Master Thesis

On the Complexity of Recognizing Similarities between Streams

Author(s):
Svagerka, Michal

Publication Date:
2018-04

Permanent Link:
https://doi.org/10.3929/ethz-b-000258241

Rights / License:
In Copyright - Non-Commercial Use Permitted
On the Complexity of Recognizing Similarities between Streams

Master Thesis
Michal Švagerka
April 10, 2018

Advisor: Prof. Dr. Juraj Hromkovič
Department of Computer Science, ETH Zürich
Abstract

Streaming algorithms is a set of approaches to solving problems on large data sets with space constraints. In this work, we formalise the notion of streaming algorithms operating on two streams and study how different models affect the complexity of problems.

Furthermore, we look at two forms of edit distances and establish how they can be calculated in a streaming setting. We establish a very simple $\tilde{O}(\log n \cdot \epsilon^{-2})$ approximation algorithm for Hamming distance, which is on par with the state-of-the-art approaches. For the edit distance where the only possible operation is one removal of some contiguous substring, we establish a probabilistic algorithm with poly-logarithmic space complexity.
Acknowledgements

I would like to thank my advisor Prof. Juraj Hromkovič for introducing the field of streaming algorithms to me and outlining some interesting problems to solve. Furthermore, I’d like to thank Przemysław Uznański for many interesting discussions on the topic.

Many thanks go to Monika Steinová and Charlotte Knierim for proofreading my thesis and helpful comments about the thesis structure.

I am very grateful to my parents who supported me during my studies.
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>1</td>
</tr>
<tr>
<td>Preliminaries</td>
<td>5</td>
</tr>
<tr>
<td>2.1 Basic notation</td>
<td>5</td>
</tr>
<tr>
<td>2.2 Streaming algorithms</td>
<td>6</td>
</tr>
<tr>
<td>2.2.1 Deterministic algorithms</td>
<td>6</td>
</tr>
<tr>
<td>2.2.2 Time complexity</td>
<td>9</td>
</tr>
<tr>
<td>2.2.3 Space complexity</td>
<td>10</td>
</tr>
<tr>
<td>2.2.4 Probabilistic algorithms</td>
<td>11</td>
</tr>
<tr>
<td>2.3 Communication complexity</td>
<td>13</td>
</tr>
<tr>
<td>Model</td>
<td>17</td>
</tr>
<tr>
<td>3.1 Taxonomy</td>
<td>17</td>
</tr>
<tr>
<td>3.1.1 Free order</td>
<td>18</td>
</tr>
<tr>
<td>3.1.2 Synchronised order</td>
<td>18</td>
</tr>
<tr>
<td>3.1.3 Serialised order</td>
<td>18</td>
</tr>
<tr>
<td>3.1.4 Sketching order</td>
<td>19</td>
</tr>
<tr>
<td>3.1.5 Interleaved order</td>
<td>19</td>
</tr>
<tr>
<td>3.2 Basic results on model hierarchy</td>
<td>20</td>
</tr>
<tr>
<td>3.3 Model incomparability</td>
<td>22</td>
</tr>
<tr>
<td>3.3.1 String equality problem</td>
<td>22</td>
</tr>
<tr>
<td>3.3.2 Common majority problem</td>
<td>22</td>
</tr>
<tr>
<td>3.3.3 Summary</td>
<td>26</td>
</tr>
<tr>
<td>3.4 Interleaved input</td>
<td>27</td>
</tr>
<tr>
<td>Hamming distance</td>
<td>31</td>
</tr>
<tr>
<td>4.1 Definition</td>
<td>31</td>
</tr>
<tr>
<td>4.2 Deterministic exact variant</td>
<td>31</td>
</tr>
</tbody>
</table>
## Contents

4.3 Deterministic approximative variant ........................................... 32  
4.4 Probabilistic exact variant ......................................................... 32  
4.5 Probabilistic approximative variant ............................................. 33  

5 Substring cut 39  
5.1 Decision problem ................................................................. 40  
5.1.1 Deterministic variant ......................................................... 40  
5.1.2 Probabilistic variant ............................................................ 41  
5.2 Cut value ................................................................................. 50  
5.3 Cut index ................................................................................. 51  

6 Conclusion 55  

Bibliography 57
Chapter 1

Introduction

The desire to open a bottle of wine has a prosaic solution – spiralling a metallic helix into the cork sealing the bottle and then pulling to extract could not possibly get any easier. An inquisitive mind might not stop there and ask: “What if you are not allowed to use a corkscrew?” This provocative question has a clean motivation, as perhaps we might find ourselves in a situation where the most mundane solution is impractical or impossible to use. For instance, sharp objects are prohibited by most major airlines in a cabin baggage, and a corkscrew may be classified as such. Our inability to carry one to our destination may provoke an alternative approach.

Similarly in the world of theoretical computer science, problems that are considered solved are often revisited in fresh contexts with some set of restrictions, which might seem arbitrary at first impression. Limiting our options this way gives us insight to the intrinsic hardness of a problem. In certain applications, the simplicity of implementing a finite automaton compared to a Turing machine may be appreciated.

Furthermore, it may lead to useful applications where the original approach would be less effective for practical reasons. An illustrative and well-studied instance of this phenomenon are distributed algorithms. Since it became apparent that increasing CPU clock rate indefinitely is not feasible, the necessity of distributing data sets across multiple devices and the ability to consolidate results of partial computations emerged. Many sequential algorithms that might even be optimal in RAM model of computing are suddenly suboptimal and give precedence to others, that perform better in such conditions.

A very simple example would be the problem of finding the maximum of a multiset. Formally, let us have an array $A$ of integers of size $10^{12}$. The goal is to find the maximum element $A_i$. This is a problem with a straightforward sequential solution — read the input array one element at a time.
and maintain the maximum found so far. We only use a constant amount of memory (apart from the input itself), and if one comparison takes 1 ns, then the whole algorithm will take roughly 1000 seconds.

Let us now divide $A$ to blocks of size $10^6$. Additionally, we impose the following restriction on our algorithm:

While reading the block $i$, we may not use any information about any other block $j \neq i$, until we finish reading that block.

It is not complicated to accommodate for this restriction — one can simply adjust the algorithm to compute and store the maximum of each block. Once that is finished, we can compute the maximum of the maxima. This alone has not gained us anything — more memory is now required and there are few extra steps performed (although this would be hardly noticeable on the running time). The tangible benefit of this construction is, however, that we can now use $10^6$ different machines and assign one block to each of them. Afterwards, the machines share their maxima and compute the final result. Even if this consolidation takes 100 ms, the whole process will finish before a single second passes. Indeed, a restriction that seemed unmotivated allowed us to achieve an algorithm that is in some settings better than the “optimal” algorithm for the general case!

In this thesis we look at streaming algorithms. The main premise of streaming algorithms is the following restriction, not dissimilar to what we have outlined for the maximum element problem:

After reading the $i$-th character of the input, a character with index $j < i$ may no longer be read.

In other words, the input for a given problem may only be read sequentially, and only once. This restriction is sometimes very limiting, but nevertheless yields algorithms for many practical applications handling very large data sets. For instance, it has shown to be a useful concept for monitoring large-scale networks, or for estimating the size of a database query response.

The first papers studying streaming algorithms (although they do not call it as such) are the 2-pass median finding algorithm by Munro and Paterson [MP80] and Flajolet-Martin algorithm [FM85] for finding the cardinality of a large set. The field was formalised and popularised by Alon, Matias and Szegedy [AMS99].

The main focus of this work is to look at applications where there is more than just a single input. In some sense, this may be viewed as a field in the intersection of distributed algorithms and streaming algorithms. Multi-stream algorithms have not been extensively studied from this perspective. Most of the literature tackles only a particular problem in an ad-hoc manner.
Some studied problems include edit (Levenshtein) distance [CGK16] and Hamming distance [PL07].

We will formalise different models with regards to how exactly can input be organised. For instance, we may have a single machine reading both input streams in parallel. In a more restricted model, that is commonly referred to as “sketching”, the input will be split across two machines and the main focus will be on reducing the amount of data that needs to be communicated in order to achieve the requested results.

Then we will study two types of edit distance, summarise the known results and improve upon them. In particular, we will first look at a metric called Hamming distance in which the only allowed modification is to change a character at a particular index. The Hamming distance is then defined as the number of characters that need to be changed in order to obtain one string from the other. This problem has already been researched and we provide the respective survey in conjunction with our results. We obtain a probabilistic $(1 + \epsilon)$-approximation algorithm using $\tilde{O}(\log n \cdot \epsilon^{-2})$ space, which is almost optimal compared to the known results. However, our algorithm is much simpler to implement than its counterparts.

The second type of edit distance we study is a non-metric measure where the only possible operation is a removal of a single contiguous block of characters. Note that this is also a special case of Levenshtein distance. We refer to this measure as \textsc{substring-cut}. We assess the complexity of three distinct questions related to \textsc{substring-cut}:

- the associated decision problem: Having two strings $S$ and $T$, is it true that one is a substring cut of another?

- the recovery problem: Having two strings $S$ and $T$, where $T$ is obtained from $S$ by removing a single contiguous substring of characters, recover this missing substring. In other words, recover the exact difference between two strings.

- the location problem: Having again two strings $S$ and $T$ obtained by the process described above, determine the position of the removed substring.

We show that all those problems are hard to solve deterministically. Furthermore, we provide an efficient probabilistic algorithm for all three subproblems that has an arbitrary small probability of failure. More precisely, we can decide the decision problem with small possibility of a false positive. In the second subproblem, the difference can be recovered exactly, again with high probability. The third subproblem we solve only approximately, but the error can be made arbitrarily small.
1. Introduction

The structure of the thesis is as follows. In the Chapter 2, we provide basic definitions and results relating to the concept of streaming algorithms. In Chapter 3, we look at streaming algorithms in greater detail, namely we compare different formalisms with regard to multiple input streams. In Chapter 4 and 5, we study different similarity measures between strings, namely Hamming distance and Substring cut respectively. Chapter 6 summarises our contribution and outlines some open problems for further research.
In this chapter, we provide the basic tools we are using throughout this thesis. In Section 2.1 we provide a list of used notation. In Section 2.2 we define the model of streaming algorithm precisely along with ways we measure their performance. In Section 2.3 we introduce the main tool shown to prove lower bounds on the complexity of certain problem – the communication complexity.

2.1 Basic notation

Throughout the thesis, we heavily use the concept of strings—finite words over a certain alphabet, usually denoted $\Sigma$.

We use:

- $|S|$ denotes the length of the string $S$ in characters.
- $\lambda$ denotes the empty string.
- $S[i]$ for the character of $S$ at index $i$. We use 1-based indexing throughout this thesis.
- $S[a..b]$ for a substring of $S$, containing elements between indices $a$ and $b$, inclusive.
- Sometimes we treat beginning and end of a string as a special character. In that case $\hat{\text{\textasciitilde}}$ and $\$$ will be used, respectively. By convention, $S[0] = \hat{\text{\textasciitilde}}$ and $S[|S| + 1] = \$.
- $A \triangleleft B$ is the notation for concatenation\footnote{There are many competing symbols for concatenation in the literature, such as $A \cdot B$, $A + B$, $A \parallel B$, $(A, B)$, $A \oplus B$ or $AB$. The notation $A \triangleleft B$ is preferred as it is, unlike the alternatives, not used for other concepts.} of two strings $A$ and $B$.
- $[n]$ is the set of positive integers from 1 to $n$, inclusive.
2. Preliminaries

2.2 Streaming algorithms

2.2.1 Deterministic algorithms

We start by defining the streaming model of computation. In order to do this, we first formalise the notion of the state of the algorithm using its memory contents (using so called cache sets). Then we continue by splitting the runtime of the algorithm into several steps, which will allow us to reason about its runtime and memory consumption.

Definition 2.1 Let $\mathcal{C}$ be a (possibly infinite) set of finite strings over alphabet \{0, 1\}, that is prefix free. This means that for every $c_1, c_2 \in \mathcal{C}$, the length of the longest common prefix of $c_1$ and $c_2$ is strictly smaller than the length of both $c_1$ and $c_2$. We then call $\mathcal{C}$ a cache set.

As we will see shortly, a streaming algorithm is a composition of several regular algorithms. Here, a regular\(^2\) algorithm is a function $f : X \rightarrow Y \cup \{\bot\}$, where $X$ is a set of inputs, $Y$ is a set of outputs and $\bot$ is a special symbol for divergence. This function needs to be accompanied with a complete and unambiguous set of instructions of that yields its function value (in case $f(x) \in Y$) or that never terminates (if $f(x) = \bot$). Formally:

Definition 2.2 Let $X$ and $Y$ be arbitrary sets. By regular algorithm we denote any computable function $f : X \rightarrow Y \cup \{\bot\}$. That is, for every input value $x \in X$, there is either an output value $y \in Y$ that can be computed by the algorithm in finite amount of steps, or the algorithm does not terminate, which is denoted by the value $\bot$.

The exact model for this regular algorithm does not matter to us, although it is required to be Turing-complete. It is important to understand that the streaming algorithm will inherit the properties and limitations of the model of the algorithms of which it is composed. In other words, when the regular algorithm operates on a pointer machine, the streaming algorithm will also operate on one. Again, the precise classification of the model is not important to us, because the running time and space characteristics are usually within polynomial factors of each other. Such granularity is sufficient for us.

