths for Small n). *The minimal padding lengths $m = N - n$ necessary for an isometric embedding for small values of n are as follows:*

n	$m = N - n$
3	2
4	3
5	4
6	6
7	7
8	9
9	10
10	12

Proof. These results are obtained through exhaustive computational search over all possible padding-length vectors $L = (|P_0|, |P_1|, \dots, |P_n|)$ satisfying the conditions (The code for this is available in [67]):

$$|P_0| = |P_n| = 0, \quad \sum_{i=0}^n |P_i| = m,$$

for various small values of n . The computational procedure enumerates all feasible padding-length vectors and verifies explicitly whether there exists a suitable binary padding configuration matching these lengths and achieving exact distance preservation.

Specifically, for each candidate padding-length vector, the algorithm checks the existence of a binary string $Y \in \{0, 1\}^m$ that can serve as padding substrings P_i with lengths given by L . The minimal padding lengths presented above are the smallest for which the existence of such Y is confirmed by computational search. \square

Using these computational findings, we establish a general lower bound on the required padding length:

Lemma 4.37. *For every integer $k \geq 1$, if $n = 9k$, then the minimum required padding length m satisfies:*

$$m \geq 10k + 2(k - 1).$$

Proof. We prove the result by induction on k .

Base case ($k = 1$): For $n = 9$, Claim 4.36 confirms that the minimum required padding length is $m = 10$, which matches the bound $10k + 2(k - 1) = 10$. Thus, the base case holds.

Inductive step: Assume the lemma holds for some $k \geq 1$; that is, for $n = 9k$, we have

$$m \geq 10k + 2(k - 1).$$

We must show that the result holds for $k + 1$. Consider $n = 9(k + 1)$. We decompose the embedding into a prefix of length $9k$ and a suffix of 9 additional bits.

By the inductive hypothesis, the prefix requires at least $10k + 2(k - 1)$ padding symbols. According to Claim 4.36, an embedding of a suffix of length 10 (which includes the additional 9 bits and one junction bit) requires at least 12 padding symbols.

Therefore, the total padding length satisfies:

$$m \geq (10k + 2(k - 1)) + 12 = 10(k + 1) + 2k.$$

This completes the inductive step and proves the lemma. □

Finally, using Lemma 4.37, we prove our main theorem:

Proof of Theorem 4.30. Let $n = 9k + i$ for some integer $k \geq 1$ and $0 \leq i \leq 8$. From Lemma 4.37, the total padding length for length $9k$ is at least

$$m \geq 10k + 2(k - 1) = 12k - 2,$$

and hence

$$N = n + m \geq 9k + i + 12k - 2 \geq 21k - 2.$$

Therefore,

$$\frac{n}{N} \leq \frac{9k + i}{21k - 2} \leq \frac{9k}{21k - 2} + \frac{8}{21k - 2} = \frac{9k}{21k} + \frac{18k}{21k(21k - 2)} + \frac{8}{21k - 2} = \frac{3}{7} + o(1).$$

This completes the proof. □

4.8 Summary

This chapter explored the fundamental question of embedding the Hamming metric space isometrically into the edit metric space, with a primary focus on binary strings. Initially, straightforward embeddings with relatively low efficiency (rate $1/\Theta(\log n)$) were introduced, followed by progressively more refined recursive embeddings, achieving rates as low as $1/\Theta(\log^* n \cdot \log \log^* n)$.

The central contribution of the chapter is the construction of a constant-rate isometric embedding leveraging synchronization strings. This groundbreaking result significantly surpasses previous rate limitations and allows algorithmic hardness results to transfer directly and optimally from the Hamming to the edit metric.

Moreover, leveraging this constant-rate embedding, the chapter established optimal hardness transfers for key problems like bichromatic closest pair, discrete 1-center, and discrete clustering and Steiner tree problems, deriving new lower bounds and NP-hardness approximation results directly in the edit metric.

Additionally, structural constraints on possible embeddings were rigorously examined, demonstrating that any isometric embedding must conform to a projective type structure, and establishing a theoretical upper bound of $3/7 + o(1)$ on achievable embedding rates.

Thus, this chapter provided both theoretical and constructive insights, positioning the Hamming-to-edit embedding as a central problem in understanding metric embeddings and complexity theory.

5

Edit and Indel distance

*“We are only as strong as we are united, as weak as we are divided.”
- Dumbledore*

5.1 Introduction

In the previous chapters, we looked into embeddings between Δ_{edit} and Δ_{Ham} metric which are computationally very different. This chapter focuses on embeddings between two very similar metrics, specifically the Δ_{indel} metric and Δ_{edit} metric. Tiskin [68] (in section 6.1) proposed a straightforward embedding from the Δ_{edit} metric to the Δ_{indel} metric. This inspired our exploration into embedding in the reverse direction i.e. from the Δ_{indel} metric to the Δ_{edit} metric. Our primary contribution in this realm is a scaled isometric embedding from the Δ_{indel} metric to Δ_{edit} , as outlined below.

Theorem 5.1 (Indel Into Edit Metrics Embedding - Approximate embedding – Statement of Theorem 5.4). *For any alphabet Σ , $n \in \mathbb{N}$ and $\varepsilon \in (0, 1]$, there exist mappings $E : \Sigma^n \rightarrow \Sigma^n$ and $E' : \Sigma^n \rightarrow (\Sigma \cup \{\$\})^N$, where $N = \Theta(n/\varepsilon)$, such that for any $X, Y \in \Sigma^n$, we have*

$$\Delta_{edit}(E(X), E'(Y)) = N - n + k, \text{ where } k \in \left[\frac{\Delta_{indel}(X, Y)}{2}, (1 + \varepsilon) \frac{\Delta_{indel}(X, Y)}{2} \right).$$

Observe that while plugging $\varepsilon \leq \frac{1}{n}$ we obtain a scaled isometry at the expense of a quadratic increase in the length of the second string. Conversely, for constant values of ε , N scales as $O(n)$ albeit with the trade-off of only approximately preserving distances within a constant factor. For intermediate values of ε , we can compromise between the accuracy and the stretch length.

Let us revisit Tiskin’s (section 6.1 of [68]) construction of the reverse embedding, namely, from Δ_{edit} into Δ_{indel} . The embedding proceeds as follows: a special character $\$$ is appended after every symbol of each string. It is easy to check that for each pair of strings, the Δ_{indel} between the embedded pair of strings equals twice the Δ_{edit} between the original pair.

In our construction, one string remains unaltered, while for the second string, we append after every symbol a block of length n consisting of the special character.

The core of the proof demonstrates the conversion of any Δ_{indel} -alignment for the input strings into an Δ_{edit} -alignment for the embedded strings, preserving the distances up to a scaling factor. This process involves replacing any deletions originally performed on the first string by substituting the characters with the special inserted character. Deletions made on the second string remain unaffected.

- This chapter presents some of the results from our paper [69]

5.1.1 Motivation

Introducing a Robust Concept of Approximation: Transitioning from Approximating Δ_{edit} into Approximating Δ_{indel}

Recall that one of the reasons we aimed to isometrically embed the Δ_{indel} metric into the Δ_{edit} metric stemmed from the abundance of approximation results for Δ_{edit} that might not easily extend to the Δ_{indel} metric. A natural approach, based on our embedding result, is to approximate the Δ_{indel} distance between X, Y by approximating the Δ_{edit} between the embedded strings. However, this is not an immediate consequence due to the substantial disparity in length between the embedded strings and the notion of approximation in this case, as detailed next:

Recall that in Theorem 5.1 the scaling mechanism is not normalized, i.e, the embedding function did not preserve normalized distances, but instead:

$$\Delta_{edit}(E_1(X), E_3(Y)) = N - n + k,$$

$$\text{where } k \in \left[\frac{\Delta_{indel}(X, Y)}{2}, \dots, (1 + \varepsilon) \frac{\Delta_{indel}(X, Y)}{2} \right]$$

where N, n are the length of the embedded strings, and $N = \Theta(\frac{n}{\varepsilon})$. Observe that the Δ_{edit} between the embedded strings lies in the range of $[N - n, N]$.

Considering the substantial difference in length between the embedded strings, an algorithm that consistently outputs the value $N - n$, regardless of the embedded strings, already yields a $1 + O(\varepsilon)$ -approximation for the distance between the embedded strings. Certainly, such an outcome provides no information about the Δ_{indel} of the original strings. Therefore, we introduce a more robust notion of approximation that generally addresses the discrepancy in string lengths:

Definition 5.2 (A Robust Notion of Approximation:). *Let $c > 1$, let Σ be a finite set, and let $X, Y \in \Sigma^*$. Define $|X| = N, |Y| = n$, and assume $N \geq n$. Define $k_{X,Y}$ such that: $\Delta_{edit}(X, Y) = N - n + k_{X,Y}$.*

An algorithm is considered to provide a robust c -approximation for Δ_{edit} if for all pairs X, Y it outputs k' such that: $k' \in [k_{X,Y}, ck_{X,Y}]$.

We assert that for any value of ε , any algorithm ALG that provides a robust c -approximation for Δ_{edit} yields an algorithm ALG' that provides $(1 + \varepsilon)c$ -approximation for Δ_{indel} . Moreover, if the running time ALG on input strings of lengths N, n is $t(N, n)$, then the running time of ALG' is $t(\frac{n}{\varepsilon}, n)$. The construction of ALG' is straightforward: on input strings X, Y we first apply the embedding, then apply ALG on the resulting strings and finally output: $2k'$.

We leave the quest of discovering a robust approximation algorithm for Δ_{edit} as an open question, which falls outside the scope of this thesis.

5.2 Embeddings between Indel distance metric and Edit distance metric

5.2.1 Tiskin’s embedding from Edit to Indel metric

Tiskin [68] (in section 6.1) first observed the existence of an embedding that maps strings from Σ^n to strings in $(\Sigma \cup \{\$\})^{2n}$. This embedding satisfies the property that for any $X, Y \in \Sigma^n$, we have $2\Delta_{edit}(X, Y) = \Delta_{indel}(E(X), E(Y))$. The embedding is straightforward: it involves appending a special character “\$” after every character in its input.

Let us delve into the intuition behind analyzing distance preservation. Given any optimal Δ_{edit} -alignment \mathcal{A} of X and Y , we can transform it into an Δ_{indel} -alignment \mathcal{A}' of $E(X)$ and $E(Y)$ effectively doubling its cost, as follows: if a character is deleted from either X or Y in \mathcal{A} , then the corresponding character along with the following \$ in $E(X)$ or $E(Y)$ are deleted in \mathcal{A}' . If $X[i]$ is substituted with $Y[j]$ in \mathcal{A} , then the corresponding characters: $E(X)[2i - 1]$ and $E(Y)[2j - 1]$ are deleted in \mathcal{A}' . Since each deletion or substitution in \mathcal{A} corresponds to deleting two characters in \mathcal{A}' , the cost of \mathcal{A}' is twice the cost of \mathcal{A} , i.e., $2\Delta_{edit}(X, Y)$. Although one must be careful, it is not difficult to prove that the resulted alignment \mathcal{A}' described above is optimal.

5.2.2 Indel to Edit Embeddings

Inspired by the embedding outlined earlier, which transforms the Δ_{edit} metric into the Δ_{indel} metric, this section introduces two separate embedding mappings that function conversely: from the Δ_{indel} metric to the Δ_{edit} metric.

In this section we introduce an embedding from the Δ_{indel} metric to the Δ_{edit} metric which is scaling isometric. Our embedding is asymmetric, implying that we embed each string in a different manner. We propose an embedding scheme that transforms strings X and Y of length n , into $E_1(X)$ and $E_2(Y)$, respectively of lengths n and $O(n^2)$. This scheme ensures that any optimal Δ_{indel} -alignment of X and Y corresponds to an optimal Δ_{edit} -alignment of $E_1(X)$ and $E_2(Y)$. The formal statement of this result is provided below.

Theorem 5.3. *For any alphabet Σ and integer $n > 0$, there exist $E_1 : \Sigma^n \rightarrow \Sigma^n$ and $E_2 : \Sigma^n \rightarrow \{\Sigma \cup \{\$\}\}^N$, where $N = \mathcal{O}(n^2)$, such that given strings $X, Y \in \Sigma^n$, we have $\Delta_{edit}(E_1(X), E_2(Y)) = N - n + \frac{\Delta_{indel}(X, Y)}{2}$.¹*

The core concept of our embedding revolves around defining E_1 as the identity function, while for $E_2(Y)$, appending the sequence $\n after each character in Y , including at the beginning, resulting in a length of $O(n^2)$ for $E_2(Y)$. This construction ensures that any optimal Δ_{indel} alignment of X and Y can be transformed into an Δ_{edit} alignment of $E_1(X)$ and $E_2(Y)$ while preserving matching characters, as elaborated below:

Given any optimal Δ_{indel} -alignment \mathcal{A} of X and Y , we can transform it into an Δ_{edit} -alignment \mathcal{A}' of $E_1(X)$ and $E_2(Y)$ as follows: If a character is deleted from

¹Our techniques can adapted easily to design embedding functions for variable length strings instead of length-preserving ones.

Y in \mathcal{A} , then the corresponding character along with the subsequent sequence of $\n in $E(Y)$ are deleted in \mathcal{A}' . If a character is deleted from X , then since each of the characters in $E(Y)$ is separated with the sequence $\n we can substitute the corresponding character in $E(X)$, with a $\$$ -symbol in $E(Y)$. This ensures that the characters of $E_1(X)$ are either matched or substituted. Consequently, we establish the optimality of this resulting Δ_{edit} alignment of $E_1(X)$ and $E_2(Y)$. The formal proof with all the details is in Section 5.2.3.

In the previous theorem, the length of one of the embedded strings grows quadratically with the input string's length due to appending $\n after each character in Y to form $E_2(Y)$. Now, a natural question arises: Can we reduce the length of the embedded string? While we currently lack knowledge of any embedding with a smaller output size which is scaling isometric, we can achieve significantly smaller embedded strings by approximately preserving the distances.

Essentially, instead of appending $\n , we can append $\k after each character in Y for $k \leq n$, resulting in a much smaller-sized embedding while maintaining distances up to a factor of $(1 + c/k)$, for some small constant $c \geq 1$. This is due to the claim that even with the smaller appending, one can still achieve a Δ_{edit} alignment of the embedded strings given an optimal Δ_{indel} alignment of the input strings, which preserves more than $(1 - \frac{c}{k})$ fraction of the matches. We formally state this approximate embedding below, with further details provided in Section 5.2.4.

Theorem 5.4 (Indel Into Edit Metrics Embedding - Approximate embedding). *For any alphabet Σ , $n \in \mathbb{N}$ and $\varepsilon \in (0, 1]$, there exist mappings $E_1 : \Sigma^n \rightarrow \Sigma^n$ and $E_3 : \Sigma^n \rightarrow (\Sigma \cup \{\$\})^N$, where $N = \Theta(n/\varepsilon)$, such that for any $X, Y \in \Sigma^n$, we have*

$$\Delta_{edit}(E_1(X), E_3(Y)) = N - n + k$$

$$\text{where } k \in \left[\frac{\Delta_{indel}(X, Y)}{2}, (1 + \varepsilon) \frac{\Delta_{indel}(X, Y)}{2} \right).$$

5.2.3 Obtaining Scaling Isometric Embedding

In this section, we describe the embedding functions E_1 and E_2 for the scaling isometric embedding from Δ_{indel} metric to Δ_{edit} metric, followed by the proof of Theorem 5.3.

The following facts, which we state without a proof, will be helpful in the proof of Theorem 5.3 and 5.4.

Fact 5.5. *For any Δ_{indel} alignment of strings X and Y , where $|X| = |Y|$, the number of deletions in X is equal to the number of deletions in Y which is equal to $\frac{\Delta_{indel}(X, Y)}{2}$.*

Fact 5.6. *For any optimal Δ_{edit} alignment of strings X and Y , where $|X| \leq |Y|$, we have $\Delta_{edit}(X, Y) = \#deletions \text{ in } X + \#deletions \text{ in } Y + \#substitutions = (|Y| - |X|) + 2 \times \#deletions \text{ in } X + \#substitutions$.*

Fact 5.7. *Let Σ, Σ' be alphabets, where $\Sigma \subseteq \Sigma'$ and strings $X, Y \in \Sigma^n$, $Y' \in \Sigma'^N$, where $N \geq n$. If Y' is obtained from Y by inserting characters from $\Sigma' \setminus \Sigma$, then $\Delta_{edit}(X, Y') \geq N - n + \frac{\Delta_{indel}(X, Y)}{2}$.*

Description of the embedding functions E_1 and E_2

We define the embedding function E_2 for strings of length n . Given a string $Y \in \Sigma^n$, $E_2(Y)$ is obtained by inserting the sequence $\n after each symbol in Y and also at the beginning. This yields in $E_2(Y) = \$^n \cdot Y[1] \cdot \$^n \cdot Y[2] \cdot \$^n \dots Y[n] \n , where $\n denotes a sequence of n dollar signs “\$”. Essentially, $E_2(Y)[i(n+1)] = Y[i]$ for all $1 \leq i \leq n$, and the remaining positions in $E_2(Y)$ are filled with “\$”. The length of the transformed string $E_2(Y)$ is $N' = n(n+1) + n = n^2 + 2n$.

Regarding E_1 , it functions as the identity function. Therefore, $E_1(X)$, simply remains the same as X , thus $|E_1(X)| = |X| = n$.

Proof of Theorem 5.3

Given strings $X, Y \in \Sigma^n$, henceforth we will denote $E_1(X)$ simply as X and $E_2(Y)$ as Y' . To prove Theorem 5.3, we demonstrate that we can construct a Δ_{edit} alignment of X and Y' with a cost of $N' - n + \frac{\Delta_{indel}(X, Y)}{2}$ given any optimal Δ_{indel} alignment of X and Y with a cost of $\frac{\Delta_{indel}(X, Y)}{2}$.

Subsequently, we establish that the constructed Δ_{edit} alignment is indeed optimal. This process involves defining a decomposition for X and Y based on their Δ_{indel} alignment, which will then be utilized to construct the required Δ_{edit} alignment for X and Y' .

Decomposition of X and Y : We start by defining blocks for X and Y given a Δ_{indel} alignment \mathcal{A} of X and Y . We partition X into blocks consisting of contiguous substrings. Each block, with the exception of the first one, starts with a contiguous matching segment followed by a contiguous deletion segment. Some blocks may have an empty deletion segment. The initial block does not contain any matching segment. These matching and deletion segments determined based on the Δ_{indel} alignment \mathcal{A} . The decomposition of Y follows a similar procedure. Left hand side of Figure 5.1 and Figure 5.2 shows the decomposition according to some optimal Δ_{indel} alignment. We formally articulate this observation below.

