DTU Compute
Department of Applied Mathematics and Computer Science

Algorithms and Data Structures for Strings, Points and Integers

or,

Points about Strings and Strings about Points

Søren Vind

Ph.D. Thesis
PHD-2015-366
ISSN: 0909-3192

DTU

DTU Compute
Department of Applied Mathematics and Computer Science
Technical University of Denmark

Richard Petersens Plads
Building 324
2800 Kongens Lyngby, Denmark
Phone +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk

PHD-2015-366
ISSN: 0909-3192

Abstract

This dissertation presents our research in the broad area of algorithms and data structures. More specifically, we show solutions for the following problems related to strings, points and integers. Results hold on the Word RAM and we measure space in w-bit words.

Compressed Fingerprints. The Karp-Rabin fingerprint of a string is a useful type of hash value that has multiple applications due to its strong properties. Given a string S of length N compressed into a straight line program (SLP) of size n, we show a $\mathcal{O}(n)$ space data structure that supports *fingerprint queries*, retrieving the fingerprint of any substring of S. Queries are answered in $\mathcal{O}(\lg N)$ time. If the compression is a Linear SLP (capturing LZ78 compression and variations), we get $\mathcal{O}(\lg \lg N)$ query time.

Our structure matches the best known query time bound for random access in SLPs, and is the first for general (unbalanced) SLPs that answers fingerprint queries without decompressing any text. We also support *longest common extension* queries, returning the length ℓ that the substrings from two given positions in S are equal. Answers are correct w.h.p. and take time $\mathcal{O}(\lg N \lg \ell)$ and $\mathcal{O}(\lg \lg N + \lg \ell \lg \lg \ell)$ for SLPs and Linear SLPs, respectively.

Dynamic Compression. In the *dynamic relative compression* scheme, we compress a string S of length N into n substrings of a given reference string of length r. We give data structures that maintain an asymptotically optimal compression in the scheme and support access, replace, insert and delete operations on S. Our solutions support each operation in $\mathcal{O}(\lg n / \lg \lg n + \lg \lg r)$ time and $\mathcal{O}(n + r)$ space; or $\mathcal{O}(\lg n / \lg \lg n)$ time and $\mathcal{O}(n + r \lg^{\varepsilon} r)$ space. They can be naturally generalized to compress multiple strings.

Our solutions obtains almost-optimal bounds, and are the first to dynamically maintain a string under a compression scheme that can achieve better than entropy compression. We also give improved results for the *substring concatenation* problem, and an extension of our structure can be used as a black box to get an improved solution to the previously studied *dynamic text static pattern* problem.

Compressed Pattern Matching. In the streaming model, input data flows past a client one item at a time, but is far too large for the client to store. The *annotated streaming model* extends the model by introducing a powerful but untrusted annotator (representing "the cloud") that can annotate input elements with additional information, sent as one-way communication to the client. We generalize the annotated streaming model to be able to solve problems on strings and present a data structure that allows us to trade off client space and annotation size. This lets us exploit the power of the annotator.

In *compressed pattern matching* we must report occurrences of a pattern of length m in a text compressed into n phrases (capturing LZ78 compression and variations). In the streaming model, any solution to the problem requires $\Omega(n)$ space. We show that the

problem can be solved in $\mathcal{O}(\lg n)$ client space in the annotated streaming model, using $\mathcal{O}(\lg n)$ time and $\mathcal{O}(\lg n)$ words of annotation per phrase. Our solution shows that the annotator let us solve previously impossible problems, and it is the first solution to a classic problem from combinatorial pattern matching in the annotated streaming model.

Pattern Extraction. The problem of extracting important patterns from text has many diverse applications such as data mining, intrusion detection and genomic analysis. Consequently, there are many variations of the *pattern extraction* problem with different notions of patterns and importance measures. We study a natural variation where patterns must 1) contain at most k don't cares that each match a single character, and 2) have at least q occurrences. Both k and q are input parameters.

We show how to extract such patterns and their occurrences from a text of length n in $\mathcal{O}(nk+k^3\text{occ})$ time and space, where occ is the total number of pattern occurrences. Our bound is the first output-sensitive solution for any approximate variation of the pattern extraction problem, with all previous solutions requiring $\Omega(n^2)$ time per reported pattern. Our algorithm is relatively simple, but requires a novel analysis technique that amortizes the cost of creating the index over the number of pattern occurrences.

Compressed Point Sets. Orthogonal range searching on a set of points is a classic geometric data structure problem. Given a query range, solutions must either count or report the points inside the range. Variants of this problem has numerous classic solutions, typically storing the points in a tree.

We show that almost any such classic data structure can be compressed without asymptotically increasing the time spent answering queries. This allows us to reduce the required space use if the point set contains geometric repetitions (copies of equal point set that are translated relative to each other). Our result captures most classic data structures, such as Range Trees, KD-trees, R-trees and Quad Trees. We also show a hierarchical clustering algorithm for ensuring that geometric repetitions are compressed.

Points with Colors. Colored orthogonal range searching is a natural generalization of orthogonal range searching which allows us to perform statistic analysis of a point set. We must store n points that each have a color (sometimes called a category) and support queries that either count or report the distinct colors of the points inside a query range.

We show data structures that support both types of queries in sublinear time, storing two-dimensional points in linear space and high-dimensional points in almost-linear space. These are the first (almost) linear space solutions with sublinear query time. We also give the first dynamic solution with sublinear query time for any dimensionality. Previous solutions answer queries faster, but require much more space.

Points with Weights in Practice. If points are each assigned a weight, it is natural to consider the *threshold range counting* problem. A data structure must store the points and be able to count the number of points within a query range with a weight exceeding

some threshold. This query appears naturally in a software system built by Milestone Systems, and allows detecting motion in video from surveillance cameras.

We implement a prototype of an index for 3-dimensional points that use little space and answers threshold queries efficiently. In experiments on realistic data sets, our prototype shows a speedup of at least a factor 30 at the expense of 10% additional space use compared to the previous approach. An optimized version of our proposed index is implemented in the latest version of the Milestone Systems software system.

Finger Predecessor. The predecessor problem is to store a set of n integers from a universe of size N to support predecessor queries, returning the largest integer in the set smaller than a given integer q. We study a variation where the query additionally receives a finger to an integer ℓ in the set from which to start the search. We show a linear space data structure that answers such *finger predecessor* queries in $\mathcal{O}(\lg\lg|\ell - q|)$ time. This generalizes and improves the $\mathcal{O}(\lg\lg N)$ time solutions for the standard predecessor problem. Our data structure is the first with a query time that only depends on the numerical distance between the finger and the query integer.

Dynamic Partial Sums. The well-studied partial sums problem is to store a sequence of n integers with support for sum and search queries. The sequence is static in the sense that its length cannot change, but the update operation can be used to change the value of an integer in the sequence by a given value. There are matching lower and upper bounds showing that the problem can be solved on the w-bit Word RAM in linear space and $\Theta(\lg n/\lg(w/\delta))$ time per operation, where δ is the maximum number of bits allowed in updates.

As a natural generalization we consider *dynamic partial sums*, allowing insertions and deletions in the sequence. Our solution requires linear space and supports all operations in optimal worst-case time $\Theta(\lg n/\lg(w/\delta))$, matching lower bounds for all supported operations. Our data structure is the first dynamic partial sums solution that matches the lower bounds, and the first to support storing integers of more than $\lg w$ bits.

Resumé

Denne afhandling præsenterer vor forskning i feltet algoritmer og datastrukturer. Mere specifikt viser vi løsninger til følgende problemer relateret til strenge, punkter og heltal. Vores grænser gælder i RAM-modellen, og vi måler plads i antal w-bit ord.

Komprimerede Fingeraftryk. En strengs Karp-Rabin fingeraftryk er en brugbar type hash-værdi, der har talrige anvendelser på grund af dens stærke egenskaber. Givet en streng S af længde N komprimeret som et Straight Line Program (SLP) af størrelse n, viser vi en datastruktur der kræver $\mathcal{O}(n)$ plads som understøtter *fingeraftryksforespørgsler* der returnerer fingeraftrykket af en given delstreng i S. Vi svarer på forespørgsler i $\mathcal{O}(\lg N)$ tid. Hvis kompressionen er en Lineær SLP (dette omfatter LZ78 kompression og varianter) kan vi svare i $\mathcal{O}(\lg\lg N)$ tid.

Vores løsning matcher den bedst kendte tidsgrænse for at tilgå et enkelt tegn i SLPer, og det er den første for generelle (ikke-balancerede) SLPer der svarer på forespørgsler uden at dekomprimere noget tekst. Vi understøtter også *længste fælles delstreng* forespørgsler, der returnerer længden ℓ som S matcher sig selv startende fra to givne positioner. Vi svarer forespørgsler korrekt med høj sandsynlighed i tid $\mathcal{O}(\lg N \lg \ell)$ og $\mathcal{O}(\lg\lg N + \lg \ell \lg\lg \ell)$ for henholdsvis SLPer og Lineære SLPer.

Dynamisk Kompression. I *dynamisk relativ kompression* komprimerer vi en streng S af længde N som n delstrenge fra en given referencestreng af længde r. Vi giver datastrukturer der vedligeholder en asymptotisk optimal kompression i denne model og understøtter operationer der tilgår, ændrer, indsætter og sletter tegn i S. Vores løsninger understøtter hver operation i $\mathcal{O}(\lg n/\lg\lg n + \lg\lg r)$ tid og $\mathcal{O}(n+r)$ plads; eller $\mathcal{O}(\lg n/\lg\lg n)$ tid og $\mathcal{O}(n + r\lg^{\varepsilon} r)$ plads. De generaliserer naturligt til at opbevare flere strenge.

Vi opnår næsten-optimale grænser, og vores løsning er den første der understøtter at dynamisk vedligeholde en streng i en type kompression der kan opnå bedre end entropi-kompression. Som en del af vores løsning viser vi forbedrede grænser for *delstrengskonkatenering* og en udvidelse af vores struktur kan anvendes til at opnå en bedre løsning for det tidligere studerede problem *dynamisk mønstergenkendelse*.

Komprimeret Mønstergenkendelse. I streamingmodellen ankommer en strøm af data et element af gangen som input til en klient der ikke har plads til at opbevare det hele. Den *annoterede streamingmodel* udvider modellen ved at introducere en kraftfuld ikke-pålidelig annotator (som repræsenterer "skyen") der kan tilføje annoteret information til input elementer, ved at sende envejskommunikation til klienten. Vi generaliserer denne model for at kunne løse problemer på strenge, og præsenterer en datastruktur der lader os afveje plads på klienten og størrelsen af annotationen. Dette tillader os at anvende annotatorens kraft.

I *komprimeret mønstergenkendelse* skal vi rapportere forekomster af et mønster af længde m i en tekst der er komprimeret som n fraser (omfatter LZ78 kompression og varianter). I streamingmodellen kræver enhver løsning til problemet $\Omega(n)$ plads. Vi viser at problemet kan løses i den annoterede streamingmodel med $\mathcal{O}(\lg n)$ klientplads og $\mathcal{O}(\lg n)$ tid og $\mathcal{O}(\lg n)$ ord annotering per frase. Dermed bryder vores resultat med pladsgrænsen i streamingmodellen, og det er den første løsning på et klassisk problem fra kombinatorisk mønstergenkendelse i den annoterede streamingmodel.

Mønsterudvinding. Der er mange forskelligartede anvendelser af at kunne udvinde vigtige mønstre fra tekst, eksempelvis i data mining, intrusion detection og genetisk analyse. Derfor er der varianter af *mønsterudvindingsproblemet*, med forskellige typer mønster og mål for vigtighed. Vi studerer en naturlig variation hvor mønstre har 1) højst k wildcards der hver matcher et tegn, og 2) et minimalt antal forekomster for at betragtes som vigtige.

Vi viser hvordan sådanne mønstre og deres forekomster kan udvindes fra en tekst af længde n i $\mathcal{O}(nk + k^3\text{occ})$ tid og plads, hvor occ er det totale antal mønsterforekomster. Vores grænse er den første løsning til en hvilken som helst ikke-eksakt variation af mønstergenkendelsesproblemet, alle tidligere løsninger kræver $\Omega(n^2)$ tid per rapporteret mønster. Vores algoritme er relativt simpel, men kræver en ny analyseteknik der amortiserer udgiften ved at konstruere indekset over det totale antal mønsterforekomster.