\(^2\)The word regular is used to contrast the notion of a streaming algorithm, which is to be defined.
2.2. Streaming algorithms

**Definition 2.3** A deterministic streaming algorithm is a tuple

\[ \mathcal{A} = (\Sigma, \mathcal{X}, C, Y, A_{\text{init}}, A_{\text{update}}, A_{\text{output}}) \],

where:

- \( \Sigma \) is a set of input characters,
- \( \mathcal{X} \) is a set of input streams, which are finite words over the alphabet \( \Sigma \),
- \( C \) is a cache set,
- \( Y \) is the set of possible output values,
- \( A_{\text{init}} : \mathbb{N} \to C \cup \{ \bot \} \) is a regular algorithm that sets up the initial cache state based on the length of the input,
- \( A_{\text{update}} : C \times \Sigma \to C \cup \{ \bot \} \) is a regular algorithm that reads an input character and updates the cache,
- \( A_{\text{output}} : C \to Y \cup \{ \bot \} \) is a regular algorithm that, after reading all characters, maps the cache state into an output value.

Armed with this definition, we now show how exactly the algorithm operates on a certain input.

**Definition 2.4** Let \( \mathcal{A} = (\Sigma, \mathcal{X}, C, Y, A_{\text{init}}, A_{\text{update}}, A_{\text{output}}) \) be a deterministic streaming algorithm and \( S = (x_1, x_2, \ldots, x_n) \in \mathcal{X} \) be a stream of length \( n \). We define \( C_{\mathcal{A}}(S) = (c_0, c_1, c_2, \ldots, c_n) \) where \( c_i \in C \cup \{ \bot \} \) to be the sequence of intermediate cache states, formally:

\[
\begin{align*}
  c_0 &= A_{\text{init}}(n) \\
  \forall i \in [n] : c_i &= \begin{cases} 
    A_{\text{update}}(c_{i-1}, x_i) & c_{i-1} \in C \\
    \bot & c_{i-1} = \bot
  \end{cases}
\end{align*}
\]

We call \( C_{\mathcal{A}}(S) \) a computation on input \( S \).

Furthermore, we define the output of the algorithm to be the result it returns:

\[
\mathcal{A}(S) := \begin{cases} 
  A_{\text{output}}(c_n) & c_n \in C \\
  \bot & c_n = \bot
\end{cases}
\]

Note that as we restricted ourselves on deterministic algorithms for the moment, \( C_{\mathcal{A}}(S) \) is indeed uniquely determined by \( \mathcal{A} \) and \( S \).

An algorithm is only useful when it solves a particular problem. Understanding how a streaming algorithm operates, we now relate it to some expected output. For this, we need the following simple definitions:
2. Preliminaries

Definition 2.5 Let $\mathcal{X}$ be a set of inputs – finite strings over an alphabet $\Sigma$. Furthermore, let $\mathcal{Y}$ be the set of outputs and $f$ a function $f : \mathcal{X} \to \mathcal{Y}$. Then the tuple

$$\mathcal{P} = (\Sigma, \mathcal{X}, \mathcal{Y}, f)$$

is called a (function) problem. That is, for every input string in $\mathcal{X}$, there is exactly one correct output from the set $\mathcal{Y}$.

A problem is called a decision problem, if $\mathcal{Y} = \{\text{true, false}\}$.

For brevity, we use notation $f_{\mathcal{P}}$ to denote the function $f$ of a problem $\mathcal{P}$.

Definition 2.6 Let $\mathcal{A} = (\Sigma, \mathcal{X}, \mathcal{C}, \mathcal{Y}, A_{\text{init}}, A_{\text{update}}, A_{\text{output}})$ be an algorithm and $\mathcal{P} = (\Sigma', \mathcal{X}', \mathcal{Y}', f)$ be a problem. We say that these two are compatible, whenever

$$\mathcal{X}' \subseteq \mathcal{X}, \mathcal{Y} \subseteq \mathcal{Y}'$$

In other words, every valid input of a problem is also a valid input to the algorithm, and every output of an algorithm is a feasible (not necessarily correct) output of a problem.

In reality, those sets may as well be equal, but it is not necessary. The above technical definition assures us that we can compare the output of the algorithm and the function. We can obtain either an exact algorithm or an approximation one.

Definition 2.7 A deterministic algorithm $\mathcal{A}$ is said to solve a compatible problem $\mathcal{P}$, if and only if for every input $S \in \mathcal{X}$ it terminates and returns the correct value:

$$\forall S \in \mathcal{X} : A(S) = f_{\mathcal{P}}(S)$$

The above definition is extended to capture approximation algorithms in a straightforward way.

Definition 2.8 Let $\alpha > 1$ be a real number. A deterministic algorithm $\mathcal{A}$ is said to approximately solve a compatible problem $\mathcal{P}$ with approximation ratio $\alpha$, if and only if for every input $S \in \mathcal{X}$ the algorithm terminates and the result is within an $\alpha$ factor of $f_{\mathcal{P}}(S)$:

$$\forall S \in \mathcal{X} : \frac{f_{\mathcal{P}}(S)}{\alpha} \leq A(S) \leq \alpha f_{\mathcal{P}}(S)$$

Note that approximation algorithms are only well-defined when the set of outputs $\mathcal{Y}$ is a subset of real numbers. In particular, it is not meaningful to define an approximation algorithm for a decision problem.
2.2. Streaming algorithms

2.2.2 Time complexity

So far there is nothing truly special about streaming algorithms as we defined them—it is evident that whatever a streaming algorithm can do, a Turing machine can do too. Conversely, a streaming algorithm is able to simulate any Turing machine as shown in the following Lemma.

Lemma 2.9 Let \( \mathcal{X} \) be a set of finite strings over alphabet \( \Sigma \) and \( g: \mathcal{X} \to \mathcal{Y} \) be a computable function. Then there exists a streaming algorithm that solves the problem \((\Sigma, \mathcal{X}, \mathcal{Y}, g)\).

Proof We define the algorithm as follows:

- \( A_{\text{init}} = \lambda \) (the cache state is empty upon initialisation),
- \( A_{\text{update}}(c, x) = c^{-}x \) (update simply appends new character to memory),
- \( A_{\text{output}}(c) = g(c) \) (the output function simulates the Turing machine, interpreting the contents of the cache as input tape).

The distinguishing property of streaming algorithms is that in assessing their complexity, we examine each step individually and observe the memory contents in between reading individual characters.

The time complexity will be defined as the complexity of the step that takes the longest time to compute.

Definition 2.10 The streaming time complexity of a computation \( C_A(S) \) is the maximum time complexity of an individual step, that is

\[
\text{time}_A(S) := \max \left\{ \text{time}(A_{\text{init}}(S)),
\max_{i=1}^n \text{time}(A_{\text{update}}(c_{i-1}, x_i)),
\text{time}(A_{\text{output}}(c_n)) \right\}.
\]

Recall that \( A_{\text{init}}, A_{\text{update}}, \) and \( A_{\text{output}} \) are regular algorithms, so their running time is evaluated in the model in which they are defined, and not in the streaming model. We then naturally extend this definition to obtain worst case time complexity of an algorithm and the complexity of a problem.

Definition 2.11 The streaming time complexity of the algorithm \( A \) is the maximum time complexity of the algorithm on all input streams of length \( n \), expressed as a function of \( n \):

\[
\text{time}_A(n) := \max \{ \text{time}_A(S) \mid S \in \mathcal{X} \cap \Sigma^n \}.
\]
2. Preliminaries

The streaming time complexity of a problem \( \mathcal{P} \) is the minimum time complexity of a streaming algorithm that solves it:

\[
\text{time}_{\mathcal{P}}(n) := \min \{ \text{time}_A(n) \mid A \text{ solves } \mathcal{P} \}.
\]

There is a clear relation of the running time of a streaming algorithm and the optimal algorithm unconstrained by the input reading order: informally speaking, a streaming algorithm cannot be faster than a regular one. Simply put, a streaming algorithm is still an algorithm and lower bounds on its running time apply in the streaming model, too.

**Lemma 2.12** Let \( \mathcal{P} \) be a problem of a time complexity \( f(n) \). Then the streaming time complexity of the problem is

\[
\text{time}_{\mathcal{P}}(n) \geq \frac{f(n)}{n + 2}.
\]

**Proof** For the sake of contradiction, let \( A \) be a streaming algorithm solve problem \( \mathcal{P} \) with complexity

\[
\text{time}_A(n) < \frac{f(n)}{n + 2}.
\]

Then the complexity of an algorithm \( A' \) simulating \( A \) is:

\[
\text{time}_{A'}(S) = \text{time}(A_{\text{init}}(S)) + \sum_{i=1}^{n} \text{time}(A_{\text{update}}(c_{i-1}, x_i)) + \text{time}(A_{\text{output}}(c_n))
\]

\[
\leq (n + 2) \cdot \text{time}_A(S) < f(n). \quad \square
\]

2.2.3 Space complexity

On the other hand, the space complexity will be defined as the maximum length of a cache state produced during the computation:

**Definition 2.13** The space complexity of a computation \( C(S) \) is the maximum size of a cache state encountered during the computation, that is:

\[
\text{space}_A(S) := \max \{|c| \mid c \in C(S)\}.
\]

The space complexity of the algorithm \( A \) is the maximum cost of the algorithm on all input streams of length \( n \), expressed as a function of \( n \):

\[
\text{space}_A(n) := \max \{ \text{space}_A(S) \mid S \in \mathcal{X} \cap \Sigma^n \}.
\]

The streaming space complexity of a problem \( f \) is the minimum space complexity of a streaming algorithm that solves it:

\[
\text{space}_{\mathcal{P}}(n) := \min \{ \text{space}_A(n) \mid A \text{ solves } \mathcal{P} \}.
\]
Note that the space complexity of an algorithm based on the definition 2.13 may be smaller than the actual amount of memory used by a Turing machine computing the same result. The reason for this is that the definition purposefully ignores the space complexity of computing $c_{i+1}$ from $c_i$. We will see later that this definition is easier to work with, as the techniques originating in the communication complexity theory are much more suited for proving lower bounds in this model. Luckily, this omission is not a critical issue—as we will see now in Definition 2.14, the general goal is to consider algorithms for which both the space complexity and time complexity is poly-logarithmic in the input size. As the space complexity of every algorithm (hence also that of each step of a streaming algorithm) is always upper bounded by its time complexity, this does affect neither the class of problems we deem efficiently solvable, nor it will change whether an algorithm is considered efficient or not.

**Definition 2.14** We call a streaming algorithm $A$ space efficient if

$$\text{space}_A(n) = O(poly(\log n) \cdot poly(\log |\Sigma|)).$$

Otherwise, it is called space inefficient. Similarly, an algorithm is called time efficient if

$$\text{time}_A(n) = O(poly(\log n) \cdot poly(\log |\Sigma|)).$$

A problem $f$ is called efficiently solvable, if there exists a streaming algorithm that solves it and is both space and time efficient.

### 2.2.4 Probabilistic algorithms

As we will see later, a lot of problems become surprisingly hard in the streaming model in the above sense, hence Monte Carlo probabilistic algorithms are often studied instead. We provide a definition below.

**Definition 2.15** A probabilistic streaming algorithm is a probability space $A^p = (\Omega_A, F_A, P_A)$

of deterministic streaming algorithms.

Note that we view probabilistic algorithm as a distribution of deterministic algorithms from which an actual algorithm is sampled each time it is invoked on a new input. Save for defining adversarial structures, this definition is equivalent to the algorithm using a random stream of data as an additional input. This formalism will be easier to work with, as with regards to space and time complexity we can just argue about each possible deterministic algorithm separately.
2. Preliminaries

Definition 2.16 Let $\delta \in (0, 1)$. A probabilistic algorithm $A^p$ is said to solve problem $P$ with error $\delta$, if and only if it solves every instance with probability at least $1 - \delta$:

$$\forall S \in \mathcal{X}: P_A(\{A(S) = f_P(S) \mid A \in \Omega_A\}) \geq 1 - \delta.$$ 

Similar to the definition of an approximation algorithm in deterministic setting, we extend the above definition.

Definition 2.17 Let $\delta \in (0, 1)$ and $\alpha > 1$. A probabilistic algorithm $A^p$ is said to approximately solve problem $P$ with error probability $\delta$ and approximation ratio $\alpha$, if and only if for every instance $S$ the result is within an $\alpha$ factor of $f_P(S)$ with probability at least $1 - \delta$:

$$\forall S \in \mathcal{X}: P_A\left(\left\{\frac{f_P(S)}{\alpha} \leq A(S) \leq \alpha f_P(S) \mid A \in \Omega_A\right\}\right) \geq 1 - \delta.$$ 

The space and time complexity of the probabilistic algorithm $A^p$ is the worst case space and time complexity of any deterministic algorithm in its sample space.

Definition 2.18 The probabilistic streaming space complexity of algorithm $A^p$ is the maximum space complexity of all deterministic algorithms in $\Omega_A$:

$$\text{space}_{A^p}(n) := \max\{\text{space}_A(n) \mid A \in \Omega_A\}$$

The probabilistic streaming time complexity of algorithm $A^p$ is the maximum time complexity of all deterministic algorithms in $\Omega_A$:

$$\text{time}_{A^p}(n) := \max\{\text{time}_A(n) \mid A \in \Omega_A\}$$

We then proceed to define the time and space complexity of a problem in a straightforward fashion as the worst case over all deterministic algorithms in the probability space, as follows.

Definition 2.19 The probabilistic streaming space complexity of a problem $P$ is defined as minimum of the space complexities of all algorithms solving the given problem with error probability at most $\delta$:

$$\text{space}_P(n, \delta) := \min\{\text{space}_{A^p}(n) \mid P\left(A^p \text{ solves } P\right) \geq 1 - \delta\}$$

The probabilistic streaming time complexity of a problem $P$ is defined as minimum of the time complexities of all algorithms solving the given problem with error probability at most $\delta$:

$$\text{time}_P(n, \delta) := \min\{\text{time}_{A^p}(n) \mid P\left(A^p \text{ solves } P\right) \geq 1 - \delta\}$$
2.3 Communication complexity

A standard technique used to give lower bounds on the space complexity of a streaming algorithm is using the concept of fooling sets originating in the study of communication complexity, pioneered by [Yao79]. We will show the results given by [Abl96] that will be used in this thesis. For more comprehensive summary of communication complexity and methods to prove lower bounds on space complexity, consult [Hro97].

