Abstract

Title of thesis	Promised streaming algorithms and finding pseudo-repetitions
	Cătălin-Ştefan Tiseanu, Master of Science, 2013
Directed by	Professor MohammadTaghi Hajiaghayi Department of Computer Science

As the size of data available for processing increases, new models of computation are needed. This motivates the study of data streams, which are sequences of information for which each element can be read only after the previous one. In this work we study two particular types of streaming variants: promised graph streaming algorithms and combinatorial queries on large words. We give an $\Omega(n)$ lower bound for working memory, where n is the number of vertices of the graph, for a variety of problems for which the graphs are promised to be forests. The crux of the proofs is based on reductions from the field of communication complexity. Finally, we give an upper bound for two problems related to finding pseudo-repetitions on words via anti-/morphisms, for which we also propose streaming versions.

Promised streaming algorithms and finding pseudo-repetitions

by

Cătălin-Ştefan Tiseanu

Thesis submitted to the Faculty of the Graduate School of the
University of Maryland, College Park, in partial fulfillment
of the requirements for the degree of
Master of Science
2013

Advisory Committee:

Professor MohammadTaghi Hajiaghayi, Chair
Professor V.S. Subrahmanian
Professor Hector Corrada-Bravo

© Copyright by
Cătălin-Ştefan Tiseanu
2013

Acknowledgments

I would like to warmly thank my advisor, Professor MohammadTaghi Hajiaghayi, for all the self-less support he has provided me towards finishing this thesis, both in terms of research advice and in terms of mentorship about life. The present work wouldn't have been possible without him. I would also like to thank Dr. Florin Manea, my former advisr at the University of Bucharest, for introducing me to the beautiful world of theoretical computer science research.

Contents

Acknowledgments **iii**

Contents **iv**

1 Introduction **1**

1.1 Summary of results 2

2 Streaming problems **3**

2.1 Motivation 3

2.2 Preliminaries 4

2.2.1 Data streams 4

2.2.2 Graph streams 5

2.2.3 Communication complexity 6

2.3 Related work 7

2.4 Lower bounds for the single pass model 10

2.5 Adapting the FRT and Racke algorithms to streaming model 17

3 Finding Pseudo-repetitions **19**

3.1 Introduction 19

3.2 Finding Pseudo-repetitions via $f - morphisms$ 21

3.2.1 Prerequisites 22

3.2.2 Solution of Problem 8 24

3.2.3 Solution of Problem 9 29

3.3 Streaming variants 35

4 Conclusion and future directions **36**

Bibliography **37**

Chapter 1

Introduction

As the size of data available for processing increases, new models of computation are needed. This motivates the study of data streams, which are sequences of information which need to be read in order, one element at a time. Streaming algorithms process these data streams, with the caveat that their working memory is significantly less than the size of the whole data stream. The field of streaming algorithms began in the eighties with results on computing basic properties of sequences, such as the K-th moment.

As the internet evolved, and more and more applications started to be modeled as graphs, the study of graph streaming algorithms commenced. Here the data stream consists of the edges of a graph, presented one at a time. Examples of real-world graphs which are too big to fit a traditional offline computation model are the friendship graph of Facebook or the call graph of Skype.

It turns out that many problems which are easy to compute in an offline setting (such as single-source shortest paths) become hard to do in a single pass streaming graph algorithm, where the memory available to such an algorithm is less than $\mathcal{O}(n)$, were n is the number of vertices of the graph. Therefore alternative models, such as the semi-streaming model, have been proposed in order to be able to study which graph problems can be solved efficiently in a streaming setting. The semi-streaming model is defined by an $\mathcal{O}(n \textit{ polylog } n)$ space restriction, which is significantly less than the memory required to store all edges for dense graphs.

In parallel with the development of graph streaming algorithms, the increase in the size of available DNA available for processing sparked the development of the field of bioinformatics. A large strand of DNA can be interpreted as a data stream, where individual elements are part of the alphabet $\{A, C, G, T\}$. One of the ways to model this computation is to consider DNA strands as words. Many important problems on DNA, such as protein folding, can be reduced to computing certain combinatorial properties, such as repetitions, on words. The notions of repetition and primitivity are defined as fundamental concepts on sequences used in a number of fields, among them being stringology and algebraic coding theory.

In this thesis, we attempt to study both types of data streaming problems. The second chapter deals with lower bounds on promised graph streaming problems. *Promised* in this context refers to the guarantee that the graph to be streamed will have a certain property (in our case the graphs are guaranteed to be forests). We show an $\Omega(n)$ lower bound for several problems, such as determining the maximum

size of a connected component. The end of the chapter presents an adaptation of the FRT and Racke tree decomposition methods for the streaming case. The second chapter deals with positive, upper bound results on studying an offline variant of repetitions on words. Specifically, we present two problems related to computing pseudo-repetitions efficiently on words. A word w is a pseudo-repetition if it equals a repeated catenation of one of its proper prefixes t and its image f(t) under some morphism or antimorphism. The chapter also presents the streaming variants of the two problems.

1.1 Summary of results

This thesis is organized as follows:

Chapter 2 presents our results related to *promised streaming graph problems*, which are based on the joint work with Hajiaghayi and Cormode. It is organized in five sections. The first one, *Motivation*, presents the motivation for considering streaming graph problems. The chapter then continues with a section on the preliminaries required for understanding the results and literature review. The third section gives a comprehensive review on the state of streaming and graph streaming problems. The fourth section, *Lower bounds for the single pass model* gives lower bounds for some problems on forests, showing that even for this restricted case it is hard to design algorithms which use less than linear memory in terms of the number of vertices. The final section gives an adaptation for the *FRT* and *Racke* algorithms to the streaming case.

Chapter 3 presents upper bounds on two problems related to computing repetitions on words. It extends on prior work we did in the area of combinatorial properties, NP-Completeness and combinatorial algorithms on words, and is based on a paper presented at *STACS 2013* [GMM$^+$13], by Gawrychowski, Manea, Mercas, Nowotka and Tiseanu. The chapter defines *anti-/morphism* and *f-repetitions* using these *anti-/morphisms*. The chapter ends by presenting two streaming variants for the two problems studied in an offline setting.

Finally, Chapter 4 presents further problems of interest related to the work presented in the thesis.

Chapter 2

Streaming problems

2.1 Motivation

The model of data streams is a natural extension of offline problems, motivated by the increase in the size of information relative to the capacity of processing. By a data stream we mean a sequence of information which needs to be read in order. The individual elements of the data stream can take multiple forms. One of the earliest problems in this area was the study on the space needed to approximate the K-th moment of a sequence of numbers using at most P passes over the input stream, as presented by Munro and Paterson [MP80]. In the paper the individual elements of the data stream were numbers.

Graph problems were considered by Henzinger and Raghavan [Rag99] in one of the earliest papers on the data-stream model. Their paper tackled a number of problems including pointer jumping problems in relationship to the degrees of various nodes in a directed layered graph. Unfortunately, many of the results showed that a large amount of space is required for these types of problems. Other early work studies the problem of counting the number of triangles in a graph and estimating common neighborhoods. As before, a significant portion of these results were negative. It appeared that more complex computation with only a small amount of space (linear) was not possible in this model.

So far, most graph algorithms have a need for access to data in an adaptive fashion. Given that the entire graph can not be stored with the amount of space provided, using a traditional graph algorithm may necessitate a large number of passes over the data. This has led to various specific stream models adapted to processing graphs, such as the Semi-Streaming, W-Stream, and Sort-Stream models. The semi-streaming model, which is the most popular in the contemporary graph streaming literature, is defined by an $\mathcal{O}(n \textit{ polylog } n)$ space restriction. Note that for dense graphs (for which $|E|$ approaches $\mathcal{O}(|V|^2)$) this represents considerably less space than that required to store the entire graph. This restriction was identified as an apparent "sweet-spot" for graph streaming in a survey article by Muthukrishnan [Mut05] and was first explored by Feigenbaum, Kannan and McGregor [FKM$^+$05]. The W-Stream and Stream-Sort models, which were described earlier, were introduced by Demetrescu and Aggarwal, respectively.

Recently, the increase in the size of the data available today for processing has

given rise to the field of Big Data in the industry, of which the Map / Reduce model is the most common, which was introduced by Dean and Ghemawat [DG08]. The Map / Reduce framework is built with computations over a large size of information in mind, for which the time of computation is not important. One of the main applications of this model is indexing a large corpus of information, which is an operation which doesn't need to be done in an online fashion, but can be rather scheduled every few days. One particularly interesting development is the appearance of frameworks which deal with large amounts of data which arrive in a streaming fashion (for example tweets). In this case the results need to be computed in an online manner as opposed to having the luxury of offline computation such as for Map / Reduce tasks. One representative streaming framework in this emerging field of real-time computation is the *Storm* framework [Mar11], an open source project maintained at Twitter.

To see the connection between real-world problems and the graph streaming instances presented in this work note that massive graphs arise naturally in many big data environments today. One of the more prominent examples are social networks, such as Facebook, Twitter or LinkedIn. The friendship graph of Facebook can be considered a streaming graph instance, since friendship relationship are generated dynamically and with a large volume. Also, follow actions on Twitter can be thought of as directed edges from one user to another. One of the major issues of classical graph algorithms, when applied to massive real-world graphs such as the web, is the need for random access to the edge set. Streaming graph algorithms on the other hand don't have this requirement.

2.2 Preliminaries

2.2.1 Data streams

A *data stream* [Rag99] is a sequence $A =< x_1, x_2, \ldots, x_n >$ of data elements x_i such that each element can be read only after the previous one (in increasing order of indices). Note that the data elements don't have to be numbers, an can be in fact, as we will see, edges of a graph.
The data stream can be of multiple types [Mut05]:

- *Time series model.* Here each x_i equals one data element of A, presented in increasing order of i. An example would be a *graph stream* where only edge additions are allowed. This is the most commonly used model.

- *Cash register model.* Here each $x_i = (j, v)$, $v > 0$ represents an increment to $A[j]$. Specifically, when x_i is encountered, $A[j]_i = A[j]_{i-1} + v$, with A_i representing the state of the stream after encountering x_i.

- *Turnstile model.* Here each $x_i = (j, u)$ represents and update to $A[j]$. Specifically, when x_i is encountered, $A[j]_i = A[j]_{i-1} + v$, with A_i representing the state of the stream after encountering x_i. In this case u can be less than 0. A *strict turnstile model* is defined as satisfying the requirement $A[j] \geq 0$, $\forall j \in [n]$. An example of a *strict turnstile model* would be a *graph stream* where both edge additions and deletions are allowed (obviously an edge can only be deleted if it has already been added to the graph).

2.2.2 Graph streams

In the following, $G = (V, E)$ represents a graph G with vertex set V and edge set E. For simplicity n represents the number of vertices and m the number of edges.

A *graph stream* is a data stream where each data element represents an edge e_i of a graph G. Specifically,

Definition 1 ([FKM+05])**.** A *graph stream* is a sequence of edges $e_{i_1}, e_{i_2}, \ldots, e_{i_m}$, where $e_{i_j} \in E$ and $i_1, i_2, \ldots, i_m$ is an arbitrary permutation of $[m] = \{1, 2, \ldots, m\}$.

The graph is revealed to the algorithm one edge at a time. Normally, it is assumed that the elements x_i of a data stream are distinct. In the case of graph streams we discern between three variants:

- *dynamic graphs*, where multiple edges are allowed between vertices, defining a multi-graph.
- *directed graphs*, where an element $x_i = (u, v)$ defines a directed edge
- *weighted graphs*, where an element $x_i = ((u, v), w)$ defines a weighted edge

Unless explicitly stated, we will be using the normal definition of undirected graphs in which multiple edges between vertices are not allowed.

The complexity of a stream graphing algorithm [FKM+05] is determined by the space it uses, the time it requires to process each edge and the number of passes it takes.

Definition 2 ([FKM+05])**.** A streaming graph algorithm computes over a graph stream using $S(n, m)$ bits of space. The algorithm may access the input stream in a sequential order (one-way) for $P(n, m)$ passes and use $T(n, m)$ time for each edge.

One particularly popular streaming graph algorithm model is the *semi-streaming graph algorithm* model, where the algorithm can use space $\mathcal{O}(n \cdot polylogn)$. This allows more than $\mathcal{O}(n)$ memory (which is not sufficient even for the simplest

problems), while at the same time not allowing the storage of the entire graph in memory.

Specifically,

Definition 3 ([FKM+05])**.** A semi-streaming graph algorithm is a *streaming graph algorithm* which computes over a graph stream using $S(n, m) = O(n \cdot polylogn)$ bits of space.

The order in which edges arrive is also subject to a choice of model. The most common scenario is the one in which the edges of the streamed graph arrive in a manner chosen by an adversary, such that it presents a worst-case scenario for the streaming algorithm. The incidence model, where edges incident on the same node arrive consecutively has also been studied. Finally, in some of the problems we study the graphs have a certain property (for example, they don't contain a cycle). We introduce this concept as *promised streaming problems*, and we define it as:

Definition 4. A promised streaming problem with respect to a property P is a streaming problem where the graph to be streamed $G = (V, E)$ obeys $P(G) = true$.