Komprimerede Punktmængder. *Ortogonal søgning* på en mængde punkter er et klassisk geometrisk datastrukturproblem. Løsninger skal enten tælle eller returnere punkter i et givet forespørgselsområde. Der er talrige klassiske løsninger på problemet der typisk opbevarer punkterne i et træ.

Vi viser at næsten alle klassiske datastrukturer til problemet kan komprimeres uden at øge tiden til at svare på forespørgsler asymptotisk. Dette lader os reducere det krævede pladsforbrug hvor punktsættet indeholder geometriske gentagelser (kopier af ens punktsæt). Vores resultat omfanger de fleste klassiske datastrukturer, såsom Range træer, KD-træer, R-træer og Quad træer. Vi viser en hierarkisk klyngealgoritme der sikrer at geometriske gentagelser kan komprimeres.

Punkter med Farver. Farvet ortogonal søgning er en naturlig generalisering af ortogonal søgning der lader os foretage statistisk analyse af punktsæt. Vi skal gemme n punkter der hver har en farve (også kaldet en kategori) og understøtte forespørgsler der enten tæller eller returnerer de unikke farver på punkterne i et forespørgselsområde.

Vi viser datastrukturer der understøtter begge typer forespørgsler i mindre end lineær tid, og gemmer to-dimensionelle punkter i lineær plads og høj-dimensionelle punkter i næsten-lineær plads. Dette er de første løsninger med (næsten) lineær pladsforbrug. Vi viser også den første dynamiske løsning med under-lineær forespørgselstid for alle dimensionaliteter. Tidligere løsninger svarer hurtigere, men kræver meget mere plads.

Punkter med Vægte i Praksis. Hvis vi tildeler hvert punkt en vægt er det naturligt at studere problemet *tærskeltælling*. En løsning til dette gemmer punkterne og understøtter at tælle punkterne i et forespørgselsområde med en vægt som overstiger en given tærskelværdi. Denne type forespørgsel optræder naturligt i software udviklet af Milestone Systems, og muliggør at finde bevægelse i overvågningsvideo.

Vi implementerer en prototype af et indeks for 3-dimensionelle punkter som bruger lidt plads og svarer effektivt på forespørgsler. I eksperimenter på realistiske datasæt bruger vores prototype 10% yderligere plads men svarer mindst 30× hurtigere på forespørgsler sammenlignet med den tidligere løsning. En optimeret løsning af vores indeks er implementeret i den seneste udgave af softwaren fra Milestone Systems.

Finger-Forgænger. I forgængerproblemet skal vi opbevare en mængde af n heltal fra et univers af størrelse N og understøtte forespørgsler efter forgængere, som returnerer det største heltal i mængden som er mindre end et givet heltal q. Vi studerer en variation hvor en forespørgsler også modtager en finger til et heltal ℓ i mængden, hvorfra en søgning kan starte. Vi viser en lineær plads datastruktur der svarer på *fingerforgænger-forespørgsler* i $\mathcal{O}(\lg\lg|\ell - q|)$ tid. Dette generaliserer og forbedrer løsningerne til standard forgænger problemet, som kræver $\mathcal{O}(\lg\lg N)$ tid. Vores datastruktur er den første løsning med et tidsforbrug der kun afhænger af den numeriske afstand mellem fingeren og forespørgslen.

Dynamiske Delsummer. Det velstuderede delsumsproblem er at gemme en sekvens af n heltal med understøttelse for forespørgslerne sum og søg. Sekvensen er statisk idet dens længde ikke kan ændres, men update operationen kan bruges til at ændre værdien af et givent heltal i sekvensen med en værdi på højst $|2^\delta|$. Der er matchende øvre og nedre grænser for problemet som viser at det kan løses på en Word RAM med w-bit ord i lineær plads og $\Theta(\lg n/\lg(w/\delta))$ tid per operation, hvor δ er det maksimale antal tilladt bits i update.

Som en naturlig generalisering studerer vi *dynamiske delsummer*, hvor vi tillader indsættelser og sletninger i sekvensen. Vores løsning kræver lineær plads og understøtter alle operationer i optimal tid $\Theta(\lg n/\lg(w/\delta))$, som matcher nedre grænser for alle understøttede operationer. Vores resultat er den første løsning til dynamiske delsummer der matcher de nedre grænser, og den første til at understøtte at gemme heltal på mere end $\lg w$ bits.

Preface

This dissertation presents results in the general topic of algorithms and data structures. It was prepared during my enrollment from October 2012 to March 2015 as a PhD student at the Department of Applied Mathematics and Computer Science in partial fulfillment of the requirements for obtaining a PhD degree at the Technical University of Denmark.

My primary focus has been on designing solutions for problems using little space. Each included paper contain at least one data structure with this particular property that solve a problem on strings, points or integers. Five papers have been peer-reviewed and published at international conferences and the remaining two are in submission.

Managed Video as a Service. My PhD scholarship was part of project *Managed Video as a Service* which was conceived in collaboration by Aalborg University, the Technical University of Denmark and industry partners Milestone Systems, Nabto and Securitas. The project was funded by a grant from the Danish National Advanced Technology Foundation (Højteknologifonden), scheduled to last three years and involve four PhD students.

My part of the project was called *Algorithms for Metadata*. The goal was to research and develop data structures for indexing massive amounts of meta data that support efficient queries. One papers is a direct result of my involvement in the project and collaboration with Milestone Systems.

Acknowledgements. I am particularly grateful to my dear advisors Philip Bille and Inge Li Gørtz for suggesting that this particular PhD position would be a great fit for me, thereby pulling me back into the academic world that I had decided to leave. Your advice have been indispensable and I hope you are not too dissappointed in my decision to leave academia. My great family and friends, colleagues, office mates and the rest of the Copenhagen Algorithms community have made the last two and a half years very enjoyable. I also owe thanks to Roberto Grossi, Raphaël Clifford and Benjamin Sach for accepting me as a guest during my research visits in Pisa and Bristol: you made me feel at home. My coauthors Philip Bille, Patrick Hagge Cording, Roberto Grossi, Inge Li Gørtz, Markus Jalsenius, Giulia Menconi, Nadia Pisanti, Benjamin Sach, Frederik Rye Skjoldjensen, Roberto Trani, and Hjalte Wedel Vildhøj made the periods of intense research insanely rewarding. And finally, thanks to Lærke for her support, love and understanding.

Thank you all. I take all responsibility, but you made this possible.

Søren Vind
Copenhagen, March 2015

Contents

Part I

Setting the Stage

1 Introduction

In this dissertation, we consider the *design and analysis of algorithms and data structures*:

> Our goal is to use computational resources efficiently. We *design* data structures and algorithms that solve *problems*, and *analyse* proposed *solutions* in a *model of computation* that lets us predict efficiency and compare different solutions on real computers.

Data Structures are the basic building blocks in software engineering; they are the organizational method we use to store and access information in our computers efficiently. An *Algorithm* specifies the steps we perform to complete some task on an input in order to produce the correct output, relying on underlying data structures. Naturally, deep knowledge and understanding of algorithms and data structures are core competences for software engineers, absolutely vital to developing efficient software.

This dissertation documents our research in this field of theoretical computer science, and later chapters each include one of the following papers that are published in or submitted to peer-reviewed conference proceedings. Papers appear in separate chapters in their original form except for formatting, meaning that notation, language and terminology has not been changed and may not be consistent across chapters. Authors are listed alphabetically as is the tradition in theoretical computer science (except for *Indexing Motion Detection Data for Surveillance Video*, which was published at a multimedia conference).

Chapter 3: ***Fingerprints in Compressed Strings*****.**
By Philip Bille, Patrick Hagge Cording, Inge Li Gørtz, Benjamin Sach, Hjalte Wedel Vildhøj and Søren Vind. At Algorithms and Data Structures Symposium (WADS) 2013 [Bil+13].

Chapter 4: ***Dynamic Relative Compression and Dynamic Partial Sums*****.**
By Philip Bille, Patrick Hagge Cording, Inge Li Gørtz, Frederik Rye Skjoldjensen, Hjalte Wedel Vildhøj and Søren Vind. In submission.

Chapter 5: ***Compressed Pattern Matching in the Annotated Streaming Model*****.**
By Markus Jalsenius, Benjamin Sach and Søren Vind. In submission.

Chapter 6: ***Output-Sensitive Pattern Extraction in Sequences*****.**
By Roberto Grossi, Giulia Menconi, Nadia Pisanti, Roberto Trani and Søren Vind. At Foundations of Software Technology and Theoretical Computer Science (FSTTCS) 2014 [Gro+14a].

Chapter 7: ***Compressed Data Structures for Range Searching*.**
By Philip Bille, Inge Li Gørtz and Søren Vind. At International Conference on Language and Automata Theory and Applications (LATA) 2015 [Bil+15].

Chapter 8: ***Colored Range Searching In Linear Space*.**
By Roberto Grossi and Søren Vind. At Scandinavian Symposium and Workshops on Algorithm Theory (SWAT) 2014 [Gro+14b].

Chapter 9: ***Indexing Motion Detection Data for Surveillance Video*.**
By Søren Vind, Philip Bille and Inge Li Gørtz. At IEEE International Symposium on Multimedia (ISM) 2014 [Vin+14].

The journal version of the following paper appeared during my PhD enrollment, but as the results were obtained and the conference version published prior to starting the PhD programme, the paper is omitted from this dissertation.

***String Indexing for Patterns with Wildcards*.**
By Philip Bille, Inge Li Gørtz, Hjalte Wedel Vildhøj and Søren Vind. At Scandinavian Symposium and Workshops on Algorithm Theory (SWAT) 2012 [Bil+12b]. In Theory of Computing Systems 2014 [Bil+14].

The dissertation is organized as follows. The following Section 1.1 is a brief general introduction to the field of algorithmic research, and may be skipped by readers familiar with the subject. Chapter 2 gives an overview of our contributions, and the later Chapters include the papers mentioned above.

1.1 Background and Context

In his pioneering work *"On Computable Numbers, with an Application to the Entscheidungsproblem"* [Tur36] from 1936, Alan Turing in a single stroke became the father of the field of *theoretical computer science*. He gave the first general model of the theoretical capabilities of computers with the *Turing Machine*, and in doing so formalized the concept of *algorithms*. In the following 80 years the computer revolution occurred and the first institutions devoted to computer science research were established around fifty years ago, causing the field to naturally expand[1]. Today, the field is founded on modern day equivalents of the concepts introduced by Turing:

Algorithms An *algorithm* is a method for solving a *problem* on an input which produces the correct output. The problem is a central concept which states the properties of the input and requirements for the output. *Sorting* is a classic example of an algorithmic problem: given as input a list of numbers, return the same numbers listed in non-decreasing order.

[1] In 1965, the Department of Computer Science was founded at Carnegie Mellon University, as possibly the first such department in the world.

Data Structures A *data structure* is a method for maintaining a representation of some data while supporting *operations* on the data. Operations typically either *update* or answer *queries* on the stored data. We may think of a data structure as a collection of algorithms operate on the stored data. An example of a data structure problem is *sorted list representation*: store a list of numbers subject to insertions and deletions, and support queries for the k'th smallest number.

Model of Computation The *model of computation*[2] is an abstract model of a computer that ignores most real-world implementation details. This allows us to reason about the performance of algorithms and data structures on real physical computers. The model of computation we use in this thesis is called the *unit-cost w-bit Word RAM*, which resembles the capabilities of modern day processors: memory is modeled as a sequence of w-bit words that can be read and written, and we can perform arithmetic and word-operations on (pairs of) words.

Generally, research in a particular problem takes two different angles. One is to prove that the problem is impossible to solve using fewer resources than some *lower bound*, meaning that *any* solution must use at least that many resources. However, in this thesis our focus is on the opposite direction, which is to show that there is a solution with some *upper bound* on the resources used.

We characterise proposed solutions using several parameters. The output of a (deterministic) solution must always adhere to its requirements and be a *correct* answer. The *performance* of a solution is the amount of resources it requires in our model of computation relative to the size n of a finite input. Typically, we split the performance into two components: the *space usage* is the number of memory words required, and the *time* is the number of unit-cost machine operations required. Here, we study *worst-case analysis*, which is the performance obtained when given an input that causes the solution to perform as poorly as possible.