**Definition 2.20** Let $P$ be a problem over alphabet $\Sigma$ on a set of inputs $X$. For $r, t \in \mathbb{N}$ let there be two sets, the set of representatives $R \subseteq \Sigma^r$ and test set $T \subseteq \Sigma^t$, such that concatenation of every pair of elements is a valid input:

$$\forall R \in R, \forall T \in T : R^\lor T \in X.$$  \hspace{1cm} (2.1)

Furthermore, every pair of representatives is distinguishable by a certain test, that is:

$$\forall R_1, R_2 \in R, R_1 \neq R_2 \exists T \in T : f_P(R_1^\lor T) \neq f_P(R_2^\lor T).$$ \hspace{1cm} (2.2)

Then the pair $(R, T)$ is called a fooling set. The order of the fooling set is integer $n := r + t$.

Constructing a fooling set is a straightforward way of proving lower bounds on space complexity, demonstrated by the following standard theorem.

**Theorem 2.21** Let $n \in \mathbb{N}$. Let $(R_n, T_n)$ be a fooling set of order $n$ for a problem $P$. Then the deterministic streaming space complexity of problem $P$ is:

$$space_P(n) \geq \lceil \log_2 |R_n| \rceil.$$  

**Proof** Assume for contradiction that there is a deterministic streaming algorithm $A$ solving $P$ using $m < \lceil \log_2 |R| \rceil$ bits of memory on problem instances of length $n$. Fix the state of memory after reading $r$ characters (recall that $r$ is the length of each representative). There are at most $2^m < |R|$ possible cache states.

As $A$ is deterministic, each representative $R$ corresponds to a single cache state. By pigeonhole principle, there are two representatives $R_1 \neq R_2$ that yield the same cache state $c_r \in C$.

Using Property 2.2 of the fooling set we select $T \in T$ such that

$$P(R_1^\lor T) \neq P(R_2^\lor T).$$

Since the output is determined only by $c_r$ and $T$, we obtain

$$P(R_1^\lor T) = P(R_2^\lor T),$$

which is a contradiction. \hfill $\square$
2. Preliminaries

For the probabilistic setting, Ablayev provides lower bounds in [Ab96]. The first result gives a lower bound proportional to a logarithm of the deterministic space complexity, suggesting that non-determinism might sometimes be exponentially more efficient than determinism.

**Theorem 2.22 (Ablayev, Theorem 4)** Let $n \in \mathbb{N}$ and $\varepsilon \in (0, 0.5)$. Let $(R_n, T_n)$ be a fooling set of order $n$ for a problem $\mathcal{P}$. Then the probabilistic streaming space complexity of solving problem $\mathcal{P}$ with probability $p = 0.5 + \varepsilon$ is:

$$\text{space}_p(n, p) \geq \log_2 \log_2 |R_n| - \log_2 \log_2 \left(1 + \frac{1}{\varepsilon}\right) - 1.$$ 

The second lower bound can yield stronger results that can be of the same order as the deterministic lower bound, but it is harder to apply as it imposes restrictions on the fooling set. More precisely, the test set has to be exponentially small with respect to the set of representatives.

**Theorem 2.23 (Ablayev, Theorem 1)** Let $n \in \mathbb{N}$ and $p \in (0.5, 1)$. Let $(R_n, T_n)$ be a fooling set of order $n$ for a problem $\mathcal{P}$. Then the probabilistic streaming space complexity of problem $\mathcal{P}$ is:

$$\text{space}_p(n, p) \geq \log_2 |R_n| - |T_n| \cdot H_b(p) - 1,$$

where $H_b(p) = -p \log_2 p - (1 - p) \log_2 (1 - p)$ is binary entropy.

Note that for the above bound to be non-trivial, we indeed need to have a rather small test set $|T_n| < \frac{\log_2 |R_n|}{H_b(p)}$.

This is especially difficult because as the following lemma shows, the test set size needs to be asymptotically optimal for the given set of representatives.

**Lemma 2.24** Let $\mathcal{P}$ be a problem with set of outputs $\mathcal{Y}$. Then for every fooling set $(\mathcal{R}, \mathcal{T})$ holds that

$$|\mathcal{T}| \geq \log_{|\mathcal{Y}|} |\mathcal{R}| = \frac{\log_2 |\mathcal{R}|}{\log_2 |\mathcal{Y}|}.$$ 

**Proof** For each $R_i \in \mathcal{R}$, define tuple

$$y_i = (\mathcal{P}(R_i^\top T))_{T \in \mathcal{T}}.$$ 

As $(\mathcal{R}, \mathcal{T})$ is a fooling set, all $y_i$ have to be distinct. Since there are at most $|\mathcal{Y}|^{|\mathcal{T}|}$ such tuples, the claim follows.

Lastly, one can also adapt all the above results to the notion of approximation algorithms, too. In order to do that, we need to strengthen the definition of a fooling set. The reason for that is simple – when looking for exact solution the wrong outputs can be arbitrarily close to the correct ones to destroy the correctness, while this is in fact alright when seeking an approximation.
Definition 2.25 Let $\mathcal{P}$ be a problem over alphabet $\Sigma$ on a set of inputs $\mathcal{X}$ and $\alpha > 1$. For $r,t \in \mathbb{N}$ let there be two sets, the set of representatives $R \subseteq \Sigma^r$ and test set $T \subseteq \Sigma^t$, such that concatenation of every pair of elements is a valid input:

$$\forall R \in R, \forall T \in T : R \overrightarrow{T} \in \mathcal{X}.$$ (2.3)

Furthermore, every pair of representatives is distinguishable by a certain test, that is:

$$\forall R_1, R_2 \in R, R_1 \neq R_2 \exists T \in T : \mathcal{P}(R_1 \overrightarrow{T}) \notin [\alpha^{-2} \cdot \mathcal{P}(R_2 \overrightarrow{T}), \alpha^2 \cdot \mathcal{P}(R_2 \overrightarrow{T})].$$ (2.4)

Then the pair $(R, T)$ is called an $\alpha$-fooling set.

We will not be rephrasing the statements of Theorems 2.21, 2.22, 2.23, but each one of them can be adapted to the notion of an $\alpha$-fooling set to produce lower bounds on the space complexity of approximation algorithms.
In this thesis, we study how the complexity of certain algorithms is affected in streaming setting. Note that unlike many previous works that studied algorithms that have only one input string, the topics we covered here operate on two or more inputs to compare them. As the general streaming model defines the complexity only on one string, we need to distinguish how exactly the input is given and how the complexity will be calculated.

In Section 3.1 we give the taxonomy of the models. In Section 3.2 we show basic reductions among those models. Then, we interpose two different problems into these models and demonstrate how the complexity of these problems varies with the model in Section 3.3 to show additional results. Lastly, we look at the randomised model, which we call interleaved, at a greater detail in Section 3.4.

3.1 Taxonomy

The main issue is the order in which we are allowed to read the characters from the streams $S$ and $T$. We expect that the complexity of the studied problems may differ significantly. We propose the following taxonomy of the input orders:

- *free* – in this model, one may choose whether to read the next character from $S$, or $T$, depending on the state of memory

- *synchronised* – in this more restricted model, one alternates between reading a constant amount of characters from $S$ and a constant amount of characters from $T$. In the simplest form, one reads the input in the following order: $(S[1], T[1], S[2], T[2], S[3], T[3], \ldots, S[n], T[n])$.

- *serialised* – in this setting, one first reads the whole contents of the stream $S$ before processing the first character from $T$. 
3. Model

- **sketching** – in this most prohibitive setting, one reads $S$ and $T$ in parallel using isolated computational devices, denoted by $C_1$ and $C_2$. The communication is only allowed after the whole contents are read.

- **interleaved** – this setting provides a reading order that is selected at random.

We will show how to adapt the Definition 2.3 of the streaming algorithm to capture the subtleties of each model and also show how they compare in terms of computing strength.

### 3.1.1 Free order

The free order is the model in which the algorithm intuitively has the most freedom when reading the input.

**Definition 3.1** In the free model of computation, the algorithm needs to select from which stream to read. In order to do that, the cache state is prefixed by a bit indicating from which stream to read. This is achievable by taking $C' = C \times \{S, T\}$ – e.g. by prefixing all the cache states by a bit specifying from which stream to read.

One can easily see that such modification does not violate the prefix-freeness of $C'$.

### 3.1.2 Synchronised order

In the synchronised order, the perceived advantage is no longer present. The order is pre-defined, but as we will prove later, it is still very useful one for many problems.

**Definition 3.2** In the synchronised model of computation, the stream from which the character was read is pre-determined, and the origin alternates. This can be achieved by setting

$$\Sigma' = \Sigma \times \{S, T\}.$$  

### 3.1.3 Serialised order

The serialised model represents a very straightforward reduction from multiple streams to a single stream.

**Definition 3.3** In the serialised model of computation, the contents of streams $S$ and $T$ are concatenated (and separated by a special character $\$ not found anywhere else in the input). The algorithm then proceeds to read this single stream in a fashion identical to Definition 2.4.

This offers a straightforward application of the communication complexity to establish lower bounds on the complexity of a problem – the set of representatives contains some subset of the inputs of the first stream, and the test set is built by selecting the input set of the second stream.
3.1. Taxonomy

3.1.4 Sketching order

The sketching model is, as we will shortly see, the weakest of our taxonomy. We can view it as two distinct machines, each operating on one of the input streams, and only after the both fully process it, they are allowed to share their internal state. In this model, a streaming algorithm produces a so-called sketch of a stream – that extracts string features relevant to the solved problem.

**Definition 3.4** In the sketching model of computation, the update function maps two cache states into an output value:

\[ A_{\text{output}} : C \times C \rightarrow \mathcal{Y} \cup \{ \perp \} \]

3.1.5 Interleaved order

The last model that we define is different from all the previous ones, as it does not define a precise order, but a distribution of orders instead.

**Definition 3.5** Let S and T be strings of length n. Let

\[ F = \{ S, S, \ldots, S, T, T, \ldots, T \} \]

be a multiset containing n elements S and n elements T. Let \( \pi : [2n] \rightarrow F \) be a multiset permutation selected uniformly at random. For every \( i \in [2n] \), the \( i \)-th character read is the first unread character from stream \( \pi(i) \). This input order is called interleaved.

![Figure 3.1: All possibilities of how an interleaved input order with \( n = 2 \) can look like. All 6 options are equiprobable.](image-url)
3. Model

![Taxonomy of the input orders and the reductions among them. Each ellipse represents one model, accompanied with a reference to the corresponding definition. Each arrow points from a stronger model (i.e. one with better complexity) to a weaker one, along with a reference to the corresponding theorem. The dashed line represents a relation where the reduction adds an additive factor to the space complexity.](image)

3.2 Basic results on model hierarchy

In this section, we show how different input orders relate to each other. There are some comparable pairs of models, i.e. the space and time complexity of problems in one model can be always lower or upper bounded by the complexity in the other one. In Figure 3.2, one can see all the relations to be proven.

**Lemma 3.6** The free model is at least as strong as the synchronised. In other words, an algorithm $A$ that operates in synchronised model can also work in the free model while having the same time and space complexity.

**Proof** We take a synchronised algorithm $A$ and modify it to work in the free model.

Let $C' = C_A \times \{S, T\}$ and put

$$A'_{\text{init}}(n) = (A_{\text{init}}(n), S)$$

$$A'_{\text{update}}((c_i, S), x_i) = (A_{\text{update}}(c_i, x_i), T)$$

$$A'_{\text{update}}((c_i, T), x_i) = (A_{\text{update}}(c_i, x_i), S).$$

$\square$
3.2. Basic results on model hierarchy

Lemma 3.7 The free model is at least as strong as serialised – that is, an algorithm $A$ that operates in serialised mode can also work in free mode while having the same time and space complexity.

Proof The free algorithm is able to choose an arbitrary order of reading the input. In particular, it may first read the contents of $S$ and only then the contents of $T$. □

Intuitively, an algorithm working in the serialised model may always choose to ignore the information about the $S$ when reading $T$, and thus restrict itself to the sketching model. Indeed, there is a clear strength reduction between the two models.

Lemma 3.8 The serialised model is at least as strong as the sketching. In other words, an algorithm $A$ that operates in sketching mode can also work in serialised mode while having the same time and space complexity.

Proof We take a sketching algorithm $A$ and modify it to work in serialised model.

Let $C' = C_A \times C_A \times \{S, T\}$ and put

$$A'_{\text{init}}(n) = (A_{\text{init}}(n), A_{\text{init}}(n), S)$$

$$A'_{\text{update}}((c_s, c_t, S), x_i) = (A_{\text{update}}(c_s, x_i), c_t, S)$$

$$A'_{\text{update}}((c_s, c_t, S), S) = (c_s, c_t, T)$$

$$A'_{\text{update}}((c_s, c_t, T), x_i) = (c_s, A_{\text{update}}(c_t, x_i), T)$$

$$A'_{\text{output}}((c_s, c_t)) = A_{\text{output}}((c_s, c_t)).$$ □

Lemma 3.9 The synchronised model is at least as strong as the sketching. In other words, an algorithm $A$ that operates in sketching mode can also work in synchronised mode while having the same time and space complexity.