2.2.3 Communication complexity

One of the main tools used to show lower bounds for graph streaming problems is taken from the field of communication complexity [Rag99]. Let X, Y and Z be finite sets and let $f : X \times Y \rightarrow Z$ be a function.

Definition 5. The *(2-party) communication model* consist of two players, Alice and Bob. Alice is given an input $x \in X$ and Bob is given an input $y \in Y$ and they want to compute $f(x, y)$. Alice does not know y while Bob does not know x. Therefore, they need to communicate (exchange bits) according to an agreed upon protocol. The communication complexity of a function f is the minimum over all communication protocols of the maximum over all $x \in X$ and all $y \in Y$ of the number of bits that need to be exchanged to compute $f(x, y)$.

The protocol can be deterministic, Monte Carlo or LasVegas. When the communication is restricted to one player sending and the other receiving, then this is called a one-way communication complexity. In a one-way protocol, it is necessary to specify which player is the sender and which the receiver. In this case only the receiver needs to be able to compute f.

Two well-known problems from the area of communication complexity are the *INDEX* and *DISJOINTNESS* problems.

Problem 1 (INDEX). Let Alice have a string $x \in \{0, 1\}^2$ and Bob have a natural number $i \in [n]$. Bob wants to compute *INDEX(x, i)* $= x_i$ by receiving a single message from Alice (one-way complexity model).

This problem has a well known lower bound of $\Omega(n)$ which need to be exchanged between Alice and Bob in the one-way communication complexity model for Bob to compute *INDEX(x, i)*. Interested readers can refer to [FKM+05] and [FKM+08] for more uses of communication complexity results used to prove lower bounds for graphs streaming algorithms.

A related but stronger result is obtained from the following problem:

Problem 2 (**DISJOINTNESS).** Let Alice have a string $x \in \{0,1\}^2$ and Bob have a string $y \in \{0,1\}^2$. Bob wants to compute whether there is an index i such that $x_i = y_i = 1$, that is *DISJOINTNESS(x, y)* = " *is there an* $1 \leq i \leq n$ *such that* $x_i = y_i = 1$?" by exchanging as many messages as needed (multi-way complexity model).

This problem also has a well known lower bound of $\Omega(n)$ bits which need to be exchanged between Alice and Bob for Bob to compute *DISJOINTNESS(x, i)*, over multiple passes. The key difference to problem *INDEX* is that the lower bound of $\Omega(n)$ bits holds even when multiple exchanges between Alice and Bob are allowed. Therefore, this allows to prove lower bounds for graph streaming problems even in the multi-pass model.

2.3 Related work

The concept of data streams and graph streaming algorithms was introduced by Raghavan et al. in [Rag99]. Their work focuses on establishing a connection between communication complexity and lower bounds for streaming algorithms, as well as analyzing the relationship between the number of passes and space complexity. Ragahvan et. al. introduce and study the *MAX*, *MAXNEIGHBOR*, *MAXTOTAL* and *MAXPATH* problems on graph streams. They show a $\Omega(kn^2)$ lower bound and a $\mathcal{O}(kn^2 \lg n)$ space solution for each of those four problems. A reduction from communication complexity problems has been used before by Alon, Mathias and Szegedy [AMS96], where they showed lower bounds for computing the k-th frequency moment of a sequence in one-pass. They draw on earlier work by Munro et al. [MP80] showed an upper bound of $n^{1/P} \lg n$ and a lower bound of $n^{1/P}$ for computing the $k - th$ largest element out of a sequence of n elements using P passes, for large enough k.

Muthukrishan gives [Mut05] a relevant study of the the state of data streaming. He defines data stream models, such as the *time series model*, *cash register model* and *turnstile model*. The study also covers the main approaches used for streaming algorithms, such as *sampling* or using *random projections*. Most of the work on graph streaming algorithms up to that point focused on the extremes of using either enough dynamic memory sufficient to store the whole graph in memory, or using

poly $\lg n$ space. However, most of the streaming graph problems turned out to be hard to do in one-pass even in a linear space model. For example, Buchsbaum, Giancarlo and Westbrook showed [BGW03] that finding two vertices which have a large enough set of common neighbors is essentially equivalent to storing the whole graph (which would need $\mathcal{O}(n^2)$ in the worst case). McGregor [McG05] studied maximum cardinality matching in graphs for which he derived an approximation algorithm using $\tilde{O}(n)$ space but a constant number of passes.

Therefore, Muthukrishnan proposed a model of $\mathcal{O}(n\ poly \lg n)$ space, where n is the number of vertices. This model allows for more than constant memory per vertex, while at the same time not allowing the storage of the whole graph (which could take $\mathcal{O}(n^2)$ for dense graphs).

Feigenbaum et. al took on Muthukrishnan suggestion with their landmark paper [FKM+05], *On graph problems in a semi-streaming model*. The paper defined *semi-streaming* graph algorithms for the first time as streaming algorithms which use $\mathcal{O}(n\ poly \lg n)$ space. The paper gives a one-pass semi-streaming algorithm for computing a bipartition for a graph, a one-pass semi-streaming algorithm for findind a maximal matching, as well a semi-streaming algorithm for unweighted bipartite graph matching that computes a $\mathcal{O}(2/3-\epsilon)$ approximation in $\mathcal{O}(\frac{\lg \frac{1}{\epsilon}}{\epsilon})$ passes. Other results include a one-pass $\mathcal{O}(1/6)$ approximation algorithm for the maximum graph weighted matching and a $\mathcal{O}(\lg n/\lg\lg n)$ approximation for diameter and shortest paths in weighted graphs. Finally, using reductions from classic communication complexity problems, the authors give a $\Omega(\lg^{1-\epsilon} n)$ lower bound for the the above two problems in unweighted graphs.

Their paper led to continuation papers studying streaming graph problems in the semi-streaming model. Zelke gives [Zel06] two semi-streaming algorithms for the deciding the k-connectivity of a graph. One of them takes $\mathcal{O}(k^2 n)$ time per edge while needing only one pass, while the other takes $\mathcal{O}(k+\alpha(n))$ time per edge but needs $k+1$ passes.

The authors of the original paper on the semi-streaming model continued their exploration of positive results and lower bounds [FKM+08]. Feigenbaum et al. give a semi-streaming one-pass algorithm for computing the $2t+1$ spanner of a graph, when $t=\Omega(\lg n/\lg\lg n)$ resolving on the open question they proposed in the previous paper. The paper also gives a lower bound of either $\mathcal{O}(k)$ passes or $\Omega(n^{1+1/k})$ for computing the first k layers of a breadth first search (BFS). Given the importance of BFS in may graph algorithms their result is a testament of the hardness of traditional graph algorithms in the semi-streaming model. Finally, the paper gives a lower bound for t-approximations of graph distance. Specifically, they prove an $\Omega(n^{1+1/t})$ lower bound for the space required to compute a t-approximation of the graph distance between two nodes, in the one-pass model. From that they derive lower bounds for computing other graph properties, such as the length of

the shortest cycle.

Given the inherent challenge of computing traditional graph properties in one-pass, even in the semi-streaming case, researchers have explored the trade-off between space and the number of passes. Demetrescu, Finocchi, and Ribichini showed [DFR09] an algorithm for computing single-source shortest paths in $\mathcal{O}((n \lg^{3/2} n)/\sqrt{s})$, using s bits when the weights of the graph are small integers. Furthermore, they show a $\mathcal{O}((n \lg n)/s)$ pass algorithm for undirected connectivity, also using $\mathcal{O}(s)$ bits of space.

As the field of streaming graph algorithm expanded, so did the methods used. Sampling is one of the main techniques used to prove (randomized) algorithms. Ahn, Guha and McGregor [AGM12] introduce the concept of *graph sketching*, that is computing linear sketches of a graph and then solving the problem offline, using traditional graph algorithms. They used random projections of a graph (which is essentially $\mathcal{O}(n^2)$ space) to a d-dimensional space, with $d << n^2$, while still preserving many of the characteristics of the graph. The paper shows that $d = O(n \, poly \lg n)$ is enough for computing connectivity, k-connectivity, bipartiteness and any constant approximation for the weight of the minimum spanning tree. If d is taken to be $\mathcal{O}(n^{1+\gamma})$, and if $\mathcal{O}(1/\epsilon)$ rounds of sketches which can be taken dependent of the previous one are allowed, then graph sparsifiers, approximate maximum weighted matching and the exact minimum spanning tree can be computed. Guruswami and Onak [GO12] proved tighter lower bounds for three p-pass streaming algorithms for the following problems on n-vertex graphs: testing if an undirected graph has a perfect matching, testing if two specific vertices are at distance at most $2(p+1)$ in an undirected graph, and testing if there is a directed path from s to t for two specific vertices s and t in a directed graph. Their paper gives an $\mathcal{O}(n^(1 + \Omega(1/p))/p^O(1)$ lower bound, which is a significant improvement given that prior to their result the best lower bound was $\mathcal{O}(n^2)$ in one pass, but no $n^{1+\Omega(1)}$ lower bound was known for any $p \geq 2$.

In parallel to graph streaming problems, other classic offline problems, such as pattern matching, were studied in a streaming model. For example, Clifford, Efremenko, Porat B. and Porat E. [CEPP08] showed that many offline pattern matching algorithms could be adapted to a streaming model using a $\mathcal{O}(\lg m)$ factor overhead in the time complexity per symbol in the stream, where m is the length of the pattern. Improvements for some pattern matching problems were made but all of them needed space linear in m. It is actually possible to show that any exact pattern matching algorithm will need $\mathcal{O}(m)$ space. In an important development of the field, Porat B. and Porat E. [PP09] showed how to do exact pattern matching using $\mathcal{O}(\lg m)$ space and $\mathcal{O}(\lg m)$ time per new stream symbol. Using a method based on fingerprints they manage to find all matches with high probability. Their approach was refined by Breslauer and Galil [BG11] to only use

constant time per new stream symbol while using the same space. Jalsenius, Porat and Sach [JPS11] made a further important development to the streaming pattern matching community by presenting a sublinear near constant time algorithm for parameterized matching in a stream, using near optimal space. Specifically they give a randomized algorithm that uses $\mathcal{O}(1)$ worst-case time per character and uses $\mathcal{O}(|\Sigma| \lg m)$ words of space. The probability that the output is correct at all text alignments is at least $1 - 1/n^c$ for any constant c. Magniez, Mathieu and Nayak [MMN10] studied the problem of computing $DYCK(s)$, in which they check if a parenthesis expression is correctly matched, where s is the number of types of parentheses. They show a one-pass randomized streaming algorithm for $DYCK(2)$ using space $\mathcal{O}(\sqrt{n \lg n})$ with $poly \lg n$ time per letter and one-sided error. An interesting find is that if two passes are allowed, one from the beginning and one from the end of the expression, then the space requirement for $DYCK(2)$ becomes $\mathcal{O}(\lg^2 n)$.

Below is a summary related to known problems in the single-pass, semi-streaming model.

Problem	Lower Bound Approximation factor	Upper Bound Approximation Factor
Weighted Matching		$\frac{1}{6}$, [FKM+05]
Unweighted Bipartite Matching		$\frac{1}{2}$, [FKM+05]
Unweighted Diameter	$\mathcal{O}((\lg n)^{1-\epsilon})$	
Graph Spanner		$(1+\epsilon)\lg n$, [FKM+05]
Planarity Testing	Exact	Exact [FKM+05]
Articulation Points	Exact	Exact [FKM+05]
Connectivity	Exact	Exact [AGM12]
k-Edge-Connectivity	Exact	Exact [AGM12]
Bipartiteness	Exact	Exact [AGM12]
Minimum Spanning Tree		$1+\epsilon$ [AGM12]

2.4 Lower bounds for the single pass model

A natural direction to explore graph streaming algorithms would be to look for problems which can be solved with sublinear working memory. As can be seen in *Related Work*, most problems on regular graphs require at least linear space. Therefore, one would be curious about the lower bounds for graph streaming problems on particular classes of graphs. To this effect, let us study problems on graphs which are forests (they donÕt have any cycles).

From now on we are going to assume that $|V| = n$. The key question we try to

explore in this section is: can we do better than linear space for some problems on forest graphs ?

One natural problem to consider would be *MAX-CONN-COMP*, which asks for the size of the largest connected component (for a graph which is a forest) :

Problem 3 (**MAX-CONN-COMP**)**.** Let $G = (V, E)$ be a forest graph. Find the smallest $k \in N$ such that there isn't any connected component in G with size larger than k.

Alternatively, the decision version of the above problem is:

Problem 4 (**MAX-CONN-COMP(k)**)**.** Let $G = (V, E)$ be a forest graph. Is there a connected component of size at least $k \geq 3$ in graph G ?

Another important problem on a tree is finding the diameter:

Problem 5 (**TREE-DIAM(k)**)**.** Let $G = (V, E)$ be a tree. Is the diameter of G at least $k \geq 3$?

Finally, we are interested in determining if a forest graph is in fact a tree:

Problem 6 (**IS-TREE**)**.** Let $G = (V, E)$ be a graph. Is G a tree ?