There are some problems for which traditional *deterministic* solutions have inherently bad worst-case performance. To work around this, *randomized* solutions relaxes the strict requirements on correctness or good worst-case performance. Instead, *Monte Carlo* solutions have worst-case performance guarantees with a low probability of producing incorrect outputs. On the contrary, *Las Vegas* solutions always produce correct outputs with a low risk of exhibiting bad performance for some inputs.

1.2 Why is this Important?

We now give a slightly contrived example that serves to illustrate why studying algorithms and data structures is important. It shows two different possible solutions for the fundamental *dictionary* data structure problem, and indicates how we can calculate their (relative) performance.

[2]Also sometimes called the *machine model*.

Imagine that you want to look up the word *"Dissertation"* in The Oxford English Dictionary in the physical book version from 1989. You remember that you used to perform this kind of query before the internet, so you would know to go to a page roughly in the middle and see if you had surpassed the entries for "D". If so, you would go to the middle part in the left half and continue like that. The property of the dictionary you use is that dictionary editors traditionally sort the entries.

However, suppose that the editors forgot to sort the words before printing your dictionary: How would you find the entry? Your only possibility would be to start on the first page and check each entry it contains. This is unfortunate: You risk that the entry is on the last page in the dictionary, forcing you to look at all pages!

The 1989 Oxford English Dictionary consist of 20 volumes with 21730 pages that contain about 300000 entries. If you can check 1 entry per second the normal (sorted) search would complete in less than 20 seconds, but the second (unsorted) search would require slightly more than 83 hours. That is, your normal search is roughly 16000 times faster than the unsorted search!

Our sorted or unsorted dictionary is a good model of how we may store data in our computer, and the search for a word is called a *membership* query. In computer science the first (sorted) search is called a *binary search*, and it can be performed exactly when the data we search in is sorted. Otherwise we must perform a *linear search* like the second (unsorted) search.

Structuring data in an ordered fashion to help search is of course not a new invention from computer science. The alphabetic ordering of information dates back at least two thousand years, and the first monolingual English dictionary from 1604 was also ordered alphabetically[3]. The example simply shows two simple possibilities for structuring data: the remainder of this dissertation discuss much more advanced data structuring techniques and algorithms for finding information in the structured data. As shown in the example, the difference in performance is dramatic, and it only grows with larger data sets. This is the main motivation for studying algorithms and data structures: we *have to* use clever algorithmic techniques to cope with the ever growing quantities of data.

[3] Interestingly, the alphabetic ordering was unusual enough to require an explanation from the editor.

2 Contributions

This chapter gives a brief overview of our contributions and shows how they fit in the larger world of algorithms and data structures. It is meant to describe important related problems from our perspective, not to provide an exhaustive history of the field. Each of the following chapters contain a paper on a problem related to either strings or points. We describe the contributions n the same order as the chapters, and finally present our additional contributions to data structures for integers.

Our included papers each contribute at least one newly-designed data structure for particular problem. In some instances, the data structure is a cornerstone in giving a solution to an algorithmic problem, while in other cases it is the singular contribution. All but one of the papers are purely theoretical, meaning that though they may perform well in practice, their efficiency are only established in theory. In a single case, the proposed data structure was implemented and tested experimentally on realistic data sets.

2.1 Model of Computation

The computation model used in all papers is the *unit-cost w-bit Word RAM* [Hag98], which models memory as an array of w-bit words that can each be read and written in constant time. To be able to index into the memory array in constant time, we require $w \geq \lg n$ where n is the problem size. Comparisons, arithmetic operations and common word-operations (as available in the `C` programming language) each take constant time.

2.2 Strings

A *string* (or text) is a sequence of characters from some *alphabet*. Classically, we store the sequence as is, with each character taking up a single word in memory. Text is core data abstraction that naturally appears in countless applications, for example in log files, text documents or DNA sequences such as `AGCAATTTGACCTAGATA`.

We may also consider storing strings in compressed form. *Phrase compression*[1] is a standard form of compression, where the text is compressed into a sequence of phrases that may each be an extension of some previous phrase in the sequence. When dealing with compressed text, we denote by n the size of the compressed text (equal to the number of phrases), and let N be the size of the decompressed text.

Typically, encoding in a phrase-compression scheme processes a text from left to right, maintaining a growing dictionary of phrases that was used to encode previous parts of the text, which may be extended. Decompression of the whole string is simple, but it is not

[1] Also sometimes called *grammar compression*.

straightforward to access a single character, as phrases may be recursively defined (i.e. they extend on each other), requiring us to process many phrases to access a character.

We include four papers on string problems. First, we study *compressed fingerprints*, which provide a useful primitive for designing algorithms that work efficiently on most types of phrase compression. Second, we consider *dynamic compression*, which is to efficiently maintain a compressed representation of a string subject to insert, delete, replace and random access operations on the string. Third, we study the *compressed pattern matching* problem, which is to find the occurrences of a given *pattern* in a given *compressed text* (i.e. the starting positions where a substring of the text matches the pattern). Finally, we consider *pattern extraction*, which may be considered the opposite of pattern matching. It is to extract unknown and *important patterns* from a given text, where importance is measured as the number of occurrences of the pattern in the text.

2.2.1 Compressed Fingerprints

In 1987, Karp and Rabin [Kar+87] proposed a classic *Monte Carlo randomized* pattern matching algorithm for uncompressed text of length N. Their approach rely on *fingerprints* to efficiently compare substrings, allowing one-sided errors: all occurrences will be reported, but there is a small risk of reporting occurrences that do not exist. Their fingerprints have subsequently been used as a cornerstone in solving many string problems [Ami+92; And+06; Col+03; Cor+05; Cor+07; Far+98; Gas+96; Kal02; Por+09].

The *Karp-Rabin fingerprint* function ϕ for a string x is given by

$$\phi(x) = \sum_{i=1}^{|x|} x[i] \cdot c^i \bmod p,$$

where $c < p$ is a randomly chosen positive integer, and p is a prime. If choosing prime p sufficiently large compared to the length of x, we get multiple key properties that make fingerprints extremely useful:

1. they support composition in constant time, allowing us to find the fingerprint of a string from the fingerprints of two halves,

2. each fingerprint can be stored in constant space,

3. equal strings have equal fingerprints, and different strings have different fingerprints with high probability.

An similarly fundamental string primitive is called *longest common extensions* (LCE), used in many string algorithms as a black box (see for example [Ami+92; And+06; Col+03; Cor+05; Cor+07; Far+98; Gas+96; Kal02; Por+09]). The answer to an LCE query on a string s is the length of the longest matching substring in s starting from two given positions positions.

For uncompressed strings, both fingerprints and answers to LCE queries can be retrieved in constant time and linear space (as well as random access to a character).

Our Contribution: *Fingerprints for Compressed Strings* The paper considers strings of length N that are compressed into a *straight line program* (SLP) with n rules. The SLP is a general phrase compression scheme that captures many compression schemes with low overhead. That is, it efficiently models many other types of phrase compression, such as the LZ family schemes invented by Lempel and Ziv [Ziv+77; Ziv+78].

We present a data structure for storing a SLP in linear compressed space $\mathcal{O}(n)$ that supports answering *fingerprint queries* that returns the Karp-Rabin fingerprint of any decompressed substring of the compressed string in $\mathcal{O}(\lg N)$ time. Our structure also supports answering LCE queries in time $\mathcal{O}(\lg \ell \lg N)$ correctly with high probability. We also give a modified data structure that works for *Linear SLPs*, which restricts SLPs to model compression schemes such as LZ78 [Ziv+78] with constant overhead. Here, we show that fingerprints can be obtained in improved time $\mathcal{O}(\lg \lg N)$ and $\mathcal{O}(n)$ space, and we support LCE queries in $\mathcal{O}(\lg \lg N + \lg \ell \lg \lg \ell)$ time. Our data structure for general SLPs matches the query time of a structure by Gagie, Gawrychowski, Kärkkäinen, Nekrich and Puglisi [Gag+12a] which only answers queries in balanced SLPs.

The property we exploit is that any fingerprint can be composed from many smaller fingerprints that we store in the data structure. To obtain linear (compressed) space, we use techniques introduced in a seminal paper by Bille, Landau, Raman, Sadakane, Satti, and Weimann [Bil+11] where they showed how to store a SLP in linear space to support random access to a character in the decompressed string in $\mathcal{O}(\lg N)$ time. We use our fingerprints for substring comparisons to answer LCE queries using an exponential search followed by a binary search.

The consequence of our result is that (probabilistically) comparing two substrings can be done in the same time as accessing a single character in SLPs. This is the same situation as for uncompressed strings, though both operations have a logarithmic slowdown for compressed strings. Our result immediately implies compressed space implementations of all previous algorithms relying on access to Karp-Rabin fingerprints or LCE queries.

Future Work For uncompressed strings, it is possible to answer both random access, fingerprint and LCE queries in constant time. This is contrary to the situation for SLPs, where we are able to answer only the first two queries in the same time $\mathcal{O}(\lg N)$, and the best solution for answering LCE queries take $\mathcal{O}(\lg^2 N)$ time. Solving this disparaty for LCE queries is an obviously interesting research direction.

For random access in SLPs Verbin and Yu [Ver+13] gave a lower bound showing that the result of [Bil+11] is almost optimal for most compression ratios. A natural question is if optimal structures exist for all compression ratios, and if fingerprint queries can be improved similarly.

2.2.2 Dynamic Compression

Classically, we store a string as a sequence of characters that each require a word, giving support for basic operations in constant time. For example, given a string s and a position i, we can access the character $s[i]$ or replace it with another alphabet symbol. We may also consider more complicated operations, such as deleting or inserting a character in

a string. Intuitively, because these operations cause all characters after the inserted or deleted character to have their position changed, such operations require superconstant time. Indeed, Fredman and Saks [Fre+89] gave a lower bound of $\Omega(\lg n/\lg\lg n)$ time per operation for supporting access, insert and delete in a string of length n.

When storing compressed strings, we normally do not support dynamic operations. The intuitive reason is that we use redundancy in the string to compress it efficiently, and any such redundancy can be removed by operations that change the compressed string. For example, it can be shown that an SLP for a string s may double in size if inserting a character in the middle of s. However, a number of results show that it is possible to maintain an entropy-compressed string representation under some dynamic operations. In particular, the structure of Grossi, Raman, Rao and Venturini [Gro+13] supports access, replace, insert and delete in optimal $\mathcal{O}(\lg n/\lg\lg n)$ time and k^{th} order empirical entropy space (plus lower order terms), which is always $\Omega(n/\lg n)$ words.

Recently, the *relative compression* scheme originally suggested by Storer and Szymanski [Sto+78; Sto+82] has gained significant interest due to its performance in practice. Here, we store an uncompressed reference string, and strings are compressed into a sequence of substrings from the reference string. It has been shown experimentally that variations of their general scheme such as the *Relative Lempel-Ziv* compression scheme exhibit extremely good performance in applications storing a large dictionary of similar strings such as genomes [Fer+13; Kur+11a; Che+12; Do+14; Hoo+11; Kur+11b; Wan+13; Deo+11; Wan+12; Och+14].

Our Contribution: *Dynamic Relative Compression and Dynamic Partial Sums* We present a new dynamic variant of relative compression that supports access, replace, insert and delete of a single character in the compressed string. Our data structure maintains an asymptotically optimal compression of size n, storing a compressed string of (uncompressed) length N relative to a given reference string of length r. Operations are supported in:

1. almost-optimal time $\mathcal{O}(\lg n/\lg\lg n+\lg\lg r)$ and linear space $\mathcal{O}(n+r)$, or
2. optimal time $\mathcal{O}(\lg n/\lg\lg n)$ and almost-linear space $\mathcal{O}(n+r\lg^{\varepsilon} r)$.

Crucially, our result can be easily generalized to store multiple strings compressed relative to the same reference string. Interestingly, the bounds we obtain almost match those for uncompressed strings. That is, we can store our strings in compressed form with almost no overhead for insert and delete operations. This is the first dynamic compression result that ensures asymptotically optimal grammar-based compression, with previous approaches only providing entropy compression. Our result implies an improved solution to the dynamic text static pattern matching problem studied by Amir, Landau, Lewenstein and Sokol [Ami+07].