Proof Informally, the algorithm in synchronised model can simply opt to ignore the additional information it learns from the second stream. Formally, for a sketching algorithm $A$ we define the following synchronised algorithm on $C' = C \times C$:

$$A'_{\text{init}}(n) = (A_{\text{init}}(n), A_{\text{init}}(n))$$

$$A'_{\text{update}}((c_s, c_t), (x_i, S)) = (A_{\text{update}}(c_s, x_i), c_t)$$

$$A'_{\text{update}}((c_s, c_t), (x_i, T)) = (c_s, A_{\text{update}}(c_t, x_i))$$

$$A'_{\text{output}}((c_s, c_t)) = A_{\text{output}}((c_s, c_t)).$$ □
3. Model incomparability

We now proceed to show the incomparability of the serialised and synchronised model. This proof is performed in a constructive way, i.e. we discuss two problems, streq and common-k-majority and show that while the former is efficiently solvable only in the synchronised model, the latter is easier to solve in the serialised model.

3.3.1 String equality problem

One of the basic problems that is studied in the field of communication complexity is streq, sometimes also called Eq. The task is to determine, for two strings S and T of the same length, whether they are equal. We will briefly discuss the complexity of this problem in deterministic setting in different input orders.

Problem streq can benefit a lot from an appropriate input order. Indeed, in the synchronised mode, the space complexity of streq is $O(\log m)$ as only constantly many characters are required to be stored. One simply compares $S[i]$ to $T[i]$ and remembers whether an error was found. Using Lemma 3.6 we show that this is the case for the free model as well.

On the opposite side of the spectrum, we can show that in serialised input order, the problem is not efficiently solvable in deterministic setting. Thanks to the result of Lemma 3.8, the result will also be proven for the sketching model.

**Lemma 3.10** The space complexity of streq in the serialised model is $\Omega(n \log |\Sigma|)$.

**Proof** Both the set of representatives and the test set are all words of length $n$:

$$\mathcal{R} = \mathcal{T} = \Sigma^n.$$  

To summarise, streq is an example of a problem that is easier to solve in synchronised model than in serialised. Note that all the cases we considered so far were constrained to determinism, as probabilistic algorithms may be significantly stronger in some cases. Indeed, we will see later in Lemma 4.5 that the problem is efficiently solvable when one allows a small error probability, regardless of the model.

3.3.2 Common majority problem

In the previous subsection, we have demonstrated that, for at least some problems, the synchronised model is stronger than serialised. In this section, we show that the converse is also true by introducing a problem that will be significantly easier in the serialised model.
3.3. Model incomparability

We outline one further measure of similarity between two streams, and that is one based on the notion of common-majority. In this problem, we are given two streams \( S \) and \( T \), representing multisets drawing elements from a similar distribution. This similarity is based on the notion of so called majority elements, i.e. elements with high occurrence count. We give the formal definition below.

**Definition 3.11** Let \( S \) be a multiset of elements from \( \Sigma \) and \#s is the number of occurrences of \( s \) in the multiset \( S \). Furthermore, let \( k \in \mathbb{N} \setminus \{1\} \). Then \( k\text{-majority} \) is the (possibly empty) set containing elements with strictly more occurrences than a fraction of the multiset:

\[
\text{majority}_k(S) = \left\{ s \in S \mid \#s > \frac{n}{k} \right\}
\]

For \( k = 2 \), this is often simply referred to as majority.

The problem \( \text{majority}_k \) is, for a single input, solvable in RAM model with \( k \log |\Sigma| \) memory by a rather simple algorithm described by Boyer-Moore in [BM91]. We give a generalisation of the algorithm without proof in Algorithm 1.

In the first pass (on lines 5-12), \( k - 1 \) candidates for the majority elements are identified. The crucial observation that makes this algorithm work is that when one removes \( k \) distinct elements from the multiset, all \( k \)-majority elements still remain a majority. However, such removal may cause new elements to become a majority in the smaller multiset. To fix these false positives, a second pass is used (see lines 17-19) where the occurrences of each candidate element are counted. The space complexity is clearly \( \Theta(k \cdot (\log |S| + \log |\Sigma|)) \) and the time complexity \( O(|S| \cdot k) \).

The two pass approach is obviously unsuitable for a streaming algorithm. In fact, this problem can even be shown to not be efficiently solvable by the exact same fooling set as in the most frequent problem studied in [AMS99].

**Lemma 3.12** The problem \( k\text{-majority} \) is not efficiently solvable in the deterministic streaming setting.

**Proof** Let \( n \) be the size of the input and \( m \) size of the alphabet. There are two cases:

- \( n \leq m \): The set of representatives will be all different subsets of the alphabet of size \( \frac{n}{2} \):

\[
\mathcal{R} = \binom{\Sigma}{\frac{n}{2}}.
\]
### Algorithm 1  Boyer-Moore majority algorithm pseudocode

```
1: procedure MAJORITY(S, k)
2:   for i ∈ [k - 1] do
3:     Register[i] ← (null, 0)
4:
5:   for s ∈ S do
6:     if ∃i: Register[i].first = s then
7:       Register[i].second += 1
8:     else if ∃j: Register[j].second = 0 then
9:       Register[j] = (s, 1)
10:      else
11:         for r ∈ Register do
12:           r.second -= 1
13:
14:   for r ∈ Register do
15:     r.second = 0
16:
17:   for s ∈ S do
18:     if ∃i: Register[i].first = s then
19:       Register[i].second += 1
20:
21: Answer ← {}
22:   for r ∈ Register do
23:     if r.second > |S| / k then
24:       Answer ← Answer ∪ {r.first}
25: return Answer
```
3.3. Model incomparability

\[ n = c \cdot m \text{ for } m \in \mathbb{N} \]: Similar to above, the set of representatives will be subsets of alphabet of size \( \frac{m}{2} \), but now, every element will be repeated \( c \) times\(^1\):

\[ \mathcal{R} = \left( \frac{\Sigma}{m/2} \right)^c. \]

We see that the length of each representative is \( \frac{n}{2} \) as well.

In both cases, the test set will consist of a single element repeated \( n/2 \) times:

\[ \mathcal{T} = \{x^{\frac{n}{2}} \mid x \in \Sigma\}. \]

Now, a concatenation of a representative \( R \in \mathcal{R} \) and test \( x^{\frac{n}{2}} \in \mathcal{T} \) yields a stream with a 2-majority if and only if \( x \in R \). The space complexity is thus \( \Omega(\min(m, n) \log m) \).

Let us instead take a look at a slightly modified problem defined on two streams: \textsc{common-k-majority}.

**Definition 3.13** Let \( S \) and \( T \) be two streams over alphabet \( \Sigma \) and let \( k \in \mathbb{N} \setminus \{1\} \). Furthermore, we know that

\[ \text{majority}_k(S) = \text{majority}_k(T) \].

The task in \textsc{common-k-majority} problem is to find the set \( \text{majority}_k(S) \).

As we will see shortly, this would be an example of a problem that is efficiently solvable in serialised order, but not in synchronised, i.e. the exact opposite of what we have already demonstrated for \textsc{streq}, showing that synchronised and serialised models are indeed incomparable.

**Lemma 3.14** The problem \textsc{common-k-majority} is not efficiently solvable in synchronised order.

**Proof** Take \( S = T \). Now the problem is equivalent to finding \( k \)-majority in a single stream, which is not efficiently solvable by Lemma 3.12. \( \Box \)

**Lemma 3.15** For constant \( k \), the problem \textsc{common-k-majority} is efficiently solvable in serialised order.

**Proof** Run the first pass of Boyer-Moore algorithm on the contents of the stream \( S \) to identify the set of candidates for majority. This set contains \( \mathcal{O}(k) \) elements. Apply the second pass on the contents of the stream \( T \) to rule out false positives. \( \Box \)

\(^1\)Due to rounding issues, we only constructed a fooling set for certain values of \( n \), but still infinitely many, which is sufficient for finding an asymptotic lower bound on complexity.
3.3.3 Summary

The results we have shown for the two problems are summarised in Figure 3.3. We see that one of the problems, streq, is efficiently solvable in synchronised model but not in serialised, while the converse is true for common majority problem. This shows that those two models are incomparable and finishes the picture anticipated by the Figure 3.2.

<table>
<thead>
<tr>
<th>Model</th>
<th>String Equality</th>
<th>Common k-Majority</th>
</tr>
</thead>
<tbody>
<tr>
<td>free</td>
<td>$\Theta(\log m)$</td>
<td>$\Theta(k \log \max(m, n))$</td>
</tr>
<tr>
<td>serialised</td>
<td>$\Theta(n \log m)$</td>
<td>$\Theta(k \log \max(m, n))$</td>
</tr>
<tr>
<td>synchronised</td>
<td>$\Theta(\log n)$</td>
<td>$\Theta(\min(m, n) \log m)$</td>
</tr>
<tr>
<td>sketching</td>
<td>$\Theta(n \log m)$</td>
<td>$\Theta(\min(m, n) \log m)$</td>
</tr>
<tr>
<td>interleaved</td>
<td>exp. $\Theta(\sqrt{n} \log m)$</td>
<td>?</td>
</tr>
</tbody>
</table>

Figure 3.3: Comparison of the space complexity of two problems described in sections 3.3.1 and 3.3.2. For simplicity, $n = |S| + |T|$ is the combined length of the streams and $m = |\Sigma|$ is the size of the input alphabet.
3.4 Interleaved input

We have proven the relations amongst all models except the interleaved. Before proceeding further, we want to take an alternative view on it.

**Lemma 3.16** Let $S$ and $T$ be strings of length $n$ being read in interleaved input order. Let the number of remaining characters to be read from $S$ and $T$ be $s$ and $t$, respectively, for $s, t \in \mathbb{N}$.

Then the probability that the next element will be from $S$ equals to $\frac{s}{s+t}$.

**Proof** In the first $k := 2n - s - t$ elements of this permutation, there are exactly $n - s$ elements from $S$, and $n - t$ elements from $T$. There are two cases on the next element in the permutation:

- $\pi(k + 1) = S$: There are $\binom{k}{n-s}$ ways of permuting the elements corresponding to the data already read (i.e. the past), and $\binom{s+t-1}{s-1}$ ways of ordering the rest (i.e. the future), past the element $\pi(k + 1) = S$. There are thus $\binom{k}{n-s} \cdot \binom{s+t-1}{s-1}$ such permutations.

- $\pi(k + 1) = T$: The number of ways of permuting the past is the same as in previous case. The number of ways of permuting the future is slightly different, namely $\binom{s+t-1}{s}$. There are thus $\binom{k}{n-s} \cdot \binom{s+t-1}{s}$ such permutations.

The sets of permutations defined above are disjoint and they cover the space of all permutations that correspond to the data seen so far. They are also all equally likely. This implies

$$
P[\pi(k + 1) = S] = \frac{\binom{k}{n-s} \cdot \binom{s+t-1}{s-1}}{\binom{k}{n-s} \cdot \binom{s+t-1}{s-1} + \binom{k}{n-s} \cdot \binom{s+t-1}{s}}
$$

$$
= \frac{(s - 1)!t!}{(s + t)!} = \frac{s}{s + t}.
$$

The above result shows that the act of selecting a multiset permutation uniformly at random is equivalent to drawing balls from a hat, where the amount of balls of each of two colours is $n$, and drawn balls are not returned in the hat. There is one further correspondence worth noting, that will be used to analyse the strength of this model.

**Definition 3.17** Let $d \in \mathbb{N}$ be a constant and $S$ be a finite set of elements from $\mathbb{Z}^d$.

A sequence $L$ of $k$ elements from $\mathbb{Z}^d$:

$$L = (v_0, v_1, \ldots, v_k)$$
is called a lattice path if each consecutive difference \(v_i - v_{i-1}\) lies in \(S\).

Additionally, if \(S = \{(1,0),(0,1)\}\) then \(L\) is called a north-east lattice path or short NE-lattice path, where \((1,0)\) represents a north step and \((0,1)\) an east step.

One can see that there is a bijection between such multiset permutations and north-east lattice paths from \((0,0)\) and \((n,n)\) – the north step represents reading an element from \(S\) and the east step an element from \(T\). This means that instead of sampling a multiset permutation at random, one can sample a north-east lattice path.

Suppose now that we have a problem that is efficiently solvable in the synchronised model. We want to assess whether we can, given a pair of streams in an interleaved order, simulate a synchronised order by performing the synchronisation ourselves. This can be simply done by maintaining a queue for the “surplus” elements, from which they are removed once the element with corresponding index from the second stream arrives. The hope is that, in expectation, the queue will not be too large and will not increase the complexity above poly-logarithmic in \(n\). Unfortunately, the following result shows that this is not true. The expected size of the queue is in order of \(\sqrt{n}\).

Furthermore, the size of the queue is at least \(\sqrt{n}\) with a constant probability.

**Theorem 3.18** Let \(L\) be a NE-lattice path from \((0,0)\) to \((n,n)\) selected uniformly at random. The lattice path passes through points

\[
L = ((0,0), (x_1, y_1), \ldots, (x_{2n-1}, y_{2n-1}), (n,n))
\]

The width of the path is defined as the maximum distance from the main diagonal, i.e.

\[
W(L) = \max_{0 \leq i \leq 2n} |x_i - y_i|.
\]

The expected width of path \(L\) is \(\Theta(\sqrt{n})\).

**Proof** Let \(\mathcal{L}\) be the set of all NE-lattice paths from \((0,0)\) to \((n,n)\). Consider diagonals \(d_1(k) : y = x + k\) and \(d_2(k) : y = x - k\). A lattice path of width at least \(k\) intersects at least one of them. The probability of that happening is:

\[
\Pr[W(L) \geq k] = \Pr[(L \cap d_1(k) \neq \emptyset) \cup (L \cap d_2(k) \neq \emptyset)]
\]

To get a lower bound on this quantity, we can simply ignore one of the diagonals:

\[
\Pr[W(L) \geq k] \geq \frac{|\{L \in \mathcal{L} | L \cap d_1(k) \neq \emptyset\}|}{|\mathcal{L}|} \tag{3.1}
\]

To obtain an upper bound, we first simplify with union bound and then use the symmetry of the diagonals:

\[
\Pr[W(L) \geq k] \leq 2 \cdot \frac{|\{L \in \mathcal{L} | L \cap d_1(k) \neq \emptyset\}|}{|\mathcal{L}|} \tag{3.2}
\]
The expression in the denominator is a well known result: there are \( \binom{2n}{n} \) north-east lattice paths from \((0,0)\) to \((n,n)\).