(note that the only non-trivial case for this problem happens when $|E| = |V| - 1$. If that is not the case we can answer the problem directly with *NO*.) A subproblem useful for the study of Problem 6 is

Problem 7 (**INDEX-SAME**)**.** Let Alice have a string $x \in 0, 1^2$ and Bob have a natural number $i \in [n-1]$. Bob wants to compute *INDEX-SAME(x, i)* $=$ $'is\ x_i = x_{i+1}?'$ by receiving a single message from Alice (one-way complexity model).

In the following we will give linear space lower bounds for all of the above problems.

Theorem 1. *Any single pass streaming graph algorithm solving* **MAX-CONN-COMP(k)** *on forest graphs needs* $\mathcal{O}(n)$ *working memory.*

Proof of Theorem 1. We are going to prove that even such a basic problem needs at least $\mathcal{O}(n)$ working memory, matching the upper bound. For this we will make use of the classic problem *INDEX* from communication complexity.

We will give a reduction of Problem 1 to Problem 4. Let $IN = (x, i)$ be an instance of *INDEX*. Let us construct the instance $G = (V = V_l \cup V_r \cup V_d, E = E_{ALICE} \cup E_{BOB})$ of *MAX-CONN-COMP(G, k)*.

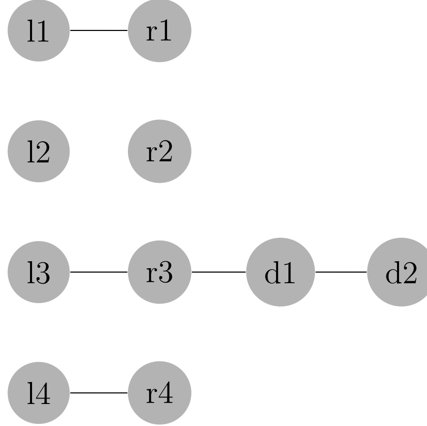
We define the vertices as:

- $V_l = \{l_1, l_2, \ldots, l_n\}$
- $V_r = \{r_1, r_2, \ldots, r_n\}$
- $V_d = \{d_1, d_2, \ldots, d_{k-2}\}$

We define the edges as:

- $E_{ALICE} = \{(l_j, r_j) | x_j = 1,\ j \in [1, n]\}$
- $E_{BOB} = \{(l_j, r_j)\} \cup \{(d_j, d_{j+1}) | j \in [1, k-3]\}$

For clarity, we present the resulting graph G for an instance $IN = (1011, 3)$ and $k = 4$.



Clearly, G has no cycles. The crux of the construction is the following: *MAX-CONN-COMP(G, k)* $= 1 \Leftrightarrow$ *INDEX(x, i)* $= 1$. To see that, notice that every connected component in G has size 1, 2, $k - 1$ or k. The only possible component of size k is the one containing $i + n$. If *INDEX(x, i)* $= 1$ then that component would have size k. Else it would have size $k - 1$. It is easy to see now that any single pass streaming algorithm A for Problem 4 using f(n) working memory, where f is an arbitrary function, implies a protocol for Problem1 where $f(n)$ bits are instance of *INDEX*. We will assume localization of x and i in the following (note that only Bob knows i). Let Alice run A on the edges encoded by x. She obtains graph $G' = (V, E_{ALICE})$. She then sends the resulting memory

image of $f(n)$ bits to Bob. Bob continues the execution of algorithm A by adding E_{BOB} and obtains the final graph $G = (V_l \cup V_r \cup V_d, E_{ALICE} \cup E_{BOB})$. If the answer $A(G, k)$ gives at the end is 1, Bob can conclude that *INDEX(x, i)* $= 1$. Else, *INDEX(x, i)* $= 0$. The total communication between Alice and Bob is $\mathcal{O}(f(n))$ bits. Since we established earlier that we need to exchange a linear number of bits between Alice and Bob to solve Problem 1 we can establish that $f(n)$ is indeed $\mathcal{O}(n)$, and that a sublinear space solution for Problem 4 does not exist. ■

It is actually possible to prove the following slightly stronger theorem:

Theorem 2. *Any multi pass streaming graph algorithm solving* **MAX-CONN-COMP(k)** *on forest graphs needs $\mathcal{O}(n)$ working memory.*

Proof of Theorem 2. One could prove a slightly stronger version of the above by a reduction from the classic communication complexity problem *DISJOINTESS*:

We will now give a reduction of Problem 2 to Problem 4.

Let $DISJ = (x, y)$ be an instance of *DISJOINTNESS*. Let us construct the instance $G = (V = V_l \cup V_m \cup V_r \cup V_d, E = E_{ALICE} \cup E_{BOB} \cup E_D)$ of *MAX-CONN-COMP(G, k)*.

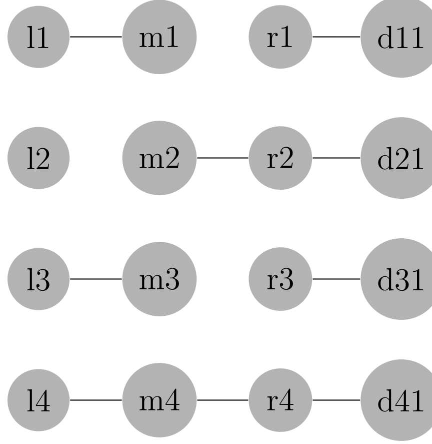
We define the vertices as:

- $V_l = \{l_1, l_2, \ldots, l_n\}$
- $V_m = \{m_1, m_2, \ldots, m_n\}$
- $V_r = \{r_1, r_2, \ldots, r_n\}$
- $V_d = \{d_{ij} | j \in [1, k-4], i \in [1, n]\}$

We define the edges as:

- $E_{ALICE} = \{(l_j, m_j) | x_j = 1,\ j \in [1, n]\}$
- $E_{BOB} = \{(m_j, r_j) | y_j = 1,\ j \in [1, n]\}$
- $E_D = \{(r_j, d_{j1}) | j \in [1, n]\} \cup \{(d_{ji}, d_{j\{i+1\}}) | j \in [1, k-4], i \in [1, n]\}$

For clarity, we present the resulting graph G for an instance $DISJ = (1011, 0101)$ and $k = 4$.



The crux of the construction is the following: *MAX-CONN-COMP(G, k)* $= k \Leftrightarrow$ $DISJOINTESS(x, i) = 1$. To see that, notice that every connected component in G has size 1, 2, $k-1$ or k. The only possible component of size k is one where both x_i and y_i are 1, for some i.
It is easy to see now that any multi pass streaming algorithm A for Problem 4 using f(n) working memory, where f is an arbitrary function, implies a protocol for Problem2 where $f(n)$ bits are exchanged. We will assume localization of x and y in the following (note that only Bob knows y). Let Alice run A on the edges encoded by x. She obtains graph $G' = (V_l \cup V_m, EDGE_{ALICE})$. She then sends the resulting memory image of $f(n)$ bits to Bob. Bob continues the execution of algorithm A by adding the edges in $EDGE_{BOB}$ and $EDGE_D$ and obtains the final graph $G = (V_l \cup V_m \cup V_r \cup V_d, EDGE_{ALICE} \cup EDGE_{BOB} \cup EDGE_D)$. If the answer $A(G, k)$ gives at the end is 1, Bob can conclude that *DISJOINTNESS(x, i)* $= 1$. Else, *DISJOINTNESS(x, i)* $= 0$. The total communication between Alice and Bob is $\mathcal{O}(f(n))$ bits. Since we established earlier that we need to exchange a linear number of bits between Alice and Bob to solve Problem 2 we can establish that $f(n)$ is indeed $\mathcal{O}(n)$, and that a sublinear space solution, even with multiple passes, for Problem 4 does not exist. ∎

Next we concern ourselves with the lower bound for Problem 5:

Theorem 3. *Any single pass streaming graph algorithm solving* **TREE-DIAM(k)** *on a tree needs* $\mathcal{O}(n)$ *working memory.*

Before we can lay out the proof for Theorem 3, we need the aid of the following theorem:

Theorem 4 (**LSUBSEQ**)**.** *Given a bit sequence* x *of size* n*, there is a single pass streaming algorithm* $LSUBSEQ$ *which uses* $\mathcal{O}(\lg n)$ *bits of memory and outputs*

(l, j) such that $x_k = 0, \forall k \in [j, j+l-1]$ and there is no $l' > l$ such that the previous property holds. In case of multiple solutions the algorithm picks the minimal j.

Proof of Theorem 4. It is quite easy to see how this problem can be solved by a single pass algorithm which maintains 3 counters (one for the start of the best sequence seen so far, one for length of the current subsequence of $0's$ and one for the best length seen so far). This requires $\mathcal{O}(\lg n)$ bits of memory.
■

Now we are ready to prove Theorem 3:

Proof of Theorem 3. In the following we will show the same lower bound as for Problem 4, by using a similar approach.

We propose the following protocol for solving $INDEX(x, i)$: Alice starts with a graph containing only the dummy vertex 0, that is $G_0 = (V_0, E_0)$ with $V_0 = 0$ and $E_0 = \emptyset$. As she streams bit x_i, she builds graph $G_i = (G_i, E_i)$ with 2 choices depending on x_i:

- if $x_i = 0$ then $V_i = V_{i-1} \cup i$ and $E_i = E_{i-1} \cup (i-1, i)$.
- if $x_i = 1$ then $V_i = V_{i-1} \cup i$ and $E_i = E_{i-1} \cup (0, i)$.

Intuitively, if $x_i = 0$ she connects a new vertex to a previous chain. Else she connects a new vertex to the dummy vertex 0. The resulting graph is a spider graph having vertex 0 as the root. Assume WLOG that G_n is not a chain (or else the problem would be trivial). Also assume WLOG that $x_1 = 1$ (or else we could stream the augmented string $1x$ of length $n + 1$). In parallel, Alice runs algorithm $LSUBSEQ(x)$. After streaming the whole input x, Alice sends the resulting memory image $Mem(G_n)$ of the graph constructed so far together with the output of $LSUBSEQ(x)$ (which takes only $\mathcal{O}(\lg n)$ bits) to Bob. Given that we want to prove a lower bound of $\mathcal{O}(n)$ bits we can afford the extra $\mathcal{O}(\lg n)$ bits. Bob receives the memory image of the graph streamed $Mem(Gn)$ so far together with the pair (l, j) given by $LSUBSEQ(x)$. Bob uses *TREE - DIAM(k)* to find the diameter of G_n. Note than in order for Bob to not alter his memory image, he creates an auxiliary copy of the memory image he has so far and runs *TREE - DIAM(k)* on it. Let the diameter that he finds be d (by iterating k over all possible values). Given that G_n is not a chain, the diameter of G_n would be composed by adding the lengths of the 2 longest branches of G_n. We already know l, which is the length of the longest branch of G_n. Therefore, we are able to get $l' = d - l$ which is the length of the second-longest branch of G_n. There are 3 cases to consider:

- if $i \in [j, j+l-1]$ then clearly $INDEX(x, i) = 0$ (vertex i is strictly part of a branch in G_n).

- if $i = j - 1$ then clearly $INDEX(x, i) = 1$ (vertex i is the start of a new branch in G_n).
- we add a chain of length $l' - 1$ to vertex i, in the same manner as for proving the lower bound for Problem 4 .

For the third case, Bob runs *TREE-DIAM(k)* again to find the diameter for this new graph. The key point to note is that the diameter increases iff $INDEX(x, i) = 0$. Note that if $INDEX(x, i) = 1$ then adding a chain of length $l'-1$ to vertex i will not increase the diameter. Let $c_{l'-1}$ be the end point of the chain added to vertex i, and let e_i be the last vertex on the branch containing i (before the chain was added). Clearly, $d_{G_n}(0, c_{l'-1}) = l'$. Also, $d_{G_n}(ei, cl' - 1) = l'-1+d_{G_n}(i, e_i) \leq l'-1+l-1 \leq d$. On the other hand, if $INDEX(x, i) = 0$ then adding a chain of length $l' - 1$ to i would create a path of length $d_{G_n}(0, c_{l'-1}) \geq l' - 1 + 2 \geq l' + 1$ because $d_{G_n}(0, i) \geq 2$. Given that there is another branch of length l in G_n (by assumption of the three cases we considered) we would get a diameter of at least $l' - 1 + 2 + l \geq l' + l + 1 > d$. Therefore, we proved that we can use *TREE-DIAM(k)* to solve $INDEX(x, i)$. Therefore, Problem 5 has a lower bound of $\mathcal{O}(n)$ bits of working memory. ■

Also, please note that the graph constructed by our protocol is connected at all times. Therefore, our result holds in a more restrictive streaming model in which we assume the streamed graph is connected at all times.