Observation 5.8. *For any Δ_{indel} alignment \mathcal{A} of $X, Y \in \Sigma^n$, we can partition X and Y into disjoint blocks based on \mathcal{A} , such that:*

1. *The number of blocks in X and Y are equal. Hence, $X = b_0^X \cdot b_1^X \cdot b_2^X \dots b_t^X$ and $Y = b_0^Y \cdot b_1^Y \cdot b_2^Y \dots b_t^Y$, where b_i^X are blocks of X and b_i^Y are blocks of Y .*
2. *$b_0^X = d_0^X$ and $b_0^Y = d_0^Y$, which may be empty.*
3. *For each $i > 0$, block b_i^X consists of a contiguous matching part m_i^X followed by a contiguous deletion part d_i^X i.e. $b_i^X = m_i^X \cdot d_i^X$. Similarly, for each block b_i^Y of Y we have, $b_i^Y = m_i^Y \cdot d_i^Y$. This means that in the alignment \mathcal{A} , the characters in m_i^X and m_i^Y are getting matched and characters in d_i^X and d_i^Y are getting deleted.*
4. *For each $i > 0$, $m_i^X = m_i^Y$.*
5. *For each $i > 0$, m_i^X is non-empty and d_i^X can be empty. The same applies for the blocks of Y .*

To define the necessary Δ_{edit} alignment between X and Y' , we begin by establishing a decomposition for both X and Y' . This decomposition is derived from an optimal Δ_{indel} alignment of X and Y . Let $\mathcal{I}^{X,Y}$ be an optimal Δ_{indel} alignment of X and Y .

Decomposition of X and Y' : According to Observation 5.8, we have a decomposition of X and Y w.r.t $\mathcal{I}^{X,Y}$. Let $X = b_0^X \cdot b_1^X \cdot b_2^X \cdots b_l^X$ and $Y = b_0^Y \cdot b_1^Y \cdot b_2^Y \cdots b_l^Y$.

Decomposition of X remains the same. We define the blocks of Y' as $b_1^{Y'}, b_2^{Y'}, \dots, b_l^{Y'}$, where $b_i^{Y'} = m_i^{Y'} \cdot c'_i \cdot d_i^{Y'}$, with $c'_i = \n . If $m_i^Y = Y[p, q]$, then $m_i^{Y'} = Y'[p(n+1), q(n+1)]$ which is simply $Y[p] \cdot \$^n \cdot Y[p+1] \cdot \$^n \cdots Y[q]$. Similarly, if $d_i^Y = Y[p, q]$, then $d_i^{Y'} = Y'[p(n+1), q(n+1) + n]$ which is $Y[p] \cdot \$^n \cdot Y[p+1] \$^n \cdots Y[q] \cdot \n (note the extra $\n at the end of $d_i^{Y'}$ which is absent in $m_i^{Y'}$). Therefore, $Y' = \$^n \cdot b_0^{Y'} \cdot b_1^{Y'} \cdot b_2^{Y'} \cdots b_l^{Y'}$. Reader can refer to Figure 5.1 for more clarity.

Based on this decomposition of X and Y' , we are now ready to define our Δ_{edit} alignment of X and Y' .

Δ_{edit} alignment of X and Y' : We proceed by defining a Δ_{edit} alignment \mathcal{A}' of X and Y' using the previously described blocks from X and Y' . Initially, we align the block b_0^X with the initial $\n block, where each character of b_0^X undergo substitution. Subsequently, for each block b_i^X where $i > 0$, we align m_i^X with $m_i^{Y'}$, ensuring that all characters in m_i^X are matched. Moreover, each d_i^X is aligned with c'_i , resulting in substitutions for each character in d_i^X . For a visual representation, please refer to Figure 5.1. Here, we observe that d_0^X aligns with the initial $\n block, each m_i^X aligns with $m_i^{Y'}$, and d_2^X and d_3^X align with c_2 and c_3 in Y' . Finally, we compute the cost of the alignment \mathcal{A}' .

Claim 5.9. $cost(\mathcal{A}') = N - n + \frac{\Delta_{indel}(X,Y)}{2}$.

Proof. Since, $|Y'| > |X|$, therefore there must be at least $|Y'| - |X| = N - n$ many deletions in any alignment of X and Y' .

In the optimal Δ_{indel} alignment $\mathcal{I}^{X,Y}$ of X and Y , we have $m_i^X = m_i^Y$, as noted in Observation 5.8. Since m_i^Y is a proper subsequence of $m_i^{Y'}$ from our embedding and decomposition, it follows that m_i^X is also a proper subsequence of $m_i^{Y'}$. Therefore, all characters in m_i^X match when aligned with $m_i^{Y'}$. Furthermore, the characters of d_i^X align with “\$” characters in c'_i , contributing to $|d_i^X|$ substitutions. Hence, the total number of substitutions is given by $\sum_{i=0}^l |d_i^X| = \frac{\Delta_{indel}(X,Y)}{2}$, using Fact 5.5. Since, in the Δ_{edit} alignment, there are only matches and substitutions in X and all deletions occur only in Y' , therefore, $cost(\mathcal{A}') = N - n + \frac{\Delta_{indel}(X,Y)}{2}$. \square

Let $\mathcal{E}^{X,Y'}$ be an optimal Δ_{edit} alignment of X and Y' and the cost of the alignment is $cost(\mathcal{E}^{X,Y'})$, i.e., $cost(\mathcal{E}^{X,Y'}) = \Delta_{edit}(X, Y')$. We will now demonstrate that \mathcal{A}' is an optimal Δ_{edit} alignment, i.e. $cost(\mathcal{A}') = cost(\mathcal{E}^{X,Y'})$.

Claim 5.10. $cost(\mathcal{E}^{X,Y'}) = cost(\mathcal{A}')$.

Proof. It is clear that $cost(\mathcal{E}^{X,Y'}) \leq cost(\mathcal{A}')$, because $\mathcal{E}^{X,Y'}$ is an optimal Δ_{edit} alignment and \mathcal{A}' is an Δ_{edit} alignment.

Let us assume, for the sake of contradiction, that $cost(\mathcal{E}^{X,Y'}) < cost(\mathcal{A}')$. In the alignment \mathcal{A}' , the characters of X are either matched or substituted. Therefore, the only way to achieve a better alignment than \mathcal{A}' is by increasing the number

of matches in X . However, matches in X and Y' can only occur with characters that are not “\$”. This implies that such matches would also exist between X and Y , which contradicts the optimality of the Δ_{indel} alignment of X and Y . Hence, we establish that $\text{cost}(\mathcal{E}^{X,Y'}) \geq \text{cost}(\mathcal{A}')$. Therefore, $\text{cost}(\mathcal{E}^{X,Y'}) = \text{cost}(\mathcal{A}')$. \square

Proof of Theorem 5.3. Using claim 5.9 and claim 5.10, we can prove the theorem. \square

Remark. *If there exists an Δ_{indel} alignment of X and Y , where the size of each deletion segment of X , i.e., $|d_i^X|$, for all $1 \leq i \leq l$ is bounded by some threshold t , then we can get an embedding of size $\mathcal{O}(nt)$.*

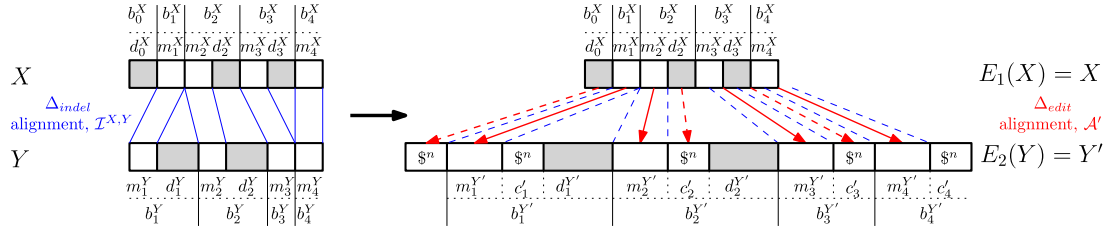


Figure 5.1 On the left side, we have the decomposition of X and Y based on the Δ_{indel} alignment. On the right side, we see the decomposition and alignment of X and Y' following our construction in Section 5.2.3. The solid red arrows indicate that all characters of m_i^X are matched, while the dotted red arrows suggest that the characters of m_i^X are substituted. The shaded cells in gray indicate deletions. On the right-hand side, the blue dotted lines indicate the alignment of m_i^X with $m_i^{Y'}$.

5.2.4 Obtaining Approximate Scaling Isometric Embedding

In this section, we introduce the embedding function E_3 and provide a proof for Theorem 5.4. While the embedding described here bears resemblance to the previous one, for the sake of completeness, we present all the details below:

Description of the function E_3

Given a string Y of length n in the alphabet Σ , we define E_3 similar to E_2 as described in Section 5.2.3. Given a parameter $0 < \varepsilon \leq 1$, we define $k = \frac{4}{\varepsilon}$.² However, for the mapping E_3 , we append $\k after every character in Y instead of appending $\n . Additionally, we append $\n at the beginning and at the end as well. Consequently, $E_3(Y)$ is represented as:

$$E_3(Y) = \$^n \cdot Y[1] \cdot \$^k \cdot Y[2] \cdot \$^k \cdot Y[3] \cdot \$^k \dots \$^k \cdot Y[n] \cdot \$^k \cdot \$^n.$$

Basically, $E_3(Y)[n + (i - 1)(k + 1) + 1] = Y[i]$ for all $1 \leq i \leq n$, and rest of the positions in $E_3(Y)$ are filled with “\$”. The length of $E_3(Y)$ is calculated as $n(k + 1) + 2n = N$. Therefore, for a fixed value of ε , we achieve a linear size embedding instead of the quadratic embedding previously.

²It’s worth noting that k must be an integer. There are two approaches to ensure this: either we choose ε such that k is an integer, or we use $k \lceil \frac{4}{\varepsilon} \rceil$. However, in the latter case, additional attention is required for the calculations, although the method remains valid.

Proof of Theorem 5.4

Given $X, Y \in \Sigma^n$, from now on we denote $E_3(Y)$ by \tilde{Y} and as in the previous embedding in Section 5.2.3, we denote $E_1(X)$ by X . The proof proceeds as follows: Let us consider an optimal Δ_{indel} alignment $\mathcal{I}^{X,Y}$ of X and Y . We aim to construct a Δ_{edit} alignment $\tilde{\mathcal{A}}$ of strings X and \tilde{Y} based on the Δ_{indel} alignment $\mathcal{I}^{X,Y}$, which preserves “most” of the matches. Subsequently, we will bound the $cost(\tilde{\mathcal{A}})$ to show that we can approximately preserve the distances, as stated in Theorem 5.4.

Similar to the proof of Theorem 5.3, here also we will utilize the decomposition of X and Y given the optimal Δ_{indel} alignment $\mathcal{I}^{X,Y}$. We will employ the decomposition from Section 5.2.3. Let $X = b_0^X \cdot b_1^X \cdot b_2^X \cdots b_l^X$ and $Y = b_0^Y \cdot b_1^Y \cdot b_2^Y \cdots b_l^Y$. We will decompose X and \tilde{Y} based on the decomposition of X and Y . The resulting decomposition of X and \tilde{Y} will then be used to establish the Δ_{edit} alignment $\tilde{\mathcal{A}}$.

Decomposition of X and \tilde{Y} : Now, let us define the blocks of X and \tilde{Y} based on the obtained blocks of X and Y . Blocks of X remain the same.

Let blocks of \tilde{Y} be $b_0^{\tilde{Y}}, b_1^{\tilde{Y}}, b_2^{\tilde{Y}}, \dots, b_l^{\tilde{Y}}$, where $b_0^{\tilde{Y}} = d_0^{\tilde{Y}}$ and for all $i > 0$, $b_i^{\tilde{Y}} = m_i^{\tilde{Y}} \cdot \tilde{c}_i \cdot d_i^{\tilde{Y}}$, $\tilde{c}_i = \k . If $m_i^Y = Y[p, q]$, then $m_i^{\tilde{Y}} = \tilde{Y}[n + (p - 1)(k + 1) + 1, n + (q - 1)(k + 1) + 1]$, which is $Y[p] \cdot \$^k \cdot Y[p + 1] \$^k \cdots \$^k \cdot Y[q]$. If $d_i^Y = Y[p, q]$, then $d_i^{\tilde{Y}} = \tilde{Y}[n + (p - 1)(k + 1) + 1, n + (q - 1)(k + 1) + 1 + k]$, which is $Y[p] \cdot \$^k \cdot Y[p + 1] \$^k \cdots \$^k \cdot Y[q] \cdot \k (note the extra $\k at the end of $d_i^{\tilde{Y}}$ which is missing in $m_i^{\tilde{Y}}$). Therefore, $\tilde{Y} = \$^n \cdot b_0^{\tilde{Y}} \cdot b_1^{\tilde{Y}} \cdot b_2^{\tilde{Y}} \cdots b_l^{\tilde{Y}} \cdot \n . Refer to Figure 5.2 for an example.

Δ_{edit} alignment of X and \tilde{Y} : Given the decomposition of X and \tilde{Y} , we can now outline our approach for constructing the Δ_{edit} alignment $\tilde{\mathcal{A}}$. The fundamental principle of our alignment strategy is to sequentially align the characters of X with those of \tilde{Y} from left to right, while carefully accounting for substitutions and deletions.

We initiate the alignment process by aligning b_0^X with the initial $\n in \tilde{Y} . Since the length of b_0^X is at most n , and none of its characters are “\$”, we perform substitutions for all characters in m_0^X .

For each subsequent block $b_i^X = m_i^X \cdot d_i^X$, from left to right, starting from block 1, we first attempt to align m_i^X with $m_i^{\tilde{Y}}$, matching as many characters of m_i^X as possible and deleting any unmatched characters in m_i^X .

Following this, we align the characters of d_i^X from left to right with the unaligned characters of \tilde{Y} , beginning from the leftmost unaligned character in \tilde{Y} . All the steps are detailed in Algorithm 16.

Algorithm 16 Getting an alignment of X and \tilde{Y} wrt edit distance metric:

Data: $X = m_0^X \cdot b_1^X \cdots b_l^X$, where $b_i^X = m_i^X \cdot d_i^X$ and $\tilde{Y} = \$^n \cdot d_0^{\tilde{Y}} \cdot b_1^{\tilde{Y}} \cdots b_l^{\tilde{Y}} \cdot \n ,
 where $b_i^{\tilde{Y}} = m_i^{\tilde{Y}} \cdot \tilde{c}_i \cdot d_i^{\tilde{Y}}$

Result: An alignment of X and \tilde{Y}

```

178 Align  $m_0^X$  to the initial  $\$^n$ ;
179  $j = 1$ ;
180 for  $i = 1 \cdots l$  do
181   if  $i \geq j$  then
182     if  $i > j$  then
183       /*  $m_i^{\tilde{Y}}$  is fully available. */
184       Align  $m_i^X$  to  $m_i^{\tilde{Y}}$ ;
185     end
186     if  $i = j$  then
187       /*  $m_i^{\tilde{Y}}$  might be partially available or not available. */
188       Align  $m_i^X$  to  $m_i^{\tilde{Y}}$  matching the maximum possible number of characters
189       of  $m_i^X$ ;
190       Delete the characters in  $m_i^X$  which cannot be matched;
191     end
192     Align the characters of  $d_i^X$  from left to right to the unaligned characters of
193      $\tilde{Y}$ , starting from  $\tilde{c}_i$  in  $\tilde{Y}$ ;
194   end
195   if  $i < j$  then
196     /*  $m_i^{\tilde{Y}}$  is not available. */
197     Delete all the characters in  $m_i^X$ .
198     Align the characters of  $d_i^X$  from left to right to the unaligned characters of
199      $\tilde{Y}$ , starting from leftmost unaligned character in  $\tilde{Y}$ ;
200   end
201   Update  $j$ , such that the leftmost unaligned character of  $\tilde{Y}$  is in block  $b_j^{\tilde{Y}}$ ;
202 end

```

Let's initiate the analysis of the Δ_{edit} alignment $\tilde{\mathcal{A}}$. First, we examine the alignment of m_i^X , and then we proceed to analyze the alignment of d_i^X for all $1 \leq i \leq l$.

While attempting to align m_i^X to $m_i^{\tilde{Y}}$, we encounter three possible scenarios regarding the alignment status of $m_i^{\tilde{Y}}$. The three scenarios are named as **fully available**, **partially available**, and **not available**. Depending on these three cases we determine how to align the characters of m_i^X :

1. **fully available:** $m_i^{\tilde{Y}}$ falls into this category if all its characters are available for alignment. In other words, none of the characters from any m_j^X , where $j < i$, have been aligned with characters from $m_i^{\tilde{Y}}$. In this scenario, we align m_i^X to $m_i^{\tilde{Y}}$, matching each character of m_i^X . This alignment is feasible because $m_i^X = m_i^Y$, and m_i^Y is a proper subsequence of $m_i^{\tilde{Y}}$ from our construction and the decomposition of X and \tilde{Y} . Thus m_i^X is also a proper subsequence of $m_i^{\tilde{Y}}$. As depicted in Figure 5.2, on the right-hand side, $m_1^{\tilde{Y}}$

and $m_5^{\tilde{Y}}$ are fully available, enabling us to match all the characters of m_1^X and m_5^X .

2. **partially available:** If some, but not all, characters from $m_i^{\tilde{Y}}$ are available for alignment, we consider it partially available. This occurs when certain characters from some d_j^X , where $j < i$, have been aligned with characters from $m_i^{\tilde{Y}}$. In this case, we align m_i^X to $m_i^{\tilde{Y}}$ and match as many characters of m_i^X as possible, deleting the rest from m_i^X . This is same as matching the largest suffix of m_i^X which appears as a proper subsequence of the remaining unaligned suffix of $m_i^{\tilde{Y}}$. The size of this largest suffix of m_i^X is exactly the number of non-\$ characters that are unaligned in $m_i^{\tilde{Y}}$.