To calculate the numerator, we use the André’s reflection [And87]: let \(L\) be a fixed lattice path that intersects the diagonal \(d_1(k)\) and let \((\hat{x}, \hat{x} + k)\) be the lattice point where this happens for the first time. If we reflect the rest of the path across \(d_1(k)\), we will obtain a path \(L'\) that ends in \((n - k, n + k)\).

We can see that this transformation is a bijection. Therefore the number of paths intersecting \(d_1(k)\) is equal to number of paths ending in \((n - k, n + k)\), which is simply \(\binom{2n}{n-k}\).

\[\begin{align*}
\mathbb{E}[W] & \geq |\sqrt{n}| \cdot \Pr[W(L) \geq |\sqrt{n}|] \\
& \geq |\sqrt{n}| \cdot \left(\frac{n - |\sqrt{n}|}{n}\right) \quad \text{(using (3.1))}
\end{align*}\]

\[= |\sqrt{n}| \cdot \prod_{i=0}^{\lfloor \sqrt{n} \rfloor - 1} \frac{n - i}{n + \lfloor \sqrt{n} \rfloor - i}\]

\[\geq |\sqrt{n}| \cdot \left(1 - \frac{\lfloor \sqrt{n} \rfloor}{n}\right)^{\lfloor \sqrt{n} \rfloor} \quad \text{(as } \lfloor \sqrt{n} \rfloor - i \geq 0)\]

\[\geq |\sqrt{n}| \cdot 2^{-2\frac{\lfloor \sqrt{n} \rfloor}{n} \cdot |\sqrt{n}|} \geq 0.25 |\sqrt{n}| \quad \text{(as } 1 - x \geq 2^{-2x} \text{ for } 0 < x \leq 0.5)\]
3. Model

which means that the expectation is lower bounded by a constant fraction of \( \sqrt{n} \).

To obtain an upper bound on the expectation, we calculate the probability of the width being at most \( k \) for every positive integer \( k \):

\[
\mathbb{E}[W] = \sum_{k=1}^{n} \Pr[W(L) \geq k]
\]

\[
\leq 2 \sum_{k=1}^{n} \binom{2n}{n-k} \left( \frac{k}{2n} \right) \tag{using (3.2)}
\]

\[
= 2 \sum_{k=1}^{n} \prod_{i=0}^{k-1} \left( 1 - \frac{k}{n + k - i} \right)
\]

\[
\leq 2 \sum_{k=1}^{n} \left( 1 - \frac{k}{2n} \right)^{k+i} \quad (k-i \leq n)
\]

\[
\leq 2 \sum_{k=1}^{n} e^{-\frac{k^2}{2n}} \quad (1-x \leq e^{-x})
\]

As the sum is bounded by Gaussian integral

\[
\sum_{k=1}^{\infty} e^{-\frac{k^2}{2n}} \leq \int_{0}^{\infty} e^{-\frac{x^2}{2n}} dx = \sqrt{\frac{n\pi}{2}},
\]

we have

\[
\mathbb{E}[W] \leq \sqrt{2\pi n}.
\]

Combining both results yields

\[
\mathbb{E}[W] = \Theta(\sqrt{n}).
\]

We conclude that by using extra \( \Theta(\sqrt{n}) \) space in expectation, one can reduce interleaved model into synchronised. For problems that are efficiently solvable on the synchronised model this gives a sublinear algorithm, although one that is not considered efficient with respect to the Definition 2.14.
In this chapter we study the problem of Hamming distance, a standard measure of similarity of two strings. Hamming distance naturally occurs in many applications, for instance in telecommunications or biology. We will first provide a formal definition and then conduct a survey of known results split into 4 sections. In each section, we consider a different combination of deterministic vs. probabilistic and exact vs. approximation algorithms. We analyse the problem on the weakest of all models – the sketching model. We also show some results which will be useful later in Chapter 5. Lastly, we show our contribution: a new probabilistic approximation algorithm for Hamming distance.

4.1 Definition

**Definition 4.1** Given two strings $S$ and $T$ of the same length $n$, their **Hamming distance** is the number of indices at which the corresponding characters are different. More formally,

$$\text{Hamming-distance}(S, T) = |\{i \in [n] \mid S[i] \neq T[i]\}|.$$

Having established the definition of the problem, we proceed to show its complexity in different settings.

4.2 Deterministic exact variant

In determinism the problem is not efficiently solvable, regardless of whether we require an exact solution or we settle for an approximation. This is due to the following correspondence to the String equality problem introduced in Section 3.3.1. Recall that we proved that this problem requires $\Theta(n)$ memory in the sketching model when seeking a deterministic solution.
4. Hamming distance

**Remark 4.2** Two strings $S$ and $T$ of the same length are equal if and only if their Hamming distance is 0.

\[ \text{STREQ}(S, T) \iff \text{HAMMING-DISTANCE}(S, T) = 0. \]

This reduction immediately yields the following result:

**Theorem 4.3** In deterministic exact setting Hamming-distance is solvable in $Θ(n \log |Σ|)$ memory.

**Proof** As determining STREQ($S, T$) has complexity of $Ω(n \log |Σ|)$, by Remark 4.2 an algorithm that solves the Hamming-distance($S, T$) problem using only $o(n \log |Σ|)$ memory would also solve the problem STREQ, which is a contradiction.

There is also an algorithm using $O(n \log |Σ|)$ memory: we simply store the whole contents of $S$ and compare the corresponding characters when reading $T$. □

### 4.3 Deterministic approximative variant

Things do not change significantly when an approximation is sought. Since the multiplicative error is irrelevant when approximating zero, we have the following result.

**Theorem 4.4** In deterministic approximative setting Hamming-distance is solvable in $Θ(n \log |Σ|)$ memory.

**Proof** For two strings $S = T$, the approximative algorithm needs to return 0 for any two-sided approximation ratio $α > 1$. We can thus use the same reduction to STREQ($S, T$) as in Theorem 4.3 to show that the complexity is $Ω(n \log |Σ|)$. □

### 4.4 Probabilistic exact variant

We first show that the special case decision problems, STREQ and IS-HAMMING-DISTANCE-1 are efficiently solvable in probabilistic setting.

**Lemma 4.5** Problem STREQ($S, T$) is solvable with failure probability at most $δ \in (0, \frac{1}{2})$ using $O(\log \frac{1}{δ})$ memory.

**Proof** Take a function $h : Σ^n \rightarrow [p]$ from a family of universal hash functions, where $n = |S| = |T|$ and $p$ is a prime in interval $(\frac{1}{2}, \frac{5}{3})$. Such a hash function can be represented using only $O(\log p) = O(\log \frac{1}{δ})$ memory, e.g. using Rabin-Karp rolling hash [KR87].
4.5 Probabilistic approximative variant

To determine whether $\text{Streq}(S, T)$, we simply compare $h(S)$ with $h(T)$ and return \textbf{Yes} if and only if $h(S) = h(T)$.

If $S = T$, we will have $h(S) = h(T)$ and the algorithm will produce correct value with probability 1. Conversely, if $S \neq T$, then since $h$ is an universal hash-function, we have $\Pr[h(S) = h(T)] = O(\delta)$. \hfill \Box

**Definition 4.6** Let $S$ be a string of length $n$. For $i \in \lceil \log n \rceil$ and $b \in \{0, 1\}$ we define a bit-indexed subsequence of $S$ based on the binary representation of an index $j$:

$$S_{b,i} = (S[j] \mid \text{i-th bit of } j \text{ is } b)_{j \in [n]}$$

**Lemma 4.7** Let $S$ and $T$ be two strings of length $n$. The following two conditions are equivalent:

- $\text{IS-hamming-distance-1}(S, T) = \text{Yes}$
- $\forall i \in \lceil \log n \rceil, \exists b \in \{0, 1\}: S_{b,i} = T_{b,i}$ and $S_{1-b,i} \neq T_{1-b,i}$

**Proof** $\Rightarrow$: Assume that $\text{HAMMING-DISTANCE}(S, T) = 1$ and the index of the character that differs is $d = b_kb_{k-1}\ldots b_1b_0$ where $k = \lceil \log n \rceil$. We can see that $S_{b,i}$ contains the character at index $i$ and therefore is different from $T_{b,i}$. Conversely, the subsequence $S_{1-b,i}$ does not contain the character at index $i$, and therefore is the same as $T_{1-b,i}$.

$\Leftarrow$: There are two cases.

Firstly, we can have $S = T$, i.e. $\text{HAMMING-DISTANCE}(S, T) = 0$. Then all corresponding bit-indexed subsequences of $S$ are equal.

Secondly, if $\text{HAMMING-DISTANCE}(S, T) \geq 2$, then we have at least two indices $d$ and $e$ for which the corresponding characters are different. Clearly, $d$ and $e$ differ in at least one bit, let it be $i$. Then $S_{d,i} \neq T_{d,i}$ and $S_{e,i} \neq T_{e,i}$ because the sequences contain the character at index $d$ and $e$, respectively. \hfill \Box

**Theorem 4.8** Problem \text{IS-hamming-distance-1} is solvable with failure probability at most $\delta$ using $O(\log n \cdot (\log \log n + \log \frac{1}{\delta}))$ memory.

**Proof** By Lemma 4.7, it suffices to check the condition $\text{Streq}(S_{b,i}, T_{b,i})$ for all values $i$ and $b$ using Lemma 4.5 with failure probability $\delta' = \frac{\delta}{2\log n}$, each with memory cost $O(\log \frac{1}{\delta'})$. By union bound, the probability of failure in any of them is at most $\delta$. \hfill \Box

### 4.5 Probabilistic approximative variant

There have been many efficient approximation algorithms for Hamming distance. There is an approach using random projections to the Euclidian space of small dimension based on Johnson Lindenstrauss lemma [JL84] in
space, and a solution based on so called $p$-stable distributions that can approximate any $p$-norm for a real $p \in (0, 2)$ using $O(\varepsilon^{-2})$ words, as described in [DIIM04]. We give an novel approach that achieves similar results up to logarithmic factors and is conceptually simpler.

To construct the algorithm for Hamming distance in a probabilistic approximation setting, we first present a result on a different, yet similar problem: number of distinct elements in a stream. We then show how to adapt algorithms solving this problem to produce an approximation of Hamming distance.

**Definition 4.9** Given stream $S$ of length $n$, over characters from alphabet $\Sigma$ of size $m$, the number-of-distinct-items is defined to be the cardinality of the set of its elements:

$$F_0 = |\{ S[i] \mid i \in [n] \}|$$

The first approximative solution for this problem using sublinear space has been shown by Flajolet and Martin in [FM85]. We list the algorithm and accompanying analysis as stated by Alon, Matias and Szegedy.

**Theorem 4.10** (Alon et al. [AMS99]) For every $\alpha > 2$, the number of distinct elements in a stream of length $n$ can be computed using $O(\log n)$ memory with 2-sided approximation ratio $\alpha$ with failure probability at most $\frac{2}{\alpha}$.

The algorithm can be outlined as follows:

1. Pick smallest integer $d$, such that $2^d > n$.
2. Let $0 < a, b < 2^d$ be two integers selected independently uniformy at random.
3. For every element of the input $S[i]$, calculate $z_i \equiv a \cdot S[i] + b \pmod{2^d}$.
4. Let $r_i := \text{C}L\text{Z}(z_i)$ be the number of leading zeroes of $z_i$.
5. Let $R := \max_{i \in [n]} (r_i)$.
6. Output $2^R$ as the approximate number of distinct items.

The intuition behind the algorithm is as follows: imagine we have $n$ distinct coloured balls, and infinite amount of bins numbered by natural numbers. Initially, all the balls are in the bin numbered 1. For each ball in the first bin, we flip a fair coin. If it comes up heads, the ball is forever kept in the first bin, otherwise it is moved into the bin number 2. We repeat this process for the second bin by moving some balls into the third bin, and so on. After some finite amount of steps, we arrive into the situation where all balls are kept in the current bin, and hence all following bins will remain empty. Let us analyse this process.
By linearity of expectation, the expected number of balls in the $k+1$-th bin is $\frac{n}{2^k}$. Assume that we do not know the total number of balls, but we know the highest numbered non-empty bin, denote this by $R$. Can we use this to estimate how many balls were there originally?

The non-trivial result on which the algorithm in Theorem 4.10 is based says that we can -- the number of balls is likely to be around $2^R$. Let us thus isolate the result on its concentration.

**Definition 4.11** Let $\Sigma$ be a ground set and $d$ a positive integer. Let $f$ be a family of functions $f_i: \Sigma \to [0, d]$ and the probability of $f_i(x) = y$ grows exponentially with $y$. More formally,

$$\forall x \in \Sigma, \forall y \in [0, d]: Pr_{f_i \in f} [f_i(x) \geq y] = 2^{-y}.$$ 

The we call $f$ a log-universal hash family.

**Lemma 4.12** Let $\beta > 1$ be a constant, and let $f$ be a log-universal hash family. Let $S \subset \Sigma$ be a fixed set such that $|S| < 2^d$. Then for the random variable $R := \max_{s \in S} f_i(s)$ we have

$$Pr[|R - \log |S|| \geq \beta] \leq 2^{1-\beta}.$$ 

For proof, consult [AMS99].