We will now show a lower bound for Problem 6. First, we need the following theorem:

Theorem 5. *Any single pass protocol for* **INDEX-SAME** *needs $\mathcal{O}(n)$ memory.*

Proof of Theorem 5. We will now prove that this problem has a lower bound of $\mathcal{O}(n)$ bits which need to be transmitted by Alice in an one-way complexity model. We will use an reduction from $INDEX$ to this problem. For this we give the following protocol for solving $INDEX$:
Let $IN = (x, j)$ be an instance of $INDEX$. Alice builds the following string x' from $IN : x' = x'_1...x'_n$ where $x'_i = 01$ if $x_i = 0$ and $x'_i = 11$ otherwise, for $i \in [n]$. Note that $|x'| = 2n$. Alice sends the representation of x' to Bob. Bob can now compute $a = INDEX - SAME(x', 2 \cdot (j - 1) + 1)$. It is easy to see now that $a = 0 \Leftrightarrow INDEX(x, j) = 0$ and vice-versa. We have now proven a lower bound of $\mathcal{O}(n)$ bits needed by any one-way protocol solving *INDEX-SAME.*
■

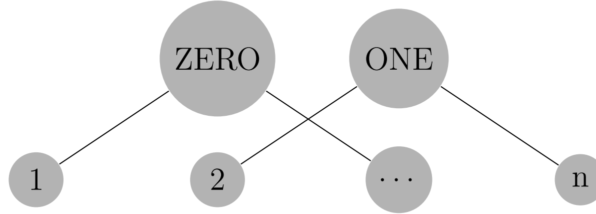
We can proceed to prove the following theorem:

Theorem 6. *Any single pass streaming algorithm solving* **IS-TREE** *needs $\mathcal{O}(n)$ working memory.*

Proof of Theorem 6. We will outline a protocol showing how *IS-TREE* can be used to solve *INDEX-SAME*.
Let $INS = (x, j)$ be an instance of *INDEX-SAME*. Alice starts with the empty graph $G = (V, E)$ with $V = \emptyset$ and $E = \emptyset$. She now streams $x_i, i \in [n]$ as follows:

- if $x_i = 0$ then she adds edge $(ZERO, i)$ to E.
- if $x_i = 1$ then she adds edge (ONE, i) to E.

Below is an example for a string x beginning with 01 and ending with 1.



After streaming the edges defined by x in the fashion described above, Alice has a graph with $n + 2$ vertices and n edges. She sends the representation of the graph to Bob. Bob now adds the edge $(j, j + 1)$ using his knowledge of j. Please note that the graph now has $n + 1$ edges, therefore it will be a tree if and only if it is a connected graph. Given the way G was constructed it is connected if and only if vertices j and $j + 1$ are either not both connected to $ZERO$ or both not connected to ONE. Notice that this is equivalent to $x_j \neq x_{j+1}$ which further means that *INDEX-SAME(x, j)* $= 1 \Leftrightarrow$ *IS-TREE(G)* $=$ NO. We showed the required reduction. ■

2.5 Adapting the FRT and Racke algorithms to streaming model

In this section we will discuss an adaption of the FRT and Racke algorithms to a single pass semi-streaming graph model. Specifically, we sketch a proof for the following two results:

Theorem 7. *For $\epsilon > 0$, and a weighted undirected graph on n vertices, whose maximum edge weight, w_{max}, and minimum edge weight w_{min}, satisfy $\lg \frac{w_{max}}{w_{min}} =$ polylog n, there is a semi-streaming algorithm that constructs a $(1+\epsilon) \lg n^2$ spanner of the graph which is a tree. The algorithm uses $\mathcal{O}(1/\epsilon \cdot n$ polylog $n)$ bits of space and worst case processing time for each edge is $\mathcal{O}(1/\epsilon \cdot n$ polylog $n)$.*

and

Theorem 8. *For $\epsilon > 0$, and a weighted undirected graph on n vertices, there is a semi-streaming algorithm that constructs a spanning tree which approximates the congestion in the original graph by $(1+\epsilon)\ \tilde{O}\ (\lg n)$. The algorithm uses $\mathcal{O}(1/\epsilon \cdot n$ polylog $n)$ bits of space.*

In the following we assume that the original graph is G.

Proof of Theorem 7. Let $d_A(x, y)$ denote the distance between vertices x and y in graph A. Note that using *Theorem* 5 in [FKM+05] gives us a $\mathcal{O}(\lg(1+\epsilon)\lg n)$ spanner of the graph. Let this graph be G'. This means that for any pair of vertices x and y, $d_G(x, y) \leq d_{G'}(x, y) \leq \lg(1+\epsilon)\lg n \cdot d_G(x, y)$. Applying the classic FRT algorithm (from [FRT03]) to graph G' we get a tree T which in expectation is a $\lg n$ spanner of G'. This means that for any pair of vertices x and y, $E[d_{G'}(x, y)] \leq E[d_T(x, y)] \leq \lg n \cdot E[d_{G'}(x, y)]$. Therefore, it must mean that for any pair of vertices x and y, $E[d_G(x, y)] \leq E[d_T(x, y)] \leq (1+\epsilon)(\lg n)^2 \cdot E[d_G(x, y)]$, making T in expectation a $(1+\epsilon)(\lg n)^2$ spanner for the original graph G, proving the theorem. ■

A similar technique yields a proof for the adaption of the Racke algorithm (described in [Räc08]) to a single pass semi-streaming model.

Proof of Theorem 8. We will start by applying the algorithm in [AG09] to get a graph G' which approximates every cut in the original graph G by a factor of $1+\epsilon$. After that we use the Racke algorithm on graph G' to get a spanning tree T (which is also a spanning tree for G) which approximates the congestion pf the edges by Õ $\lg n$ for G', giving the combined congestion result for graph G as stated in the theorem.

Too see that in more detail, consider what happens to the load of the edges in T, per Racke algorithm. Localize T as the spanning tree (note that it is a spanning tree for both G and G'). Denote $load_X(e)$ as the load of edge e in graph X. Take an edge $e \in G'$. If $e \notin T$, then $load_{G'}(e) = 0$, and $load_G(e)$ would also be 0. If $e \in G'$, note that the way the load is calculated guarantees that $load_{G'}(e)$ is within a factor of $1+\epsilon$ from $load_G(e)$, because G' approximates the weight of every cut within a factor of $1+\epsilon$. Combining both cases, we get that the load of the edges in G' ■

Chapter 3

Finding Pseudo-repetitions

3.1 Introduction

The notions of repetition and primitivity are fundamental concepts on sequences used in a number of fields, among them being stringology and algebraic coding theory. A word is a repetition (or power) if it equals a repeated catenation of one of its prefixes. We consider a more general concept here, namely *pseudo-repetitions in words.* A word w is a pseudo-repetition if it equals a repeated catenation of one of its proper prefixes t and its image $f(t)$ under some morphism or antimorphism (for short "anti-/morphism") f, thus $w \in t\{t, f(t)\}^+$.

Pseudo-repetitions, introduced in a restricted form by Czeizler et al. [CKS10], lacked so far a well-developed algorithmic part. Given that the motivation for studying these objects originates from bioinformatics, where efficient algorithms are crucial, producing such tools seems not only natural but even necessary. This work is aimed to fill this gap. We investigate the following two basic algorithmic problems: decide whether a word w is a pseudo-repetition for an anti-/morphism f and find all k-powers of pseudo-repetitions occurring as factors in a word w, for an f as above; in these problems w is given as input, while f, although of unrestricted form, is fixed, thus not a part of the input. We establish algorithms and complexity bounds for these problems for various types of anti-/morphisms thereby improving significantly the results from [CKX12]. Apart from the application of standard stringology tools, like suffix arrays, we extend the toolbox by nontrivial applications of results from combinatorics on words.

Background and Motivation. The motivation of introducing pseudo-repetition and pseudo-primitivity in [CKS10] originated from the field of computational biology, namely the facts that the Watson-Crick complement can be formalized as an antimorphic involution and both a single-stranded DNA and its complement (or its image through such an involution) basically encode the same information. Until now, pseudo-repetitions were considered only in the cases of involutions, following the original motivation, and the results obtained were mostly of combinatoric nature (e.g., generalizations of the Fine and Wilf theorem - see, e.g., [CKS10, MMN12]).

A natural extension of these concepts is to consider anti-/morphisms in general, which is done in this paper. Considering that the notion of repetition is central in combinatorics of words and the plethora of applications that this concept has

(see [Lot97]), the study of pseudo-repetitions seems even more attractive, at least from a theoretical point of view. While the biological motivation seems appropriate only for the case of antimorphic involutions, the general problem of identifying pseudo-repetitions can be seen as a formalization of scenarios where we are interested in identifying sequences having a hidden repetitive structure. Indeed, as each pseudo-repetition is an iterated catenation of a factor and its encoding through some simple function, such words have an intrinsic, yet not obvious, repetitive structure.

Some Basic Concepts. For more detailed definitions we refer to [Lot97].

Let V be a finite alphabet; V^* denotes the set of all words over V and V^k the set of all words of length k. The *length* of a word $w \in V^*$ is denoted by $|w|$. The *empty word* is denoted by λ. We denote by $\mathrm{alph}(w)$ the alphabet of all letters that occur in w. A word $u \in V^*$ is a *factor* of $v \in V^*$ if $v = xuy$, for some $x, y \in V^*$; we say that u is a *prefix* of v, if $x = \lambda$, and a *suffix* of v, if $y = \lambda$. We denote by $w[i]$ the symbol at position i in w, and by $w[i..j]$ the factor of w starting at position i and ending at position j, consisting of the catenation of the symbols $w[i], \ldots, w[j]$, where $1 \leq i \leq j \leq n$; we define $w[i..j] = \lambda$ if $i > j$. Also, we write $w = u^{-1}v$ when $v = uw$. The powers of a word w are defined recursively by $w^0 = \lambda$ and $w^n = ww^{n-1}$ for $n \geq 1$. If w cannot be expressed as a nontrivial power of another word, then w is *primitive*. A *period* of a word w over V is a positive integer p such that $w[i] = w[j]$ for all i and j with $i \equiv j \pmod p$. By $per(w)$ we denote the smallest period of w.

The following classical result is extensively used in our investigation:

Theorem 9 (Fine and Wilf [FW65])**.** *Let u and v be in V^*. If two words $\alpha \in u\{u,v\}^+$ and $\beta \in v\{u,v\}^+$ have a common prefix of length greater than or equal to $|u| + |v| - \gcd(|u|,|v|)$, then u and v are powers of a common word of length $\gcd(|u|,|v|)$.*

A function $f : V^* \to V^*$ is a morphism if $f(xy) = f(x)f(y)$ for all $x, y \in V^*$; f is an antimorphism if $f(xy) = f(y)f(x)$ for all $x, y \in V^*$. In order to define a morphism or an antimorphism it is enough to give the definitions of $f(a)$ for all $a \in V$. An anti-/morphism $f : V^* \to V^*$ is an involution if $f^2(a) = a$ for all $a \in V$. We say that f is *uniform* if there exists a number k with $f(a) \in V^k$ for all $a \in V$; if $k = 1$ then f is called *literal*. If $f(a) = \lambda$ for some $a \in V$, then f is called *erasing*, otherwise *non-erasing*.

We say that a word w is an *f-repetition*, or, alternatively, an f-power, if w is in $t\{t, f(t)\}^+$, for some prefix t of w. If w is not an f-power, then w is *f-primitive*. As an example, the word $ACGTAC$ is primitive from the classical point of view (i.e., $\mathbf{1}$-primitive, where $\mathbf{1}$ is the identical anti-/morphism) as well as f-primitive for the morphic involution f defined by $f(A) = T$, $f(C) = G$, $f(T) = A$, and $f(G) = C$. However, for the antimorphic involution $f(A) = T$ and $f(C) = G$ (which is, in

fact, a formalization of the Watson-Crick complement, from biology), we get that $ACGTAC = AC \cdot f(AC) \cdot AC$, thus, it is an f-repetition.

Finally, the computational model we use to design and analyse our algorithms is the standard unit-cost RAM (Random Access Machine) with logarithmic word size, which is generally used in the analysis of algorithms.

3.2 Finding Pseudo-repetitions via $f - morphisms$

In the upcoming algorithmic problems, we assume that the words we process are sequences of integers (called letters, for simplicity). In general, if the input word has length n then we assume its letters are in $\{1, \ldots, n\}$, so each letter fits in a single memory-word. This is a common assumption in algorithmics on words (see, e.g., the discussion in [KSB06]).

In the first problem, which seems to us the most interesting one in the general context of pseudo-repetitions, we approach the fundamental problem of deciding whether a word is an f-repetition, for a fixed anti-/morphism f.

Problem 8. Let $f : V^* \to V^*$ be an anti-/morphism. Given $w \in V^*$, decide whether this word is an f-repetition.

We solve this problem in the general case of erasing anti-/morphisms in $\mathcal{O}(n \lg n)$ time. However, in the particular case of uniform anti-/morphisms we obtain an optimal solution running in linear time. The latter covers the biologically motivated case of involutions from [CKS10]. This optimal result seems interesting to us, as it shows that pseudo-repetitions can be detected as fast as repetitions, if the way we encode the repeated factor (i.e., the function f) is simple enough, yet not the identity. We also extend our results to a more general form of Problem 8, testing whether $w \in \{t, f(t)\}^+$ for a proper factor t of w. Except for the most general case (of erasing anti-/morphisms), where we solve this problem in $\mathcal{O}(n^{1+\frac{1}{\lg \lg n}} \lg n)$ time, we preserve the same time complexity as we obtained for Problem 8.