More technically, let $m_i^{\tilde{Y}} = \tilde{Y}[p, q]$. Since, $m_i^{\tilde{Y}}$ is partially available, therefore, the leftmost unaligned character in \tilde{Y} is in $m_i^{\tilde{Y}}$. Let $\tilde{Y}[p']$ be the leftmost unaligned character in \tilde{Y} . So, $\tilde{Y}[p', q]$ is available for alignment. Now, we need to find the largest suffix of m_i^X that can match into $\tilde{Y}[p', q]$. For that let's count the number of non-\$ characters in $\tilde{Y}[p', q]$, which is $\lfloor \frac{(q-p')}{k+1} \rfloor + 1$. The extra 1 non-\$ character is $\tilde{Y}[q]$ because the last character of every $m_j^{\tilde{Y}}$ is non-\$. Now, we match the last $\lfloor \frac{(q-p')}{k+1} \rfloor + 1$ many characters of m_i^X and delete the rest.

In Figure 5.2, $m_2^{\tilde{Y}}$ is partially unaligned, resulting in two missed matches in m_2^X , with only the last character of m_2^X being matched.

3. **not available:** $m_i^{\tilde{Y}}$ is not available if none of its characters are available for alignment. In this case, all characters from some m_j^X , where $j < i$, have been aligned with all characters from $m_i^{\tilde{Y}}$. Here, we cannot match any character of m_i^X , therefore, we delete the entire m_i^X . We again refer to Figure 5.2, where $m_3^{\tilde{Y}}$ and $m_4^{\tilde{Y}}$ are not available, resulting in the failure to match any characters from m_3^X and m_4^X .

Let us now examine the alignment of d_i^X . As stated earlier, the characters of d_i^X are aligned from left to right with the unaligned characters of \tilde{Y} , beginning from the leftmost unaligned character in \tilde{Y} . The characters in d_i^X either get matched or substituted but are not deleted. During this process, it's possible that we align characters of d_i^X with characters in $m_j^{\tilde{Y}}$ for some $j > i$. Consequently, certain characters in $m_j^{\tilde{Y}}$, which are not "\$", become unavailable for matching with the corresponding characters in m_j^X , resulting in missed matches. These missed matching characters in X may be deleted from X during our alignment. Our goal is to quantify the number of deletions in X , which is exactly equal to the number of missed matches caused by aligning the characters of d_i^X for all $i > 0$ to the matching segments of \tilde{Y} , i.e., $m_j^{\tilde{Y}}$ for $j > i$. For each $1 \leq i \leq l$, let S_i denote the total number of characters from d_i^X that gets aligned with characters from $m_j^{\tilde{Y}}$ for all $j > i$.

In order to analyze the $cost(\tilde{\mathcal{A}})$, it's crucial to count the number of deletions in X and \tilde{Y} , along with the number of substitutions. The number of deletions in \tilde{Y} and substitutions concerning the alignment $\tilde{\mathcal{A}}$ are straightforward to analyse, as discussed in the proof of Claim 5.13.

for alignment and the leftmost unaligned character can be non- $\$$ character. Considering that every $k + 1$ characters in \tilde{Y} have at most one character which is not “ $\$$ ”, S_i is bounded by $\lceil \frac{|d_i^X|}{k+1} \rceil$.

From the above analysis of the alignment of d_i^X , we have the following claim.

Claim 5.11. *For each i , $S_i \leq \lceil \frac{|d_i^X|}{k+1} \rceil$.*

Using the above claim and the observation that the total number of deletions in X is exactly $\sum_{i=1}^l S_i$, we can now prove the following claim.

Claim 5.12. *For the alignment $\tilde{\mathcal{A}}$, the total number of deletions in $X = \sum_i S_i < \frac{1}{k}(\Delta_{indel}(X, Y))$.*

Proof. Given that the number of deletions in X is $\frac{\Delta_{indel}(X, Y)}{2}$ (as per Fact 5.5), by Claim 5.11, we have $\sum_i S_i \leq \sum_i \lceil \frac{|d_i^X|}{k+1} \rceil < \sum_i \frac{2|d_i^X|}{k} \leq \frac{\Delta_{indel}(X, Y)}{k} = \frac{\Delta_{indel}}{4} \times \varepsilon$, given $k = \frac{4}{\varepsilon}$ for any $0 < \varepsilon \leq 1$. \square

We can finally bound the cost of alignment $\tilde{\mathcal{A}}$ in the following claim.

Claim 5.13. $\Delta_{edit}(X, \tilde{Y}) \leq cost(\tilde{\mathcal{A}}) < \tilde{N} - n + \frac{\Delta_{indel}}{2} + \frac{\Delta_{indel}}{2} \varepsilon$.

Proof. Since $\tilde{\mathcal{A}}$ is an Δ_{edit} alignment of X and \tilde{Y} , therefore $cost(\tilde{\mathcal{A}})$ is less than the cost of an optimal Δ_{edit} alignment of X and \tilde{Y} , which is $\Delta_{edit}(X, \tilde{Y})$.

Using Fact 5.6, we know that $\Delta_{edit}(X, \tilde{Y}) = \#\text{deletions in } X + \#\text{deletions in } \tilde{Y} + \#\text{substitutions} = (\tilde{N} - n) + 2 \times \#\text{deletions in } X + \#\text{substitutions}$.

Since, all the substitutions are in d_i^X for all $0 \leq i \leq l$, therefore, $\#\text{substitutions} \leq \sum_i |d_i^X| = \frac{\Delta_{indel}(X, Y)}{2}$. Applying the bounds for $\#\text{deletions in } X$ from Claim 5.12, we get $\Delta_{edit}(X, \tilde{Y}) < \tilde{N} - n + \frac{\Delta_{indel}}{2} + \frac{\Delta_{indel}}{2} \varepsilon$. \square

Proof of Theorem 5.4. From Fact 5.7, we have $\Delta_{edit}(X, \tilde{Y}) \geq \tilde{N} - n + \frac{\Delta_{indel}(X, Y)}{2}$, as \tilde{Y} is obtained from Y by inserting special character “ $\$$ ” which is not in Σ . From Claim 5.13, we have $\Delta_{edit}(X, \tilde{Y}) < \tilde{N} - n + (1 + \varepsilon) \frac{\Delta_{indel}(X, Y)}{2}$. \square

5.3 Summary

In this chapter, we investigated the relationship between the Δ_{indel} and Δ_{edit} distance metrics through the lens of metric embeddings. While previous work by Tiskin established a scaled isometric embedding from the Δ_{edit} metric into the Δ_{indel} metric by inserting special characters, our focus was on the reverse direction—embedding Δ_{indel} into Δ_{edit} .

We proposed two main embedding constructions:

1. Exact Embedding (Theorem 5.3):

We presented a scaling isometric and asymmetric embedding from the Δ_{indel} metric into the Δ_{edit} metric. In this construction, one string remains unchanged while the other is expanded quadratically in length by appending blocks of a special character after each symbol. This ensures that the Δ_{edit} distance between the embedded strings exactly reflects the Δ_{indel} distance between the originals, up to a scaling factor and additive term depending on the length.

2. Approximate Embedding (Theorem 5.4):

To address the quadratic blowup, we introduced an approximate embedding with tunable accuracy via a parameter ε . By appending shorter blocks of special characters, we achieved a linear-sized embedding that approximates the Δ_{indel} distance within a $(1 + \varepsilon)$ factor, with the embedding size scaling as $\Theta(n/\varepsilon)$.

Additionally, we motivated the need for a more nuanced definition of approximation to account for length discrepancies in embedded strings. We introduced a *robust approximation* framework tailored for this setting and showed that any robust approximation algorithm for Δ_{edit} implies an approximation algorithm for Δ_{indel} via our embedding.

Together, these contributions deepen the understanding of structural connections between Δ_{edit} and Δ_{indel} , and open pathways for transferring algorithmic insights and hardness results across the two metrics.

6

Alphabet Reduction

“You fail to recognize that it matters not what someone is born, but what they grow to be!”
— *Dumbledore*

6.1 Introduction

This chapter investigates techniques for reducing the alphabet size of strings while approximately, or in some cases exactly preserving distances under metrics such as Δ_{indel} and Δ_{edit} . Our central motivation is to understand whether strings over a general alphabet can be embedded into strings over a smaller, possibly binary, alphabet such that their pairwise distances are well-preserved.

Previously, in Chapter 4, Section 4.3.1, we introduced an alphabet reduction technique in the context of the Δ_{edit} metric, under the assumption that the alignment must conform to a specific structure. In contrast, the reductions presented in this chapter make no assumptions about the alignment type, thus addressing a more general and challenging setting.

We formally ask:

Can strings over an arbitrary alphabet be embedded into binary strings in a way that approximately preserves distances such as Δ_{edit} or Δ_{indel} ?

More precisely, we aim to design an embedding $E : \Sigma^n \rightarrow \{0, 1\}^m$ such that for all $x, y \in \Sigma^n$,

$$\Delta_{edit}(x, y) \approx \Delta_{edit}(E(x), E(y)),$$

with a central goal of minimizing the output length m , while allowing small distortions (or no distortion) in the distances.

The motivation for such embeddings arises from two complementary directions:

1. **Algorithmic Applications:** Many algorithmic problems involving Δ_{edit} or Δ_{indel} become significantly easier over small, especially binary, alphabets. An effective alphabet reduction would allow us to leverage existing tools and algorithms designed for binary alphabets, applying them to strings over larger alphabets without the need to redesign solutions from scratch.

Such embeddings could thus serve as a bridge, enabling transfer of efficient techniques across alphabet sizes and potentially shedding light on what structural properties make large-alphabet spaces harder to work with.

2. **Lower Bounds:** On the flip side, proving lower bounds for problems like Δ_{edit} and Δ_{indel} is often easier over larger alphabets. If we can embed large alphabet strings into binary strings while preserving distances, then lower bounds obtained in the large-alphabet setting could carry over to the binary case, thereby strengthening the results. For instance, in the breakthrough result by [5], it was shown that computing the exact edit distance in truly subquadratic time is unlikely under the Strong Exponential Time Hypothesis (SETH), but the result was initially proved for strings over large alphabets. It was only later extended to binary strings in [70]. If an optimal isometric embedding into binary strings had existed, such lower bounds would have followed immediately.

This chapter presents new results that address the above question. Some of these results are drawn from our paper [69], and they explore both theoretical limitations and constructive embeddings that enable effective alphabet reduction.

6.1.1 Our Results

In the sequel, we utilize the notation Δ_{indel} to represent the Indel distance between a pair of strings, Δ_{edit} to denote their edit distance. Additionally, we use $\tilde{\Delta}_{indel}$ and $\tilde{\Delta}_{edit}$ to represent the normalized Indel distance between a pair of strings (for precise definitions, refer to Section 6.1.2).

Alphabet Reduction - Succinct Embedding:

Our first result pertains to alphabet reduction achieving normalized scaled isometric embedding. In this context, as elaborated in Section 6.2, any embedding contracts the normalized distances of some of the pairs. Consequently, we shift our focus to approximate normalized scaled isometric embedding, where we permit slight distortions in the normalized distances. Our main result is outlined below:

Theorem 6.1 (Alphabet Reduction - Succinct Embedding). *Let Γ be a finite alphabet, and let $0 < \varepsilon < 1/4$. There exists an alphabet Σ , where $|\Sigma| = \mathcal{O}(\frac{1}{\varepsilon^2})$ and there exists $E : \Gamma^* \rightarrow \Sigma^*$ satisfying:*

$$\forall X, Y \in \Gamma^* : \tilde{\Delta}_{indel}(E(X), E(Y)) \in \left[(1 - \varepsilon) \tilde{\Delta}_{indel}(X, Y), \tilde{\Delta}_{indel}(X, Y) \right].$$

Moreover, for every $X \in \Gamma^n$ we have: $|E(X)| = \mathcal{O}(n \log(|\Gamma|))$.

Observe that embedded string length is optimal up to logarithmic factors. The size of the alphabet Σ must exceed $1/\varepsilon$, as otherwise, by a claim proved later, there will be a pair of strings whose distance will be contracted by at least $(1 - \frac{1}{|\Sigma|})$ -factor.

As a result, we find that when focusing on Δ_{indel} approximation, we can, without loss of generality, limit our scope to a constant alphabet size without significantly impacting the quality of the approximation. This is captured in the following corollary which we provide without a proof.

Corollary 6.2. *Let Γ be a finite alphabet, and let $0 < \varepsilon < 1/4$. Suppose that there exists an algorithm ALG that provides a c -factor approximation for the Δ_{indel} metric for any pairs of strings in Σ of total length N , where $|\Sigma| = \mathcal{O}(1/\varepsilon^2)$, running in time $t(N)$.*

Then there exists an algorithm ALG' that, given any pair of string in Γ of total length n , provides a $c + \varepsilon$ -approximation for their Δ_{indel} distance in time $t(n \log |\Gamma|)$.

The proof strategy of Theorem 6.1 is as follows: Initially, we construct an error-correcting code within the smaller alphabet Σ , where $|\Sigma| = \mathcal{O}(1/\varepsilon^2)$. This code ensures that every distinct pair of codewords shares only a small portion of LCS, indicating a significant Δ_{indel} between them. The code comprises $|\Gamma|$ words, with its dimension being logarithmic in the size of Γ . We interpret this code as a mapping from characters within Γ to short strings in Σ . The existence of such a code can be demonstrated using the probabilistic method. The construction is almost tight in terms of the smaller alphabet size: it is not hard to show that for any code $C \subseteq \Sigma^k$, if $|C| > |\Sigma|$, then there exist $c \neq c' \in C$, satisfying: $\Delta_{\text{indel}}(c, c') < (1 - \frac{1}{|\Sigma|})2k$.

Employing the code construction, we embed input strings in a straightforward and natural manner: Consider a string residing in Γ^* , then each of its characters is sequentially encoded using the code. This encoding ensures that identical characters are mapped to the same codeword, while the code's distance guarantees that distinct characters may have only a few shared matches, akin to the global nature of Δ_{indel} alignments. If there were no matches between distinct characters' encodings, any alignment for the embedded strings could be converted into an alignment between the input strings without any distortion in the normalized costs. However, these few matches between non-matching codewords introduce a slight distortion and complicate the proof. In subsection 6.2.1 we outline our findings regarding normalized scaled alphabet reductions, covering both lower and upper bounds. Section 6.2.2 discusses our upper bounds, while Section 6.2.3 addresses lower bounds.

Alphabet Reduction - Binary Alphabets:

Our previous method relied on a local approach, wherein the characters of the string from the large alphabet were encoded sequentially and independently. It appears that such a local strategy may not result in a normalized scaled isometric embedding into a small, particularly binary alphabet. As codes in such alphabets have a relative distance of at most $1/2$, and this distance affects the distortion of the normalized distances. Therefore, achieving alphabet reduction into binary alphabets requires a scaling that is not normalized, as well as a more global approach that does not encode the characters sequentially and independently. However, there's a caveat: the encoding still needs to be performed independently on each string rather than on a pair of strings. Specifically, if we take a particular string X its encoding will remain the same regardless of the second string Y . This aspect forms the focal point of our forthcoming result.

For clarity, we present our results for the Δ_{indel} metric, with a similar approach applicable to the Δ_{edit} metric. Our main result states that one can embed Δ_{indel} over any alphabet Σ into binary alphabet. We employ asymmetric embedding and prove that the distances are preserved up to some scaling function. The dimension of the embedded strings is quasi-polynomial, making this result more of a proof

of concept at the moment. Nonetheless, we find it conceptually intriguing and pose the question of decreasing the target dimension as an open question. Our main result is as follows:

Theorem 6.3 (Informal statement of Theorem 6.13). *For any alphabet Σ and for every $n \in \mathbb{N}$ there exist functions $G, H : \Sigma^n \rightarrow \{0, 1\}^N$, $f : [n] \rightarrow [N]$ where $N = n^{\mathcal{O}(\log n)}$ such that for any $X, Y \in \Sigma^n$,*

$$\Delta_{\text{indel}}(G(X), H(Y)) = f(\Delta_{\text{indel}}(X, Y)).$$

Our initial consideration revolves around the fact that deciding whether $\Delta_{\text{indel}}(X, Y) \leq k$ for two strings $X, Y \in \Sigma^n$ and a threshold parameter k , can be accomplished by a Turing machine utilizing $\log(n)$ -space. This capability can then be translated into a formula of $\log^2(n)$ -depth.

The foundation of our construction converts this formula into a pair of binary strings X', Y' of quasi-polynomial length, where X' (respectively, Y') depends solely on X (Y) and the Indel distance between X', Y' is contingent on the distance between X, Y . This is achieved by recursively transforming the formula into such a pair of strings gate by gate. Two essential components, referred to as *AND*- and *OR*-gadgets, implement this process.

The input for the *AND*-gadget consists of two pairs of strings $(X_0, Y_0), (X_1, Y_1)$, which can be thought of as outputs from previous levels. Here the X_i 's only depend on X and the Y_i 's only depend on Y . Moreover, we are guaranteed that $\Delta_{\text{indel}}(X_i, Y_i)$ can only take two values $\{F, T\}$, where $F < T$. The goal is to concatenate the X_i into a single string X and the Y_i 's into a different string Y such that: $\Delta_{\text{indel}}(X, Y)$ can also take value in $\{F', T'\}$, where $F' < T'$ and: $\Delta_{\text{indel}}(X, Y) = T'$ if and only if $\Delta_{\text{indel}}(X_0, Y_0) = T \wedge \Delta_{\text{indel}}(X_1, Y_1) = T$. Similarly for the *OR*-gadget. Our construction of the LCS instance from the Formula Evaluation is similar to that of Abboud and Bringmann [71] which considers reduction of the Formula Satisfiability to LCS. The purpose of our reduction is different, though. This result is presented in Section 6.3.

6.1.2 Preliminaries and Notations

In this section we introduce the notations that is used throughout the rest of the chapter.

Normalized Distance: To assess the distance between each pair of strings in a standardized manner, it is advantageous to express it as a normalized value within the range $[0, 1]$. To achieve this, we introduce the following definition:

$$\tilde{\Delta}_{\text{edit}}(X, Y) = \frac{\Delta_{\text{edit}}(X, Y)}{\max(|X|, |Y|)}, \quad \tilde{\Delta}_{\text{indel}}(X, Y) = \frac{\Delta_{\text{indel}}(X, Y)}{|X| + |Y|}$$

6.2 Alphabet Reduction - Succinct Embedding

Within this section, we tackle the task of embedding strings from a sizable alphabet into a smaller one while preserving the global nature of the original metric space. Our main result demonstrates that it's possible to embed strings from any large alphabet Γ into strings of a smaller alphabet Σ , where the length of the strings

remains approximately unchanged, and the normalized distances are distorted by at most a factor of $(1 + \varepsilon)$. The size of the alphabet Σ increases quadratically with $1/\varepsilon$. To provide clarity, we present our results for the Δ_{indel} metric; a similar approach can be applied to the Δ_{edit} metric. This section is structured as follows: In subsection 6.2.1 we outline our findings regarding normalized scaled alphabet reductions, covering both lower and upper bounds. Section 6.2.2 discusses our upper bounds, while Section 6.2.3 addresses lower bounds.