**Corollary 4.13** Putting $F_0 = 2^R$ and $\alpha = 2^\beta$ we obtain

$$Pr \left[ \frac{1}{\alpha} \leq \frac{F_0}{|S|} \leq \alpha \right] \geq 1 - \frac{2}{\alpha}.$$ 

To use this result to provide an approximation algorithm for Hamming distance, we need to establish a relation between it and the number of distinct items and then we adapt the presented algorithm. To do that, we add a single word to the definition of Hamming distance, one that keeps the definition equivalent, but hints the key insight to the reduction. Compare the two Definitions 4.1 and 4.14 -- their difference is underlined.

**Definition 4.14** Given two strings $S$ and $T$ of the same length $n$, their Hamming distance is the number of distinct indices at which the corresponding characters are different.

We will see how to isolate the indices at which the strings differ from those where they are the same in the following theorem, which also states the algorithm.
4. Hamming distance

Theorem 4.15 Let \( \alpha > 2, 0 < \beta < \frac{\alpha - 2}{\alpha \log n} \) and \( S \) and \( T \) be two streams of length \( n \). The Hamming-distance \((S, T)\) can be calculated using \( O(\log n \log \beta^{-1}) \) memory up to an approximation ratio \( \alpha \) with failure probability at most \( \frac{2}{\alpha} + \beta \log n \).

Proof Select a random function \( f \) from a family of log-universal hash functions to encode indices of characters in \( S \) into \( d + 1 \) buckets. This splits the strings \( S \) and \( T \) into \( d + 1 \) subsequences where every corresponding pair of subsequences consists of some (potentially zero) characters at some positions in \( S \) and \( T \). Denote these by \( S_0, S_1, \ldots, S_d, T_0, T_1, \ldots, T_d \).

For every \( i \) calculate \( \text{streq}(S_i, T_i) \) using Theorem 4.5, each with failure probability at most \( \delta = \beta \), and denote

\[
R := \max_{i \in [0, d]} \{ i | \neg \text{streq}(S_i, T_i) \}.
\]

By union bound, the probability that any of the invocations of \( \text{streq} \) returns incorrect value is at most \( \beta \log n \). We now assume that this does not happen, and we want to show that this constructions gives us the requested approximation of Hamming with error at most \( \frac{2}{\alpha} \). This combined would yield the claimed result.

Let \( I \) be the set of indices on which \( S[i] \neq T[i] \), note that

\[
\text{Hamming-distance}(S, T) = |I|.
\]

We can clearly see that the substrings \( S_i \) and \( T_i \) that are different are exactly those for which we can find a index witnessing the difference. Formally,

\[
\forall i \in I : \text{streq}(S_{f(i)}, T_{f(i)}) = \text{False}
\]

and conversely

\[
\text{streq}(S_j, T_j) = \text{False} \Rightarrow \exists i \in I : S[i] \neq T[i].
\]

This implies that

\[
R = \max_{i \in I} f(i).
\]

Since \( f \) is a log-universal hash function, the values \( f(i) \) are exponentially distributed. Hence using Lemma 4.12 and Corollary 4.13 we obtain

\[
\Pr\left[ \frac{1}{\alpha} \leq \frac{2^R}{|I|} \leq \alpha \right] \geq 1 - \frac{2}{\alpha},
\]

and therefore \( 2^R \) is an unbiased and concentrated estimator of Hamming distance. \( \square \)
The above theorem gives us an algorithm for \((2 + \epsilon)\)-approximation algorithm for Hamming distance using \(O(\log n (\log \log n + \log 1/\epsilon))\) memory with a constant failure probability. We will now turn this into \((1 + \epsilon)\)-approximation algorithm by employing a standard technique of repetition of experiments.

We now step back to the problem of counting the number of distinct items. The original result by Flajolet and Martin has been improved in the so called LogLog algorithm [DF03] for the NUMBER-OF-DISTINCT-ITEMS problem, with a revised analysis (and new name HyperLogLog) in [FFGea07]. The idea is simple – the ground set is split into \(m\) disjoint subsets and the Algorithm 4.10 is used on all of them separately. Finally, the result is averaged by taking harmonic mean and a corrective multiplicative factor is applied. Putting \(m = \Theta(1/\epsilon^2)\) yields an \((1 + \epsilon)\)-approximation algorithm with constant failure probability. Since the underlying mechanism of “buckets” for substrings is used as a black box, the analysis holds for our Hamming distance algorithm, too.

Note that there are further improvements in the HyperLogLog, this time titling it HyperLogLog++, presented in [HNH13]. These results further improve the concentration and correct some biases present for very low cardinalities (hence small Hamming distances). Unfortunately, the techniques employed can no longer be applied to sketching for Hamming distance in a straightforward way. Adapting these ideas is a potential direction for further research.

To summarise, we have achieved a \(\tilde{O}(\epsilon^{-2} \log n)\) space approximation algorithm for Hamming distance using sketches of substring chosen at random and reduction to NUMBER-OF-DISTINCT-ITEMS problem.
Chapter 5

Substring cut

In this section, we study the properties of the Substring-cut problem in various settings. In the Substring-cut problem, one has two strings $S$ and $T$, where $|S| > |T|$, and aims to determine whether $T$ can be constructed from $S$ by cutting one of its substrings and concatenating the remaining prefix and suffix. More formally, for string $S$ of length $|S| = n$ and string $T$ of length $|T| = n - k$, does there exist an index $i \in [0, n - k]$ such that $T = \text{Concatenate}(S[1..i], S[i + k + 1..n])$?

If yes, we will say that $T$ is a substring cut of $S$. Furthermore, we will call the index $i$ by the term cut index, the value $k$ as cut length and we refer to the substring $S[i + 1..i + k]$ as cut value.

Lemma 5.1 In the RAM model of computation, this Substring-cut can be solved in $\mathcal{O}(n)$ using $\mathcal{O}(1)$ memory.

Proof The answer is Yes if and only if

$$|\text{Longest-common-prefix}(S, T)| + |\text{Longest-common-suffix}(S, T)| \geq n - k.$$  

□

In the sketching model this is complicated by the fact that one may not read contents of $S$ and $T$ simultaneously. Furthermore, one can easily see that determining $\text{Longest-common-prefix}(S, T)$ is equivalent to finding largest non-negative integer $k$, such that \text{Streq}(S[1..k], T[1..k]). The problem \text{Streq} not being efficiently solvable in the deterministic setting provides an intuition that this naïve approach will probably not work for our problem.

The problem Substring-cut comes in multiple variants, depending what the assumptions on the strings $S$ and $T$ are and what exactly is the question asked. Firstly, one may be interested to simply in the answer to the decision problem, i.e. determining whether a continuous substring can be omitted from $S$ to obtain $T$. This is studied in the Section 5.1. Secondly, one might
want to recover the cut value, which is studied in Section 5.2. Thirdly, we may be interested in the cut index when one exists, which we examine in Section 5.3.

### 5.1 Decision problem

In this section, the decision variant of the Substring-cut problem is studied. As mentioned previously, in the streaming setting we are given two streams \( S \) and \( T \), and we are interested whether a contiguous substring of \( S \) can be removed from \( S \) to form the stream \( T \).

#### 5.1.1 Deterministic variant

In the deterministic setting, no efficient algorithm exists when \( |T| \) is sufficiently large as is demonstrated by the following lower bound.

**Lower bound**

We show a lower bound using a standard technique of constructing a fooling set. The set of representatives is \( \mathcal{R} = \{ r_x := \text{Concatenate}(x, a, b^k) \mid x \in \Sigma^{n-k-1} \} \), for fixed characters \( a \neq b \in \Sigma \), and the test set is \( \mathcal{T} = \{ t_x := \text{Concatenate}(x, a) \mid x \in \Sigma^{n-k-1} \} \).

We observe that \((\mathcal{R}, \mathcal{T})\) is indeed a fooling set, as for each \( x \neq y \) we have

\[
\text{Substring-cut}(r_x, t_x) = \text{Yes} \neq \text{No} = \text{Substring-cut}(r_y, t_x).
\]

The size of the fooling set is \( m^{n-k-1} \). The communication complexity of distinguishing the representatives is thus

\[
\log m^{n-k-1} = \Omega((n-k) \cdot \log m).
\]

As a direct consequence, the problem Substring-cut is not efficiently solvable when

\[
|T| = n - k = \omega(\text{polylog}(n)).
\]

**Upper bound**

Now we show that the lower bound we achieved is tight. We construct an algorithm solving the problem in \( O(|T| \cdot \log m) \) streaming complexity. We distinguish two cases:
5.1. Decision problem

1. $|T| = \Theta(|S|)$: We can simply store the whole contents of the stream $S$ and $T$ and use Lemma 5.1.

2. $|T| = o(|S|)$: As $k$, the size of the cut substring, grows so significantly that the length of the remaining string becomes small enough, some parts of the stream $S$ become irrelevant.

Recall that in Substring-cut we are interested in index $i \in [1, |T|]$, such that $T = \text{CONCATENATE}(S[1..i], S[i + k + 1..n])$. This means that only the first $|T|$ and last $|T|$ characters of $S$ can influence the result—everything else would have been in the cut substring. We can thus ignore the middle portion of the string and only store a prefix and a suffix of $S$ of appropriate length, as well as the whole stream $T$. By Lemma 5.1, the answer can be found in $O(|T| \log m)$ time and memory.

We have thus established that the complexity of Substring-cut has complexity $\Theta(|T| \cdot \log m)$. If $|T|$ happens to be polylogarithmic in $|S|$ and the value of $|T|$ (or a good upper bound) is known while processing the stream $S$, the problem becomes efficiently solvable.

5.1.2 Probabilistic variant

In reality, we would expect $|T|$ to be quite large, as the case of it being small yields a somewhat degenerate problem. We can thus interpret the complexity $\Theta(|T|)$ to signify that Substring-cut is not efficiently solvable in deterministic setting. That being said, we look deeper into the problem to see whether a good probabilistic algorithm exists.

In the following, we achieve a space and time efficient algorithm when the size of the cut is small. More precisely, an algorithm poly-logarithmic in the size of the input stream and the size of the alphabet, and polynomial in the length of the cut. We are able to achieve $1 + \varepsilon$ approximation ratio for arbitrary positive $\varepsilon$ with high probability.

Let us first examine the case where $|S| - |T| = k = 1$, that is, there is exactly one element removed from $S$ to form $T$, while all the other elements retain their relative ordering. We show that there is an efficient algorithm for deciding such property and later show how to adapt it to a larger cut.

In order to do this, we will study certain subsequences of $S$ and $T$. Given that the cut character is somewhere in the middle of the stream $S$, we cannot reliably construct a subsequence of $S$ and $T$ given indices of elements as we did when approximating Hamming distance in Section 4.5 and expect them to be equal or even similar. More promising option is to filter elements of $S$ and $T$ based on their value. Let us start with some definitions.
Basic definitions

We will now show how to filter certain values from the stream and then extract a necessary condition for the existence of Substring-cut.

**Definition 5.2** Let $C \subset \Sigma$ be a proper subset of values of the alphabet and $S$ a sequence of elements from the universe $\Sigma$. The subsequence of $S$ that contains all the elements from the set $C$, and none from its complement, is called a domain filtered subsequence. We denote it by

$$S[C] = (s_i | i \in [|S|], s_i \in C)$$

Furthermore, we define the index sequence of a set $C$ to be the sequence of indices to the sequence $S$ on which exactly the elements from $C$ reside. For convenience, we will be prepending 0 and appending $|S| + 1$ to the index sequence. In other words, the special characters ‘$\hat{}$’ and ‘$\dollar$’ will be considered as belonging to every set. Formally:

$$S_I[C] = \text{CONCATENATE}((0), (i | i \in [|S|], s_i \in C), (|S| + 1))$$

For clarity, we demonstrate the above definition on a simple example.

**Example 5.3** Let $S = "dbcaccacb"$ be a sequence, and $C = \{ 'a', 'b' \}$. Then, $S[C] = "baab"$,

and

$$S_I[C] = (0, 2, 4, 7, 9, 10).$$

The role of these constructs in connection to our problem is evident—given the cut value $x$ all domain filtered subsequences having $x \notin C$ are exactly the same for $S$ and $T$. In continuation to the previous example, let $T = "dbcaccab"$. Clearly, $T$ can be obtained from $S$ by removing the element with index 8, and $x = 'c'$. One can verify that indeed $T[C] = S[C]$.

This yields a necessary condition for the problem Substring-cut. There are however $2^{|\Sigma| - 1}$ subsets of $\Sigma$ that do not contain $x$ and we cannot check the condition for all of them. Furthermore, the condition is not even sufficient, as the following counterexample shows. Let $S = "aab"$ and $T = "ba"$. The condition holds for all subsets of $\{ 'a', 'b' \}$ not containing ‘$a’ (there is only one such non-empty set, and $S[\{ 'b' \}] = ( 'b' ) = T[\{ 'b' \})$, but $T$ is not a Substring-cut of $S$. We will address both of these problems: first we strengthen the condition in such a way that it is both necessary and sufficient, and then we reduce the amount of subsets of $\Sigma$ that need to be checked.

Instead of the domain filtered subsequence itself, we will look at the corresponding index sequence. We will need the following definition, whose significance will be clear later.
Definition 5.4 Let $A$ and $B$ be two integer sequences of the same length. We call $A$ and $B$ 1-similar, if and only if there is an index $\rho > 0$, such that

- $\forall i < \rho : a_i = b_i$ (the arrays have some non-empty common prefix) and
- $\forall i \geq \rho : a_i = b_i + 1$ (anything not part of this prefix differs by one).

Verifying that two sequences are 1-similar is not straightforward in the streaming model, as we cannot afford to store it explicitly, and hashing techniques we have employed previously are doomed to fail. However, we can reduce this problem to Hamming distance by looking at the difference array.

Definition 5.5 Let $A = (a_i)_{i=0}^{n}$ be a sequence of integers having $a_0 = 0$. Then the sequence $\text{diff}(A) = (a_{i+1} - a_i)_{i=0}^{n-1}$ is called the difference array of $A$, and $A$ is called the prefix sum array of $\text{diff}(A)$.