The difficulty in solving the problem comes from the new dynamic operations that change the string, forcing us to modify the representation. This may break both our desired time bound for operations and our asymptotically optimal compression ratio.

Our solution rely on two key properties: 1) modifications are local and can be reduced to accessing and changing at most five substrings in the compression sequence; and 2) a *maximal compression* is enough to maintain an asymptotically optimal compression relative to a given reference string. This lets us solve the problem by reducing it to two core data structure problems. First, we ensure compression maximality by detecting if neighboring substrings among the five touched substrings can be merged into one. To support efficient access operations, we maintain the dynamically changing sequence of substring lengths. These data structure problems are known as *substring concatenation* and *dynamic partial sums*, and we give new general solutions that significantly improve on previous solutions to both.

In substring concatenation, we must preprocess a text of length r to support queries asking for an occurrence of a substring of the text which is the concatenation of two other substrings in the text (if it exists). We show two different solutions. The first requires $\mathcal{O}(\lg\lg r)$ time and linear space and is a simple application of some of our previous work [Bil$^+$14]. The second solution requires constant time and $\mathcal{O}(r \lg^\varepsilon r)$ space, reducing the problem to answering one-dimensional range emptiness queries on nodes of a suffix tree. We describe our dynamic partial sums data structure in Section 2.4.2.

Future Work A natural question is if we can get the best of both worlds for substring concatenation queries, answering queries in constant time and linear space? Such a data structure can be plugged into our current solution for dynamic relative compression, resulting in a linear space and optimal time solution that closes the problem. The presented compression scheme is relatively simple, and an experimental performance comparison against other possible approaches would be highly interesting.

2.2.3 Compressed Pattern Matching

A classic string problem is *pattern matching*, where we must find occurrences of a *pattern* in a text. Traditionally a pattern consist of characters from the same alphabet as the text, and a pattern occurrence is the location of a substring in the text that is equal to the pattern. In *compressed pattern matching*, the text of length N is compressed into n phrases and we must report the occurrences of a pattern of length m in the decompressed text.

Good solutions exist for both the classic and compressed problem. There are multiple optimal solutions to the classic variant, some of which even work in the *streaming model* [Mun$^+$80; Fla$^+$85; Alo$^+$99]. In this model of computation, the text is received in a stream one character at a time, only sublinear space $o(N)$ is allowed and we must report occurrences immediately. Known solutions for compressed pattern matching do *not* work in the streaming model. In fact, there is a space lower bound for compressed pattern matching in the streaming model, showing that it requires $\Omega(n)$ space.

In order to model the current state of affairs in computing with easy access to massive computational power over the internet, the *annotated streaming model* introduced by Chakrabarti, Cormode and McGregor [Cha$^+$09; Cha$^+$14] expands the classic streaming model by introducing an untrustworthy *annotator*. This annotator is a theoretical abstraction of "the cloud", with unbounded computational resources and storage, and it

assists a *client* in solving some problem by providing an *annotated data stream* that is transmitted along the normal input stream (i.e. the client-annotator communication is one-way). Software and hardware faults, as well as intentional attempts at deceit by the annotator are modeled by assuming that the annotator cannot be trusted. In this model, the relevant performance parameters are the client space, the time to process a phrase and the amount of annotation sent per input element.

Our Contribution: *Compressed Pattern Matching in the Annotated Streaming Model* We give a randomized trade-off solution to compressed pattern matching in the annotated streaming model which require $\mathcal{O}(\lg n)$ space on the client, and uses $\mathcal{O}(\lg n)$ words of annotation and time per phrase received (at one end of the trade-off). The result is possibly the first to show the power of the annotator in solving traditional problems on strings (existing work in the model focuses on graph algorithms). The implication of the result is that the power of the annotator is highly useful even though it cannot be trusted. In particular, our solution circumvent the $\Omega(n)$ space lower bound at the cost of annotation.

We solve the problem by a careful application of techniques for pattern matching in compressed strings, providing a simple initial solution in linear space and constant time per phrase with no annotation. We reduce the space use with a new randomized annotated data structure that generally allows us to trade client space for annotation, allowing us to store and access arbitrary information in logarithmic client space with logarithmic annotation overhead.

Future Work There are multiple natural directions to go in extending our work in the annotated streaming model. Results that decrease the amount of annotation per input element would be of great interest. It seems likely that our result can be extended to dictionary matching, where matches with any pattern from a dictionary of patterns must be reported.

Pattern matching for more powerful pattern such as regular expressions may be worth investigating. Intuitively, access to an untrusted annotator with unbounded computational power should help, as finding such patterns typically require more resources than classic patterns.

2.2.4 Pattern Extraction

In *pattern extraction*, the task is to extract the "most important" patterns from a text S. We define the occurrence of a pattern in S as in *pattern matching*, and the importance of a pattern depends on its statistical relevance like its number of occurrences.

Pattern extraction should *not* be confused with pattern matching. In fact, we may consider the problems inverse: the former gets a text S as input and extracts patterns P and their occurrences from S, where both are unknown (and P meets some criteria); the latter gets S and a given pattern p as input, and finds unknown occurrences of p in S.

As is the case in pattern matching, one can imagine many ways of generalizing the standard patterns which consist of text characters, making the problem more real-world

applicable. Indeed, many approximate pattern variations and measures of importance exist in the pattern extraction world [Com$^+$13; Cun$^+$12; Apo$^+$11; Fed$^+$14; Esk04; Ili$^+$05; Gro$^+$11; Sag98; Ukk09; Ari$^+$07]. The natural variation we study allows patterns that contain the special don't care character $\star$, which means that the position can be ignored (so $\star$ matches any single character in S). For example, $\texttt{TA}\star\texttt{C}$ is such a pattern for DNA sequences that e.g. matches strings TATC and TAGC.

Our Contribution: *Output-Sensitive Pattern Extraction in Sequences* We consider extraction of patterns with at most k don't cares and at least q occurrences. We call such patterns *motifs* and note that both k and q are input parameters. More precisely, we show how to extract *maximal motifs*, meaning that we do not report motifs that can be made more precise (by extending them or replacing a don't care by an alphabet symbol) without losing occurrences. We extract all maximal motifs in $\mathcal{O}(nk + k^3\text{occ})$ time and space, where occ is the their total number of occurrences.

In comparison, existing approaches require polynomial time in n to report each pattern. That is, we give the first truly *output-sensitive* bounds for any approximate variation of the pattern extraction problem (we note that k is typically a small constant). This is in the data structure sense of output-sensitivity where the time to output a single occurrence cannot depend on the size of the input. Our algorithm is relatively simple to implement, but the analysis is complex.

We suggest an index called a *motif trie* that compactly stores the set of all maximal motifs $\mathcal{M}$. Each (maximal) motif can be represented as $\mathcal{O}(k)$ characters from a motif alphabet. The motif trie uses this property to store $\mathcal{M}$ in a generalized suffix tree, representing motifs using the motif alphabet. Note that it is easy to extract $\mathcal{M}$ from the motif trie and the difficulty is how to construct it efficiently. We do so level-wise, using an oracle that reveals the children of a node efficiently using properties of the motif alphabet and one-dimensional intervals. To determine the time spent by our algorithm, we use a novel analysis that amortizes the construction cost for the entire trie over the number of occurrences it stores.

A Curious Result This result started not as a theoretical piece of work, but as an implementation. Roberto Trani was a student of Roberto Grossi who created a high-performing pattern extraction implementation. The approach was relatively simple, but analyzing why it had such good performance turned out to be complicated. The presented result is a theoretical algorithm that uses some of the originally implemented ideas and extend them to be able to prove worst-case time bounds. Interestingly, the history of the result indicates that (a variant of) the solution performs well in practice, though the presented version has not been implemented.

Future Work The motif trie is a natural construction, and our construction algorithm may be used to create other similar indices. Interestingly, the presented algorithm is essentially a slow way to generate a traditional suffix tree. A natural question is if a similar construction and analysis can help understand the *frequent itemset problem*, which

is to extract frequently occurring subsets of items from a collection of sets of items. Here algorithms also perform bad in the worst case, but often have reasonable performance in practice.

2.3 Points

Much real world data such as locations of landmarks, visual data, or visitors to websites contains natural spatial information. Though it has no immediate spatial information, even more data can be represented spatially. For example, each row in a relational database may be thought of as a multidimensional point (with one dimension for each column). Consequently, geometric data structures is an important and well-studied research topic.

One of the core problems is *orthogonal range searching*, where we must store a set of d-dimensional points P, possibly with support for point insertions or deletions. There are two standard types of queries that takes as input a d-dimensional rectangle R. A *reporting* query must return a list of points from P contained in R, and the answer to a *counting* query is the number of points from P contained in R. Reporting query times have a necessary occ term, which is the time spent listing the points in the output. We note that the range searching problems are relatively well-understood for two-dimensional points, but there are many open questions for the high-dimensional variants.

Three included papers consider variations of range searching for points in at least two dimensions. First, we consider *compressed range searching*, which is to store the points in a compressed form while supporting standard range searching queries. Second, we study *colored range searching* where the points each have a single color and a query answer must list or count the distinct colors inside a query rectangle. Finally, we give experimental results on supporting *threshold range counting*, where each point has a weight and we must count the points exceeding some weight threshold inside a query rectangle.

2.3.1 Compressed Point Sets

The orthogonal range searching problem has been studied extensively over the last 40 years, with a variety of classic solutions [Ben75; Ben79; Ore82; Ben$^+$80; Lue78; Lee$^+$80; Gut84; Cla83; Kan$^+$99; Kre$^+$91; Gae$^+$98; Bay$^+$72; Arg$^+$08; Rob81; Pro$^+$03; Com79; Epp$^+$08] (Samet presents an overview in [Sam90]). They typically store the points in a tree, using a strategy for dividing the d-dimensional universe into smaller areas and possibly storing multiple copies of the same point to answer queries efficiently.

The point set may contain geometric repetitions, that is, copies of point subsets that are identical to each other but where each copy is offset by some translation vector. Range searching on such points sets arise naturally in several scenarios such as data and image analysis [Tet$^+$01; Paj$^+$00; Dic$^+$09], GIS applications [Sch$^+$08; Zhu$^+$02; Hae$^+$10; Dic$^+$09], and when compactly representing sparse matrices and web graphs [Gal$^+$98; Bri$^+$09; Gar$^+$14; Ber$^+$13].

Our Contribution: *Compressed Data Structures for Range Searching* We show how to exploit geometric repetitions to compress almost all classic data structures based on trees. We show that any query on the original data structure can be answered by simulation on the compressed representation with only constant overhead.

We give a tree model that capture almost all classic range searching data structures with constant overhead. We then show how to represent this model efficiently, compressing it using a minimal DAG representation. As the classic data structures are not designed to be compressed, they may be un-compressible even though the underlying point set is. To compensate for this, we also give a new hierarchical clustering algorithm ensuring that geometric repetitions are indeed compressed (under certain constraints). The result is a new compressed data structure.

Future Work Our clustering algorithm requires $\mathcal{O}(N^2 \lg D)$ time to cluster N points with a maximum distance of D between points. This is prohibitive for large point sets, and improvements would be interesting. Furthermore, the clustering algorithm may be modified to allow good clustering of more point distributions, such as subsets of points that are identical under rotation or scaling. Our general tree model supports storing such repetitions (if they can be described by a constant number of words per edge). An open question is how to efficiently find such repetitions and construct a corresponding tree.

2.3.2 Points with Colors

We consider *colored range searching* as a natural variation of range searching where each point has a single color. Colors are picked from some color alphabet, and two points can have the same color. The problem is sometimes called generalized or categorical range searching, as we can think of the colors as categories assigned to the points. Queries ask for information about the distinct colors of points contained in a query range R (instead of the actual points). That is, an answer to a reporting query is the distinct colors in R, and a counting query must return the number of distinct colors in R.

Known solutions to colored range searching have much worse query performance than their non-colored counterparts. Intuitively, this is because points in the query range may have non-unique colors, meaning that additional filtering is required to output distinct color information. Observe that we can always obtain linear space and time by simply storing the points in a list and scanning through it to find distinct colors of points in the range.