6.2.1 Normalized Scaled Isometric Embedding - Our Finding

An embedding E is said to preserve lengths, if there exists: $\ell : \mathbb{N} \rightarrow \mathbb{N}$ such that: $\forall X \in \Gamma^n : |E(X)| = \ell(n)$. It is non-shrinking if $\ell(n) \geq n$. It is natural to focus on non-shrinking embeddings otherwise if the embedding maps from a large alphabet to smaller one some distinct strings will get mapped to the same string.

The following claim shows that any length-preserving embedding, mapping strings from large alphabet into strings of smaller one, necessarily contracts the normalized distances between certain pairs of strings. The proof of the claim is deferred to Section 6.2.3.

Claim 6.4. *Let Γ, Σ be finite alphabets, such that: $|\Gamma| > |\Sigma|$, and let $E : \Gamma^* \rightarrow \Sigma^*$, which is a length-preserving embedding, then for any $n \in \mathbb{N}$ we have:*

$$\exists X, Y \in \Gamma^n : \tilde{\Delta}_{indel}(E(X), E(Y)) < \tilde{\Delta}_{indel}(X, Y).$$

Therefore, we redirect our attention to approximate embedding, where we allow for a slight distortion in distances. Our main result is as follows:

Theorem 6.1 (Alphabet Reduction - Succinct Embedding). *Let Γ be a finite alphabet, and let $0 < \varepsilon < 1/4$. There exists an alphabet Σ , where $|\Sigma| = \mathcal{O}(\frac{1}{\varepsilon^2})$ and there exists $E : \Gamma^* \rightarrow \Sigma^*$ satisfying:*

$$\forall X, Y \in \Gamma^* : \tilde{\Delta}_{indel}(E(X), E(Y)) \in \left[(1 - \varepsilon)\tilde{\Delta}_{indel}(X, Y), \tilde{\Delta}_{indel}(X, Y) \right].$$

Moreover, for every $X \in \Gamma^n$ we have: $|E(X)| = \mathcal{O}(n \log(|\Gamma|))$.

Note that the distortion is “one-sided” in the sense that the normalized distances of the embedded strings cannot surpass the normalized distance of the original strings. However, for the lower bound, a $(1 - \varepsilon)$ -factor may be incurred. Furthermore, we demonstrate that for any embedding, the normalized distances cannot be uniformly scaled by a fixed factor. In particular, we demonstrate that there exist pairs of strings whose normalized distances are reduced, while for other pairs, their normalized distances converge to each other arbitrarily closely. This, in turn, illustrates that we cannot deduce the value of $\tilde{\Delta}_{indel}(X, Y)$ directly from the value of $\tilde{\Delta}_{indel}(E(X), E(Y))$ by a simple scaling.

Claim 6.5. *Let Γ, Σ be finite alphabets, such that: $|\Gamma| > |\Sigma|$, and let $E : \Gamma^* \rightarrow \Sigma^*$, which is a length-preserving non-shrinking embedding. We have:*

1. For any $n \in \mathbb{N}$, there exist $X, Y \in \Gamma^n$ such that

$$\tilde{\Delta}_{indel}(E(X), E(Y)) \leq \left(1 - \frac{1}{|\Sigma|}\right) \tilde{\Delta}_{indel}(X, Y).$$

2. For any sequence Z_1, Z_2, \dots where $|Z_n| = n$,

$$\lim_{n \rightarrow \infty} \tilde{\Delta}_{\text{indel}}(E(Z_n), E(\Lambda)) - \tilde{\Delta}_{\text{indel}}(Z_n, \Lambda) = 0,$$

where Λ denotes the empty string.

6.2.2 Upper Bounds

The crux of Theorem 6.1 lies in the existence of an error correcting code with respect to the Δ_{indel} metric, even when the alphabet size is small. More specifically, given a proximity parameter $\varepsilon > 0$, and $|\Gamma|$, we pick a set C of strings residing in Σ^k of cardinality $|\Gamma|$, with large pairwise distance. The construction of C follows a greedy approach reminiscent of the Gilbert-Varshamov bound [72, 73]. Properties of the code are summarized in the next statement.

Lemma 6.6. *For any $\varepsilon < 1/2$, let Σ be a finite alphabet satisfying $|\Sigma| > 32/\varepsilon^2$. For every $n \in \mathbb{N}$, there exists $k \in \mathbb{N}$ with $k = \mathcal{O}(\log n)$ for which the following conditions are satisfied:*

1. There exists a code $C_{n,\varepsilon} \subseteq \Sigma^k$ with $|C_{n,\varepsilon}| = n$.
2. $\forall c \neq c' \in C_{n,\varepsilon} : \Delta_{\text{indel}}(c, c') \geq (2 - \varepsilon)k$.

We use the following simple fact to prove Lemma 6.6:

Proposition 6.7. *For any integer $k \geq 1$ and any real $0 < \varepsilon < 1/2$, $\binom{k}{\lfloor \varepsilon k \rfloor} \leq 2^{(\varepsilon \log(1/\varepsilon) + 2\varepsilon)k}$.*

Proof of Proposition 6.7. Let $\varepsilon' = \lfloor \varepsilon k \rfloor / k$ so $\varepsilon' \leq \varepsilon$. It is well known that $\binom{k}{\varepsilon' k} \leq 2^{H(\varepsilon')k}$, where $H(x) = x \log(1/x) + (1-x) \log 1/(1-x)$ is the binary entropy function. So $\binom{k}{\lfloor \varepsilon k \rfloor} = \binom{k}{\varepsilon' k} \leq 2^{H(\varepsilon')k} \leq 2^{H(\varepsilon)k}$, as $H(\cdot)$ is increasing on the interval $[0, 1/2]$. Notice, $\log 1/(1-\varepsilon) = \log(1 + \frac{\varepsilon}{1-\varepsilon}) \leq \log e^{\frac{\varepsilon}{1-\varepsilon}} = \frac{\varepsilon}{1-\varepsilon} \cdot \log e$, where we use the inequality $1+x \leq e^x$ for any real x . Hence, $H(\varepsilon) \leq \varepsilon \log(1/\varepsilon) + \varepsilon \cdot \log e \leq \varepsilon \log(1/\varepsilon) + 2\varepsilon$. \square

Proof of Lemma 6.6.

The proof employs the probabilistic method, demonstrating that there exists a random set $C \subseteq \Sigma^k$ meeting the specified criteria with a non-zero probability, thereby establishing its existence. The parameters of the constructions are as follows: pick an integer k from the interval $[\frac{2}{\varepsilon} \log n, \frac{1}{\varepsilon} + \frac{2}{\varepsilon} \log n]$ so that εk is an integer and let $|\Sigma| = \lceil 32/\varepsilon^2 \rceil$.

Our initial insight is that, given any $X, Y \in \Sigma^k$, if for all subsets $I_1, I_2 \subset [\ell]$ where $|I_1| = |I_2| = \varepsilon k$, it holds that: $X_{I_1} \neq Y_{I_2}$ then $LCS(X, Y) < \varepsilon k$ and $\Delta_{\text{indel}}(X, Y) \geq (2 - 2\varepsilon)k$.

Next, for X, Y chosen uniformly at random, and for a fixed choice of $I_1, I_2 \subset [\ell]$ where $|I_1| = |I_2| = \varepsilon k$ we have that:

$$\Pr[X_{I_1} = Y_{I_2}] = |\Sigma|^{-\varepsilon k}$$

Applying a union bound to all possible choices of I_1, I_2 we obtain:

$$\Pr[\forall I_1, I_2 : X_{I_1} = Y_{I_2}] = |\Sigma|^{-\varepsilon k} \binom{k}{\varepsilon k}^2 \leq |\Sigma|^{-\varepsilon k} 2^{2(\varepsilon \log(1/\varepsilon) + 2\varepsilon)k} \leq 2^{-\varepsilon k},$$

where the final inequality is a consequence of our choice of Σ where $\log |\Sigma| \geq 2 \log(1/\varepsilon) + 5$.

In summary, with a probability of at most $2^{-\varepsilon k}$, it follows that for uniformly random X and Y in Σ^k , their LCS surpasses εk . Ultimately, consider a set $C \subseteq \Sigma^k$ with $|C| = n$. Applying a union bound to all pairs within C , we deduce that the probability of the existence of a pair $c \neq c'$ such that $LCS(c, c') > \varepsilon k$ is at most $\binom{|C|}{2} 2^{-\varepsilon k} < 1$. The last inequality arises from our choice of $k \geq \frac{2}{\varepsilon} \log n$, and this establishes the existence of C , as claimed. \square

Endowed with the existence of such an error correcting code we assign a distinct codeword to each of the characters in the larger alphabet Γ . The embedding procedure is as follows: Given a string $X \in \Gamma^n$, we embed it into a string in $(\Sigma^k)^n$, where the encoding of X is formed by concatenating the codewords assigned to each of its characters. The presentation of the embedding, along with its proof of correctness, is provided in Section 6.2.2.

The Embedding

Let Γ be an alphabet, and let ε be a proximity parameter, and let $C := C_{|\Gamma|, \varepsilon}$ be the code whose existence is guaranteed by Lemma 6.6. We interpret the code C as a function mapping characters from Γ into (short) strings in Σ^k . The embedding proceeds as follows: each string in Γ^* is encoded sequentially character by character, where the encoding of each character is performed using the code C . Our main technical lemma is as follows:

Lemma 6.8. *For any finite alphabet Γ and $\varepsilon > 0$, consider the encoding $C_{|\Gamma|, \varepsilon} : \Gamma \rightarrow \Sigma^k$ as implied by Lemma 6.6. For any string $X \in \Gamma^*$ define: $E(X) = C(X_1) \dots C(X_n)$ (where $n = |X|$).*

Then for any pair of strings $X, Y \in \Gamma^$ the following inequality holds:*

$$(1 - 48\varepsilon)\Delta_{\text{indel}}(X, Y) \leq \Delta_{\text{indel}}(E(X), E(Y)) \leq \Delta_{\text{indel}}(X, Y)$$

Moreover, for every $X \in \Gamma^n$ we have: $|E(X)| = \mathcal{O}(n \log |\Gamma|)$.

In the sequel, an Δ_{indel} -alignment converting $E(X)$ into $E(Y)$ is simply referred as an alignment. In the course of the proof, we introduce the concepts of blocks and block-structured alignments. For $X \in \Gamma^n$ we define the i -th block of $E(X)$ to be the substring of $E(X)$ corresponding to X_i , namely it equals $C(X_i)$. Furthermore, given an alignment \mathcal{A} that transforms $E(X)$ into $E(Y)$, we label it as a *block-structured* alignment if, for each i -th block in $E(X)$, the alignment either fully matches all the characters of the block to some block j in $E(Y)$ or entirely deletes the i -th block. It is clear that block-structured alignments for the embedded strings correspond one-to-one with alignments for the original strings, and their normalized distance remains unchanged. To prove our main technical lemma we transform any alignment converting $E(X)$ to $E(Y)$ into a block-structured one without significantly increasing its cost.

We will introduce certain notations to facilitate the presentation of the proof. For any $X \in \Gamma^n$ we employ lowercase letters such as: i, j, k etc. to represent indices of X_i . We utilize tuples from $[n] \times [k]$ to represent indices of $E(X)$, where the first index signifies the block index denoted by lowercase letters, and the second one describes the index within the block represented by a lowercase Greek letter.

The following claim, stated without a formal proof, will be useful in the subsequent proof.

Claim 6.9. *For an alignment \mathcal{A} transforming $E(X)$ into $E(Y)$, we have that the set of matching characters has to be monotone, indicating that for $(i, \alpha) < (i', \alpha')$ in lexicographical order if (i, α) is matched to (j, β) and (i', α') to (j', β') by \mathcal{A} , we must have: $(j, \beta) < (j', \beta')$.*

Given an alignment \mathcal{A} , we partition $E(Y)$ into n segments based on its matching blocks in $E(X)$. Define the i -th segment as follows: if no character of the i -th block of $E(X)$ is matched under \mathcal{A} , then the i -th segment is empty. Otherwise, let j denote the first block of $E(Y)$ that includes a matching character for one of the characters in the i -th block. If i is the smallest block containing a matching coordinate within j , then the i -th segment starts at $(j, 1)$, otherwise it starts at the first coordinate within the j -th block matching with the i -th block.

The ending point of the segment is defined similarly: Let j' be the initial block of $E(Y)$ containing a match for the $(i+1)$ -th block of $E(X)$. If the i -th block does not match any of the j' -th coordinates, then the i -th segment ends at $(j-1, k)$. Otherwise, it ends at the coordinate preceding the first match of $(i+1)$ and j . We define the starting point of the first non-empty segment as $(1, 1)$ and the end point of the last non-empty segment as (n, k) . See Figure 6.1 (in appendix) for an illustration.

Additionally, we define $cost_{\mathcal{A}}(i)$, the cost of the i -th block, as the sum of unmatched coordinates in $E(X)$ within its i -th block and the unmatched coordinates in $E(Y)$ within its i -th segment. For example, in the illustration provided in Figure 6.1 (in appendix), we have $cost_{\mathcal{A}}(1) = 0 + 3$ since all the characters of the first block of $E(X)$ are matched, and there are 3 unmatched coordinates in the first segment of $E(Y)$. As the decomposition of $E(Y)$ results in disjoint parts, the sum of the costs across the different blocks equals the cost of \mathcal{A} , which we denote by: $cost(\mathcal{A})$.

Converting Into Block-Structured Alignments In this section, we introduce an algorithm that takes an arbitrary alignment \mathcal{A} transforming $E(X)$ into $E(Y)$ as input and produces an alignment \mathcal{A}^* . The resulting alignment \mathcal{A}^* is block-structured, and its cost does not substantially exceed that of \mathcal{A} .

To design the new matching we need few definitions. We say that blocks i and j are *partially matched* by an alignment \mathcal{A} if there exists a pair (i, α) and (j, β) matched by \mathcal{A} . Furthermore, i and j are *significantly matched* by \mathcal{A} if more than εk characters in the i -th block of X are matched into the j -th block; we say that i and j are *perfectly matched* if \mathcal{A} matches every character in $E(X)$ into a character in $E(Y)$. The algorithm operates in two stages. In the first stage, the algorithm iteratively takes two significantly matched blocks that are not perfectly matched, removes all matches that the characters of the two blocks participate in,

and introduces a perfect match between the two blocks. In the second stage, we remove all the matches from blocks that are not matched perfectly.

A key observation is that if i and j are significantly matched then we have the following inequality: $\Delta_{indel}(C(X_i), C(Y_j)) < (2 - \varepsilon)k$. Therefore, by the distance guarantee regarding C , we must have $X_i = Y_j$. The algorithm is given next. For the sake of the analysis its second stage is divided into two cases.

Algorithm 17 Converting Into Block-Structured Alignments:

Data: An alignment \mathcal{A} converting $E(X)$ into $E(Y)$

Result: An alignment \mathcal{A}' that is block-structured, converting $E(X)$ into $E(Y)$

```

197  $\mathcal{A}' \leftarrow \mathcal{A}$ ;
    ;
198 for  $i = 1 \dots |X|$  do
199     if  $i$  has some significant match then
200          $j \leftarrow$  smallest block in  $E(Y)$  that significantly matches  $i$ ;
201         Remove all matches incident with blocks  $i$  and  $j$  from  $\mathcal{A}'$ , and add a perfect
           match between the two blocks;
202     end
203 end
204  $i \leftarrow 1$  ;
205 while  $i \leq |X|$  do
206     if  $i$  has a partial and not perfect match then
207         if  $i$  is partially matched to more than a single block then
208             Delete from  $\mathcal{A}'$  all matches of characters from the  $i$ -th block of  $E(X)$ ;
209              $i++$ ;
210         end
211         else
212              $j \leftarrow$  smallest  $E(Y)$ -block that partially matches  $i$ ;
213              $i' \leftarrow$  smallest  $E(X)$  that does not match with the  $j$ -th block;
214             Delete from  $\mathcal{A}'$  all matches of characters from the  $j$ -th block of  $E(Y)$ ;
215              $i \leftarrow i'$ ;
216         end
217     end
218 end

```

The Correctness of the Algorithm We break down the proof of correctness into three claims. Claim 6.10 states that the output produced by Algorithm 17 is a block-structured alignment. The subsequent two claims provide bounds on the cost difference between the input and output alignments.

Claim 6.10. *The alignment \mathcal{A}' produced by Algorithm 17 from any alignment \mathcal{A} converting $E(X)$ into $E(Y)$ is a block-structured alignment converting $E(X)$ into $E(Y)$.*

The proof of Lemma 6.8 is derived from the following claims, let us first state the claims.

Claim 6.11. *Let $\varepsilon < 1/4$ and let \mathcal{A} be any alignment converting $E(X)$ into $E(Y)$. Let \mathcal{A}_I be the resulting alignment obtained by applying the algorithm described in stage I on \mathcal{A} with a proximity parameter of ε . Then,*

$$\text{cost}(\mathcal{A}_I) \leq (1 + 4\varepsilon)\text{cost}(\mathcal{A}).$$

Claim 6.12. *Let \mathcal{A}_I be any resulting alignment obtained by applying the algorithm described in stage I on some alignment \mathcal{A} with a proximity parameter of ε . Let \mathcal{A}_{II} be the resulting alignment obtained by applying the algorithm described in stage II on \mathcal{A}_I with a proximity parameter of ε . Then,*

$$\text{cost}(\mathcal{A}_{II}) \leq (1 + 4\varepsilon)\text{cost}(\mathcal{A}_I).$$

The proofs of claims 6.10, 6.11 and 6.12 are given in Section ??.

Proof of Lemma 6.8. [using Claim 6.11 and Claim 6.12]

Let OPT represent the normalized cost of the optimal alignment between X and Y , and \widetilde{OPT} denote the normalized cost of the optimal alignment between $E(X)$ and $E(Y)$. Notice that any alignment between X and Y can be paired with an alignment between $E(X)$ and $E(Y)$ having the same cost. Consequently, we have: $\widetilde{OPT} \leq OPT$. To complete the argument, it remains to establish that: $(1 - 48\varepsilon)OPT \leq \widetilde{OPT}$, which can be achieved by demonstrating: $OPT \leq (1 + 24\varepsilon)\widetilde{OPT}$.