Two other natural algorithmic problems are related to the fundamental combinatorial property of freeness of words, in the context of pseudo-repetitions. More precisely, we are interested in identifying the factors of a word which are pseudo-repetitions.

Problem 9. Let $f : V^* \to V^*$ be an anti-/morphism and $w \in V^*$ a given word.
(1) Enumerate all (i, j, ℓ), $1 \leq i, j, \ell \leq |w|$, such that there exists t with $w[i..j] \in \{t, f(t)\}^\ell$.
(2) Given k, enumerate all (i, j), $1 \leq i, j \leq |w|$, so there exists t with $w[i..j] \in \{t, f(t)\}^k$.

Question (2) was originally considered in [CKX12], while the first one is its natural generalisation. Our approach to question (1) is based on constructing data structures which enable us to retrieve in constant time the answer to queries $rep(i, j, \ell)$: "Is there $t \in V^*$ such that $w[i..j] \in \{t, f(t)\}^\ell$?", for $1 \le i \le j \le n$ and $1 \le \ell \le n$, where $n = |w|$. For unrestricted f, one can produce such data structures in $\mathcal{O}(n^{3.5})$ time. When f is non-erasing, the time taken to construct them is $\mathcal{O}(n^3)$, while when f is a literal anti-/morphism we can do it in time $\mathcal{O}(n^2)$. Once we have these structures, we can identify in $\Theta(n^3)$ time, in the general case, all the triples (i, j, ℓ) such that $w[i..j] \in \{t, f(t)\}^\ell$, answering (1) in $\mathcal{O}(n^{3.5})$ time. Similarly, for f non-erasing (respectively, literal) we answer question (1) in $\Theta(n^3)$ (respectively, $\Theta(n^2 \lg n)$) time and show that there are input words on which every algorithm solving this question has a running time asymptotically equal to ours (including the preprocessing time). Unfortunately, the time bound obtained for most general case is not tight.

Exactly the same data structures are used in the simplest case of literal anti-/morphisms to answer the more particular question (2). We obtain an algorithm that outputs in $\mathcal{O}(n^2)$ time, for given w and k, all pairs (i, j) such that $w[i..j] \in \{t, f(t)\}^k$; this time bound is shown to be tight. Taking advantage of the fact that k is given as input (so fixed throughout the algorithm) we can refine our solution for question (1) in order to get a $\Theta(n^2)$-time solution of question (2) for f non-erasing, again a tight bound, and a $\mathcal{O}(n^2 k)$-time solution for the general case. Our results improve significantly the algorithmic results reported in [CKX12].

3.2.1 Prerequisites

We begin this section by presenting several number theoretic properties. For the lemmas for which a proof is not shown here, the interested reader can consult the paper on which this chapter is based [GMM+13]. Lemma 1 is used in the time complexity analysis of our algorithms, while Lemma 2 and its corollary are utilised in the solutions of Problem 9. Given two natural numbers k and n, we write $k \mid n$ if k divides n. We denote by $d(n)$ the number of divisors of n and by $\sigma(n)$ their sum.

Lemma 1. Let n be a natural number. The following statements hold:
(1) $\sum_{1 \le \ell \le n} d(\ell) \in \Theta(n \lg n)$, $\sum_{1 \le \ell \le n} d(\ell) \ge n \lg n$, $d(n) \in o(n^\epsilon)$ for all $\epsilon > 0$ (see [Apo76]); (2) $\sigma(n) \in \mathcal{O}(n \lg \lg n)$ (see [Apo76]); (3) $\sum_{1 \le \ell \le n} (n - \ell + 1) d(\ell) \in \Theta(n^2 \lg n)$. ■

Lemma 2. Let n be a natural number. We can compute in $\mathcal{O}(n^3)$ time a three dimensional array $T[k][m][\ell]$, with $1 \le k, m, \ell \le n$, where $T[k][m][\ell] = 1$ if and only if there exists a divisor s of ℓ and the numbers k_1 and k_2 such that $k_1 + k_2 = k$ and $k_1 s + k_2 s m = \ell$. ■

Corollary 1. Let R be a fixed natural constant, and n and k be given natural numbers. We can compute in $\mathcal{O}(n \lg n)$ time a matrix $T_k[m][\ell]$ with $1 \leq m \leq R$ and $1 \leq \ell \leq n$, where $T_k[m][\ell] = 1$ if and only if there exists a divisor s of ℓ and the numbers k_1 and k_2 such that $k_1 + k_2 = k$ and $k_1 s + k_2 sm = \ell$. The constant hidden by the $\mathcal{O}$-notation depends on R. ■

We briefly present the data structures we use. For a word u with $|u| = n$ over $V \subseteq \{1, \ldots, n\}$ we can build in linear time a suffix array structure as well as data structures allowing us to return in constant time the answer to queries "How long is the longest common prefix of $u[i..n]$ and $u[j..n]$?", denoted $LCPref(u[i..n], u[j..n])$. For more details, see [Gus97, KSB06], and the references therein. Also, for u and an anti-/morphism f, we compute an array len with n elements defined as $len[i] = |f(u[1..i])|$, for $1 \leq i \leq n$. For f non-erasing we also compute an array inv, having $|f(u)|$ elements, such that $inv[i] = j$ if $len[j] = i$ and $inv[i] = -1$ otherwise. These computations are done in $\mathcal{O}(n)$ time. Note the following result:

Lemma 3. Let $w \in V^*$ be a word of length n. We compute the values $per[i]$, the period of $w[1..i]$, for all $i \in \{1, \ldots, n\}$ in linear time $\mathcal{O}(n)$. Also, we compute the values $per[i][j]$, the period of $w[i..j]$, for all $i, j \in \{1, \ldots, n\}$ in quadratic time $\mathcal{O}(n^2)$. ■

Next we show an important property of pseudo-repetitions, for non-erasing morphisms.

Lemma 4. Let f be a non-erasing anti-/morphism, and x, y, z be words over V such that $f(x) = f(z) = y$. If $\{x, y\}^* x \{x, y\}^* \cap \{z, y\}^* z \{z, y\}^* \neq \emptyset$ then $x = z$.

Proof. We sketch the proof only for the case when f is a morphism; a similar argument works for antimorphisms. If $\{x, y\}^* x \{x, y\}^* \cap \{z, y\}^* z \{z, y\}^* \neq \emptyset$ then we may assume without losing generality there exists w such that $w = xw'$, $w' \in \{x, y\}^*$, and $w \in \{z, y\}^* z \{z, y\}^*$.

If z is a prefix of w, as $f(x) = f(z)$ and f is non-erasing, we get easily that $x = z$.

Assume now that $w = yzw''$ with $w'' \in \{z, y\}^*$. It is not hard to see that from $|x| \leq |y|$ and $w = xw'$ we obtain that $|x|$ is a period of y, and, thus, $y = x^\ell u$ where $\ell > 0$ and u is a prefix of x. If y and x are powers of the same word v, then $x = v^{k_1}$, $y = v^{k_2}$ and $u = v^{k_3}$, so z is also a power of v. Since $f(x) = f(z)$ we conclude again that $x = z$. Further, assume that x and y are not powers of the same word. Hence, u is a proper prefix of x, i.e., $x = uv$ for $u \neq \lambda \neq v$. Consequently, w' has a prefix of the form $x^p y$, with $p \geq 0$, and it follows that after the first $|y|$ symbols of w both the factor vu and the factor z occur (as vu occurs after the first $|y| - |x|$ symbols of w'). Since $|vu| = |x|$ we get easily that $z = vu$. So, $|z| = |x|$,

$y = f(z) = f(vu) = f(v)f(u)$ and $y = f(x) = f(u)f(v)$. It follows that y is a power of a primitive word t. By an involved case analysis, it follows that x is a power of the same primitive word as y, a contradiction.

In the case when $w = yyzw''$ for some $w'' \in \{z, y\}^*$, we can apply Theorem 9 to the prefix of length $2|y|$ of w (which is a prefix of a word from $x\{x, y\}^*$, as well) and obtain that x and y are powers of the same word. Once again, we obtain that $z = x$. ■

The next lemmas provide insights to the combinatorial properties of f-repetitions, for f a general morphism, and are utilised in showing the soundness and efficiency of our algorithms. When using them, we take x to be the shorter and y the longer of the words t and $f(t)$.

Lemma 5. Let x and y be words over V such that x and y are not powers of the same word. If $w \in \{x, y\}^*$ then there exists a unique decomposition of w in factors from $\{x, y\}$. ■

Lemma 6. Let $x, y \in V^+$ and $w \in \{x, y\}^* \setminus \{x\}^*$ be words such that $|x| \leq |y|$ and x and y are not powers of the same word. Let $M = \max\{p \mid x^p \text{ is a prefix of } w\}$ and $N = \max\{p \mid x^p \text{ is a prefix of } y\}$. Then $M \geq N$. Moreover, if $M = N$ then $w \in y\{x, y\}^*$ holds, while if $M > N$ then either it is the case that $w \in x^{M-N}y\{x, y\}^* \setminus x^{M-N-1}yxV^*$, or we have $w \in x^{M-N-1}y\{x, y\}^+ \setminus x^{M-N}yV^*$ and $N > 0$. ■

3.2.2 Solution of Problem 8

§*A general solution.* We first assume that f is a morphism and let $n = |w|$. We construct in linear time the word $x = wf(w)$ of length $m = n + |f(w)|$ (which is in $\mathcal{O}(n)$); note that the length of x (hence, the constant hidden by the $\mathcal{O}$-notation) depends on the fixed morphism f. Moreover, we build in $\mathcal{O}(n)$ time data structures enabling us to answer *LCPref* queries for x.

Using these data structures, Algorithm 1 tests whether w is an f-repetition or not.

Following the comments inserted in its description, it is not hard to see that Algorithm 1 is sound. In the following, we compute its complexity. The step where we test whether w is a repetition takes $\mathcal{O}(n)$ time, as it can be done by locating the occurrences of w in ww. Further, note that the computation in each of the steps $6-9$ of the algorithm can be executed in constant time using the data structures we already constructed. Indeed, for some $s \leq n$, we can compute the largest ℓ such that $w[s..\ell]$ is a power of x in constant time as follows. In the worst case, $\ell = s - 1$, or, in other words, $w[s..\ell] = \lambda$, when x does not occur at position s. Otherwise, ℓ is the largest number less than or equal to $LCPref(w[s..n], w[s + |x|..n])$ such that

Algorithm 1 $Test(w, f)$: decides whether w is an f-repetition

1: Test whether there exists a word x such that $w = x^k$, with $k \geq 2$. If yes, then we halt and decide that w is an f-repetition. Otherwise, go to step 2.
{If the result of the test is positive we decide that w is an f-repetition, as repetitions can be seen as trivial f-repetitions. The algorithm continues for w primitive.}
2: **for** $t = w[1..i]$, such that $i < n$, $len[i] \geq 1$, t and $f(t)$ are not powers of some $x \in V^*$ **do**
3: Set $x = t$ and $y = f(t)$ if $i \leq len[i]$ or $x = f(t)$ and $y = t$, otherwise;
4: Set $s = i + 1$, $\ell'_i = |y|$, $\ell''_i = |x|$; {We have $\ell'_i = \max\{len[i], i\}$ and $\ell''_i = \min\{i, len[i]\}$}
5: If $s = n + 1$ halt and decide that w is an f-repetition;
6: Compute $M = \max\{p \mid x^p$ is a prefix of $w[s..n]\}$, $N = \max\{p \mid x^p$ is a prefix of $y\}$;
7: If $w[s..n] = x^M$, set $s = n + 1$, go to step 5;{If $w[s..n] \in \{x\}^+$ then $w \in t\{t, f(t)\}^*$}
8: If $x^{M-N}y$ occurs at position s, set $s = (M - N)\ell''_i + \ell'_i$, go to step 5;
9: If $M > N$ and $x^{M-N-1}yx$ occurs at position s, set $s = (M - N - 1)\ell''_i + \ell'_i$, go to step 5;
{By Lemma 6, $w[s..n]$ should have either $x^{M-N}y$ or $x^{M-N-1}yx$ as prefix. By Lemma 6, if $x^{M-N-1}yx$ occurs at position s, we shall check whether $w[s..n] \in x^{M-N-1}y\{x, y\}^+$.}
{If none of the above holds, we get that $w[s..n] \notin \{t, f(t)\}^+$, so $w \notin t\{t, f(t)\}^+$.}
10: **end for**
11: Halt and decide that w is not an f-repetition.