Curiously, colored counting is considered harder than colored reporting as the former problem is not decomposable: an answer cannot be determined from the number of colors in two halves of the query range. This is opposed to reporting, where an answer can be obtained by querying smaller parts of the range, merging the results and removing duplicates. For the standard problems both types of queries are decomposable, and solutions to counting are generally most efficient. Kaplan, Rubin, Sharir and Verbin [Kap$^+$07] give a hardness reduction which shows that a solution to two dimensional colored range counting in polylogarithmic time per query and $\mathcal{O}(n \operatorname{polylg} n)$ space would improve the best known matrix multiplication algorithm. This would be a major breakthrough, and

the reduction suggest that even in two dimensions, no polylogarithmic time and almost-linear space solution for counting may exist.

Our Contribution: *Colored Range Searching in Linear Space* Our goal is to give data structures that support both types of queries in little space. We give the first linear space data structure for two-dimensional points that answers queries in sublinear time. We also give a high-dimensional structure with almost-linear space of $\mathcal{O}(n \lg\lg^d n)$ and the first dynamic solutions for any dimensionality. This can be compared to previous results that have large space use of $\Omega(n^d)$ for counting and $\Omega(n \lg^{d-1} n)$ for reporting to get polylogarithmic query bounds.

Our result is obtained by partitioning points into groups with logarithmiccally many color. We further partition each group into buckets, each containing a polylogarithmic number of points. The points in each bucket are stored in a modified range tree data structure with color information stored in auxiliary data structures for each node. To answer a query, we solve the problem independently in each bucket and merge bucket results.

Future Work The colored range searching problem has many interesting loose ends and possible variations. First, the hardness result in [Kap+07] indicates that it may be possible to achieve "real" large lower bounds for colored range counting. Second, if focusing on upper bounds, improved query bounds in little space or with some restriction on the number of colors would be very interesting. We believe that especially the two-dimensional case can be improved by tabulating answers for small sets of points. An open question is if there is a linear space data structure with sublinear query time for at least 3 dimensional points?

2.3.3 Points with Weights in Practice

Another natural variation of range searching arise if points each have an integer weight. We may consider *threshold queries* asking us to report or count the number of points in a query area where the weight of each reported point must exceed some threshold.

To the best of our knowledge, no previous work has been done on obtaining results for this particular variation of range searching. However, we can think of the weight as a coordinate in a new integer dimension and the queries as being unbounded in one direction in that new dimension. This allows us to use known range searching data structures to answer queries at the expense of increasing the dimensionality of the points stored. Other related work show how to count the sums of weights [Cha90], or how to report the top-k heaviest points inside a query area [Rah+11].

Our Contribution: *Indexing Motion Detection Data for Surveillance Video* We suggest a data structure for 3-dimensional points that supports *threshold range counting* queries. Our data structure is in turn used to answer *motion detection* queries on video from a statically mounted surveillance camera to find "interesting clips" that contain a

lot of motion in an area of the frame. The problem appears in a software system by Milestone Systems, and in our paper we give experimental results on an implemented prototype.

To see why threshold range counting can help answer motion detection queries, observe that each pixel in 2-dimensional grayscale video from a surveillance camera may be considered a weighted 3-dimensional point (the timestamp in the video is the third dimension). The difference between two frames in the video is a matrix with large weights in areas where many pixels changed value by a lot, meaning that something significant happened. By storing such pixel differences a large answer to threshold range counting implies motion in the video.

Our solution reduces the resolution of allowed queries. It stores histograms that summarize information from the video and allow us to answer queries efficiently. The histograms are stored in compressed form on disk to reduce the space use.

The previous solution implemented by Milestone Systems answered queries by decompressing the entire relevant portion of the compressed video file and computing the answer from the decompressed frames. We implemented a prototype of our solution in Python and performed a number of experiments with realistic data sets. The index is space-efficient in practice (using around 10% of space required by the video file), quick to construct and answers queries efficiently (speedup of at least a factor 30).

Real World Implications A modified version of the suggested solution is in use by Milestone Systems. Their implementation is a restricted and optimized version of the prototype, implementing automatic noise detection to filter out image noise. This allows them to reduce the query resolution further. The end result is a performance improvement of a factor 100 compared to their previous solution, for a space requirement of about 1KB per second of video. The new data structure has enabled them to use the search function as an integral part of their product, as it can now deliver results in real time. This is contrary to previously where it was so slow that it was useless in practice.

Future Work The reason we study threshold range searching in the first place is that data structures for storing moving points are quite complicated and seem to perform relatively poorly in practice. What we *actually* want is a simple kinetic data structure that supports efficient threshold queries: does such a structure exist?

2.4 Integers

An important and particularly well-studied area of data structure research is *integer data structures*. Generally, we must store a sequence of n integers, here denoted X, and support operations on the sequence[2]. In our work, we suggest new solutions to variants of two core problems called *predecessor* and *partial sums*.

[2]Sometimes we consider sets of integers, but to unify our presentation we here only discuss sequences.

2.4.1 Finger Predecessor

We must store a sequence X of n integers to support the *predecessor* query $\mathsf{pred}(q)$, returning the largest element in X smaller than q. The *successor* query $\mathsf{succ}(q)$, returning the smallest element larger than q, must also be supported. Elements in X are drawn from a universe of size N (remember that $w \geq \lg N$). A line of papers by van Emde Boas, Kaas and Zijlstra; Johnson; and Willard [Emd+76; Emd77; Wil83; Joh81] showed the existence of data structures for the problem with $\mathcal{O}(\lg\lg N) = \mathcal{O}(\lg w)$ query time and $\mathcal{O}(N)$ space use. Later, Mehlhorn and Näher [Meh+90] showed how to reduce the space use to $\mathcal{O}(n)$ by using perfect hashing. The resulting data structure additionally supports insertions and deletions in the sequence in $\mathcal{O}(\lg\lg N)$ expected time.

A finger search tree supports *finger predecessor* queries, which is a predecessor query where the data structure is given a reference to an existing element $\ell \in X$ to start the search from. Dietz and Raman [Die+94] showed that queries, insertions and deletions in such a tree can be supported in time $\mathcal{O}(\lg d)$, where d is the number of integers between the query answer and ℓ. Andersson and Thorup [And+00] subsequently improved the query time to $\mathcal{O}(\sqrt{\lg d / \lg\lg d})$.

Our Contribution: *Fingerprints in Compressed Strings* We give a new data structure for answering finger predecessor queries in bounds that depend on the absolute distance between the query integer q and the finger ℓ. For this variation, we give a solution supporting insertions and deletions with query time $\mathcal{O}(\lg\lg|\ell - q|)$, which is an improvement over the general solution when the reference element is close to the query point. Our data structure is used in the paper to reduce the query time for multiple close queries to improve the time to answer LCE queries.

Future Work A new result by Pătraşcu and Thorup [Păt+14] essentially closes the classic variant of the predecessor problem, and existing structures are optimal in almost all cases. It is an open problem to construct optimal succinct solutions that use space proportional to the information theoretic lower bound.

2.4.2 Dynamic Partial Sums

In the *partial sums* problem, we must store a sequence X of n integers to support three operations: *update* the value of an integer at position i by adding some value Δ to it, find the *sum* of the first i elements, and *search* for the smallest sum larger than some integer q. It is natural to consider the search operation as a successor query among the prefix sums of X. The problem is extremely well-studied, with many papers studying variations and showing improved upper and lower bounds [Die89; Ram+01; Hus+03; Fre+89; Hon+11; Hus+96; Fen94; Păt+04; Cha+10]. In 2004, Pătraşcu and Demaine [Păt+04] gave a linear space solution in the w-bit Word RAM model with optimal query time $\mathcal{O}(\lg n / \lg(w/\delta))$ per operation and showed a matching lower bound. They restrict the possibly negative Δ argument to the update operation to be at most δ bits, so $|\Delta| \leq 2^{\delta}$, and show that $\delta = o(\lg^{\epsilon} n)$ for a solution to be faster than $\mathcal{O}(\lg n)$ time per operation.

Dynamic partial sums is a generalization of partial sums where we must additionally support insertions and deletions of integers, changing the length of X. However, almost no solutions include support for such modifications, though individual integers can be updated. The only known solutions to dynamic partial sums are by Hon, Sadakane and Sung [Hon+11] and Navarro and Sadakane [Nav+14]. Their solutions are succinct and use $\mathcal{O}(\lg n / \lg \lg n)$ time per operation, but only work for sequences of small integers of at most $\lg w$ bits. This is exponentially smaller than the w-bit integers supported in the static data structure by Pătraşcu and Demaine [Păt+04].

Our Contribution: ***Dynamic Relative Compression and Dynamic Partial Sums*** We give a new linear space data structure for the dynamic partial sums problem that additionally supports splitting and merging neighboring integers in sequence X. Our data structure is the first to support full w-bit integers, and it supports all operations in optimal worst-case time $\mathcal{O}(\lg n / \lg(w/\delta))$, matching the lower bound for non-dynamic partial sums of $\Omega(\lg n / \lg(w/\delta))$ time per operation by Pătraşcu and Demaine [Păt+04].

Since insert and delete can be used to implement update, the lower bound implies that the new operations are supported in optimal time, and the δ update restriction implies that only integers smaller than 2^δ can be inserted or deleted. Any dynamic partial sums solution solves list representation (support access, insert and delete on a list), and we match the lower bound of $\Omega(\lg n / \lg \lg n)$ due to Fredman and Saks [Fre+89] for elements of $\delta \leq \lg^\varepsilon n$ bits where $\varepsilon < 1$. This is optimal when also supporting sum.

The main difficulty in supporting modifications is that indices may change, which causes elements in the sequence to shift left or right. Our data structure builds and improves on the techniques used by Pătraşcu and Demaine [Păt+04] in their non-dynamic optimal-time data structure. They precompute sums and store them in an array coupled with a small data structure for updates. This supports all non-dynamic operations in constant time for small sequences of polylogarithmic length, and a standard reduction is used to store the full sequence in a B-tree. In our construction we first simplify their approach, only storing some representative elements in a dynamic predecessor data structure by Pătraşcu and Thorup [Păt+14]. All other elements are stored in a small update data structure. We then show how to modify this representation to efficiently support insertions and deletions, changing the way we store representatives and supporting modifications on the update data structure.

Future Work Our new data structure almost closes the dynamic partial sums problem, matching amortized lower bounds for all operations. Two open problems remain. First, as the lower bounds are amortized, some operations may be implementable in less time: for example, Chan and Pătraşcu [Cha+10] give a linear space solution to the non-dynamic variant with a faster increment/decrement update operation and optimal query time for sum. Second, our data structure uses linear space in words. Though succinct solutions exist for the partial sums problem, previous solutions to dynamic partial sums only support very small integers, so a natural question is if our data structure can be made succinct.

Part II

Strings

3 Fingerprints in Compressed Strings

Philip Bille[1] Patrick Hagge Cording[1] Inge Li Gørtz[1*]
Benjamin Sach[2] Hjalte Wedel Vildhøj[1] Søren Vind[1†]

[1] Technical University of Denmark
[2] University of Warwick

Abstract

The Karp-Rabin fingerprint of a string is a type of hash value that due to its strong properties has been used in many string algorithms. In this paper we show how to construct a data structure for a string S of size N compressed by a context-free grammar of size n that answers fingerprint queries. That is, given indices i and j, the answer to a query is the fingerprint of the substring $S[i, j]$.

We present the first $O(n)$ space data structures that answer fingerprint queries without decompressing any characters. For Straight Line Programs (SLP) we get $O(\lg N)$ query time, and for Linear SLPs (an SLP derivative that captures LZ78 compression and its variations) we get $O(\lg \lg N)$ query time. Hence, our data structures has the same time and space complexity as for random access in SLPs. We utilize the fingerprint data structures to solve the longest common extension problem in query time $O(\lg N \lg \ell)$ and $O(\lg \ell \lg \lg \ell + \lg \lg N)$ for SLPs and Linear SLPs, respectively. Here, ℓ denotes the length of the LCE.

*Supported by a grant from the Danish Council for Independent Research | Natural Sciences.
†Supported by a grant from the Danish National Advanced Technology Foundation.

3.1 Introduction

Given a string S of size N and a Karp-Rabin fingerprint function ϕ, the answer to a FINGERPRINT(i,j) query is the fingerprint $\phi(S[i,j])$ of the substring $S[i,j]$. We consider the problem of constructing a data structure that efficiently answers fingerprint queries when the string is compressed by a context-free grammar of size n.