Consider the optimal alignment \mathcal{A} that transforms X into Y , and let \mathcal{A}' be the alignment generated by Algorithm 17 when applied to \mathcal{A} . According to Claims 6.11 and 6.12, we obtain:

$$\begin{aligned} \frac{1}{2|E(X)|} \cdot \text{cost}(\mathcal{A}') &\leq \frac{1}{2|E(X)|} \cdot (1 + 4\varepsilon)^2 \text{cost}(\mathcal{A}) \\ &\leq \frac{1}{2|E(X)|} \cdot (1 + 24\varepsilon) \text{cost}(\mathcal{A}) \\ &= (1 + 24\varepsilon)OPT. \end{aligned}$$

We conclude the proof by noting that: $\widetilde{OPT} \leq \frac{1}{2|E(X)|} \cdot \text{cost}(\mathcal{A}')$. □

Proof of Claim 6.10. We first prove that \mathcal{A}' converts $E(X)$ into $E(Y)$. Observe that \mathcal{A} converts $E(X)$ into $E(Y)$, and \mathcal{A}' is derived by alternately performing a perfect match for blocks with a significant match and removing other matched characters from \mathcal{A} , it is evident that the resulting alignment, \mathcal{A}' , indeed converts $E(X)$ into $E(Y)$.

Next we claim that the only matched characters are within blocks which are perfectly matched. Consider each block i : If i has a significant match under \mathcal{A} , then under \mathcal{A}' , it is matched perfectly to a single block j . Next, we need to demonstrate that in \mathcal{A}' , no partial matching exists.

For blocks i that are partially matched under \mathcal{A} into a *single* block j , or vice versa, it is straightforward to verify the absence of partial matching in \mathcal{A}' . However, consider a block i that is matched into several blocks under \mathcal{A} , and one of these blocks has several partial matches as well. In this scenario, due to the monotonicity property (as per Claim 6.9), it is either the case that i is matched to several blocks, and then only the last matched block j is matched to several

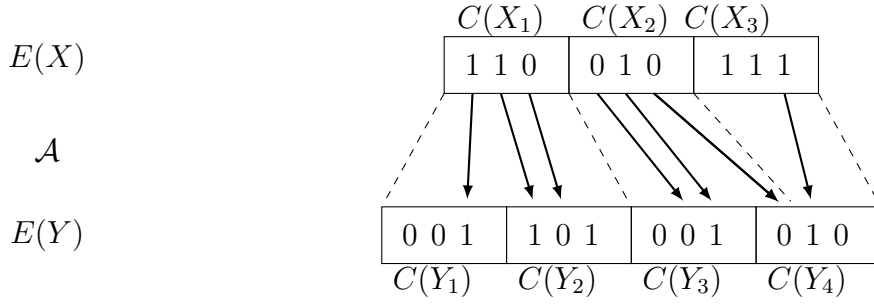


Figure 6.1 An illustration of the matching between the strings $E(X)$ and $E(Y)$. Arrows indicate matching coordinates, and dashed lines represent the beginning/end points of the segments. The first segment starts at $(1, 1)$ and ends at $(2, 3)$, as the first matched coordinate in the second block of $E(X)$ is mapped to the third block, and no character from the first block of $E(X)$ is mapped to that block. The second segment starts at $(3, 1)$ and ends at $(4, 1)$ (as the first matched coordinate in the second block of $E(X)$ is mapped to the third block, and there exists a character from the second block of $E(X)$ that is mapped to that block)

blocks, or vice versa.

Let us analyze the first case; the second one follows by the same reasoning. In this case, the algorithm first removes all the matches involving the coordinates of the i -th block during the i -th iteration. Subsequently, in the following iteration, it removes all the matches involving the coordinates of the j -th block. In total, this results in a block-structured alignment. \square

Proof of Claim 6.11. We show that for each block $i \in [n]$, the cost of the i -th block under \mathcal{A} and \mathcal{A}_I does not change substantially. Specifically, as we scan the blocks from left to right, we establish that for each block, $cost_{\mathcal{A}_I}(i) \leq (1 + 4\epsilon)cost_{\mathcal{A}}(i)$. The proof of Claim 6.11 follows by summing the cost differences over the various blocks.

Consider $i \in [n]$ and assume that the i -th block of $E(X)$ significantly matches the j -th block of $E(Y)$, with j being the smallest one that satisfies this condition. In stage I, the algorithm opts to establish a perfect match between i and j . This may impact (i) the cost of the i -th block, (ii) the cost of any block $i' > i$ that has partial match with the j -th block of $E(Y)$ and (iii) the cost of any block $i' < i$ that has partial match with the j -th block of $E(Y)$.

As for the i -th block, we affirm that its cost under \mathcal{A}_I can only decrease. Given that the j -th block of $E(Y)$ significantly matched with i , and as observed earlier, we have $X_i = Y_j$. During phase I, all the characters in $C(X)_i$ are matched to the characters in $C(Y)_j$, so there is no increase in the number of deleted characters in the i -th block. However, in \mathcal{A}_I , the characters in blocks $j' > j$, which were previously matched under \mathcal{A} to characters in the i -th block of $E(X)$, are deleted and no longer matched under \mathcal{A}_I . Nevertheless, each deletion can be accounted for by each new matching in the j -th block.

Consider blocks $i' > i$ that were partially matched to the j -th block of $E(Y)$. Under \mathcal{A}_I all these matches are deleted. Nevertheless, each such a deletion can be accounted against a character of the i -th block of $E(X)$ that is deleted by \mathcal{A} and is matched under \mathcal{A}_I .

It is left to consider any block $i' < i$ that has a partial match with the j -th

block of $E(Y)$. We claim that the i' -th block lacks a significant match with any of the blocks in $E(Y)$. For the j -th block, this follows from the description of phase I. Regarding previous blocks, if a significant match existed, then in prior iterations, it would have been perfectly mapped to a preceding block, and consequently, it would not have any matching with the j -th block.

We next claim that the cost of i' -th block is at least $(1 - 2\varepsilon)k$. The proof is conducted through case analysis based on the number of blocks in $E(Y)$ that were partially matched into this block:

If the characters of the block were mapped solely to the j -th block, then in \mathcal{A} , at least $(1 - \varepsilon)k$ of its characters are deleted, and the proof follows. If it was mapped to one additional block, then under \mathcal{A} , at least $(1 - 2\varepsilon)k$ of its characters are deleted, and the proof follows. The remaining case to analyze is when it was mapped to at least three blocks; denote the first three of them by $j_1 < j_2 < j_3$.

Consider the block j_2 , we claim that at least $(1 - \varepsilon)k$ of its characters are deleted by \mathcal{A} . Since it does not have a significant match with i' , at most εk of its characters are matched into the i' -th block of $E(X)$ by \mathcal{A} . We claim that all the others are deleted. Since some characters of the i' -th block are matched with j_1 , there is no matching between characters from j_2 to any block preceding i' otherwise the matching is not monotone, see Figure 6.2 for illustration. By similar reasoning these characters cannot be matched to block which following i' . Hence the characters which are not matched to i' are deleted, as claimed. In total we conclude that: $cost_{\mathcal{A}}(i') \geq (1 - \varepsilon)k$.

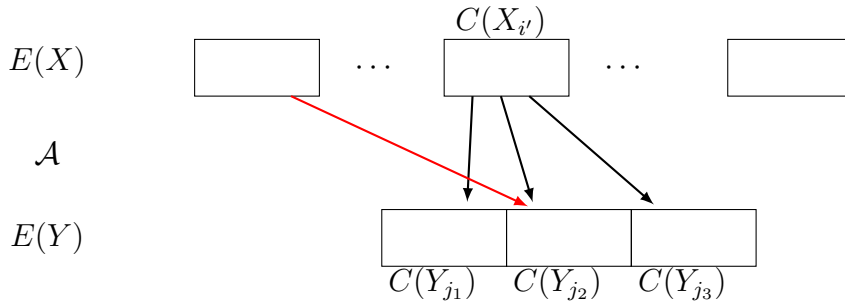


Figure 6.2 Block i' partially matches blocks j_1, j_2, j_3 in $E(Y)$. Consequently, no other block in $E(X)$ can be partially matched with j_2 . The red line illustrates the prohibited matching.

In either case, the cost of the i' -th block is at least $(1 - 2\varepsilon)k$, as claimed. Observe that during the i -th iteration at most ε characters of the i' -th block were deleted and this quantity were deleted in the j -th block of $E(Y)$ we get:

$$cost_{\mathcal{A}_1}(i') \leq cost_{\mathcal{A}}(i') + 2\varepsilon k \leq \left(1 + \frac{2\varepsilon}{1 - 2\varepsilon}\right) cost_{\mathcal{A}}(i') \leq (1 + 4\varepsilon) cost_{\mathcal{A}}(i'),$$

where the last inequality follows from the fact that $\varepsilon < 1/4$. □

Proof of Claim 6.12. The proof of Claim 6.12 involves some subtleties. Let us briefly discuss why before delving into the detailed proof. We aim to replicate the ideas presented in the proof of Claim 6.11 and identify the point of failure. Consider the i -th iteration of the algorithm and assume that i has a partial and not perfect matching. By applying the previous arguments, we deduce that the costs

of the blocks prior to i that have a partial match to j do not change significantly. However, after removing these matches, most of the characters in the j -th block of $E(Y)$ will be deleted. In the modified alignment \mathcal{A}_{II} , the assigned cost to the i -th block of $E(X)$ may account for all these deletions (in both $E(X)$ and $E(Y)$). Therefore, the cost of the i -th block may increase from approximately k in \mathcal{A}_{I} to approximately $2k$ in \mathcal{A}_{II} , resulting in a potential 2-factor increase in the cost. Thus, a more global argument is required, one that doesn't analyze the cost of each block independently.

Indeed, let us examine the i -th iteration of the algorithm and assume that i has a partial and not perfect match. Let j be the smallest block matched to i under \mathcal{A}_{I} . For simplicity we refer to the i -th block $E(X)$ as the i -th block and the j -th block of $E(Y)$ as the j -th block.

Firstly, note that under \mathcal{A}_{I} , if there are characters in the i -th block matched outside the j -th block, then they must be matched within blocks larger than j . The choice of j ensures that the partial matching blocks of i can only be to blocks greater or equal to j . Characters in the j -th block matched outside the i -th block must be matched within blocks larger than i , since otherwise, the algorithm should have deleted the matching characters between the i -th and j -th blocks in previous iterations.

During the i -th iteration, we claim that only one of two possibilities arises: either the i -th block contains more than a single partial match, or the j -th block contains more than a single partial match. Simultaneous occurrence of these possibilities is precluded, as such a scenario would violate the monotonicity of the alignment. We will proceed to prove these cases separately.

Case I: the i -th block contains more than a single partial match:

Initially, note that in this scenario, during the i -th iteration, the algorithm removes all the matched characters associated with the i -th block. Let m represent the number of blocks that have a partial match with the i -th block of $E(X)$ under \mathcal{A}_{I} . Observe that in \mathcal{A}_{II} we delete all the matched characters in these m blocks which may contribute an extra factor of at most: $2\epsilon mk$.

Now, if $m = 1$, then, under \mathcal{A}_{I} , at least $(1 - \epsilon)k$ characters were deleted in both the i -th block of $E(X)$ and the j -th block of $E(Y)$. Due to the definition of the i -th segment of $E(Y)$ and since the first matched character in the j -th block is matched to the i -th block, the i -th segment of $E(Y)$ contains the j -th block, and hence $\text{cost}_{\mathcal{A}_{\text{I}}}(i) \leq (2 - 2\epsilon)k$. Thus, and since $\epsilon < 1/4$:

$$\text{cost}_{\mathcal{A}_{\text{II}}}(i) \geq \text{cost}_{\mathcal{A}_{\text{I}}}(i) - 2\epsilon k \geq \text{cost}_{\mathcal{A}_{\text{I}}}(i) \left(1 - \frac{\epsilon}{1 - \epsilon}\right) \geq \text{cost}_{\mathcal{A}_{\text{I}}}(i)(1 - 2\epsilon)$$

For $m > 1$, under \mathcal{A}_{I} , we observe that: (i) at least $(1 - m\epsilon)k$ characters of the i -th block are deleted; (ii) for a block in $E(Y)$ that has a partial match to i , except perhaps for the last block, at least $(1 - \epsilon)k$ of its characters are deleted (as per the monotonicity property, its characters are matched only to the i -th block). Hence, we have:

$$\text{cost}_{\mathcal{A}_{\text{II}}}(i) \geq (1 - m\epsilon)k + (m - 1)(1 - \epsilon)k \geq (1 - 2\epsilon)mk,$$

Here, the first term is due to the deletion in the i -th block, and the second term is due to the $E(Y)$ blocks. Combining these, we obtain:

$$\text{cost}_{\mathcal{A}_{\text{II}}}(i) \geq \text{cost}_{\mathcal{A}_{\text{I}}}(i) - 2\epsilon mk \geq \text{cost}_{\mathcal{A}_{\text{I}}}(i) \left(1 - \frac{2\epsilon}{1 - 2\epsilon}\right) \geq \text{cost}_{\mathcal{A}_{\text{I}}}(i)(1 - 4\epsilon)$$

Case II: The j -th block contains more than a single partial match:

Alternatively, if i has a single partial match to j , and j has multiple ones, then the algorithm deletes all the matched characters involving the j -th block. Here, the argument involves evaluating the cost of all $E(X)$ blocks that are partially matched into the j -th block of $E(Y)$. The claim is that their total cost does not change much, and the proof follows a similar approach to that in Case I. In particular if j has a partial matching with the blocks $i_1 < \dots < i_m$, we claim that: $\sum_{k=1\dots m} \text{cost}_{\mathcal{A}_{\text{II}}}(i_k) \geq (m - 1)(1 - \epsilon)k$. The proof can be concluded using the same idea of the previous case, we omit the details. Hence:

$$\sum_{k=1\dots} \text{cost}_{\mathcal{A}_{\text{II}}}(i_k) \geq \sum_{k=1\dots m} \text{cost}_{\mathcal{A}_{\text{I}}}(i_k) - 2\epsilon mk \geq (1 - 4\epsilon) \cdot \left(\sum_{k=1\dots m} \text{cost}_{\mathcal{A}_{\text{I}}}(i_k)\right).$$

In summary, in both cases, the cost of affected blocks in each iteration does not decrease beyond a $(1 - 4\epsilon)$ fraction of their original cost. Due to the disjoint nature of the set of affected blocks over different iterations, we conclude that the total cost of the new alignment is at least $(1 - 4\epsilon)$ of the cost of the original alignment, as asserted. □

6.2.3 Lower Bounds

Proof of Claim 6.4.

For any value of $n \in \mathbb{N}$, define A_n as the set of length n strings composed of a single character from Γ . Clearly, $|A_n| = |\Gamma|$ and moreover, for every distinct pair of strings in A_n , their Indel distance is $2n$.

Now consider any embedding $E : \Gamma^* \rightarrow \Sigma^*$. Since $|A_n| = |\Gamma| > |\Sigma|$, by the pigeonhole principle there exist $X \neq Y \in A_n$ satisfying: $E(X)_1 = E(Y)_1$ ¹. Hence, $\Delta_{\text{indel}}(E(X), E(Y)) < 2\ell(n)$ while $\Delta_{\text{edit}}(X, Y) = 2n$. □

Proof of Claim 6.5.

1. Fix $n \in \mathbb{N}$ and consider any embedding $E : \Gamma^* \rightarrow \Sigma^*$. For any $X \in \Gamma^n$, define the value $p(E(X)) \in \Sigma$ as the plurality value among $\{E(X)_i\}_{i \in \mathbb{N}}$ (ties are broken arbitrarily). Observe that the character $p(E(X))$ appears at least $\frac{\ell(n)}{|\Sigma|}$ times in the string $E(X)$. Furthermore, for any $X, Y \in \Sigma^n$ if: $p(E(X)) = p(E(Y))$, then we get: $LCS(E(X), E(Y)) \geq \frac{\ell(n)}{|\Sigma|}$ and hence: $\Delta_{\text{indel}}(E(X), E(Y)) \leq \left(1 - \frac{1}{|\Sigma|}\right) 2\ell(n)$.

As in the proof of Claim 6.4, define A_n as the set of strings composed of a single character from Γ . Recall that $|A_n| = |\Gamma|$ and moreover, for every distinct pair of points in A_n , their LCS distance is $2n$.

¹ $E(X)_i$ is the i^{th} character of the string $E(X)$.

Since $|A_n| = |\Gamma| > |\Sigma|$, by the pigeonhole principle there exist $X \neq Y \in A_n$ satisfying: $p(E(X)) = p(E(Y))$, yielding: $\Delta_{indel}(E(X), E(Y)) \leq \left(1 - \frac{1}{|\Sigma|}\right) 2\ell(n)$, whereas $\Delta_{indel}(X, Y) = 2n$, as claimed.

2. Let $k = |E(\Lambda)|$. For $Z \in \Gamma^n$, we have: $\Delta_{indel}(E(Z), E(\Lambda)) \geq \ell(n) - k$ so $\tilde{\Delta}_{indel}(E(Z), E(\Lambda)) \geq 1 - \frac{k}{\ell(n)} \geq 1 - \frac{k}{n}$. On the other hand, $\Delta_{indel}(Z, \Lambda) = n$ so $\tilde{\Delta}_{indel}(Z, \Lambda) = 1$.

□

6.3 Alphabet Reduction - Binary Alphabets

In this section we show a reduction of Δ_{edit} and Δ_{indel} over an arbitrary alphabet to the binary alphabet. The reduction expands the strings super-polynomially, but one can think of it as a proof of concept that more efficient reduction might exist. The main theorem of this section is the following statement which is a formal statement of Theorem 6.3. For ease of presentation it is beneficial to think about Longest Common Subsequence instead of Δ_{indel} . That is how we state the theorem here.