$\ell - s + 1$ is divisible by $|x|$. This strategy is used in step 6 to compute M and N. The verification from step 7 takes clearly constant time: we just check whether $n - s + 1 = M|x|$. Moreover, step 8 and 9 can also be implemented in constant time using *LCPref* queries; indeed, we know that x^{M-N} occurs at position s, and then we just have to check whether y occurs at position $s + (M - N)|x|$ by a *LCPref* query, for step 8, or, respectively, whether yx occurs at position $s + (M - N - 1)|x|$ by two *LCPref* queries, for step 9. Further, the iterative process in steps $3 - 9$ is executed for each prefix $w[1..i]$ of w, and during each iteration the algorithm makes at most $\mathcal{O}(\lfloor \frac{n}{\ell'_i} \rfloor)$ steps, as s can take at most $\lfloor \frac{n}{\ell'_i} \rfloor$ different values (in the cycle defined by the "go to" instruction from step 8). Since $\ell'_i \geq i$, the overall time complexity of the algorithm is upper bounded by $\mathcal{O}(\sum_{1 \leq i \leq n} \lfloor \frac{n}{i} \rfloor)$. Thus, the time complexity of Algorithm 1 is $\mathcal{O}(n \lg n)$. As a side note, in the case when f is

erasing, $w \in t\{t, f(t)\}^+$ for some t with $f(t) = \lambda$ if and only if $w \in \{t\}^+$, that is, w is a repetition. Hence, we run the iterative process starting in step 2 only for prefixes $w[1..i]$ with $len[i] \geq 1$.

The case when f is an antimorphism is similar. We take $x = wf(w)$, build the same data structures, and proceed just as in the former case. As the single difference, now we have $w[s+1..s+len[i]] = f(w[1..i])$ iff $LCPref(s+1, m-len[i]+1) = len[i]$, where $m = |x|$.

When f is uniform we can easily obtain a more efficient algorithm. In this case, $|t|$ divides n, so we only need to run the iterative instruction for the prefixes $w[1..i]$ of w with $i \mid n$. Hence, the total running time of the algorithm is, in this case, upper bounded by $\mathcal{O}(\sum_{i|n} \frac{n}{i}) \in \mathcal{O}(n \lg\lg n)$, by Lemma 1.

§*A linear time solution for the case when f is uniform.* We can obtain an even faster solution for Problem 8 for the case when f is uniform by using some more intricate precomputed data structures in order to speed-up Algorithm 1. To this end, we analyse again the computation performed by Algorithm 1 on an input word w.

The main phase of the algorithm is the following. For a prefix $t = w[1..i]$ of w with $i \mid n$ we run a cycle (steps $5-9$) that extends iteratively a prefix $w[1..s-1]$, where $s \geq i+1$, of the word w such that the newly obtained prefix is in $t\{t, f(t)\}^*$. However, at each iteration the prefix is extended with a word of the form $t^k f(t)$, with $k \geq 0$. As k can be actually equal to 0, we can only say that the number of iterations of the cycle is upper bounded by $\frac{n}{|f(t)|} \leq \frac{n}{|t|}$. Here we plug in our speed-up strategy: we try to extend the prefix in each of the iterations of the cycle from steps $5-9$ with a word that belongs to $\{t, f(t)\}^\alpha$ for some fixed number α that depends on n, but not on t. In this way, we upper bound the number of iterations of the cycle by $\frac{n}{\alpha|t|}$, and the overall complexity of the algorithm by $\mathcal{O}(\frac{n \lg\lg n}{\alpha})$. Finally, in order to obtain an algorithm solving Problem 8 in linear time, we choose $\alpha = \lceil \lg\lg n \rceil$.

Let $R = |f(a)|$, for $a \in \mathrm{alph}(w)$; as f is uniform, the definition of R does not depend on the choice of a from V, and we also have $R = \frac{|f(u)|}{|u|}$, $\forall u \in V^*$. Let $r_t = \max\{\ell \mid t^\ell \text{ prefix of } f(t)\}$. Clearly, $r_t \leq R$ and we can assume without losing generality that $\alpha > R$. Indeed, this holds for $n > 2^{2^R}$, which is the case when we want to optimise Algorithm 1; for smaller n the algorithm runs in constant time $\mathcal{O}(1)$, as $n \lg\lg n \leq R2^{2^R}$ and R is constant (f being fixed).

It only remains to show how we can implement efficiently the above mentioned extension of the prefix. First, note that there exists a constant C such that $\frac{(\lg n)^4 (\lg\lg n)^2}{n} \leq C$ for all n. Therefore, running the original form of Algorithm 1 for the prefixes t of w with $|t| > \frac{n}{(\lg n)^2 \lg\lg n}$ and $|t| \mid n$ (that is, at most $(\lg n)^2 \lg\lg n$

prefixes) takes $\mathcal{O}(n)$ time. Therefore, from now on, we only consider prefixes t such that $|t| \mid n$, $|t| < \frac{n}{(\lg n)^2 \lg\lg n}$, and, assuming that the input word is not a repetition, t and $f(t)$ are not powers of the same word.

Now consider a prefix t, as above. There are $2^\alpha \in \mathcal{O}(\lg n)$ words in $\{t, f(t)\}^\alpha$. Every such word can be encoded by a bit-string of length α: each occurrence of t is encoded by 0 and an occurrence of $f(t)$ by 1. Denote these bit-strings $v_1, \dots, v_{2^\alpha}$, and let $\overline{v}_i$ be the word encoded by v_i, for all $1 \leq i \leq 2^\alpha$. Further, for a bit-string v_ℓ we can determine by binary search two values b_ℓ and e_ℓ such that all the suffixes contained in the suffix array of w between the positions b_ℓ and e_ℓ have the word $\overline{v}_\ell t^{r_t}$ as a prefix. From Theorem 9, applied for two strings $\overline{v}_i t^{r_t}$ and $\overline{v}_j t^{r_t}$ with $i \neq j$, and the facts that t and $f(t)$ are not powers of the same word and r_t is the maximal power of t occurring as a prefix of $f(t)$, we get that the intervals $[b_i, e_i]$ and $[b_j, e_j]$ are disjoint. The time needed to compute these values for each ℓ is $\mathcal{O}(\lg n \lg\lg n)$, as a comparison between the word $\overline{v}_\ell t^{r_t}$ and a suffix of w can be done in $\mathcal{O}(\lg\lg n)$ by looking at the encoding v_ℓ and the string t^{r_t} (a prefix of $f(t)$) and, consequently, comparing only the factors of length $|t|$ and $|f(t)|$ of $\overline{v}_\ell t^{r_t}$ with those of the words from the suffix array. Thus, the time needed to compute b_ℓ and e_ℓ for all ℓ is $\mathcal{O}((\lg n)^2 \lg\lg n)$. Next, we construct a set E_t containing the values e_ℓ ordered increasingly, while keeping track for each e_ℓ of the corresponding values of ℓ and b_ℓ. Note that E_t contains $\mathcal{O}(\lg n)$ integers from $\{1, \dots, n\}$.

We need one more result before concluding this preprocessing phase. We want to store a static set $S \subseteq \{1, \dots, n\}$ so that finding the successor in S of a given $x \in \{1, \dots, n\}$ takes constant time. Thus, we use a static d-ary tree of depth 2, where $d = \lceil n^{0.5} \rceil$, so that the whole tree has n leaves corresponding to different values of x. We mark all leaves corresponding to the elements of S, and remove all nodes with no marked leaf in the corresponding subtree. At each remaining inner node v we store a table of length d where for each child of v (both remaining and already removed) we store the successor of the rightmost leaf in its corresponding subtree. The total size of the structure is $\mathcal{O}(|S| n^{0.5})$ and we can construct it in the same time if we start with an empty S and add its elements one-by-one, creating new inner nodes when necessary. Furthermore, using the tables we can find the successor of any x in $\mathcal{O}(1)$ time by traversing the path from the root of the tree towards x as long as the nodes exist and taking the minimum of the successors stored for these nodes. If we store each E_t in this way the query time is constant and the total construction time and space is in $\mathcal{O}(d(n) n^{0.5} \lg n) \subseteq \mathcal{O}(n)$, where the final upper bound follows from Lemma 1.

By the previously given explanations, this entire preprocessing takes linear time. We now use it to solve in linear time Problem 8.

Assume now that we run step 5 of the algorithm for some prefix t of w as above and the word $w[s..n]$ with $s \leq n - (\alpha + r_t)|t| + 1$. There is at most one ℓ such that

the index i_s of $w[s..n]$ in the suffix array of w is between b_ℓ and e_ℓ (that is, $\overline{v}_\ell t^{r_t}$ is a prefix of $w[s..n]$). This ℓ can be found, if it exists, in $\mathcal{O}(1)$ using the precomputed data structures (i.e., the sets E_t, organised as described above): return the value ℓ such that e_ℓ is the minimal element of E_t greater than or equal to i_s and $b_\ell \leq i_s$. Then, we repeat the procedure for the word $w[s'..n]$ where $w[s..s'-1] = \overline{v}_\ell$, but only if $n - s' + 1 \geq (\alpha + r_t)|t|$ or $s' = n + 1$. If $n - s' + 1 \leq (\alpha + r_t)|t|$ we run the processing of the original algorithm. Clearly, this process takes $\mathcal{O}(\frac{n}{\alpha t} + 2\alpha)$ steps for each t, so the complete algorithm runs in $\mathcal{O}(n)$ time. We only have to show that it works correctly, i.e., it decides whether $w \in t\{t, f(t)\}^+$. The soundness is proven by the following remark. If $w[s..n]$ starts with $\overline{v}_j t^{r_t}$ for some $j \leq 2^\alpha$, then it is enough to consider in the next iteration only the word $w[s + |\overline{v}_j|..n]$, and no other word $w[s + |\overline{v}_k|..n]$ where $k \leq 2^\alpha$ such that $\overline{v}_k$ is also a prefix of $w[s..n]$. Indeed, if there exists v_k leading to a solution, we get a contradiction with either the fact that r_t is the maximal power of t occurring as a prefix of $f(t)$, or with the fact that t and $f(t)$ are not powers of the same word.

To conclude, this implementation of Algorithm 1 runs in optimal linear time for f uniform.

§*Summary.* We were able to show the following theorem.

Theorem 10. *Let $f : V^* \to V^*$ be an anti-/morphism. Given $w \in V^*$, one can decide whether $w \in t\{t, f(t)\}^+$ in $\mathcal{O}(n \lg n)$ time. If f is uniform we only need $\mathcal{O}(n)$ time.* ■

The more general problem of testing whether there exists t with $w \in \{t, f(t)\}\{t, f(t)\}^+$ for f a fixed anti-/morphism is also worth considering. Solving this problem seems to require a different strategy than the one in Algorithm 1. There we take prefixes t of w, which determine uniquely $f(t)$, and check whether $w \in t\{t, f(t)\}^*$. Here, a prefix y does not determine uniquely, in general, a factor x such that $f(x) = y$, so more possibilities have to be considered when checking whether there exists t such that $w \in f(t)\{t, f(t)\}^*$. However, the cases of f non-erasing and uniform anti-/morphisms have solutions based on results in the line of Lemmas 4 and 5, leading to similar complexities as for Problem 8. The case of erasing anti-/morphisms is solved by a more involved algorithm, based on both combinatorics on words and number theoretic insights.

Theorem 11. *Let $f : V^* \to V^*$ be an anti-/morphism. Given $w \in V^*$, we decide whether $w \in \{t, f(t)\}\{t, f(t)\}^+$ in $\mathcal{O}(n^{1+\frac{1}{\lg \lg n}} \lg n)$ time. If f is non-erasing we solve the problem in $\mathcal{O}(n \lg n)$ time, while when f is uniform we only need $\mathcal{O}(n)$ time.* ■

3.2.3 Solution of Problem 9

Recall that our approach to solve the first question of Problem 9 is based on constructing, for the input word w, data structures that enable us to obtain in constant time the answer to queries $rep(i,j,\ell)$: "Is there $t \in V^*$ such that $w[i..j] \in \{t, f(t)\}^\ell$?", for all $1 \leq i \leq j \leq |w|$ and $1 \leq \ell \leq |w|$. Moreover, a solution for the second question is derived directly from this strategy: we only need to construct similar data structures, that allow us to answer, this time, queries $rep(i,j,\ell)$ for a single ℓ, given as input of the problem together with w.

§*The case of erasing morphisms.* We start by presenting the solution of the first question of the problem. Given an arbitrary anti-/morphism f and a word w of length n, we can construct the aforementioned data structures in $\mathcal{O}(n^{3.5})$ time. More precisely, we construct an oracle-structure that already contains the answers to every possible query.

We only give an informal description of our construction. Assume that $|w| = n$. The idea is to compute the $n \times n \times n$ three dimensional array M such that $M[i][j][k] = 1$ if there exists a word t with $w[i..j] \in \{t, f(t)\}^k$, and $M[i][j][k] = 0$, otherwise. We proceed as follows.

Let i be a position in w. We first consider the prefixes t of $w[i..n]$ such that t and $f(t)$ are not powers of the same word. Note that, for such a prefix t of $w[i..n]$, with $t \neq \lambda \neq f(t)$, and $j > i$ there is at most one number k such that $w[i..j] \in t\{t, f(t)\}^{k-1}$. The set of these prefixes is partitioned in $n^{0.5} + 1$ sets $S_{i,\delta} = \{t \mid |f(t)| = \delta\}$, for $1 \leq \delta \leq n^{0.5}$, and $S_i = \{t \mid |f(t)| > n^{0.5}\}$; note that some of these sets may actually be empty. Further, for each δ we compute $f_{i,\delta} = \max\{k \mid x^k \text{ is a suffix of } w[1..i], |x| = \delta\}$.