The fingerprint of a string is an alternative representation that is much shorter than the string itself. By choosing the fingerprint function randomly at runtime it exhibits strong guarantees for the probability of two different strings having different fingerprints. Fingerprints were introduced by Karp and Rabin [Kar$^+$87] and used to design a randomized string matching algorithm. Since then, they have been used as a central tool to design algorithms for a wide range of problems (see e.g., [Ami$^+$92; And$^+$06; Col$^+$03; Cor$^+$05; Cor$^+$07; Far$^+$98; Gas$^+$96; Kal02; Por$^+$09]).

A fingerprint requires constant space and it has the useful property that given the fingerprints $\phi(S[1,i-1])$ and $\phi(S[1,j])$, the fingerprint $\phi(S[i,j])$ can be computed in constant time. By storing the fingerprints $\phi(S[1,i])$ for $i = 1 \ldots N$ a query can be answered in $O(1)$ time. However, this data structure uses $O(N)$ space which can be exponential in n. Another approach is to use the data structure of Gąsieniec et al. [Gas$^+$05] which supports linear time decompression of a prefix or suffix of the string generated by a node. To answer a query we find the deepest node that generates a string containing $S[i]$ and $S[j]$ and decompress the appropriate suffix of its left child and prefix of its right child. Consequently, the space usage is $O(n)$ and the query time is $O(h+j-i)$, where h is the height of the grammar. The $O(h)$ time to find the correct node can be improved to $O(\lg N)$ using the data structure by Bille et al. [Bil$^+$11] giving $O(\lg N + j - i)$ time for a FINGERPRINT(i,j) query. Note that the query time depends on the length of the decompressed string which can be large. For the case of *balanced* grammars (by height or weight) Gagie etal. [Gag$^+$12a] showed how to efficiently compute fingerprints for indexing Lempel-Ziv compressed strings.

We present the first data structures that answer fingerprint queries on general grammar compressed strings without decompressing any characters, and improve all of the above time-space trade-offs. Assume without loss of generality that the compressed string is given as a Straight Line Program (SLP). An SLP is a grammar in Chomsky normal form, i.e., each nonterminal has exactly two children. A Linear SLP is an SLP where the root is allowed to have more than two children, and for all other internal nodes, the right child must be a leaf. Linear SLPs capture the LZ78 compression scheme [Ziv$^+$78] and its variations. Our data structures give the following theorem.

Theorem 3.1.1. *Let S be a string of length N compressed into an SLP G of size n. We can construct data structures that support* FINGERPRINT *queries in:*

(i) $O(n)$ *space and query time* $O(\lg N)$

(ii) $O(n)$ *space and query time* $O(\lg \lg N)$ *if G is a Linear SLP*

Hence, we show a data structure for fingerprint queries that has the same time and space complexity as for random access in SLPs.

Our fingerprint data structures are based on the idea that a random access query for i produces a path from the root to a leaf labelled $S[i]$. The concatenation of the substrings produced by the left children of the nodes on this path produce the prefix $S[1,i]$. We store the fingerprints of the strings produced by each node and concatenate these to get the fingerprint of the prefix instead. For Theorem 3.1.1(i), we combine this with the fast random access data structure by Bille et al. [Bil$^+$11]. For Linear SLPs we use the fact that the production rules form a tree to do large jumps in the SLP in constant time using a level ancestor data structure. Then a random access query is dominated by finding the node that produces $S[i]$ among the children of the root, which can be modelled as the predecessor problem.

Furthermore, we show how to obtain faster query time in Linear SLPs using finger searching techniques. Specifically, a finger for position i in a Linear SLP is a pointer to the child of the root that produces $S[i]$.

Theorem 3.1.2. *Let S be a string of length N compressed into an SLP G of size n. We can construct an $O(n)$ space data structure such that given a finger f for position i or j, we can answer a* FINGERPRINT(i,j) *query in time $O(\lg\lg D)$ where $D = |i-j|$.*

Along the way we give a new and simple reduction for solving the finger predecessor problem on integers using any predecessor data structure as a black box.

In compliance with all related work on grammar compressed strings, we assume that the model of computation is the RAM model with a word size of $\lg N$ bits.

Longest Common Extension in Compressed Strings

As an application we show how to efficiently solve the longest common extension problem (LCE). Given two indices i,j in a string S, the answer to the LCE(i,j) query is the length ℓ of the maximum substring such that $S[i,i+\ell] = S[j,j+\ell]$. The compressed LCE problem is to preprocess a compressed string to support LCE queries. On uncompressed strings this is solvable in $O(N)$ preprocessing time, $O(N)$ space, and $O(1)$ query time with a nearest common ancestor data structure on the suffix tree for S [Har$^+$84]. Other trade-offs are obtained by doing an exponential search over the fingerprints of strings starting in i and j [Bil$^+$12a]. Using the exponential search in combination with the previously mentioned methods for obtaining fingerprints without decompressing the entire string we get $O((h+\ell)\lg\ell)$ or $O((\lg N+\ell)\lg\ell)$ time using $O(n)$ space for an LCE query. Using our new (finger) fingerprint data structures and the exponential search we obtain Theorem 3.1.3.

Theorem 3.1.3. *Let G be an SLP of size n that produces a string S of length N. The SLP G can be preprocessed in $O(N)$ time into a Monte Carlo data structure of size $O(n)$ that supports LCE queries on S in*

(i) $O(\lg\ell\lg N)$ *time*

(ii) $O(\lg\ell\lg\lg\ell+\lg\lg N)$ *time if G is a Linear SLP.*

Here ℓ denotes the LCE value and queries are answered correctly with high probability. Moreover, a Las Vegas version of both data structures that always answers queries correctly can be obtained with $O(N^2/n \lg N)$ preprocessing time with high probability.

We furthermore show how to reduce the Las Vegas preprocessing time to $O(N \lg N \lg\lg N)$ when all the internal nodes in the Linear SLP are children of the root (which is the case in LZ78). The following corollary follows immediately because an LZ77 compression [Ziv+77] consisting of n phrases can be transformed to an SLP with $O(n \lg \frac{N}{n})$ production rules [Cha+05; Ryt03].

Corollary 3.1.3.1. *We can solve the* LCE *problem in $O(n \lg \frac{N}{n})$ space and $O(\lg \ell \lg N)$ query time for LZ77 compression.*

Finally, the LZ78 compression can be modelled by a Linear SLP G_L with constant overhead. Consider an LZ78 compression with n phrases, denoted $r_1, \ldots, r_n$. A terminal phrase corresponds to a leaf in G_L, and each phrase $r_j = (r_i, a)$, $i < j$, corresponds to a node $v \in G_L$ with r_i corresponding to the left child of v and the right child of v being the leaf corresponding to a. Therefore, we get the following corollary.

Corollary 3.1.3.2. *We can solve the* LCE *problem in $O(n)$ space and $O(\lg \ell \lg\lg \ell + \lg\lg N)$ query time for LZ78 compression.*

3.2 Preliminaries

Let $S = S[1, |S|]$ be a string of length $|S|$. Denote by $S[i]$ the character in S at index i and let $S[i, j]$ be the substring of S of length $j - i + 1$ from index $i \geq 1$ to $|S| \geq j \geq i$, both indices included.

A Straight Line Program (SLP) G is a context-free grammar in Chomsky normal form that we represent as a node-labeled and ordered directed acyclic graph. Each leaf in G is labelled with a character, and corresponds to a terminal grammar production rule. Each internal node in G is labeled with a nonterminal rule from the grammar. The unique string $S(v)$ of length $size(v) = |S(v)|$ is *produced* by a depth-first left-to-right traversal of $v \in G$ and consist of the characters on the leafs in the order they are visited. We let $root(G)$ denote the root of G, and $left(v)$ and $right(v)$ denote the left and right child of an internal node $v \in G$, respectively.

A Linear SLP G_L is an SLP where we allow $root(G_L)$ to have more than two children. All other internal nodes $v \in G_L$ have a leaf as $right(v)$. Although similar, this is not the same definition as given for the Relaxed SLP by Claude and Navarro [Cla+11]. The Linear SLP is more restricted since the right child of any node (except the root) must be a leaf. Any Linear SLP can be transformed into an SLP of at most double size by adding a new rule for each child of the root.

We extend the classic *heavy path decomposition* of Harel and Tarjan [Har+84] to SLPs as in [Bil+11]. For each node $v \in G$, we select one edge from v to a child with maximum size and call it the *heavy edge*. The remaining edges are *light edges*. Observe that $size(u) \leq size(v)/2$ if v is a parent of u and the edge connecting them is light. Thus,

the number of light edges on any path from the root to a leaf is at most $O(\lg N)$. A *heavy path* is a path where all edges are heavy. The heavy path of a node v, denoted $H(v)$, is the unique path of heavy edges starting at v. Since all nodes only have a single outgoing heavy edge, the heavy path $H(v)$ and its leaf $leaf(H(v))$, is well-defined for each node $v \in G$.

A *predecessor data structure* supports predecessor and successor queries on a set $R \subseteq U = \{0, \ldots, N-1\}$ of n integers from a universe U of size N. The answer to a *predecessor query* $\mathsf{pred}(q)$ is the largest integer $r^- \in R$ such that $r^- \leq q$, while the answer to a *successor query* $\mathsf{succ}(q)$ is the smallest integer $r^+ \in R$ such that $r^+ \geq q$. There exist predecessor data structures achieving a query time of $O(\lg \lg N)$ using space $O(n)$ [Emd$^+$76; Meh$^+$90; Wil83].

Given a rooted tree T with n vertices, we let $depth(v)$ denote the length of the path from the root of T to a node $v \in T$. A *level ancestor data structure* on T supports *level ancestor queries* $\mathrm{LA}(v, i)$, asking for the ancestor u of $v \in T$ such that $depth(u) = depth(v) - i$. There is a level ancestor data structure answering queries in $O(1)$ time using $O(n)$ space [Die91] (see also [Ber$^+$94; Als$^+$00; Ben$^+$04]).

Fingerprinting

The Karp-Rabin fingerprint [Kar$^+$87] of a string x is defined as $\phi(x) = \sum_{i=1}^{|x|} x[i] \cdot c^i \bmod p$, where c is a randomly chosen positive integer, and $2N^{c+4} \leq p \leq 4N^{c+4}$ is a prime. Karp-Rabin fingerprints guarantee that given two strings x and y, if $x = y$ then $\phi(x) = \phi(y)$. Furthermore, if $x \neq y$, then with high probability $\phi(x) \neq \phi(y)$. Fingerprints can be composed and subtracted as follows.

Lemma 3.2.1. *Let $x = yz$ be a string decomposable into a prefix y and suffix z. Let N be the maximum length of x, c be a random integer and $2N^{c+4} \leq p \leq 4N^{c+4}$ be a prime. Given any two of the Karp-Rabin fingerprints $\phi(x)$, $\phi(y)$ and $\phi(z)$, it is possible to calculate the remaining fingerprint in constant time as follows:*

$$\phi(x) = \phi(y) \oplus \phi(z) = \phi(y) + c^{|y|} \cdot \phi(z) \bmod p$$
$$\phi(y) = \phi(x) \ominus_s \phi(z) = \phi(x) - \frac{c^{|x|}}{c^{|z|}} \cdot \phi(z) \bmod p$$
$$\phi(z) = \phi(x) \ominus_p \phi(y) = \frac{\phi(x) - \phi(y)}{c^{|y|}} \bmod p$$

In order to calculate the fingerprints of Theorem 3.2.1 in constant time, each fingerprint for a string x must also store the associated exponent $c^{|x|} \bmod p$, and we will assume this is always the case. Observe that a fingerprint for any substring $\phi(S[i, j])$ of a string can be calculated by subtracting the two fingerprints for the prefixes $\phi(S[1, i-1])$ and $\phi(S[1, j])$. Hence, we will only show how to find fingerprints for prefixes in this paper.

3.3 Basic Fingerprint Queries in SLPs

We now describe a simple data structure for answering FINGERPRINT$(1, i)$ queries for a string S compressed into a SLP G in time $O(h)$, where h is the height of the parse tree for S. This method does not unpack the string to obtain the fingerprint, instead the fingerprint is generated by traversing G.