Theorem 6.13. *For any integer $n \geq 1$, any alphabet Σ of size at most n^3 , there exist integers S, R, N where $N = n^{\mathcal{O}(\log n)}$ and functions $G, H, G', H' : \Sigma^n \rightarrow \{0, 1\}^N$ such that for any $X, Y \in \Sigma^n$,*

$$\begin{aligned} LCS(X, Y) &= \frac{LCS(G(X), H(Y)) - R}{S} \\ \Delta_{edit}(X, Y) &= \frac{LCS(G'(X), H'(Y)) - R}{S}. \end{aligned}$$

Hence, for any pair of strings X, Y one can recover $\Delta_{edit}(X, Y)$ from $\Delta_{indel}(G'(X), H'(Y))$ over a binary alphabet. Both mappings G, H and G', H' can be computed efficiently in the length of their output. Indeed, they will be defined explicitly below. We remark that the bound n^3 on the size of Σ is essentially arbitrary and could be replaced for example by a bound 2^n without change in the other parameters (except for multiplicative constants). However, the n^3 bound allows for hashing any large alphabet by a random pair-wise independent hash function to an alphabet of size n^3 without affecting the distance of any given pair of strings except with probability $< 1/n$.

In order to prove the theorem we will need several auxiliary functions. We say that a 0-1 string is *balanced* if it contains the same number of 0's and 1's. We say a formula ϕ is *normalized* if it consists of alternating layers of binary *AND* and *OR* and all of its literals are at the same depth; each literal is either a constant, a variable or its negation.

We define two functions $g, h : \{0, 1\}^* \times \{\text{normalized formulas}\} \rightarrow \{0, 1\}^*$ and two threshold functions $f, t : \{\text{normalized formulas}\} \rightarrow \mathbb{N}$ as follows: Let us consider sets of variables $U = \{u_1, \dots, u_p\}$ and $V = \{v_1, \dots, v_q\}$, and let $A = \{a_1, \dots, a_p\}$ where a_i is the assignment to the variable u_i for all $1 \leq i \leq p$, and $B = \{b_1, \dots, b_q\}$ where b_i is the assignment to the variable v_i for all $1 \leq i \leq q$.

Let $\phi(U, V)$ be a normalized formula which is defined over two disjoint sets of variables $U = \{u_1, \dots, u_p\}$ and $V = \{v_1, \dots, v_q\}$. Let $A \in \{0, 1\}^p, B \in \{0, 1\}^q$ where A and B are interpreted as assignments for U and V respectively. We define two functions g, h , such that g gets as an input a pair (ϕ, A) and outputs a string in $\{0, 1\}^*$, similarly h takes a pair (ϕ, B) as its input and outputs a string in $\{0, 1\}^*$. We also define threshold functions f, t which take such a formula as input and output a natural number. The crux of the construction is that for any assignment A for U and B for V we have that if ϕ is satisfied by the assignment pair A, B then $LCS(g(\phi, A), h(\phi, B)) = t(\phi)$, otherwise $LCS(g(\phi, A), h(\phi, B)) = f(\phi)$

We establish the recursive definitions of g, h, f and t based on the depth of the formula. The base case is when ϕ is either a constant 0, 1 or single literals $u_i, \neg u_i, v_j, \neg v_j$, where, $u_i \in U, v_j \in V$. Here by $\neg 0$ we understand symbol 1, and similarly by $\neg 1$ we understand symbol 0.

	$\phi = u_i$	$\phi = \neg u_i$	$\phi = v_j$	$\phi = \neg v_j$	$\phi = 1$	$\phi = 0$
$g(A, \phi)$	$\neg a_i a_i$	$a_i \neg a_i$	0 1	0 1	0 1	1 0
$h(B, \phi)$	0 1	0 1	$\neg b_j b_j$	$b_j \neg b_j$	0 1	0 1
$t(\phi)$	2	2	2	2	2	2
$f(\phi)$	1	1	1	1	1	1

and further inductively:

	$\phi = \phi_0 \text{ OR } \phi_1$							
$g(A, \phi)$	$1^{k/2}$	1^{4k}	$g(A, \phi_0)$	1^{4k}	0^{4k}	$g(A, \phi_1)$	0^{4k}	$0^{k/2}$
$h(B, \phi)$	$0^{k/2}$	0^{4k}	$h(B, \phi_0)$	0^{4k}	1^{4k}	$h(B, \phi_1)$	1^{4k}	$1^{k/2}$
$t(\phi)$	$9k + T$							
$f(\phi)$	$9k + F$							

	$\phi = \phi_0 \text{ AND } \phi_1$								
$g(A, \phi)$	0^{T+F}	$1^{11k+T+F}$	0^{5k}	$g(A, \phi_0)$	0^k	1^k	0^k	$g(A, \phi_1)$	0^{5k}
$h(B, \phi)$	0^{T+F}	0^{5k}	$h(B, \phi_0)$	0^k	1^k	0^k	$h(B, \phi_1)$	0^{5k}	$1^{11k+T+F}$
$t(\phi)$	$13k + 3T + F$								
$f(\phi)$	$13k + 2T + 2F$								

where $k = |g(x, \phi_0)|$, $T = t(\phi_0)$, and $F = f(\phi_0)$.

Key properties of our functions are summarized in the next lemma.

Lemma 6.14. *Let $\phi(U, V)$ be a balanced formula of depth d with set of variables $U = \{u_1, \dots, u_p\}$ and $V = \{v_1, \dots, v_q\}$. For every two assignments $A, A' \in \{0, 1\}^p$ to variables U , we have $|g(A, \phi)| = |g(A', \phi)|$. Similarly, for every two assignments $B, B' \in \{0, 1\}^q$ to variables V , $|h(B, \phi)| = |h(B', \phi)|$. Additionally, $|g(A, \phi)| = |h(B, \phi)| \leq 30^d$.*

Furthermore, the following holds:

1. If $\phi(A, B)$ is true then $LCS(g(A, \phi), h(B, \phi)) = t(\phi)$.
2. If $\phi(A, B)$ is false then $LCS(g(A, \phi), h(B, \phi)) = f(\phi)$.

Finally, $f(\phi) < t(\phi)$.

In order to prove the above lemma we also need two gadgets which we call the *AND*-gadget and the *OR*-gadget. We need the lemmas on these gadgets which analyze the composition of *AND* and *OR*.

Lemma 6.15 (OR-gadget). *Let $k, T, F \geq 1$ be integers where k is even and $k/2 \leq F < T$. Let $X_0, Y_0, X_1, Y_1 \in \{0, 1\}^k$ be balanced strings where $LCS(X_0, Y_0), LCS(X_1, Y_1) \in \{T, F\}$. Let*

$$X' = 1^{k/2} \ 1^{4k} \ X_0 \ 1^{4k} \ 0^{4k} \ X_1 \ 0^{4k} \ 0^{k/2}, \quad (6.1)$$

$$Y' = 0^{k/2} \ 0^{4k} \ Y_1 \ 0^{4k} \ 1^{4k} \ Y_0 \ 1^{4k} \ 1^{k/2}. \quad (6.2)$$

Let $T' = 9k + T$ and $F' = 9k + F$. If $LCS(X_0, Y_0) = T$ or $LCS(X_1, Y_1) = T$ then $LCS(X', Y') = T'$, and otherwise $LCS(X', Y') = F'$.

Proof of Lemma 6.15. The analysis considers how the maximum matching between X' and Y' can look like. For ease of notation, we divide X' and Y' into 8 blocks each so $X' = b_1^{X'} b_2^{X'} \dots b_8^{X'}$ and $Y' = b_1^{Y'} b_2^{Y'} \dots b_8^{Y'}$ where:

$$b_1^{X'} \ b_2^{X'} \ b_3^{X'} \ b_4^{X'} \ b_5^{X'} \ b_6^{X'} \ b_7^{X'} \ b_8^{X'} \quad (6.3)$$

$$X' = 1^{k/2} \ 1^{4k} \ X_0 \ 1^{4k} \ 0^{4k} \ X_1 \ 0^{4k} \ 0^{k/2}, \quad (6.4)$$

$$Y' = 0^{k/2} \ 0^{4k} \ Y_1 \ 0^{4k} \ 1^{4k} \ Y_0 \ 1^{4k} \ 1^{k/2} \quad (6.5)$$

$$b_1^{Y'} \ b_2^{Y'} \ b_3^{Y'} \ b_4^{Y'} \ b_5^{Y'} \ b_6^{Y'} \ b_7^{Y'} \ b_8^{Y'} \quad (6.6)$$

that is $b_3^{X'} = X_0, b_6^{X'} = X_1, b_3^{Y'} = Y_1, b_6^{Y'} = Y_0$ and all the other $b_i^{X'}$'s and $b_i^{Y'}$'s are either blocks of 0's or 1's.

First we consider two significant matchings of X' and Y' . Consider a matching that matches symbols from $b_1^{X'}$ to symbols in $b_3^{Y'}$, $b_2^{X'}$ to $b_5^{Y'}$, $b_3^{X'}$ to $b_6^{Y'}$, $b_4^{X'}$ to $b_7^{Y'}$ and $b_6^{X'}$ to $b_8^{Y'}$. Since $LCS(b_3^{X'}, b_6^{Y'}) = LCS(X_0, Y_0)$, $LCS(b_1^{X'}, b_3^{Y'}) = LCS(Y_1, 1^{k/2}) = k/2$ and $LCS(b_6^{X'}, b_8^{Y'}) = LCS(X_1, 1^{k/2}) = k/2$ largest such matching has size $8k + LCS(X_0, Y_0) + k/2 + k/2 = 9k + LCS(X_0, Y_0)$. Notice, this is either T' or F' . Similarly, a matching that matches symbols from $b_3^{X'}$ to $b_1^{Y'}$, $b_5^{X'} b_6^{X'} b_7^{X'}$ to $b_2^{Y'} b_3^{Y'} b_4^{Y'}$ and $b_8^{X'}$ to $b_6^{Y'}$ will have size $9k + LCS(X_1, Y_1)$. Thus, we only need to argue that there is no matching larger than

$$9k + \max(LCS(X_0, Y_0), LCS(X_1, Y_1))$$

Consider a maximum matching of X' to Y' . Assume that the matching starts by matching a symbol 1 in X' and Y' . Without changing the cost of the matching we can replace that edge by matching the left-most 1's in X' and Y' . In X' , the first 1 is in $b_1^{X'}$ so we can assume that there is a symbol from $b_1^{X'}$ which is matched somewhere. Thus the blocks $b_1^{Y'}$ and $b_2^{Y'}$ are unmatched. Furthermore, one can assume without loss of generality that all symbols matched from $b_1^{X'}$ are matched to 1's in $b_3^{Y'} = Y_0$ as Y_0 contains enough 1's and replacing each matched pair between $b_0^{X'}$ and $b_3^{Y'} \dots b_8^{Y'}$ by a matching pair which uses the leftmost unmatched 1

in $b_3^{Y'}$ (going over edges from left to right) will not affect the cost of the matching. So without loss of generality all matched symbols from $b_1^{X'}$ are matched into $b_3^{Y'} = Y_1$.

We are now concerned with matching symbols after $b_1^{X'}$. Matching any symbol from $b_4^{Y'}$ will block at least $4k$ 1's in $b_1^{X'}b_2^{X'}$ from being matched anywhere. If $b_4^{Y'}$ is matched only to $b_3^{X'}$ then we will match at most $2k$ 0's from $b_3^{Y'}, b_4^{Y'}, b_6^{Y'}$, and at most $5k$ 1's from $b_3^{X'} \cdots b_8^{X'}$ so altogether the matching will be of size at most $7k$. If $b_4^{Y'}$ is matched to something in $b_5^{X'} \cdots b_8^{X'}$ then at most $k/2$ 1's from $b_5^{Y'} \cdots b_8^{Y'}$ will be matched so even if all $5k$ 0's from $b_3^{Y'} \cdots b_8^{Y'}$ were matched the overall matching will be of size at most $6k$. Similar argument applies if we were to match 0's from $b_3^{Y'}$, the matching would be smaller than $7k$. Thus we can assume that no 0 is matched from $b_3^{Y'}b_4^{Y'}$ if $b_1^{X'}$ matches something.

Hence, we can assume that if the matching starts by 1 then our maximum matching matches 1's in $b_1^{X'}b_2^{X'}$ greedily from left to right that is $b_1^{X'}b_2^{X'}$ is completely matched to 1's in $b_3^{Y'}$ and $b_5^{Y'}$ which gives $4.5k$ 1's. It remains to match $b_3^{X'} \cdots b_8^{X'}$ to $b_6^{Y'}b_7^{Y'}b_8^{Y'}$. Matching 0's from $b_6^{Y'} = Y_0$ to anywhere in $b_5^{X'} \cdots b_8^{X'}$ will cut-off all but $k/2$ 1's in $b_7^{Y'}b_8^{Y'}$ from being matched. This would result in a small matching. Hence, $b_6^{Y'}$ must match $b_3^{X'}b_4^{X'}$ so no 0 in $b_5^{X'} \cdots b_8^{X'}$ will be matched. Thus we can assume that all 1's in $b_7^{Y'}b_8^{Y'}$ are greedily matched from right to left into $b_6^{X'}$ and then $b_4^{X'}$.

This leaves $b_3^{X'} = X_0$ to be matched to $b_6^{Y'} = Y_0$. This can be done by a matching of size $LCS(X_0, Y_0)$. This is one of the two good matching we have considered initially so it has size $9k + LCS(X_1, Y_1)$.

If our matching starts by 0, a completely symmetric argument gives that its size is going to be $9k + LCS(X_1, Y_1)$. So there is no matching larger than $9k + \max\{LCS(X_0, Y_0), LCS(X_1, Y_1)\}$. It follows that the cost of the largest matching is F' if both $LCS(X_0, Y_0) = LCS(X_1, Y_1) = F$ and T' otherwise. \square

Notice, both X' and Y' are balanced in the above lemma. Also, $|X'| = |Y'| = 19k$.

Lemma 6.16 (AND-gadget). *Let $k, T, F \geq 1$ be integers where k is even and $k/2 \leq F < T$. Let $X_0, Y_0, X_1, Y_1 \in \{0, 1\}^k$ be balanced strings where $LCS(X_0, Y_0), LCS(X_1, Y_1) \in \{T, F\}$. Let*

$$X' = 0^{T+F} 1^{11k+T+F} 0^{5k} X_0 0^k 1^k 0^k X_1 0^{5k}, \quad (6.7)$$

$$Y' = 0^{T+F} 0^{5k} Y_0 0^k 1^k 0^k Y_1 0^{5k} 1^{11k+T+F}. \quad (6.8)$$

Let $T' = 13k+3T+F$ and $F' = 13k+2T+2F$. If $LCS(X_0, Y_0) = LCS(X_1, Y_1) = T$ then $LCS(X', Y') = T'$, and otherwise $LCS(X', Y') = F'$.

Proof of Lemma 6.16. Again, the proof proceeds by a case by case analysis of a maximum matching between X' and Y' . WLOG we can focus only on maximum matchings that match the initial block 0^{T+F} in X' to the same initial block in Y' . We divide X' and Y' into blocks:

$$b_0^{X'} \quad b_8^{X'} \quad b_1^{X'} \quad b_2^{X'} \quad b_3^{X'} \quad b_4^{X'} \quad b_5^{X'} \quad b_6^{X'} \quad b_7^{X'} \quad (6.9)$$

$$X' = 0^{T+F} 1^{11k+T+F} 0^{5k} X_0 \quad 0^k 1^k \quad 0^k X_1 \quad 0^{5k}, \quad (6.10)$$

$$Y' = 0^{T+F} 0^{5k} Y_0 \quad 0^k 1^k \quad 0^k Y_1 \quad 0^{5k} 1^{11k+T+F} \quad (6.11)$$

$$b_0^{Y'} \quad b_1^{Y'} \quad b_2^{Y'} \quad b_3^{Y'} \quad b_4^{Y'} \quad b_5^{Y'} \quad b_6^{Y'} \quad b_7^{Y'} \quad b_8^{Y'} \quad (6.12)$$

There are two significant matchings between X' and Y' we will analyze first. Consider a matching that matches all 1's in X' and Y' . Such a matching necessarily matches some 1 from block $b_8^{X'}$ to a 1 in $b_8^{Y'}$. That prevents all the 0's from $b_1^{X'} \dots b_7^{X'}$ to be matched and similarly for 0's from $b_1^{Y'} \dots b_7^{Y'}$. Hence, the size of such a matching is exactly $T+F+(11k+T+F)+k/2+k+k/2 = 13k+2T+2F = F'$, the number of ones plus the size of $b_0^{X'}$. It is thus clear, that any matching that matches some 1 from $b_8^{X'}$ to a 1 in $b_8^{Y'}$ has size at most F' .

The other significant matching is a matching that matches $b_i^{X'}$ to $b_i^{Y'}$, for $i = 1, \dots, 7$, so that each pair of blocks is matched in the best possible way. Such a matching has size $T + F + 13k + LCS(X_0, Y_0) + LCS(X_1, Y_1)$. Thus if $LCS(X_0, Y_0) = LCS(X_1, Y_1) = T$ we get an overall matching of size $13k+3T+F = T'$.

Now, our goal is to argue that any maximum matching of $b_1^{X'} \dots b_7^{X'}$ to $b_1^{Y'} \dots b_7^{Y'}$ has size at most $13k + LCS(X_0, Y_0) + LCS(X_1, Y_1)$. Since the length of $b_1^{X'} \dots b_7^{X'}$ and $b_1^{Y'} \dots b_7^{Y'}$ is $15k$ and $13k + LCS(X_0, Y_0) + LCS(X_1, Y_1) \geq 14k$ if a matching leaves at least $k + 1$ symbols in either one of the strings unmatched it cannot be maximum. If a matching would match a 1 from the central block $b_4^{X'}$ to either $b_2^{Y'} = Y_0$ or $b_6^{Y'} = Y_1$, at least $k + 1$ symbols would be left unmatched in $b_1^{Y'} \dots b_7^{Y'}$ so the matching would not be maximum. So a maximum matching must match 1's in the central block $b_4^{X'}$ to 1's in $b_4^{Y'}$ if it matches them at all. Clearly, the best is to match all of them. (If a matching would not match the central 1's then it can either be increased by matching the 1's or not. In the latter case it must be matching some symbol from $b_4^{X'}$ to a symbol in $b_1^{Y'} b_2^{Y'} b_6^{Y'} b_7^{Y'}$ (or vice versa for $b_4^{Y'}$) hence leaving unmatched at least $k + 1$ symbols in one of the strings.)