We first deal with the case when $t \in S_i$, for $1 \leq i \leq n$. We compute for each j the number k such that $w[i..j] \in t\{t, f(t)\}^{k-1}$; this can be done in constant time (for each j) using *LCPref*-queries, as in the previous algorithms. More precisely, for some j we only need to look at the corresponding value for $j - |t|$ and $j - |f(t)|$, increase them with 1 (if they are defined) and store as the value corresponding to j the one obtained from $j - |t|$ if t occurs as a suffix of $w[i..j]$ or the one corresponding to $j - |f(t)|$ if $f(t)$ occurs as a suffix of $w[i..j]$ (due to Lemma 5, at most one case holds); if none of these values was defined, or neither t nor $f(t)$ occurs as a suffix of $w[i..j]$, the value corresponding to j remains undefined. This entire process takes linear time. Then, for j such that $w[i..j] \in t\{t, f(t)\}^{k-1}$ and all $k' \in \{0, 1, \ldots, f_{i,\delta}\}$, where $\delta = |f(t)|$, we set $M[i - k'\delta][j][k + k'] = 1$. It is not hard to see that for $\delta > n^{0.5}$ we have $f_{i,\delta} < n^{0.5}$, so the process described above takes $\mathcal{O}(n^{0.5})$ time for each j. Now, we repeat the process for all $i \in \{1, \ldots, n\}$ and all prefixes t from S_i and discover all the factors $w[i'..j']$ and numbers k such that $\{f(t), t\}^k$, with $|f(t)| > n^{0.5}$. The time needed to do the computations described above is $\mathcal{O}(n^{3.5})$.

Further, we consider the case of the words of the sets $S_{i,\delta}$, for some fixed $\delta < n^{0.5}$ and all $1 \le i \le n$. For each i, for each t in $S_{i,\delta}$, and for each j we compute and put the pairs (i,k) such that $w[i..j] \in t\{t,f(t)\}^{k-1}$ in a list R^δ_j. This takes roughly $\mathcal{O}(n^3)$ time. Note that the number of elements of the list R^δ_j is also bounded by n^2, as for each i we have a unique decomposition of $w[i..j]$ in k parts, starting with a prefix t.

Now, for each j (and, recall, that δ is fixed), we build an $n \times n$ matrix T^δ_j, initially with all the entries set to 0. Now we partition this matrix in *diagonal arrays* obtained as follows: for ℓ from 1 to n and for p from 1 to n, if the element $T^\delta_j[\ell][p]$ is not stored already in a diagonal array, we construct a new diagonal array that stores the elements $T^\delta_j[\ell][p], T^\delta_j[\ell-\delta][p+1], \ldots T^\delta_j[\ell-d\delta][p+d]$, for $0 \le d < \frac{\ell}{\delta}$. While constructing this matrix we can keep track for each element of the array it belongs to. This procedure takes, clearly, $\mathcal{O}(n^2)$ time. These arrays partition the elements of the matrix T^δ_j so the total number of their elements is n^2.

To continue, for each element (i,k) of the list R^δ_j, we check in which diagonal array (i,k) is and memorise that we should mark (i.e., set to 1) in this array the consecutive elements $T^\delta_j[i][k], T^\delta_j[i-\delta][k+1], \ldots, T^\delta_j[i-f_{i,\delta}\delta][k+f_{i,\delta}]$. This, again, can be done in $\mathcal{O}(n^2)$ time, as we only need to memorise the first and the last of these elements (called, in the following, margins). When we are done we have to mark r_d groups of consecutive elements in each diagonal array d, where $\sum_d r_d \in \mathcal{O}(n^2)$. To do the marking we sort the margins of the groups associated with each diagonal array, with the counting sort algorithm, and then traverse each array, keeping track of how many groups contain each of its elements, and mark the elements appearing in at least one group. Sorting the lists of intervals takes $\mathcal{O}(\sum_d r_d) = \mathcal{O}(n^2)$ time, and, thus, the marking takes $\mathcal{O}(n^2)$ time in total. Once the elements of all groups are marked, for all i and k we set $M[i][j][k] = 1$ if and only if $T^\delta_j[i][k] = 1$.

The overall complexity of the computation described above for a fixed δ is $\mathcal{O}(n^3)$. As we iterate through all $\delta \le n^{0.5}$, we get that this case requires $\mathcal{O}(n^{3.5})$, as well. Now, we know all triples (i,j,k) such that $w[i..j] \in \{t,f(t)\}^k$ and t and $f(t)$ are not powers of the same word.

Further, we consider the case of triples (i,j,k) such that $w[i..j] \in \{t,f(t)\}^k$, where t and $f(t)$ are powers of the same word. By Lemma 3 we compute in $\mathcal{O}(n^2)$ time the periods of all the factors $w[i..j]$ of w and of the factors $f(w[i..j])$ of $f(w)$. We also compute in cubic time the array T from Lemma 2. Now we can check in constant time using *LCPref* queries whether $per(t) = p$, $p \mid |t|$, and $f(w[i..i+p-1])$ is a power of $w[i..i+p-1]$ (i.e., t and $f(t)$ are powers of the same word). If this is the case, we compute $m = \frac{|f(w[i..i+p-1])|}{p}$ and set $M[i][j][k] = 1$ if and only if $T[k][m][j-i+1] = 1$. Indeed, $M[i][j][k] = 1$ if and only if there exists s, k_1, k_2 such that $s \mid j-i+1$, $k_1+k_2 = k$, and $w[i..j] = ((w[i..i+p-1])^s)^{k_1}(f((w[i..i+p-1])^s))^{k_2}$,

that is, $sk_1 + smk_2 = j - i + 1$, which is equivalent to $T[k][m][j-i+1] = 1$

There is a simple case that remained to be discussed. If $f(w[i..j]) = \epsilon$, then $M[i][j][k] = 1$, for all $k \geq 1$. Identifying and memorising all such factors takes $\mathcal{O}(n^3)$ time.

By the above case analysis, we conclude that we can compute all the non-zero entries of the matrix M in $\mathcal{O}(n^{3.5})$ time. The answer to $rep(i,j,k)$ is given by the entry $M[i][j][k]$.

Finally, we consider the case when we search f-repetitions with k factors, for a fixed k. This time, we compute a two dimensional matrix M_k such that $M_k[i][j] = M[i][j][k]$, defined previously. Fortunately, M_k can be computed much quicker than the whole matrix M. According to Corollary 1 the case of t and $f(t)$ being factors of the same word can be implemented in quadratic time (the constant R from the statement of the corollary can be taken as the maximum length of $f(a)$, for all letters $a \in \text{alph}(w)$). Further, when t and $f(t)$ are not periods of the same word we just need to compute, for each i, t and j the number k' such that $w[i..j] \in t\{t, f(t)\}^{k'-1}$ and check (in constant time) whether $f(t)^{k-k'}$ is a suffix of $w[1..i]$; if all these hold, we get that $M_k[i][j] = 1$. However, note that we do not need to go through all the possible values of j. Indeed, we first generate all the prefixes of $w[i..n]$ that have the form t^ℓ with $\ell \leq k$ and see if one of them is longer than $|t| + |f(t)|$. If yes, we try to extend the longest such prefix with t or $f(t)$ iteratively until we use k factors t or $f(t)$ in the constructed word. By Lemma 5 we obtain in this process only $\mathcal{O}(k)$ such words, and these are exactly the prefixes of $w[i..n]$ that can be expressed as the catenation of at most k factors t and $f(t)$; in other words, this process provides a set that contains all the values j for which $M_k[i][j] = 1$. According to these, the whole process of computing the non-zero entries of the matrix M' takes $\mathcal{O}(n^2 \cdot k)$ time. Note that the answer to a query $rep(i,j,k)$ is given by $M_k[i][j]$; as we already mentioned, we only ask queries for the value k given as input.

§*The case of non-erasing morphisms.* For f non-erasing, the oracle matrix M described previously can be computed in $\mathcal{O}(n^3)$ time, where $|w| = n$. Initially, we set $M[i][j][k] = 0$, for $i, j, k \in \{1, \ldots, n\}$.

As in the case of erasing morphisms, by Lemma 3 we compute (and store) in quadratic time the periods of all the factors $w[i..j]$ of w and of the factors $f(w[i..j])$ of $f(w)$. We also compute in cubic time the array T from Lemma 2.

First we analyse the simplest case. We can check in constant time using *LCPref* queries whether $per(w[i..j]) = p$, $p \mid (j-i+1)$, and $f(w[i..i+p-1])$ is a power of $w[i..i+p-1]$. If so, we compute $m = \frac{|f(w[i..i+p-1])|}{p}$ and set $M[i][j][k] = 1$ if and only of $T[k][m][j-i+1] = 1$.

Further we present the more complicated cases.

First, let i be a number from $\{1, \dots, n\}$. We want to detect the factors $w[i..j]$ that belong to $t\{t, f(t)\}^{k-1}$ for some prefix t of $w[i..n]$ such that t and $f(t)$ are not powers of the same word (this case was already covered) and $k \geq 2$. To do this we try all the possible prefixes t of $w[i..n]$. Once we choose such a $t = w[i..\ell]$ we set $M[i][\ell][1] = 1$. Further, starting from the pair $(\ell, 1)$, we compute, by backtracking, all the pairs (m, e) such that $w[i..m] \in t\{t, f(t)\}^{e-1}$; basically, from the pair (m, e) we obtain the pairs $(m + |t|, e + 1)$ if $w[m + 1..m + |t|] = t$ and the pair $(m + |f(t)|, e + 1)$ if $w[m + 1..m + |f(t)|] = f(t)$. By Lemma 5 we obtain exactly one pair of the form $(m, \cdot)$ (as there is an unique decomposition of $w[i..m]$ into factors t and $f(t)$ as long as t and $f(t)$ are not powers of the same word). Therefore, computing all these pairs takes linear time. Further, if we obtained the pair (m, k) we set $M[i][m][k] = 1$.

The whole process just described can be clearly implemented in $\mathcal{O}(n^3)$ time. At this point we know all the possible triples (i, j, k) such that $w[i..j] \in t\{t, f(t)\}^{k-1}$ for some t. It remains to find also the triples (i, j, k) such that $w[i..j] \in f(t)\{t, f(t)\}^{k-1}$ for some t.

In this case, for each $i \in \{1, \dots, n\}$ we go through all the prefixes $y = w[i..\ell]$ of $w[i..n]$ and assume that $y = f(t)$. Further, we compute a set of pairs (m, e) such that $w[i..m] = y^e$; this can be done easily in linear time, using *LCPref*-queries. Now, for each of these pairs, say (m, e), we try to find a factor $t = w[m + 1..m']$ such that $f(t) = y$ and t and y are not powers of the same word. Once we found such a factor t (which can be done in constant time using *LCPref* queries and the array *inv*) we store the pair $(m + |t|, e + 1)$ and starting from this pair we compute, as in the previous case, all the pairs (m'', e') such that $w[m + 1..m''] \in t\{t, y\}^{e'-e-1}$. The key remark regarding this process is that, by Lemma 4, no two pairs having the first component equal to m'' are obtained for a fixed i. As the number of values that m'' may take is upper bounded by n, the entire computation of these pairs takes $\mathcal{O}(n)$ time. Once this is completed, we set $M[i][m][k] = 1$ for each (m, k) obtained.

In this way we identified all the triples (i, j, k) such that $w[i..j] \in \{t, f(t)\}^k$, for some t, in cubic time and stored in the array M the answers to all the possible *rep*-queries.

Now, consider the case when we search f-repetitions with k factors, for a given k and f non-erasing. The computation goes on exactly as in the case of general morphisms with the only difference that when we consider the prefix t of a word $w[i..n]$ we can restrict our search to the prefixes t shorter than $\frac{n}{k}$. Thus, the overall complexity of computing the entries of the matrix M_k decreases to $\mathcal{O}(n \cdot \frac{n}{k} \cdot k) = \mathcal{O}(n^2)$ time. Again, the answers to a query $rep(i, j, k)$ for the given value k is given by the entry $M_k[i][j]$ of the matrix M_k.

§*The case of literal morphisms.* In the case when f is literal, we are able to construct faster some data structures enabling us to answer *rep* queries. More precisely, we do not need to construct the entire oracle structure, but only some less complex matrix allowing us to retrieve in constant time the answers to our queries. To this end, we first create for the word $wf(w)$ the same data structures as in the initial solution of Problem 8. Further, we define an $n \times n$ matrix M such that for $1 \leq i, d \leq n$ the element $M[i][d] = (j, i_1, i_2)$ stores the beginning index of the longest word $w[j..i]$ contained in $\{t, f(t)\}^+$ for some word t of length d, as well as the last occurrences $w[i_1..i_1 + |t| - 1]$ of t and $w[i_2..i_2 + |f(t)| - 1]$ of $f(t)$ in $w[j..i]$, such that d divides both $i - i_1 + 1$ and $i - i_2 + 1$. If there exist t and t' with $t \neq t'$ and $w[j..i] \in \{t, f(t)\}^k \cap \{t', f(t')\}^k$, we have $t = f(t')$ and $f(t) = t'$; in this case, $M[i][d]$ equals (j, i_1, i_2) if $i_1 > i_2$ or (j, i_2, i_1), otherwise. The array M can be computed in $\mathcal{O}(n^2)$ time by dynamic programming. Intuitively, $M[i][d]$ is obtained in constant time from $M[i-d][d]$ using *LCPref* queries on $wf(w)$.