The data structure stores $size(v)$ and the fingerprint $\phi(S(v))$ of the string produced by each node $v \in G$. To compose the fingerprint $f = \phi(S[1, i])$ we start from the root of G and do the following. Let v' denote the currently visited node, and let $p = 0$ be a variable denoting the size the concatenation of strings produced by left children of visited nodes. We follow an edge to the right child of v' if $p + size(left(v')) < i$, and follow a left edge otherwise. If following a right edge, update $f = f \oplus \phi(S(left(v')))$ such that the fingerprint of the full string generated by the left child of v' is added to f, and set $p = p + size(left(v'))$. When following a left edge, f and p remains unchanged. When a leaf is reached, let $f = f \oplus \phi(S(v'))$ to include the fingerprint of the terminal character. Aside from the concatenation of fingerprints for substrings, this procedure resembles a random access query for the character in position i of S.

The procedure correctly composes $f = \phi(S[1, i])$ because the order in which the fingerprints for the substrings are added to f is identical to the order in which the substrings are decompressed when decompressing $S[1, i]$.

Since the fingerprint composition takes constant time per addition, the time spent generating a fingerprint using this method is bounded by the height of the parse tree for $S[i]$, denoted $O(h)$. Only constant additional space is spent for each node in G, so the space usage is $O(n)$.

3.4 Faster Fingerprints in SLPs

Using the data structure of Bille et al. [Bil+11] to perform random access queries allows for a faster way to answer FINGERPRINT$(1, i)$ queries.

Lemma 3.4.1 ([Bil+11])**.** *Let S be a string of length N compressed into a SLP G of size n. Given a node $v \in G$, we can support random access in $S(v)$ in $O(\lg(size(v)))$ time, at the same time reporting the sequence of heavy paths and their entry- and exit points in the corresponding depth-first traversal of $G(v)$.*

The main idea is to compose the final fingerprint from substring fingerprints by performing a constant number of fingerprint additions per heavy path visited.

In order to describe the data structure, we will use the following notation. Let $V(v)$ be the left children of the nodes in $H(v)$ where the heavy path was extended to the right child, ordered by increasing depth. The order of nodes in $V(v)$ is equal to the sequence in which they occur when decompressing $S(v)$, so the concatenation of the strings produced by nodes in $V(v)$ yields the prefix $P(v) = S(v)[1, L(v)]$, where $L(v) = \sum_{u \in V(v)} size(u)$. Observe that $P(u)$ is a suffix of $P(v)$ if $u \in H(v)$. See Figure 3.1 for the relationship between u, v and the defined strings.

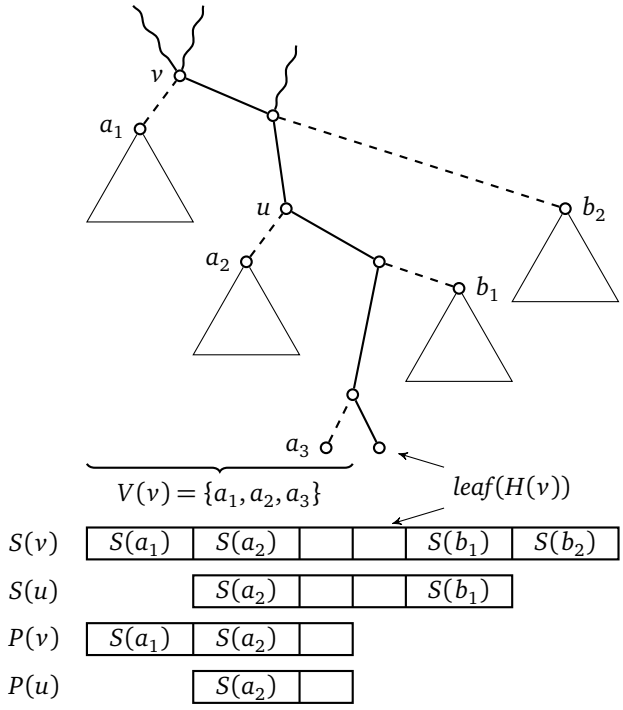
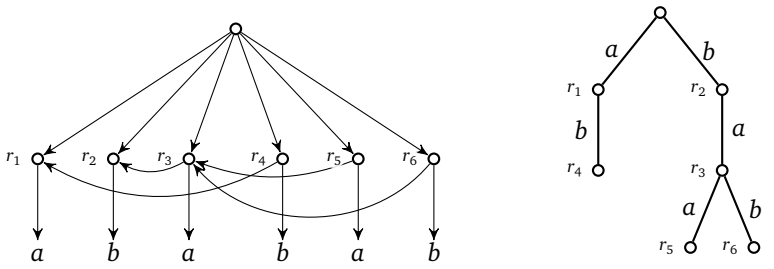


Figure 3.1: Figure showing how $S(v)$ and its prefix $P(v)$ is composed of substrings generated by the left children a_1, a_2, a_3 and right children b_1, b_2 of the heavy path $H(v)$. Also illustrates how this relates to $S(u)$ and $P(u)$ for a node $u \in H(v)$.

Let each node $v \in G$ store its unique outgoing heavy path $H(v)$, the length $L(v)$, $size(v)$, and the fingerprints $\phi(P(v))$ and $\phi(S(v))$. By forming heavy path trees of total size $O(n)$ as in [Bil+11], we can store $H(v)$ as a pointer to a node in a heavy path tree (instead of each node storing the full sequence).

The fingerprint $f = \phi(S[1,i])$ is composed from the sequence of heavy paths visited when performing a single random access query for $S[i]$ using Lemma 3.4.1. Instead of adding all left-children of the path towards $S[i]$ to f individually, we show how to add all left-children hanging from each visited heavy path in constant time per heavy path. Thus, the time taken to compose f is $O(\lg N)$.

More precisely, for the pair of entry- and exit-nodes v, u on each heavy path H traversed from the root to $S[i]$, we set $f = f \oplus (\phi(P(v)) \ominus_s \phi(P(u))$ (which is allowed because $P(u)$ is a suffix of $P(v)$). If we leave u by following a right-pointer, we additionally set $f = f \oplus \phi(S(left(u)))$. If u is a leaf, set $f = f \oplus \phi(S(u))$ to include the fingerprint of the terminal character.



(a) Linear SLP. (b) Dictionary tree.

Figure 3.2: A Linear SLP compressing the string `abbaabbaabab` and the dictionary tree obtained from the Linear SLP.

Remember that $P(v)$ is exactly the string generated from v along H, produced by the left children of nodes on H where the heavy path was extended to the right child. Thus, this method corresponds exactly to adding the fingerprint for the substrings generated by all left children of nodes on H between the entry- and exit-nodes in depth-first order, and the argument for correctness from the slower fingerprint generation also applies here.

Since the fingerprint composition takes constant time per addition, the time spent generating a fingerprint using this method is bounded by the number of heavy paths traversed, which is $O(\lg N)$. Only constant additional space is spent for each node in G, so the space usage is $O(n)$. This concludes the proof of Theorem 3.1.1(i).

3.5 Faster Fingerprints in Linear SLPs

In this section we show how to quickly answer FINGERPRINT$(1,i)$ queries on a Linear SLP G_L. In the following we denote the sequence of k children of $root(G_L)$ from left to right by $r_1,\ldots,r_k$. Also, let $R(j)=\sum_{m=1}^{j} size(r_m)$ for $j=0,\ldots,k$. That is, $R(j)$ is the length of the prefix of S produced by G_L including r_j (and $R(0)$ is the empty prefix).

We also define the dictionary tree F over G_L as follows. Each node $v \in G_L$ corresponds to a single vertex $v^F \in F$. There is an edge (u^F, v^F) labeled c if $u = left(v)$ and $c = S(right(v))$. If v is a leaf, there is an edge $(root(F), v^F)$ labeled $S(v)$. That is, a left child edge of $v \in G_L$ is converted to a parent edge of $v^F \in F$ labeled like the right child leaf of v. Note that for any node $v \in G_L$ except the root, producing $S(v)$ is equivalent to following edges and reporting edge labels on the path from $root(F)$ to v^F. Thus, the prefix of length a of $S(v)$ may be produced by reporting the edge labels on the path from $root(F)$ until reaching the ancestor of v^F at depth a.

The data structure stores a predecessor data structure over the prefix lengths $R(j)$ and the associated node r_j and fingerprint $\phi(S[1,R(j)])$ for $j=0,\ldots,k$. We also have a doubly linked list of all r_j's with bidirectional pointers to the predecessor data structure and G_L. We store the dictionary tree F over G_L, augment it with a level ancestor data

structure, and add bidirectional pointers between $v \in G_L$ and $v^F \in F$. Finally, for each node $v \in G_L$, we store the fingerprint of the string it produces, $\phi(S(v))$.

A query FINGERPRINT$(1,i)$ is answered as follows. Let $R(m)$ be the predecessor of i among $R(0), R(1), \ldots, R(k)$. Compose the answer to FINGERPRINT$(1,i)$ from the two fingerprints $\phi(S[1,R(m)]) \oplus \phi(S[R(m)+1,i])$. The first fingerprint $\phi(S[1,R(m)])$ is stored in the data structure and the second fingerprint $\phi(S[R(m)+1,i])$ can be found as follows. Observe that $S[R(m)+1,i]$ is fully generated by r_{m+1} and hence a prefix of $S(r_{m+1})$ of length $i-R(m)$. We can get r_{m+1} in constant time from r_m using the doubly linked list. We use a level ancestor query $u^F = \text{LA}(r^F_{m+1}, i-R(m))$ to determine the ancestor of r^F_{m+1} at depth $i-R(m)$, corresponding to a prefix of r_{m+1} of the correct length. From u_F we can find $\phi(S(u)) = \phi(S[R(m)+1,i])$.

It takes constant time to find $\phi(S[R(m)+1,i])$ using a single level ancestor query and following pointers. Thus, the time to answer a query is bounded by the time spent determining $\phi(S[1,R(m)])$, which requires a predecessor query among k elements (i.e. the number of children of $root(G_L)$) from a universe of size N. The data structure uses $O(n)$ space, as there is a bijection between nodes in G_L and vertices in F, and we only spend constant additional space per node in G_L and vertex in F. This concludes the proof of Theorem 3.1.1(ii).

3.6 Finger Fingerprints in Linear SLPs

The $O(\lg\lg N)$ running time of a FINGERPRINT$(1,i)$ query is dominated by having to find the predecessor $R(m)$ of i among $R(0), R(1), \ldots, R(k)$. Given $R(m)$ the rest of the query takes constant time. In the following, we show how to improve the running time of a FINGERPRINT$(1,i)$ query to $O(\lg\lg|j-i|)$ given a finger for position j. Recall that a finger f for a position j is a pointer to the node r_m producing $S[j]$. To achieve this, we present a simple linear space finger predecessor data structure that is interchangeable with any other predecessor data structure.

3.6.1 Finger Predecessor

Let $R \subseteq U = \{0, \ldots, N-1\}$ be a set of n integers from a universe U of size N. Given a finger $f \in R$ and a query point $q \in U$, the *finger predecessor problem* is to answer finger predecessor or successor queries in time depending on the universe distance $D = |f-q|$ from the finger to the query point. Belazzougui et al. [Bel+12] present a succinct solution for solving the finger predecessor problem relying on a modification of z-fast tries. Other previous work present dynamic finger search trees on the word RAM [Kap+13; And+07]. Here, we use a simple reduction for solving the finger predecessor problem using any predecessor data structure as a black box.

Lemma 3.6.1. *Let $R \subseteq U = \{0, \ldots, N-1\}$ be a set of n integers from a universe U of size N. Given a predecessor data structure with query time $t(N,n)$ using $s(N,n)$ space, we can solve the finger predecessor problem in worst case time $O(t(D,n))$ using space $O(s(N, \frac{n}{\lg N})\lg N)$.*

Proof. Construct a complete balanced binary search tree T over the universe U. The leaves of T represent the integers in U, and we say that a vertex *span* the range of U represented by the leaves in its subtree. Mark the leaves of T representing the integers in R. We remove all vertices in T where the subtree contains no marked vertices. Observe that a vertex at height j span a universe range of size $O(2^j)$. We augment T with a level ancestor data structure answering queries in constant time. Finally, left- and right-neighbour pointers are added for all nodes in T.