So a maximum matching of $b_1^{X'} \dots b_7^{X'}$ to $b_1^{Y'} \dots b_7^{Y'}$ matches all 1's in $b_4^{X'}$ and $b_4^{Y'}$ to each other. Hence, without loss of generality it matches also the neighboring 0's, so $b_3^{X'} b_4^{X'} b_5^{X'}$ is matched to $b_3^{Y'} b_4^{Y'} b_5^{Y'}$. WLOG we can also assume that $b_1^{X'}$ perfectly matches $b_1^{Y'}$, and $b_7^{X'}$ perfectly matches $b_7^{Y'}$. Hence a maximum matching of $b_1^{X'} \dots b_7^{X'}$ to $b_1^{Y'} \dots b_7^{Y'}$ matches the corresponding $b_i^{X'}$ and $b_i^{Y'}$ for $i = 1, 3, 4, 5, 7$. The best we can do on the remaining parts consisting of A_i 's and Y_i 's is $LCS(X_0, Y_0) + LCS(X_1, Y_1)$. Hence, a maximum matching of $b_1^{X'} \dots b_7^{X'}$ to $b_1^{Y'} \dots b_7^{Y'}$ has cost $13k + LCS(X_0, Y_0) + LCS(X_1, Y_1)$.

It remains to consider a matching that matches some 1 from $b_8^{X'}$ to $b_1^{Y'} \dots b_7^{Y'}$ but not to $b_8^{Y'}$. Such a matching necessarily has to leave $b_1^{Y'}$ unmatched which is $5k$ symbols. If $b_8^{Y'}$ does not match anything in $b_1^{X'} \dots b_7^{X'}$, the best matching we can get has size at most $10k = |b_2^{Y'} \dots b_7^{Y'}|$. If $b_8^{Y'}$ matches something in $b_1^{X'} \dots b_7^{X'}$, it can contribute at most $2k$ additional edges matching 1's. Either way, the matching will fall short of the required $13k$ edges. \square

Again, X' and Y' in the above lemma are balanced. Indeed, the initial block of $T + F$ zeros has the sole purpose of making them balanced. Also $|X'| = |Y'| = 26k + 2T + 2F \leq 30k$.

Proof of Lemma 6.14. The claims regarding the length of $g(A, \phi)$ and $h(B, \phi)$ follow easily by induction on the depth of the formula using the fact that the formula is normalized. All normalized formulas of the same depth that have top gate *AND* give strings of the same size, and similarly all normalized formulas of the same depth that have top gate *OR* give strings of the same size. Indeed, in the base case $d = 1$, all the output strings are of length 2. Then either we apply *AND* composition on strings of the same length generated for the left and right

sub-formulas or we apply *OR* composition. In each case the length of the strings for the sub-formulas are the same so the resulting strings are also of the same size. In each step the length of the strings multiplies by a factor of at most 30, so the output strings are of length at most 30^d .

The claim about $LCS(g(A, \phi), h(B, \phi))$ being either $f(\phi)$ or $t(\phi)$ also follows by induction on the depth of ϕ . In the base case, $d = 1$ and ϕ is a literal, so the claim is obvious from the definition of $g(A, \phi)$ and $h(B, \phi)$ for literals. For higher depths $d > 1$, the claim follows inductively by Lemma 6.16 if ϕ has top gate *AND*, or by Lemma 6.15 if ϕ has top gate *OR*.

The final claim $f(\phi) < t(\phi)$ follows inductively as well. \square

Proof of Theorem 6.13. First, we prove the claim regarding $LCS(X, Y)$. For a string $X \in \Sigma^n$, let \bar{X} be its (natural) binary encoding using $3n \log n$ bits, and for an integer $k \leq n$, let \bar{k} be its encoding in unary using n bits. There is a non-deterministic Turing machine running in logarithmic space that given inputs $X, Y \in \Sigma^n$ and $k \in \mathbb{N}$ checks whether $LCS(X, Y) \geq k$. This is because the question whether $LCS(X, Y) \geq k$ or not can be efficiently reduced to a reachability question on a directed graph. For fixed n , this non-deterministic computation can be turned into a normalized boolean formula $\phi_n(U, V, W)$ of depth $d = \mathcal{O}(\log^2 n)$ which takes as its input the binary encoding of X, Y and k such that $\phi_n(\bar{X}, \bar{Y}, \bar{k})$ is true if and only if $LCS(X, Y) \geq k$, where $\phi_n(\bar{X}, \bar{Y}, \bar{k})$ is the evaluation of ϕ_n with the assignment $U = \bar{X}, V = \bar{Y}$ and $W = \bar{k}$. Let $M = n \cdot |g(\bar{X}, \phi_n(U, V, \bar{1}))|$, where X is an arbitrary string of length n . Notice, M depends only on the depth of $\phi_n(U, V, W)$ not on the actual assignment of variables. Set $N = (3n - 2)M$.

Define

$$\begin{aligned} G(X) &= g(\bar{X}, \phi_n(U, V, \bar{1})) \cdot 0^M 1^M \cdot g(\bar{X}, \phi_n(U, V, \bar{2})) \cdot 0^M 1^M \cdots g(\bar{X}, \phi_n(U, V, \bar{n})), \\ H(Y) &= h(\bar{Y}, \phi_n(U, V, \bar{1})) \cdot 0^M 1^M \cdot h(\bar{Y}, \phi_n(U, V, \bar{2})) \cdot 0^M 1^M \cdots h(\bar{Y}, \phi_n(U, V, \bar{n})). \end{aligned}$$

Note that for a fixed value of k , $\phi_n(U, V, \bar{k})$ represents a formula that has two sets of variables U and V , where U depends only on X and V depends only on Y . Clearly, $G(X)$ depends solely on the string X , while $H(Y)$ depends on Y .

Observe that for any k smaller or equal than $LCS(X, Y)$, we have $\phi_n(\bar{X}, \bar{Y}, \bar{k})$ is true and hence by Lemma 6.14 we get:

$$LCS(g(\bar{X}, \phi_n(U, V, \bar{k})), h(\bar{Y}, \phi_n(U, V, \bar{k}))) = T$$

For k that exceeds $LCS(X, Y)$ this evaluates to F .

Furthermore, we can get a matching between $G(X)$ and $H(Y)$ of size

$$\begin{aligned} &2M(n - 1) + LCS(X, Y) \cdot T + (n - LCS(X, Y)) \cdot F \\ &= 2M(n - 1) + nF + LCS(X, Y) \cdot (T - F), \end{aligned}$$

by matching optimally the consecutive blocks of $G(X)$ and $H(Y)$. Any other matching that would try to match $g(\bar{X}, \phi_n(U, V, \bar{i}))$ to $h(\bar{Y}, \phi_n(U, V, \bar{j}))$ for $i \neq j$ will leave at least $2M$ symbols unmatched so it will be worse. Hence,

$$LCS(G(X), H(Y)) = 2M(n - 1) + nF + LCS(X, Y) \cdot (T - F)$$

Setting $R = 2M(n - 1) + nF$ and $S = (T - F)$ gives the required relationship.

The proof for Δ_{edit} is the same. If we make sure that the normalized formula for LCS and Δ_{edit} are both of the same depth, then the parameters S and R will be identical. \square

6.4 Summary

In this chapter, we explored the problem of reducing the alphabet size of strings while preserving edit-based distances such as Δ_{indel} and Δ_{edit} . We considered both exact and approximate embeddings, with particular attention to the trade-offs between alphabet size, distortion, and output length.

Our key contributions are as follows:

1. **Succinct Alphabet Reduction:** We showed that strings over any finite alphabet Γ can be embedded into strings over a smaller alphabet Σ of size $\mathcal{O}(1/\varepsilon^2)$, such that the normalized Δ_{indel} distance is preserved up to a $(1 - \varepsilon)$ factor. The embedding uses error-correcting codes over the smaller alphabet and keeps the output string length within a logarithmic factor of the input length.
2. **Lower Bounds:** We established limitations of alphabet reduction by proving that any length-preserving embedding into a smaller alphabet must contract distances for some string pairs. This justifies the need to consider approximate embeddings.
3. **Binary Alphabet Embedding:** We presented a quasi-polynomial embedding into the binary alphabet that exactly preserves the Δ_{indel} distance via a scaling function. Although the dimensionality is currently high, this result serves as a proof of concept, showing the theoretical feasibility of such reductions.

These results demonstrate that meaningful alphabet reduction is achievable with low distortion, even into constant-size or binary alphabets. The techniques provide a foundation for designing more efficient algorithms over small alphabets and raise compelling open problems, chief among them being whether the dimensionality of binary embeddings can be substantially reduced.

7

Conclusion

*“The stories we love best do live in us forever.”
- J.K. Rowling, 2011 speech at King’s Cross*

In this thesis, we presented an extensive exploration into the structural and algorithmic properties of string metrics, specifically focusing on edit distance and its variants. Each chapter contributed significantly towards advancing both the theoretical understanding and practical algorithmic solutions involving these metrics.

In Chapter2, we tackled the problem of embedding the edit distance metric into the Hamming metric, addressing the limitations inherent in naive methods. We introduced the CGK embedding and subsequently developed a novel decomposable embedding scheme. This scheme leverages randomized decomposition into succinctly represented grammar-based blocks, achieving near-optimal distortion. Crucially, our method supports efficient incremental updates and parallelization, making it particularly suitable for dynamic and streaming contexts. The chapter also introduced an encoding based on Karp-Rabin hashing, establishing a robust theoretical foundation for further algorithmic applications.

Chapter3 built upon these embedding results, showcasing practical algorithmic applications. We developed succinct and efficient edit distance sketches and rolling sketches capable of handling dynamic updates. Leveraging our decomposable embedding approach, we significantly reduced complexity compared to prior methods. Notably, we presented a highly efficient randomized streaming algorithm for k -edit approximate pattern matching, substantially improving the previous state-of-the-art complexities from $\tilde{O}(k^5)$ space and $\tilde{O}(k^8)$ time to $\tilde{O}(k^2)$ for both metrics. These achievements underscore the utility of decomposable embeddings as fundamental primitives in algorithm design for sublinear and approximate string processing.

In Chapter4, our focus shifted to embedding from the Hamming metric into the edit metric. Here, we achieved a significant theoretical breakthrough by constructing a constant-rate isometric embedding using synchronization strings. This result allowed for optimal transfer of algorithmic hardness from the Hamming metric, establishing new lower bounds and hardness results in the edit metric domain. Additionally, we rigorously examined structural constraints on possible embeddings, providing important theoretical limits on achievable embedding rates.

Chapter5 explored the relationship between the insertion-deletion (Indel) met-

ric and the edit (ED) metric. We established exact and approximate embeddings from the Indel metric into the ED metric, highlighting essential trade-offs between embedding accuracy and complexity. Our approximate embedding efficiently addressed the quadratic length blow-up issue inherent in exact embeddings. Moreover, we introduced the robust approximation framework, enabling meaningful transfer of approximation algorithms and complexity results between these two fundamental metrics.

Finally, Chapter 6 investigated alphabet reduction techniques for edit-based metrics, providing both exact and approximate methods. We successfully demonstrated that significant alphabet reduction is possible with controlled distortion, even achieving embeddings into constant-sized or binary alphabets. We also provided crucial lower bounds illustrating inherent limitations of length-preserving embeddings. Our findings pave the way for designing algorithms over reduced alphabets, posing intriguing open problems regarding further dimensionality reduction.

Collectively, these contributions represent a cohesive and comprehensive advancement in our understanding of metric embeddings for edit distance and related metrics. The theoretical frameworks, algorithmic innovations, and structural insights developed in this thesis open multiple avenues for future research, particularly in refining embedding complexities, expanding applications in streaming algorithms, and further exploring the profound connections between diverse metric spaces.

7.1 Future directions

Several intriguing open problems emerged from this research:

Chapter 2 There remains a considerable gap between known upper and lower bounds on distortion for edit-to-Hamming embeddings. An important challenge is to design an optimal embedding. Furthermore, constructing a decomposable embedding with significantly lower distortion could drastically improve algorithmic applications presented in Chapter 3.

Chapter 4 There still exists a small gap between the lower and upper bounds of the embedding rate. Although not fully detailed in this thesis, the current best-known lower bound rate of $1/8$ relies heavily on synchronization strings. A fascinating question is whether simpler constructions without synchronization strings can be developed. Such simpler embeddings could potentially lead to explicit constructions of synchronization strings over smaller alphabets.

Chapter 5 Despite progress, no known symmetric isometric (or scaled isometric) embedding exists from Indel into edit metrics. Determining whether such symmetric embeddings can be achieved remains an open and significant theoretical question.

Chapter 6 Finally, a key open problem is constructing an alphabet reduction embedding to a binary alphabet for edit/Indel metrics with optimal rate. Exist-

ing solutions either suffer from high dimensionality or approximate distortions. Achieving optimal rate with practical dimensionality could significantly advance both theoretical understanding and practical algorithmic applications.

Collectively, these open problems present exciting directions for future research in metric embeddings, complexity theory, and string algorithms.

Bibliography

1. WAGNER, Robert A.; FISCHER, Michael J. The String-to-String Correction Problem. *J. ACM*. 1974, vol. 21, no. 1, pp. 168–173. ISSN 0004-5411.
2. MASEK, William J.; PATERSON, Michael S. A faster algorithm computing string edit distances. *Journal of Computer and System Sciences*. 1980, vol. 20, no. 1, pp. 18–31. ISSN 0022-0000.
3. GRABOWSKI, Szymon. New tabulation and sparse dynamic programming based techniques for sequence similarity problems. *Discrete Applied Mathematics*. 2016, vol. 212, pp. 96–103.
4. LANDAU, Gad M.; MYERS, Eugene W.; SCHMIDT, Jeanette P. Incremental String Comparison. *SIAM J. Comput.* 1998, vol. 27, no. 2, pp. 557–582. ISSN 0097-5397.
5. BACKURS, Arturs; INDYK, Piotr. Edit Distance Cannot Be Computed in Strongly Subquadratic Time (Unless SETH is False). In: *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing*. Portland, Oregon, USA: ACM, 2015, pp. 51–58. STOC '15. ISBN 978-1-4503-3536-2.
6. CHAKRABORTY, Diptarka; DAS, Debarati; GOLDENBERG, Elazar; KOUCKÝ, Michal; SAKS, Michael E. Approximating Edit Distance within Constant Factor in Truly Sub-Quadratic Time. In: *59th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2018*. 2018, pp. 979–990. Available from DOI: 10.1109/FOCS.2018.00096.
7. KOUCKÝ, Michal; SAKS, Michael E. Constant factor approximations to edit distance on far input pairs in nearly linear time. In: MAKARYCHEV, Konstantin; MAKARYCHEV, Yury; TULSIANI, Madhur; KAMATH, Gautam; CHUZHOUY, Julia (eds.). *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing, STOC 2020*. ACM, 2020, pp. 699–712. Available from DOI: 10.1145/3357713.3384307.
8. BRAKENSIEK, Joshua; RUBINSTEIN, Aviad. Constant-factor approximation of near-linear edit distance in near-linear time. In: MAKARYCHEV, Konstantin; MAKARYCHEV, Yury; TULSIANI, Madhur; KAMATH, Gautam; CHUZHOUY, Julia (eds.). *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing, STOC 2020*. ACM, 2020, pp. 685–698. Available from DOI: 10.1145/3357713.3384282.

9. ANDONI, Alexandr; NOSATZKI, Negev Shekel. Edit Distance in Near-Linear Time: it's a Constant Factor. In: IRANI, Sandy (ed.). *61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020, Durham, NC, USA, November 16-19, 2020*. IEEE, 2020, pp. 990–1001. Available from DOI: 10.1109/FOCS46700.2020.00096.
10. BHATTACHARYA, Sudatta; KOUCKÝ, Michal. Locally consistent decomposition of strings with applications to edit distance sketching. In: *ArXiv*. ABC, 2023, pp. 000–999.
11. CHAKRABORTY, Diptarka; GOLDENBERG, Elazar; KOUCKÝ, Michal. Streaming algorithms for embedding and computing edit distance in the low distance regime. In: WICHS, Daniel; MANSOUR, Yishay (eds.). *Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2016, Cambridge, MA, USA, June 18-21, 2016*. ACM, 2016, pp. 712–725. Available from DOI: 10.1145/2897518.2897577.
12. BATU, Tuğkan; ERGUN, Funda; SAHINALP, Cenk. Oblivious String Embeddings and Edit Distance Approximations. In: *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithm*. Miami, Florida: Society for Industrial and Applied Mathematics, 2006, pp. 792–801. SODA '06. ISBN 0-89871-605-5.
13. OSTROVSKY, Rafail; RABANI, Yuval. Low distortion embeddings for edit distance. *J. ACM*. 2007, vol. 54, no. 5, p. 23. Available from DOI: 10.1145/1284320.1284322.
14. JOWHARI, Hossein. Efficient Communication Protocols for Deciding Edit Distance. In: *Algorithms - ESA 2012 - 20th Annual European Symposium, Ljubljana, Slovenia, September 10-12, 2012. Proceedings*. 2012, pp. 648–658.
15. CORMODE, Graham; MUTHUKRISHNAN, S. The string edit distance matching problem with moves. In: *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms, January 6-8, 2002, San Francisco, CA, USA*. 2002, pp. 667–676.
16. SAHINALP, Süleyman Cenk; VISHKIN, Uzi. Symmetry breaking for suffix tree construction. In: *Proceedings of the Twenty-Sixth Annual ACM Symposium on Theory of Computing, 23-25 May 1994, Montréal, Québec, Canada*. ACM, 1994, pp. 300–309. Available from DOI: 10.1145/195058.195164.
17. BIRENZWIGE, Or; GOLAN, Shay; PORAT, Ely. Locally Consistent Parsing for Text Indexing in Small Space. In: CHAWLA, Shuchi (ed.). *Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms, SODA 2020, Salt Lake City, UT, USA, January 5-8, 2020*. SIAM, 2020, pp. 607–626. Available from DOI: 10.1137/1.9781611975994.37.
18. KARP, Richard M.; RABIN, Michael O. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*. 1987, vol. 31, no. 2, pp. 249–260. Available from DOI: 10.1147/rd.312.0249.
19. COLE, Richard; VISHKIN, Uzi. Deterministic coin tossing and accelerating cascades: micro and macro techniques for designing parallel algorithms. In: *Proceedings of the eighteenth annual ACM symposium on Theory of computing (STOC)*. 1986, pp. 206–219. Available from DOI: 10.1145/12130.12151.