It is evident that when $A$ is an index sequence a $B$ is the sequence of distances of these elements, then

$$\text{diff}(A) = B.$$ 

There is thus a clear correspondence between those two arrays. As we show in the following lemma, we can test 1-similarity of some pair of arrays by computing the Hamming distance of their difference arrays.

Lemma 5.6 Let $A$ and $B$ be two prefix sum arrays of the same length $n$, such that $a_n = b_n + 1$. Then the following two conditions are equivalent:

- $A$ and $B$ are 1-similar, and
- the Hamming distance between $\text{diff}(A)$ and $\text{diff}(B)$, the respective difference arrays, is one.

Proof $\Rightarrow$: Trivial - one just computes the difference arrays of $A$ and $B$, there will be exactly one difference at index $\rho$.

$\Leftarrow$: As $A$ and $B$ are prefix sum arrays, $a_0 = b_0 = 0$, also $\sum_i \text{diff}(A)_i = a_n$ and $\sum_i \text{diff}(B)_i = b_n$. Let $\rho$ be the index of the only element in $\text{diff}(A)$, such that $\text{diff}(A)_\rho \neq \text{diff}(B)_\rho$. Clearly, $\text{diff}(A)_\rho = \text{diff}(B)_\rho + 1$, since this is the only difference and the sums of the arrays differ by one. Therefore, we obtain

$$\forall i < \rho : a_i - b_i = \sum_{j=0}^{i} \text{diff}(A)_j - \sum_{j=0}^{i} \text{diff}(B)_j = \sum_{j=0}^{i} (\text{diff}(A)_j - \text{diff}(B)_j) = 0$$

and

$$\forall i \geq \rho : a_i - b_i = \sum_{j=0}^{i} (\text{diff}(A)_j - \text{diff}(B)_j) + (\text{diff}(A)_\rho - \text{diff}(B)_\rho) = 1$$

$\square$
We can use the definition of 1-similarity to formulate the following necessary and sufficient condition for **Substring-cut**:

**Lemma 5.7** Let $S$ and $T$ be two streams. Then the following two conditions are equivalent:

- $T$ is a **Substring-cut** of $S$ with cut length 1,
- there is an element $x \in \Sigma$, such that for $C = \Sigma \setminus \{x\}$ it holds that $S[C] = T[C]$ and $(S_I[C], T_I[C])$ is a pair of 1-similar sequences.

**Proof** $\Rightarrow$: Set $x$ to be the cut value and $\rho$ be its index in $S$.

$\Leftarrow$: Since, $S[C] = T[C]$ verifies that no two elements of $C$ ever swap place. As the arrays are 1-similar, by Lemma 5.6 the 1-similarity implies that the Hamming distance of the difference arrays of $S_I[C]$ with $T_I[C]$ is 1. In the terms of the streams $S$ and $T$ this means that every pair of consecutive elements of $C$ but one has the same number of occurrences of $x$ between them. Remember that we treat both the start and end of the stream as an element from $C$ to simplify the argument. There exists just one pair for which the difference of their indices is different. Thanks to the fact that the index sequence has the length of the stream (+1) as the last element, and the index sequence is prefix sum array of the difference arrays we are looking at, the distance of the two consecutive elements from $C$ decreases by one in $T$. This exactly means that one $x$ was removed from $S$ to form $T$. $\square$

We have now established a necessary and sufficient condition for the problem **Substring-cut**: index sequences of domain $C$ in $S$ are 1-similar to those in $T$. We also have a way of efficiently verifying 1-similarity by computing the Hamming distance of the difference arrays of the said index sequences. However, as we do not know the cut value of $x$ in advance, we still have a prohibitively large class of sets $C$ to check. To tackle this problem, we first state the following definition and few intermediate results.
Distinguishing classes of sets

**Definition 5.8** Let \( \mathcal{C} \) be a class of subsets \( C \in \Sigma \). Let \( a \) and \( b \) be non-negative integers, such that \( a + b \leq |\Sigma| \). We call the class \( \mathcal{C} \) to be \((a,b)\)-distinguishing if for any disjoint sets \( A \) and \( B \) having \( |A| = a \) and \( |B| = b \), there exists a set \( C \in \mathcal{C} \) that contains all elements from \( A \), but none from \( B \). Formally,

\[
\forall A, B \subseteq \Sigma, |A| = a, |B| = b, A \cap B = \emptyset, \exists C \in \mathcal{C} : C \cap (A \cup B) = A
\]

The \((a,b)\)-distinguishing classes of subsets have a very useful property that we will soon use: by intersecting carefully selected sets from the class, we can isolate a small number of elements from all the remaining ones.

**Lemma 5.9** Let \( \mathcal{C} \) be an \((a,b)\)-distinguishing class of subsets of \( \Sigma \) for \( a, b \geq 1 \). Every set of at most \( a \) elements can be retrieved as the intersection of all subsets containing it:

\[
\forall A \subseteq \Sigma, |A| \leq a : \bigcap_{C \in \mathcal{C}} A \subseteq C
\]

**Proof** Assume for contradiction that the intersection contains some extra element \( x \notin A \). Take \( A' \supseteq A \) such that \( |A'| = a \) and \( x \notin A' \), and \( B' \subseteq \Sigma \) such that \( |B'| = b \), \( x \in B' \) and \( A' \cap B' = \emptyset \). As \( \mathcal{C} \) is \((a,b)\)-distinguishing, there exists a set \( C \in \mathcal{C} \) containing all elements from \( A' \) but none from \( B' \). Such set \( C \) contains \( A \) as subset, and belongs to the intersection. Since \( x \notin A' \), \( x \) is not in the intersection, which is a contradiction. \( \square \)

Next we show the existence of distinguishing class of subsets of reasonable size, along with a probabilistic way of obtaining one.

**Lemma 5.10** Let \( \Sigma \) be an alphabet of size \( m \), and \( a, b \) positive integer constants. Then there exists a \((a,b)\)-distinguishing class of subsets of size \( \mathcal{O}(\log m) \). Additionally, such class can be constructed efficiently with error probability at most \( \delta = m^{-1} \).

**Proof** Take \( \alpha \in \mathbb{N} \) to be fixed later, and select \( \alpha \) random subsets \( C_i \) of \( \Sigma \), by sampling every element to each subset independently uniformly at random with probability \( \gamma := \frac{a}{a+b} \). Let all \( C_i \) form the class \( \mathcal{C} \). We show that with probability strictly greater than zero \( \mathcal{C} \) is an \((a,b)\)-distinguishing class. This fact would imply that there exists at least one class of size \( \alpha \) having the desired property.

Fix sets \( A, B \subseteq U, |A| = a, |B| = b \) and fix one \( C_i \in \mathcal{C} \). Since every element from the universe belongs to each set with independent probability \( \gamma \), we have that

\[
Pr[C_i \cap (A \cup B) = A] = \gamma^a (1 - \gamma)^b = \frac{a^a b^b}{(a+b)^{a+b}}.
\]
5. Substring cut

Since all sets $C_i$ were chosen independently at random, the probability of all of them simultaneously not being able to distinguish $A$ and $B$ is exactly the product of the probability of such even happening for each of them. That is:

$$Pr \left[ \forall C_i \in C : C_i \cap (A \cup B) \neq A \right] = \left( 1 - \frac{a^a b^b}{(a+b)^{a+b}} \right)^a$$

This is the probability that fixed sets $A$ and $B$ are not distinguished, but we are interested in all possible sets $A$ and $B$. Therefore, we are interested in:

$$Pr[C_{\text{fail}}] := Pr \left[ C \text{ is not } a,b\text{-distinguishing} \right]$$

$$= Pr \left[ \bigcup_{A,B \subseteq \Sigma} \forall C_i \in C : C_i \cap (A \cup B) \neq A \right]$$

$$\leq \sum_{A,B \subseteq \Sigma} Pr \left[ \forall C_i \in C : C_i \cap (A \cup B) \neq A \right] \quad \text{(union bound)}$$

$$= \binom{m}{a} \cdot \binom{m-a}{b} \cdot \left[ 1 - \frac{a^a b^b}{(a+b)^{a+b}} \right]^a$$

$$\leq \frac{m^a}{a!} \cdot \frac{m^b}{b!} \cdot e^{-\frac{a^a b^b}{(a+b)^{a+b}}}$$

Now we fix

$$\alpha = \frac{(a+b)^{a+b}}{a^a b^b} \left[ (a+b) \log m - \log a! - \log b! - \log \delta \right]$$

in order to obtain $Pr[C_{\text{fail}}] < \delta$ for any $\delta \in (0,1)$. This means that the probability that $C$ is $(a,b)$-distinguishing is larger than zero, and hence there exists some class $C$ of such size that is $(a,b)$-distinguishing. $\square$

In streaming setting we cannot possibly afford to store the subsets, let alone to construct them explicitly. We can overcome this difficulty quite easily by describing each subset by a hash function from a family of $(a+b)$-wise independent hash functions as described in [CSRL01], and observe that the analysis still holds.

The following technical lemma demonstrates the use of this concept for our problem. Simply put, it says that we can instead of checking the index subsequence for 1-similarity given a large alphabet that contains everything but one character, we can check many smaller alphabets to achieve the same thing. This is where the concept of distinguishing classes comes handy.
Lemma 5.11 Let $C$ be a $(2,1)$-distinguishing class of subsets of $\Sigma$ that is closed with respect to complement, $S$ and $T$ two streams such that $|S| = |T| + 1$. Then the following conditions are equivalent:

- $\exists x \in \Sigma : S_I[\Sigma \setminus \{x\}]$ and $T_I[\Sigma \setminus \{x\}]$ are 1-similar, and $S[\Sigma \setminus \{x\}] = T[\Sigma \setminus \{x\}]$
- $\exists x \in \Sigma : \forall C \in C, x \notin C : S_I[C]$ and $T_I[C]$ are 1-similar, and $S[C] = T[C]$.

Proof $\Rightarrow$: Denote $C_x = \Sigma \setminus \{x\}$. When $x \notin C$, then $C \subseteq C_x$. $S_I[C]$ and $T_I[C]$ are thus subsequences of 1-similar subsequences $S_I[C_x]$ and $T_I[C_x]$, therefore they are 1-similar, too.

Furthermore, for arbitrary $C$ we see that $S[C]$ is a domain filtered subsequence of $S[\Sigma \setminus \{x\}]$, and therefore $T[C] = S[C]$.

$\Leftarrow$: By Lemma 5.9 we know that the intersection of all sets containing $x$ in $C$ is exactly $\{x\}$, and using de Morgan’s laws and the fact that $C$ is closed on complement we obtain:

$$\bigcup_{C \in C \setminus \{x\}} C = \Sigma \setminus \{x\}.$$

In other words, the union of all sets that do not contain value $x$ is the whole universe excluding element $x$ and therefore we can use their corresponding index sequences to recover the index sequences $S_I[C_x]$ and $T_I[C_x]$. We do this by simply merging the sequences (and eliminating duplicates). We only need to show that they are 1-similar and that the corresponding subsequences are equal.

To show that $S_I[\Sigma \setminus \{x\}] = T_I[\Sigma \setminus \{x\}]$ we first argue that they have the same length. Indeed, every element $y \in \Sigma, y \neq x$ has clearly the same number of occurrences in it. To show that no two distinct $y, z$ ever swap place, we use the fact that $C$ is $(2,1)$-distinguishing, and therefore there is a set $C_{yz} \in C$ which contains $y$ and $z$, but not $x$, and by assumption $S[C_{yz}] = T[C_{yz}]$.

What is left to show is that in $S_I[\Sigma \setminus \{x\}]$ there is no pair of consecutive elements $y, z$ (not necessarily distinct) for which the number of $x$s between them in $T$ is larger than that in $S$. Assume that this is not the case.

Take any $C \in C$ such that $y, z \in C$ and $x \notin C$ (we know that such set exists, since $C$ is $(2,1)$-distinguishing). At the corresponding positions, the difference arrays of $S_I[C]$ and $T_I[C]$ have the distance between these occurrences of $y$ and $z$. As there is one more $x$ in the string $T$, the distance in $T_I[C]$ is larger by one. Since $S_I[C]$ and $T_I[C]$ are 1-similar, the Hamming distance of their difference arrays is 1 (by Lemma 5.6), and therefore every other pair of corresponding elements has the same value. Summing these, we obtain $|S| + 1 = |T|$ which is a contradiction. \qed
Algorithm outline

Now that we have all the necessary tools, we combine them to obtain the algorithm.

Theorem 5.12 Problem Substring-cut for \( k = 1 \) is efficiently solvable in the probabilistic setting.

Proof We generate a random class \( C \) of subsets of \( \Sigma \) — by Lemma 5.10 this is a (2,1)-distinguishing class with failure probability \( \delta = \frac{1}{m} \) when its size is

\[
\alpha = \frac{(a + b)^{a+b}}{a^n b^b} \left[ (a + b) \log m - \log a! - \log b! - \log \delta \right] < 27 \log m.
\]

For every \( C \in C \) and its complement, we test the 1-similarity of the corresponding index sequences and equality of domain filtered substrings. We can do this efficiently by employing Lemma 5.6 and the probabilistic exact algorithms for Is-HAMMING-DISTANCE-1 and STREQ described in Section 4.4. When the answer for both \( C \) and \( \overline{C} \) is FALSE, we know that the difference between \( S \) and \( T \) is in at least two distinct characters, therefore the answer is FALSE. Otherwise \( T \) is Substring-cut of \( S \) using the results of Lemmas 5.7 and 5.11.

The total space complexity is \( \Theta(\log m \log n (\log \log n + \log \delta^{-1})) \).

The Algorithm 2 summarises the pseudo-code of the algorithm.