The matrix M helps us answer *rep*-queries in constant time. Indeed, the answer to a query $rep(i, j, k)$ is yes if and only if $k \mid j - i + 1$ and the first component of the triple $M[j][\frac{j-i+1}{k}]$ is lower than or equal to i, and no, otherwise.

§*Solving Problem 9.* We now give the final solutions for the two questions of Problem 9.

Let us begin with the first question. It is straightforward how one can use the computed data structures to identify, given a word w of length n, the triples (i, j, k) such that the factor $w[i..j]$ is in $\{t, f(t)\}^k$ for some t. Indeed, we return the solution-set comprising all the triples (i, j, k) for which the answer to $rep(i, j, k)$ is yes. The time needed to do so is $\Theta(n^3)$ (without the preprocessing), as we go through all possible triples (i, j, k) and check whether $rep(i, j, k)$ returns yes or no. Furthermore, any algorithm solving this problem needs $\Omega(n^3)$ operations in the worst case. Take, for instance, the non-erasing uniform morphism f defined by $f(a) = aa$ and $w = a^n$. It follows that $w[i..j]$ is in $\{a, f(a)\}^k$, for all i and j with $\lfloor (j - i + 1)/2 \rfloor \leq k \leq j - i + 1$; hence, for these w and f we have $\Theta(n^3)$ triples (i, j, k) in the solution set of our problem.

For f a literal anti-/morphism, we propose a $\Theta(n^2 \lg n)$ algorithm solving the discussed problem. Using the Sieve of Eratosthenes, we compute in $\mathcal{O}(n \lg n)$ time the lists of divisors for all numbers ℓ with $1 \leq \ell \leq n$. Further, for each pair $(i, i + \ell - 1)$ with $\ell \geq 1$ and all $d \mid \ell$ we check whether $rep(i, i + \ell - 1, d)$ returns yes. If so, the triple $(i, i + \ell - 1, d)$ is one of those we were looking for. Clearly, the algorithm is correct. Its complexity is $\mathcal{O}(n \lg n) + \Theta(\sum_{1 \leq \ell \leq n} (n - \ell + 1) d(\ell))$. Following Lemma 1, the overall complexity of this algorithm is $\Theta(n^2 \lg n)$. Moreover, any algorithm solving this problem does $\Omega(n^2 \lg n)$ operations in the worst case: for $w = a^n$ and the anti-/morphism $f(a) = a$, a correct algorithm returns exactly

$\sum_{1\leq \ell \leq n}(n-\ell+1)d(\ell) \in \Theta(n^2 \lg n)$ triples. This proves our claim.

In the case of the second question of our problem, we proceed as follows. Recall that, in this case, we are given both a word w and a number k. To identify the pairs (i,j) such that the factor $w[i..j]$ is in $\{t, f(t)\}^k$ for some t we just have to go through all the possible values for i and j and check the answer of the query $rep(i,j,k)$. Clearly, this takes $\Theta(n^2)$ time. The preprocessing, in which the data structures needed to answer *rep* queries are built, takes in the more efficient case of non-erasing morphisms $\mathcal{O}(n^2)$ time, as well; in the general case, the preprocessing takes $\mathcal{O}(n^2k)$ time, and this is more than the time needed to actually answer all the queries. This improves in a more general framework the results reported in [CKX12], where the same problem was solved in time $\mathcal{O}(n^2 \lg n)$. Finally, note that the bounds obtained for non-erasing morphisms are tight, since all the factors of length $k\ell$ of $w = a^n$ are equal to $(a^\ell)^k$, thus being solutions to our problem, no matter what anti-/morphism f is used. Hence, the number of elements in the solution-set of question (2) of Problem 9 for w is in $\Theta(n^2)$.

§*Summary.* Before concluding this section, recall that the key idea in our approach is to solve both parts of Problem 9 using *rep* queries. In order to assert the efficiency of this method note that, once data structures allowing us to answer such queries are constructed, our algorithms solve the two parts of Problem 9 efficiently. In particular, no other algorithm solving any of the two questions of Problem 9 can run faster than ours (excluding the preprocessing part), in the worst case. Hence, in general, a faster preprocessing part yields a faster complete solution for the problem. However, in the case of non-erasing and, respectively, literal anti-/morphisms (which includes the biologically motivated case of involutions) the preprocessing is as time-consuming as the part where we use the data structures we previously constructed to actually solve the questions of the problem. Thus, the time bounds obtained in these cases are tight.

Theorem 12. *Let $f : V^* \to V^*$ be an anti-/morphism and $w \in V^*$ a given word, $|w| = n$.*
(1) One can identify in time $\mathcal{O}(n^{3.5})$ the triples (i,j,k) with $w[i..j] \in \{t, f(t)\}^k$, for a proper factor t of $w[i..j]$.
(2) One can identify in time $\mathcal{O}(n^2k)$ the pairs (i,j) such that $w[i..j] \in \{t, f(t)\}^k$ for a proper factor t of $w[i..j]$, when k is also given as input.
For a non-erasing f we solve (1) in $\Theta(n^3)$ time and (2) in $\Theta(n^2)$ time. For a literal f we solve (1) in $\Theta(n^2 \lg n)$ time and (2) in $\Theta(n^2)$ time.

3.3 Streaming variants

Given that size of the words can be quite large, such as in real-world situations where we model DNA strands, the natural need for formulating problems in a streaming model arises. To this end we propose the two following streaming variants:

Problem 10. Given two words T and P over V , and an antimorphic involution $f : V^* \to V^*$, identify all the factors P' of T such that $P \longrightarrow_f^* P'$.

and

Problem 11. Given two words T and P over V , and an antimorphic involution $f : V^* \to V^*$, identify all the factors P of T that are obtained by non-overlapping f-rotations from P.

We hope to explore the solution for this problems in future work.

Chapter 4

Conclusion and future directions

The present work attempted to give further insights in graph streaming problems and variants of problems amenable to streaming.

Chapter 1 explained the importance of studying graph streaming problems, even when we are given a promised property about them, and gave a comprehensive literature review. Chapter 2 presented four instances of promised streaming graph problems on forests. A lower bound of $\Omega(n)$ working memory was presented for both problems. The chapter ended with presenting an adaptation of the classic FRT and Racke methods for the streaming case. In Chapter 3, we presented two exact problems related to finding combinatorial properties in strings, and the suggested a formulation of those problems in the streaming model. We note that out methods deal with finding strongly repeated structure in strings, and are applicable to a variety of real-world scenarios.

The current work ends with presenting future problems of interest in the area of streaming graph algorithms:

Problem 12. Given a streaming graph instance, compute the size of a maximal matching (and implicitly, a 2-approximation of the maximum matching).

We suspect that solving Problem 12 deterministically in a single pass requires at least $\Omega(n)$ space, where n is the number of vertices of the streaming graph instance.

Another interesting problem to consider for when the graph is promised to be a forest is:

Problem 13. Is it possible to characterize streaming graph problems for which the instances are promised to be forests by *property* such that only problems which satisfy that property need $\Omega(n)$ working memory ?

Finally, it would be interesting to consider streaming problems derived from testing combinatorial properties on strings, such as the ones defined at the end of Chapter 3.

Bibliography

[AG09] Kook Ahn and Sudipto Guha. Graph sparsification in the semi-streaming model. *Automata, Languages and Programming*, pages 328–338, 2009.

[AGM12] Kook Jin Ahn, Sudipto Guha, and Andrew McGregor. Analyzing graph structure via linear measurements. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 459–467. SIAM, 2012.

[AMS96] Noga Alon, Yossi Matias, and Mario Szegedy. The space complexity of approximating the frequency moments. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 20–29. ACM, 1996.

[Apo76] T. M. Apostol. *Introduction to analytic number theory.* Springer, 1976.

[BG11] Dany Breslauer and Zvi Galil. Real-time streaming string-matching. In *Combinatorial Pattern Matching*, pages 162–172. Springer, 2011.

[BGW03] Adam L Buchsbaum, Raffaele Giancarlo, and Jeffery R Westbrook. On finding common neighborhoods in massive graphs. *Theoretical Computer Science*, 299(1):707–718, 2003.

[CEPP08] Raphaël Clifford, Klim Efremenko, Benny Porat, and Ely Porat. A black box for online approximate pattern matching. In *Combinatorial Pattern Matching*, pages 143–151. Springer, 2008.

[CKS10] E. Czeizler, L. Kari, and S. Seki. On a special class of primitive words. *Theoret. Comput. Sci.*, 411:617–630, 2010.

[CKX12] E. Chiniforooshan, L. Kari, and Z. Xu. Pseudopower avoidance. *Fundam. Informat.*, 114(1):55–72, 2012.

[DFR09] Camil Demetrescu, Irene Finocchi, and Andrea Ribichini. Trading off space for passes in graph streaming problems. *ACM Transactions on Algorithms (TALG)*, 6(1):6, 2009.

[DG08] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[FKM+05] Joan Feigenbaum, Sampath Kannan, Andrew McGregor, Siddharth Suri, and Jian Zhang. On graph problems in a semi-streaming model. *Theoretical Computer Science*, 348(2):207–216, 2005.

[FKM+08] Joan Feigenbaum, Sampath Kannan, Andrew McGregor, Siddharth Suri, and Jian Zhang. Graph distances in the data-stream model. *SIAM Journal on Computing*, 38(5):1709–1727, 2008.

[FRT03] Jittat Fakcharoenphol, Satish Rao, and Kunal Talwar. A tight bound on approximating arbitrary metrics by tree metrics. In *Proceedings of the thirty-fifth annual ACM symposium on Theory of computing*, pages 448–455. ACM, 2003.

[FW65] N. J. Fine and H. S. Wilf. Uniqueness theorem for periodic functions. *Proc. of the American Mathemat. Soc.*, 16:109–114, 1965.

[GMM+13] Pawel Gawrychowski, Florin Manea, Robert Mercas, Dirk Nowotka, Catalin Tiseanu, Natacha Portier, and Thomas Wilke. Finding pseudo-repetitions. In *30th International Symposium on Theoretical Aspects of Computer Science (STACS 2013)*, volume 20, pages 257–268. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2013.

[GO12] Venkatesan Guruswami and Krzysztof Onak. Superlinear lower bounds for multipass graph processing. *arXiv preprint arXiv:1212.6925*, 2012.

[Gus97] D. Gusfield. *Algorithms on strings, trees, and sequences: computer science and computational biology*. Cambridge University Press, New York, NY, USA, 1997.

[JPS11] Markus Jalsenius, Benny Porat, and Benjamin Sach. Parameterized matching in the streaming model. *arXiv preprint arXiv:1109.5269*, 2011.

[KSB06] J. Kärkkäinen, P. Sanders, and S. Burkhardt. Linear work suffix array construction. *J. ACM*, 53:918–936, 2006.

[Lot97] M. Lothaire. *Combinatorics on Words*. Cambridge University Press, 1997.

[Mar11] Nathan Marz. A storm is coming. http://engineering.twitter.com/2011/08/storm-is-coming-more-details-and-plans.html, August 2011.

[McG05] Andrew McGregor. Finding graph matchings in data streams. *Approximation, Randomization and Combinatorial Optimization. Algorithms and Techniques*, pages 611–612, 2005.

[MMN10] Frédéric Magniez, Claire Mathieu, and Ashwin Nayak. Recognizing well-parenthesized expressions in the streaming model. In *Proceedings of the 42nd ACM symposium on Theory of computing*, pages 261–270. ACM, 2010.

[MMN12] F. Manea, R. Mercas, and D. Nowotka. Fine and Wilf's theorem and pseudo-repetitions. In *MFCS*, volume 7464 of *LNCS*, pages 668–680. Springer, 2012.

[MP80] J. Ian Munro and Mike S Paterson. Selection and sorting with limited storage. *Theoretical computer science*, 12(3):315–323, 1980.

[Mut05] S Muthukrishnan. *Data streams: Algorithms and applications*. Now Publishers Inc, 2005.

[PP09] Benny Porat and Ely Porat. Exact and approximate pattern matching in the streaming model. In *Foundations of Computer Science, 2009. FOCS'09. 50th Annual IEEE Symposium on*, pages 315–323. IEEE, 2009.

[Räc08] Harald Räcke. Optimal hierarchical decompositions for congestion minimization in networks. In *Proceedings of the 40th annual ACM symposium on Theory of computing*, pages 255–264. ACM, 2008.

[Rag99] Monika R Henzinger Prabhakar Raghavan. Computing on data streams. In *External Memory Algorithms: Dimacs Workshop External Memory and Visualization, May 20-22, 1998*, volume 50, page 107. Amer Mathematical Society, 1999.

[Zel06] Mariano Zelke. k-connectivity in the semi-streaming model. *arXiv preprint cs/0608066*, 2006.