20. LINIAL, Nathan. Distributive Graph Algorithms–Global Solutions from Local Data. In: *28th Annual Symposium on Foundations of Computer Science, FOCS*. IEEE Computer Society, 1987, pp. 331–335. Available from DOI: 10.1109/SFCS.1987.20.
21. LINIAL, Nathan. Locality in Distributed Graph Algorithms. *SIAM J. Comput.* 1992, vol. 21, no. 1, pp. 193–201. Available from DOI: 10.1137/0221015.
22. GANESH, Arun; KOCIUMAKA, Tomasz; LINCOLN, Andrea; SAHA, Barna. How Compression and Approximation Affect Efficiency in String Distance Measures. In: *Proceedings of the 2022 ACM-SIAM Symposium on Discrete Algorithms, SODA*. 2022, pp. 2867–2919. Available from DOI: 10.1137/1.9781611977073.112.
23. CLIFFORD, Raphaël; KOCIUMAKA, Tomasz; PORAT, Ely. The streaming k -mismatch problem. In: *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019*. SIAM, 2019, pp. 1106–1125. Available from DOI: 10.1137/1.9781611975482.68.
24. BHATTACHARYA, Sudatta; KOUCKÝ, Michal. Streaming k -edit approximate pattern matching via string decomposition. *arXiv preprint arXiv:2305.00615*. 2023.
25. BELAZZOUGUI, Djamel; ZHANG, Qin. Edit Distance: Sketching, Streaming, and Document Exchange. In: *2016 IEEE 57th Annual Symposium on Foundations of Computer Science (FOCS)*. 2016, pp. 51–60. Available from DOI: 10.1109/FOCS.2016.15.
26. JIN, Ce; NELSON, Jelani; WU, Kewen. An Improved Sketching Algorithm for Edit Distance. In: *38th International Symposium on Theoretical Aspects of Computer Science, STACS 2021*, 2021, vol. 187, 45:1–45:16. LIPIcs. Available from DOI: 10.4230/LIPIcs.STACS.2021.45.
27. KOCIUMAKA, Tomasz; PORAT, Ely; STARIKOVSKAYA, Tatiana. Small-space and streaming pattern matching with k edits. In: *2021 IEEE 62nd Annual Symposium on Foundations of Computer Science (FOCS)*. 2021, pp. 885–896. Available from DOI: 10.1109/FOCS52979.2021.00090.
28. KUSHILEVITZ, Eyal; OSTROVSKY, Rafail; RABANI, Yuval. Efficient search for approximate nearest neighbor in high dimensional spaces. In: *Proceedings of the thirtieth annual ACM symposium on Theory of computing*. 1998, pp. 614–623.
29. FEIGENBAUM, Joan; ISHAI, Yuval; MALKIN, Tal; NISSIM, Kobbi; STRAUSS, Martin J; WRIGHT, Rebecca N. Secure multiparty computation of approximations. *ACM transactions on Algorithms (TALG)*. 2006, vol. 2, no. 3, pp. 435–472.
30. PORAT, Ely; LIPSKY, Ohad. Improved Sketching of Hamming Distance with Error Correcting. In: *Combinatorial Pattern Matching, 18th Annual Symposium, CPM*. Springer, 2007, vol. 4580, pp. 173–182. Available from DOI: 10.1007/978-3-540-73437-6_19.

31. KOCIUMAKA, Tomasz; PORAT, Ely; STARIKOVSKAYA, Tatiana. Small-space and streaming pattern matching with k edits. In: *2021 IEEE 62nd Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE, 2022, pp. 885–896.
32. KNUTH, Donald E.; JR., James H. Morris; PRATT, Vaughan R. Fast Pattern Matching in Strings. *SIAM J. Comput.* 1977, vol. 6, no. 2, pp. 323–350.
33. BOYER, Robert S; MOORE, J Strother. A fast string searching algorithm. *Communications of the ACM*. 1977, vol. 20, no. 10, pp. 762–772.
34. PORAT, Benny; PORAT, Ely. Exact and approximate pattern matching in the streaming model. In: *2009 50th Annual IEEE Symposium on Foundations of Computer Science*. IEEE, 2009, pp. 315–323.
35. CLIFFORD, Raphaël; FONTAINE, Allyx; PORAT, Ely; SACH, Benjamin; STARIKOVSKAYA, Tatiana. The k -mismatch problem revisited. In: *Proceedings of the twenty-seventh annual ACM-SIAM symposium on Discrete algorithms SODA*. SIAM, 2016, pp. 2039–2052.
36. BRESLAUER, Dany; GALIL, Zvi. Real-time streaming string-matching. *ACM Transactions on Algorithms (TALG)*. 2014, vol. 10, no. 4, pp. 1–12.
37. CLIFFORD, Raphaël; FONTAINE, Allyx; PORAT, Ely; SACH, Benjamin; STARIKOVSKAYA, Tatiana. Dictionary matching in a stream. In: *Algorithms-ESA 2015: 23rd Annual European Symposium, Patras, Greece, September 14-16, 2015, Proceedings*. Springer, 2015, pp. 361–372.
38. GOLAN, Shay; PORAT, Ely. Real-time streaming multi-pattern search for constant alphabet. In: *25th Annual European Symposium on Algorithms (ESA 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.
39. GOLAN, Shay; KOPELOWITZ, Tsvi; PORAT, Ely. Towards optimal approximate streaming pattern matching by matching multiple patterns in multiple streams. In: *45th International Colloquium on Automata, Languages, and Programming (ICALP 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.
40. GAWRYCHOWSKI, Paweł; STARIKOVSKAYA, Tatiana. Streaming dictionary matching with mismatches. *Algorithmica*. 2019, pp. 1–21.
41. STARIKOVSKAYA, Tatiana. Communication and streaming complexity of approximate pattern matching. In: *28th Annual Symposium on Combinatorial Pattern Matching (CPM 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.
42. RADOSZEWSKI, Jakub; STARIKOVSKAYA, Tatiana. Streaming k -mismatch with error correcting and applications. *Information and Computation*. 2020, vol. 271, p. 104513.
43. GOLAN, Shay; KOCIUMAKA, Tomasz; KOPELOWITZ, Tsvi; PORAT, Ely. The Streaming k -Mismatch Problem: Tradeoffs Between Space and Total Time. In: GØRTZ, Inge Li; WEIMANN, Oren (eds.). *31st Annual Symposium on Combinatorial Pattern Matching (CPM 2020)*. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020, vol. 161, 15:1–15:15. Leibniz International Proceedings in Informatics (LIPIcs). ISBN 978-3-95977-149-8. ISSN 1868-8969. Available from DOI: 10.4230/LIPIcs.CPM.2020.15.

44. BACKURS, Arturs; INDYK, Piotr. Edit distance cannot be computed in strongly subquadratic time (unless SETH is false). In: *Proceedings of the forty-seventh annual ACM symposium on Theory of computing STOC*. 2015, pp. 51–58.
45. LANDAU, Gad M; VISHKIN, Uzi. Efficient string matching with k mismatches. *Theoretical Computer Science*. 1986, vol. 43, pp. 239–249.
46. GALIL, Zvi; GIANCARLO, Raffaele. Improved string matching with k mismatches. *ACM SIGACT News*. 1986, vol. 17, no. 4, pp. 52–54.
47. AMIR, Amihood; LEWENSTEIN, Moshe; PORAT, Ely. Faster algorithms for string matching with k mismatches. *Journal of Algorithms*. 2004, vol. 50, no. 2, pp. 257–275.
48. CHARALAMPOPOULOS, Panagiotis; KOCIUMAKA, Tomasz; WELLNITZ, Philip. Faster approximate pattern matching: A unified approach. In: *2020 IEEE 61st Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE, 2020, pp. 978–989.
49. GAWRYCHOWSKI, Pawel; UZNANSKI, Przemyslaw. Towards unified approximate pattern matching for hamming and l_1 distance. In: *45th International Colloquium on Automata, Languages, and Programming (ICALP 2018)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2018.
50. CHAN, Timothy M; GOLAN, Shay; KOCIUMAKA, Tomasz; KOPELOWITZ, Tsvi; PORAT, Ely. Approximating text-to-pattern Hamming distances. In: *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing*. 2020, pp. 643–656.
51. LANDAU, Gad M; VISHKIN, Uzi. Fast parallel and serial approximate string matching. *Journal of algorithms*. 1989, vol. 10, no. 2, pp. 157–169.
52. COLE, Richard; HARIHARAN, Ramesh. Approximate string matching: A simpler faster algorithm. *SIAM Journal on Computing*. 2002, vol. 31, no. 6, pp. 1761–1782.
53. CHARALAMPOPOULOS, Panagiotis; KOCIUMAKA, Tomasz; WELLNITZ, Philip. Faster pattern matching under edit distance: a reduction to dynamic puzzle matching and the Seaweed Monoid of permutation matrices. In: *2022 IEEE 63rd Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE, 2022, pp. 698–707.
54. HAEUPLER, Bernhard; SHAHRASBI, Amirbehshad. Synchronization Strings and Codes for Insertions and Deletions - a Survey. 2021. Available from arXiv: 2101.00711.
55. HAEUPLER, Bernhard; SHAHRASBI, Amirbehshad. Synchronization Strings: Codes for Insertions and Deletions Approaching the Singleton Bound. *J. ACM*. 2021, vol. 68, no. 5, 36:1–36:39. Available from DOI: 10.1145/3468265.
56. HAEUPLER, Bernhard; SHAHRASBI, Amirbehshad; VITERCIK, Ellen. Synchronization Strings: Channel Simulations and Interactive Coding for Insertions and Deletions. In: *Proceedings of the International Conference on Automata, Languages, and Programming (ICALP)*. 2018, 75:1–75:14.

57. HAEUPLER, Bernhard; SHAHRASBI, Amirbehshad. Synchronization Strings: Explicit Constructions, Local Decoding, and Applications. In: *Proceedings of the ACM Symposium on Theory of Computing (STOC)*. 2018, pp. 841–854.
58. HAEUPLER, Bernhard; SHAHRASBI, Amirbehshad; SUDAN, Madhu. Synchronization Strings: List Decoding for Insertions and Deletions. In: *Proceedings of the International Conference on Automata, Languages, and Programming (ICALP)*. 2018.
59. CHENG, Kuan; HAEUPLER, Bernhard; LI, Xin; SHAHRASBI, Amirbehshad; WU, Ke. Synchronization Strings: Highly Efficient Deterministic Constructions over Small Alphabets. In: *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 2019, pp. 2185–2204.
60. RUBINSTEIN, Aviad. Hardness of approximate nearest neighbor search. In: DIAKONIKOLAS, Ilias; KEMPE, David; HENZINGER, Monika (eds.). *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2018, Los Angeles, CA, USA, June 25-29, 2018*. ACM, 2018, pp. 1260–1268. Available from DOI: 10.1145/3188745.3188916.
61. KARTHIK, CS; MANURANGSI, Pasin. On closest pair in euclidean metric: Monochromatic is as hard as bichromatic. *Combinatorica*. 2020, vol. 40, no. 4, pp. 539–573.
62. ABOUD, Amir; BATENI, MohammadHossein; COHEN-ADDAD, Vincent; KARTHIK C. S.; SEDDIGHIN, Saeed. On Complexity of 1-Center in Various Metrics. In: MEGOW, Nicole; SMITH, Adam D. (eds.). *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques, APPROX/RANDOM 2023, September 11-13, 2023, Atlanta, Georgia, USA*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023, vol. 275, 1:1–1:19. LIPIcs. Available from DOI: 10.4230/LIPICs.APPROX/RANDOM.2023.1.
63. COHEN-ADDAD, Vincent; KARTHIK C. S.; LEE, Euiwoong. Johnson Coverage Hypothesis: Inapproximability of k-means and k-median in L_p -metrics. In: NAOR, Joseph (Seffi); BUCHBINDER, Niv (eds.). *Proceedings of the 2022 ACM-SIAM Symposium on Discrete Algorithms, SODA 2022, Virtual Conference / Alexandria, VA, USA, January 9 - 12, 2022*. SIAM, 2022, pp. 1493–1530. Available from DOI: 10.1137/1.9781611977073.63.
64. FLEISCHMANN, Henry L.; GAVVA, Surya Teja; KARTHIK C. S. On Approximability of Steiner Tree in L_p -metrics. In: WOODRUFF, David P. (ed.). *Proceedings of the 2024 ACM-SIAM Symposium on Discrete Algorithms, SODA 2024, Alexandria, VA, USA, January 7-10, 2024*. SIAM, 2024, pp. 1669–1703. Available from DOI: 10.1137/1.9781611977912.67.
65. COHEN-ADDAD, Vincent; FEUILLOLEY, Laurent; STARIKOVSKAYA, Tatiana. Lower bounds for text indexing with mismatches and differences. In: Society for Industrial and Applied Mathematics, 2019. SODA '19.
66. BHATTACHARYA, Sudatta; DEY, Sanjana; GOLDENBERG, Elazar; HABIB, Mursalin; HAEUPLER, Bernhard; C.S, Karthik; KOUCKÝ, Michal. *Constant Rate Isometric Embedding of Hamming Metric into Edit Metric*. [N.d.]. submitted to FOCS'25.

67. *GitHub Repository* [<https://github.com/sudatta18107/Ham-to-ED.git>]. [N.d.].
68. TISKIN, Alexander. Semi-local string comparison: Algorithmic techniques and applications. *Mathematics in Computer Science*. 2008, vol. 1, pp. 571–603.
69. BHATTACHARYA, Sudatta; DEY, Sanjana; GOLDENBERG, Elazar; KOUCKÝ, Michal. Many Flavors of Edit Distance. In: *44th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2024)*. 2024. ISBN 978-3-95977-355-3. ISSN 1868-8969. Available from DOI: 10.4230/LIPIcs.FSTTCS.2024.11.
70. BRINGMANN, Karl; KÜNNEMANN, Marvin. Quadratic Conditional Lower Bounds for String Problems and Dynamic Time Warping. In: *2015 IEEE 56th Annual Symposium on Foundations of Computer Science*. 2015, pp. 79–97. Available from DOI: 10.1109/FOCS.2015.15.
71. ABOUD, Amir; BRINGMANN, Karl. Tighter Connections Between Formula-SAT and Shaving Logs. In: CHATZIGIANNAKIS, Ioannis; KAKLAMANIS, Christos; MARX, Dániel; SANNELLA, Donald (eds.). *45th International Colloquium on Automata, Languages, and Programming, ICALP 2018, July 9-13, 2018, Prague, Czech Republic*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018, vol. 107, 8:1–8:18. LIPIcs. Available from DOI: 10.4230/LIPICS.ICALP.2018.8.
72. GILBERT, E. N. A comparison of signalling alphabets. *The Bell System Technical Journal*. 1952, vol. 31, no. 3, pp. 504–522. Available from DOI: 10.1002/j.1538-7305.1952.tb01393.x.
73. VARSHAMOV, Rom Rubenovich. Estimate of the number of signals in error correcting codes. *Doklady Akad. Nauk, SSSR*. 1957, vol. 117, pp. 739–741.

List of Figures

2.1	The hierachical decomposition of x	35
2.2	Decomposition of $B^x(\ell, i)$ after compression and split.	36
3.1	Rolling sketch.	45
3.2	The alignment of text and pattern grammars after arrival of some text symbol. The pattern P is represented by grammars G_1^P, \dots, G_r^P . Grammars G_1^P, \dots, G_{r-R}^P are encoded by Enc and sent to the CKP-match algorithm as its pattern. The current text T is represented by the sequence of grammars $G_1^T, \dots, G_s^T, G_1^a, \dots, G_t^a$. Grammars G_1^T, \dots, G_s^T are encoded and committed to the CKP-match algorithm as its text. Grammars G_1^a, \dots, G_t^a are active grammars of the text, and might change as more symbols are added to the text.	68
5.1	On the left side, we have the decomposition of X and Y based on the Δ_{indel} alignment. On the right side, we see the decomposition and alignment of X and Y' following our construction in Section 5.2.3. The solid red arrows indicate that all characters of m_i^X are matched, while the dotted red arrows suggest that the characters of m_i^X are substituted. The shaded cells in gray indicate deletions. On the right-hand side, the blue dotted lines indicate the alignment of m_i^X with $m_i^{Y'}$	109
5.2	On the left side, we have the decomposition of two strings X and Y of length 20 each based on the Δ_{indel} alignment. On the right side, we see the decomposition and alignment of $E_1(X) = X$ and $E_3(Y) = \tilde{Y}$ as constructed by our Algorithm 16, where $k = 1$. Matching between characters is indicated by solid red arrows, while substitutions are denoted by dotted red arrows. The deleted cells or characters are shaded in grey while the substituted cells are shaded in green.	113

- 6.1 An illustration of the matching between the strings $E(X)$ and $E(Y)$. Arrows indicate matching coordinates, and dashed lines represent the beginning/end points of the segments. The first segment starts at $(1, 1)$ and ends at $(2, 3)$, as the first matched coordinate in the second block of $E(X)$ is mapped to the third block, and no character from the first block of $E(X)$ is mapped to that block. The second segment starts at $(3, 1)$ and ends at $(4, 1)$ (as the first matched coordinate in the second block of $E(X)$ is mapped to the third block, and there exists a character from the second block of $E(X)$ that is mapped to that block) 126
- 6.2 Block i' partially matches blocks j_1, j_2, j_3 in $E(Y)$. Consequently, no other block in $E(X)$ can be partially matched with j_2 . The red line illustrates the prohibited matching. 127

List of Tables

2.1	Table of parameters	38
-----	-------------------------------	----

List of Abbreviations

1. $\tilde{\mathcal{O}}$ - \mathcal{O} that hides *polylog* factors.
2. Δ_{Ham} - Hamming distance
3. Δ_{edit} - Edit distance
4. Δ_{indel} - Indel distance

List of Publications

1. Sudatta Bhattacharya and Michal Koucký, “Locally consistent decomposition of strings with applications to edit distance sketching”, in Proceedings of the 55th Annual ACM Symposium on Theory of Computing (STOC 2023)
2. Sudatta Bhattacharya and Michal Koucký, “Streaming k-Edit Approximate Pattern Matching via String Decomposition”, in 50th International Colloquium on Automata, Languages, and Programming (ICALP 2023).
3. Sudatta Bhattacharya, Sanjana Dey, Elazar Goldenberg and Michal Koucký, “Many Flavours of Edit distance”, to appear in FSTTCS 2024.
4. Sudatta Bhattacharya, Sanjana Dey, Elazar Goldenberg, Mursalin Habib, Bernhard Haeupler, Karthik C.S and Michal Koucký, “Constant Rate Isometric Embedding of Hamming Metric into Edit Metric”, preprint submitted to FOCS 2025.
5. Sudatta Bhattacharya, Zdeněk Dvořák, and Fariba Noorizadeh, “Chromatic number of intersection graphs of segments with two slopes”, in EuroComb 2023.

*For everyone who stayed with me through all seven chapters.
Mischief managed.!!!*