Corollary 5.13 Problem Substring-cut is efficiently solvable in probabilistic setting when \( k = \text{polylog}(n) \).

Proof Let us construct an alphabet consisting of all \( k \)-tuples of elements from \( \Sigma \). We know that the cut value is a \( k \)-tuple. What we don’t know is its alignment in the string \( S \) and therefore we need to try each of \( k \) alignments independently. Then we use Theorem 5.12 for \( n' = \frac{n}{k} \) and \( m' = m^k \). The total complexity of the algorithm is \( \Theta(k^2 \log n \log m (\log \log n + \log p^{-1})) \).
5.1. Decision problem

Algorithm 2 Algorithm pseudocode for $k = 1$

1: procedure SUBSTRING–CUT–INIT(state, sets)
2:     for $i \in \{\text{sets}\}$ do
3:         $C_i = \text{random subset of } \Sigma$
4:     for $b \in \{0, 1\}$ do
5:         state.prev$_{i,b} = 0$
6:         Is–HAMMING–1–INIT(state.hamming$_{i,b}$)
7:         STREQ–1–INIT(state.streq$_{i,b}$)
8:
9: procedure SUBSTRING–CUT–UPDATE(state, index, value)
10:     for $C_i \in \mathcal{C}$ do
11:         $b \leftarrow \text{value} \in C_i$
12:         dist $\leftarrow$ index $-$ state.prev$_{i,b}$
13:         Is–HAMMING–1–UPDATE(state.hamming$_{i,b}$, dist)
14:         STREQ–UPDATE(state.streq$_{i,b}$, $C_i$)
15:         state.prev$_{i,b} \leftarrow$ index
16:
17: procedure SUBSTRING–CUT–OUTPUT(stateS, stateT)
18:     for $C_i \in \mathcal{C}$ do
19:         for $b \in \{0, 1\}$ do
20:             result$_{i,b} \leftarrow$ Is–HAMMING–1–OUTPUT(stateS.hamming$_{i,b}$, stateT.hamming$_{i,b}$)
21:             result$_{i,b} \leftarrow$ result$_{i,b}$ $\land$ STREQ–OUTPUT(stateT.streq$_{i,b}$, stateS.streq$_{i,b}$)
22:         if result$_{i,0} =$False, result$_{i,1} =$False then
23:             return False
24:         return Probably True
5. Substring cut

5.2 Cut value

Suppose that we know that Substring-cut\((S, T)\) is True. How do we find the cut value, i.e. the part of the string that was removed? For instance consider strings \(S = abacd\) and \(T = abcd\). \(T\) can be obtained from \(S\) by cutting out the second occurrence of the character \(a\) and concatenating the rest. Is it possible to learn the value \(a\)?

\[
S = \begin{array}{cccc}
\text{a} & \text{b} & \text{a} & \text{c} & \text{d} \\
\end{array}
\]
\[
T = \begin{array}{cccc}
\text{a} & \text{b} & \text{c} & \text{d} \\
\end{array}
\]

Figure 5.1: Illustration of the cut value for strings \(S = abacd\) and \(T = abcd\). The cut value is the string \(a\).

Using the result from Section 5.1, it is not difficult to recover the cut value exactly without changing the asymptotic complexity. We first solve the case where \(k = 1\). Recall that in order to decide the Substring-cut problem, we take some subsets \(C \subset \Sigma\) (from the distinguishing class) and determine whether the domain filtered subsequences are the same and index subsequences are 1-similar. This is always performed for the set and its complement independently. Furthermore, on one of these subsets, the substring will be different, because of the missing character \(x \in \Sigma\), which is the cut value. If we had the subsets stored explicitly and the alphabet was small, we would be able to use Lemma 5.9 to recover the cut value. However, this is not possible in general case. We will show how to augment the class \(C\) to facilitate such an extraction.

In order to do that, assume that there is a binary encoding of the alphabet \(\Sigma\) that uses \(\lceil \log |\Sigma| \rceil\) bits. For every bit position, define two subsets containing all characters having that bit 0 or 1, respectively. More precisely

\[
\forall i \in \{0, 1, \ldots, \lceil \log |\Sigma| \rceil - 1\}, \forall b \in \{0, 1\} : C_{i,b} = \{c \in \Sigma \mid c \& 2^i = b \cdot 2^i\}
\]

where \& is the bitwise and. We add all these sets into the class \(C\) and perform the Substring-cut algorithm. As \(C_{i,0}\) is complement of \(C_{i,1}\), exactly one of them will have the corresponding domain filtered subsequences equal, and the other will have a mismatch. Denote \(b(i)\) to be the value for which \(C_{i,b(i)}\) is a mismatch. Then it is clear that the cut value encoded in binary is

\[
x = (b(\lceil \log |\Sigma| \rceil - 1), \ldots, b(1), b(0)).
\]

We have added only \(\Theta(\log \Sigma)\) many sets, and the space and time complexity remains the same. Similarly to Corollary 5.13, applying the same trick on a binary encoding of \(\Sigma^k\) yields the cut value even if \(k > 1\).
5.3 Cut index

Finally, suppose again that we know that $\text{SUBSTRING-CUT}(S, T)$ is TRUE. How do we find the cut index? In other words, what is the index $i$ such that

$$T = S[1..i] \check{\sim} S[i + k + 1..n]?$$

Since the value we are after is a quantity, we are interested in its approximation. Note that approximating the index $i$ is not precisely what we might want. Since $i$ is equal to the length of longest common prefix, one might naturally ask why did we choose to approximate the prefix and completely omitted the suffix. Observe that the common suffix can be very short, and in that case we do not learn a good approximation of its value just from approximating the common prefix. Similarly, if a prefix is short, we suddenly lose our ability to provide good estimate of its length if we just reverted it. Therefore, we are interested in both quantities:

$$L_p(S, T) = |\text{LONGEST-COMMON-PREFIX}(S, T)|,$$

$$L_s(S, T) = |\text{LONGEST-COMMON-SUFFIX}(S, T)|.$$

We now show how to estimate $L_p(S, T)$ and then how to adapt this technique to $L_s(S, T)$.

**Lemma 5.14** Let $\varepsilon > 0$ and $\delta > 0$. Calculating a $(1 + \varepsilon)$-approximation of $L_p(S, T)$ with probability at least $1 - \delta$ can be done in $O(\log n \cdot \frac{1}{\varepsilon} \cdot (\log \log n + \log \frac{1}{\delta}))$ space.

**Proof** For every $j \in [0, \log_{1+\varepsilon}(n)]$ consider the prefixes of length $(1 + \varepsilon)^j$. Denote these by $S_j$ and $T_j$. We compare each $S_j$ to the corresponding $T_j$ using the algorithm established by Lemma 4.5 with $\delta' = \frac{\delta}{\log_{1+\varepsilon} n}$. We then return the value $(1 + \varepsilon)^{j'}$ where $j'$ is the maximum $j$ for which

$$\text{STREQ}(S_j, T_j) = \text{TRUE}.$$

The probability of error in any of the runs of STREQ is at most $\delta$. It is easy to see that if all calls to STREQ return the correct value, the returned prefix is a $(1 + \varepsilon)$-approximation of the longest common prefix. $\square$

If $n$ and $k$ are known before reading the input, we can easily adapt the technique to calculate approximation of $L_s(S, T)$ — the longest common suffix. If $k$ is unknown, but a good upper bound is present, we can simply bruteforce all values of $k$. Furthermore, we will show a technique that works even if $n$ is unknown.
Lemma 5.15 Let $\varepsilon > 0$ and $\delta > 0$. Calculating a $(1 + \varepsilon)$-approximation of $L_S(S, T)$ with probability at least $1 - \delta$ can be done in $O(\log n \cdot \frac{1}{\varepsilon} \cdot (\log \log n + \log \frac{1}{\delta}))$.

Proof We will store information about some set of suffixes after reading a character. Ideally, we want to have suffixes of lengths $(1 + \varepsilon)^j$ for each $j$, but this is not possible to do precisely. The reason for this is that in a streaming setting, we can only extend the suffix by appending a character, but we cannot possibly shorten it by removing a character from its beginning, since we cannot afford to store it (this would in turn be equivalent to storing the whole input). Instead, we will maintain the relative differences between suffix lengths small enough.

Formally, let the lengths of suffixes after reading $i$ characters be

$$S_i = (S_{i,1}, S_{i,2}, \ldots, S_{i,k})$$

in increasing order. The following properties are to be maintained:

(a) $S_{i,1} = 0$ in order to correctly approximate the value of zero

(b) $S_{i,k} = i$ in order to correctly approximate the value of $i$

(c) $\forall j \in [2, k]: S_{i,j} \leq \max(S_{i,j-1} + 1, (1 + \varepsilon)S_{i,j-1})$ to approximate all values in between

Furthermore, we aim to maintain the size of the set $S_i$ to be as small as possible. The greedy strategy to calculate $S_{i+1}$ from $S_i$ when reading the $i + 1$-th input character is as follows:

(i) First create a new suffix of length 1 with the newly read character:

$$S_{i+1,1} = 1.$$

(ii) Then append the newest character to all current suffixes. That is

$$S_{i+1,j+1} = S_{i,j} + 1.$$

(iii) If the relative difference of three consecutive suffix sizes is too small, remove the middle one. That is, if there exists an index $j$ such that

$$S_{i+1,j+1} \leq (1 + \varepsilon)S_{i+1,j-1},$$

remove the suffix with size $S_{i+1,j}$.

One can easily see that the suffix of length 1 and the suffix containing the whole string will always be present, hence conditions (a) and (b) are satisfied. The condition (c) is also trivially satisfied due to inequality (iii). The
above construction will, in conjunction with Lemma 5.14 yield, a $(1 + \varepsilon)$ approximation of the longest common suffix.

What remains to show is that there are not too many suffixes being stored. Let us call a suffix small if its length is of size at most $\varepsilon^{-1}$ small, and large otherwise. There are $\frac{1}{\varepsilon}$ small suffixes, which are all necessarily stored. For large suffixes,

$$S_{n,j} + 1 \leq (1 + \varepsilon)S_{n,j},$$

hence the condition (c) can be simplified as

$$S_{n,j} \leq (1 + \varepsilon)S_{n,j-1}.$$

Thanks to the step (iii) of the algorithm, we have

$$S_{n,j+2} > (1 + \varepsilon)S_{n,j}.$$  

Since the maximum size equals $n$, there are at most $2\log_{1+\varepsilon} n$ large suffixes. Combined together, there are at most $O(\log n \cdot \frac{1}{\varepsilon})$ suffixes stored at any point in time, and the lemma follows. □

The Lemmas 5.14 and 5.15 allow us to efficiently approximate the cut location with only a small error probability, thus completing the picture of the problem SUBSTRING-CUT.
Conclusion

In this chapter, we are going to summarise the results achieved in this thesis and discuss possible extensions and suggestions for future research.

In Chapter 3 we have discussed multiple ways how multiple streams can be organised as an input. We established a taxonomy of input orders, with four deterministic ones and one stochastic.

We established a clear hierarchy among the deterministic classes, with the free order (Definition 3.1) being the least restrictive and sketching (Definition 3.4) being the weakest (but most practical at the same time). The synchronised (Definition 3.2) and serialised (Definition 3.3) being in the middle. These reductions have been proven in section 3.2.

Furthermore, the synchronised and serialised order are in general incommparable. This incomparability was proved by studying two problems, streq in Section 3.3.1 and common-majority Section 3.3.2. We showed that streq is easily solvable in the synchronised model, but the standard well-known result about its linear space complexity applies in the serialised model. On the other hand, we adapted the Boyer-Moore algorithm to solve the problem common-majority efficiently in the serialised model by leveraging the information obtained while reading the first of the two streams, while in the synchronised order the problem remains difficult.

Additionally, we attempted to capture the essence of input orders that may originate in sources that deliver data with variable speed in the interleaved model. We tried to reduce this to a more deterministic model, but we showed that this reduction is unfortunately not efficient in the average case by the means of Theorem 3.18.

In Chapter 4 we studied the problem of Hamming-distance. This is a problem where already very strong results were known beforehand. We have achieved an approximation algorithm that is on par with the current state-of-
6. Conclusion

the-art techniques, up to logarithmic factor, that is very simple to implement. It is an open question whether this novel approach might be improved even further, or perhaps applied to other metrics.

In particular, it might be possible to reduce the size of the hash in each bucket from \( O(\log \log n) \) to \( O(1) \). This will introduce a non-negligible probability of a collision and will complicate the analysis. However, the probability of the collision can be calculated exactly, and in that case the value returned can be de-biased. How exactly is the concentration of the result affected by this adjustment is an interesting question.

Lastly, we emphasised exact probabilistic algorithms for Hamming-distance at most 2, which proved helpful in the following chapter.

In Chapter 5, we introduced and formalised a special case of Levenshtein distance, called the Substring-cut. Recall that in this measure of similarity\(^1\) allows for a single operation — cutting one contiguous substring from string \( S \) and concatenating the rest to form string \( T \). We were able to design an efficient probabilistic algorithm for the appropriate decision problem. Furthermore, the algorithm may also be used to recover the exact string that was omitted, and also approximately find the location where this occurred. All these questions can be answered with arbitrary high probability.

An interesting open question is whether this approach may be optimised to get rid of the quadratic dependency on the length of the substring removed. Furthermore, it would be helpful to adapt it in such way that two or more cuts may occur in the string, or the string may be subjected to some small noise (that is, the remaining parts have small Hamming distance). This could in theory lead to a potent sketch for Levenshtein distance.

---

\(^1\)We purposely avoid the word metric, as it does not satisfy metric axioms.
Bibliography


Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

ON THE COMPLEXITY OF RECOGNIZING SIMILARITIES
BETWEEN STREAMS

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):

ŠVAGERKA

First name(s):

MICHAL

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date

10.4.2018

Signature(s)

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.