Each internal node $v \in T$ at height j store an instance of the given predecessor data structure for the set of marked leaves in the subtree of v. The size of the universe for the predecessor data structure equals the span of the vertex and is $O(2^j)$[1].

Given a finger $f \in R$ and a query point $q \in U$, we will now describe how to find both $\mathsf{succ}(q)$ and $\mathsf{pred}(q)$ when $q < f$. The case $q > f$ is symmetric. Observe that f corresponds to a leaf in T, denoted f_l. We answer a query by determining the ancestor v of f_l at height $h = \lceil \lg(|f - q|) \rceil$ and its left neighbour v_L (if it exists). We query for $\mathsf{succ}(q)$ in the predecessor data structures of both v and v_L, finding at least one leaf in T (since v spans f and $q < f$). We return the leaf representing the smallest result as $\mathsf{succ}(q)$ and its left neighbour in T as $\mathsf{pred}(q)$.

Observe that the predecessor data structures in v and v_L each span a universe of size $O(2^h) = O(|f - q|) = O(D)$. All other operations performed take constant time. Thus, for a predecessor data structure with query time $t(N, n)$, we can answer finger predecessor queries in time $O(t(D, n))$.

The height of T is $O(\lg N)$, and there are $O(n \lg N)$ vertices in T (since vertices spanning no elements from R are removed). Each element from R is stored in $O(\lg N)$ predecessor data structures. Hence, given a predecessor data structure with space usage $s(N, n)$, the total space usage of the data structure is $O(s(N, n) \lg N)$.

We reduce the size of the data structure by reducing the number of elements it stores to $O(\frac{n}{\lg N})$. This is done by partitioning R into $O(\frac{n}{\lg N})$ sets of consecutive elements R_i of size $O(\lg N)$. We choose the largest integer in each R_i set as the representative g_i for that set, and store that in the data structure described above. We store the integers in set R_i in an atomic heap [Fre$^+$93; Hag98] capable of answering predecessor queries in $O(1)$ time and linear space for a set of size $O(\lg N)$. Each element in R keep a pointer to the set R_i it belongs to, and each set left- and right-neighbour pointers.

Given a finger $f \in R$ and a query point $q \in U$, we describe how to determine $\mathsf{pred}(q)$ and $\mathsf{succ}(q)$ when $q < f$. The case $q > f$ is symmetric. We first determine the closest representatives g_l and g_r on the left and right of f, respectively. Assuming $q < g_l$, we proceed as before using g_l as the finger into T and query point q. This gives $p = \mathsf{pred}(q)$ and $s = \mathsf{succ}(q)$ among the representatives. If g_l is undefined or $g_l < q < f \leq g_r$, we select $p = g_l$ and $s = g_r$. To produce the final answers, we perform at most 4 queries in the atomic heaps that p and s are representatives for.

All queries in the atomic heaps take constant time, and we can find g_l and g_r in constant time by following pointers. If we query a predecessor data structure, we know

[1]The integers stored by the data structure may be shifted by some constant $k \cdot 2^j$ for a vertex at height j, but we can shift all queries by the same constant and thus the size of the universe is 2^j.

that the range it spans is $O(|g_l - q|) = O(|f - q|) = O(D)$ since $q < g_l < f$. Thus, given a predecessor data structure with query time $t(N,n)$, we can solve the finger predecessor problem in time $O(t(D,n))$.

The total space spent on the atomic heaps is $O(n)$ since they partition R. The number of representatives is $O(\frac{n}{\lg N})$. Thus, given a predecessor data structure with space usage $s(N,n)$, we can solve the finger predecessor problem in space $O(s(N,\frac{n}{\lg N})\lg N)$. □ □

Using the van Emde Boas predecessor data structure [Emd$^+$76; Meh$^+$90; Wil83] with $t(N,n) = O(\lg\lg N)$ query time using $s(N,n) = O(n)$ space, we obtain the following corollary.

Corollary 3.6.1.1. *Let $R \subseteq U = \{0,\ldots,N-1\}$ be a set of n integers from a universe U of size N. Given a finger $f \in R$ and a query point $q \in U$, we can solve the finger predecessor problem in worst case time $O(\lg\lg|f-q|)$ and space $O(n)$.*

3.6.2 Finger Fingerprints

We can now prove Theorem 3.1.2. Assume wlog that we have a finger for i, i.e., we are given a finger f to the node r_m generating $S[i]$. From this we can in constant time get a pointer to r_{m+1} in the doubly linked list and from this a pointer to $R(m+1)$ in the predecessor data structure. If $R(m+1) > j$ then $R(m)$ is the predecessor of j. Otherwise, using Corollary 3.6.1.1 we can in time $O(\lg\lg|R(m+1)-j|)$ find the predecessor of j. Since $R(m+1) \geq i$ and the rest of the query takes constant time, the total time for the query is $O(\lg\lg|i-j|)$.

3.7 Longest Common Extensions in Compressed Strings

Given an SLP G, the longest common extension (LCE) problem is to build a data structure for G that supports longest common extension queries $LCE(i,j)$. In this section we show how to use our fingerprint data structures as a tool for doing LCE queries and hereby obtain Theorem 3.1.3.

3.7.1 Computing Longest Common Extensions with Fingerprints

We start by showing the following general lemma that establishes the connection between LCE and fingerprint queries.

Lemma 3.7.1. *For any string S and any partition $S = s_1 s_2 \cdots s_t$ of S into k non-empty substrings called phrases, $\ell = \mathrm{LCE}(i,j)$ can be found by comparing $O(\lg \ell)$ pairs of substrings of S for equality. Furthermore, all substring comparisons $x = y$ are of one of the following two types:*

Type 1 *Both x and y are fully contained in (possibly different) phrase substrings.*

Type 2 *$|x| = |y| = 2^p$ for some $p = 0,\ldots,\lg(\ell)+1$ and for x or y it holds that*

(a) The start position is also the start position of a phrase substring, or

(b) The end position is also the end position of a phrase substring.

Proof. Let a position of S be a *start* (*end*) position if a phrase starts (ends) at that position. Moreover, let a comparison of two substrings be of *type 1* (*type 2*) if it satisfies the first (second) property in the lemma. We now describe how to find $\ell = \text{LCE}(i,j)$ by using $O(\lg \ell)$ type 1 or 2 comparisons.

If i or j is not a start position, we first check if $S[i,i+k] = S[j,j+k]$ (type 1), where $k \geq 0$ is the minimum integer such that $i+k$ or $j+k$ is an end position. If the comparison fails, we have restricted the search for ℓ to two phrase substrings, and we can find the exact value using $O(\lg \ell)$ type 1 comparisons.

Otherwise, $\text{LCE}(i,j) = k + \text{LCE}(i+k+1, j+k+1)$ and either $i+k+1$ or $j+k+1$ is a start position. This leaves us with the task of describing how to answer $\text{LCE}(i,j)$, assuming that either i or j is a start position.

We first use $p = O(\lg \ell)$ type 2 comparisons to determine the biggest integer p such that $S[i,i+2^p] = S[j,j+2^p]$. It follows that $\ell \in [2^p, 2^{p+1}]$. Now let $q < 2^p$ denote the length of the longest common prefix of the substrings $x = S[i+2^p+1, i+2^{p+1}]$ and $y = S[j+2^p+1, j+2^{p+1}]$, both of length 2^p. Clearly, $\ell = 2^p + q$. By comparing the first half x' of x to the first half y' of y, we can determine if $q \in [0, 2^{p-1}]$ or $q \in [2^{p-1}+1, 2^p-1]$. By recursing we obtain the exact value of q after $\lg 2^p = O(\lg \ell)$ comparisons.

However, comparing $x' = S[a_1, b_1]$ and $y' = S[a_2, b_2]$ directly is not guaranteed to be of type 1 or 2. To fix this, we compare them indirectly using a type 1 and type 2 comparison as follows. Let $k < 2^p$ be the minimum integer such that $b_1 - k$ or $b_2 - k$ is a start position. If there is no such k then we can compare x' and y' directly as a type 1 comparison. Otherwise, it holds that $x' = y'$ if and only if $S[b_1-k, b_1] = S[b_2-k, b_2]$ (type 1) and $S[a_1-k-1, b_1-k-1] = S[a_2-k-1, b_2-k-1]$ (type 2). □ □

Theorem 3.1.3 follows by using fingerprints to perform the substring comparisons. In particular, we obtain a Monte Carlo data structure that can answer a LCE query in $O(\lg \ell \lg N)$ time for SLPs and in $O(\lg \ell \lg \lg N)$ time for Linear SLPs. In the latter case, we can use Theorem 3.1.2 to reduce the query time to $O(\lg \ell \lg \lg \ell + \lg \lg N)$ by observing that for all but the first fingerprint query, we have a finger into the data structure.

3.7.2 Verifying the Fingerprint Function

Since the data structure is Monte Carlo, there may be collisions among the fingerprints used to determine the LCE, and consequently the answer to a query may be incorrect. We now describe how to obtain a Las Vegas data structure that always answers LCE queries correctly. We do so by showing how to efficiently verify that the fingerprint function ϕ is *good*, i.e., collision-free on all substrings compared in the computation of $\text{LCE}(i,j)$. We give two verification algorithms. One that works for LCE queries in SLPs, and a faster one that works for Linear SLPs where all internal nodes are children of the root (e.g. LZ78).

3.7.2.1 SLPs

If we let the phrases of S be its individual characters, we can assume that all fingerprint comparisons are of type 2 (see Theorem 3.7.1). We thus only have to check that ϕ is collision-free among all substrings of length $2^p, p = 0, \ldots, \lg N$. We verify this in $\lg N$ rounds. In round p we maintain the fingerprint of a sliding window of length 2^p over S. For each substring x we insert $\phi(x)$ into a dictionary. If the dictionary already contains a fingerprint $\phi(y) = \phi(x)$, we verify that $x = y$ in constant time by checking if $\phi(x[1, 2^{p-1}]) = \phi(y[1, 2^{p-1}])$ and $\phi(x[2^{p-1} + 1, 2^p]) = \phi(y[2^{p-1} + 1, 2^p])$. This works because we have already verified that the fingerprinting function is collision-free for substrings of length 2^{p-1}. Note that we can assume that for any fingerprint $\phi(x)$ the fingerprints of the first and last half of x are available in constant time, since we can store and maintain these at no extra cost. In the first round $p = 0$, we check that $x = y$ by comparing the two characters explicitly. If $x \neq y$ we have found a collision and we abort and report that ϕ is not good. If all rounds are successfully verified, we report that ϕ is good.

For the analysis, observe that computing all fingerprints of length 2^p in the sliding window can be implemented by a single traversal of the SLP parse tree in $O(N)$ time. Thus, the algorithm correctly decides whether ϕ is good in $O(N \lg N)$ time and $O(N)$ space. We can easily reduce the space to $O(n)$ by carrying out each round in $O(N/n)$ iterations, where no more than n fingerprints are stored in the dictionary in each iteration. So, alternatively, ϕ can be verified in $O(N^2/n \lg N)$ time and $O(n)$ space.

3.7.2.2 Linear SLPs

In Linear SLPs where all internal nodes are children of the root, we can reduce the verification time to $O(N \lg N \lg \lg N)$, while still using $O(n)$ space. To do so, we use Theorem 3.7.1 with the partition of S being the root substrings. We verify that ϕ is collision-free for type 1 and type 2 comparisons separately.

Type 1 Comparisons. We carry out the verification in rounds. In round p we check that no collisions occur among the p-length substrings of the root substrings as follows: We traverse the SLP maintaining the fingerprint of all p-length substrings. For each substring x of length p, we insert $\phi(x)$ into a dictionary. If the dictionary already contains a fingerprint $\phi(y) = \phi(x)$ we verify that $x = y$ in constant time by checking if $x[1] = y[1]$ and $\phi(x[2, |x|]) = \phi(y[2, |y|])$ (type 1).

Every substring of a root substring ends in a leaf in the SLP and is thus a suffix of a root substring. Consequently, they can be generated by a bottom up traversal of the SLP. The substrings of length 1 are exactly the leaves. Having generated the substrings of length p, the substrings of length $p + 1$ are obtained by following the parents left child to another root node and prepending its right child. In each round the p length substrings correspond to a subset of the root nodes, so the dictionary never holds more than n fingerprints. Furthermore, since each substring is a suffix of a root substring, and

the root substrings have at most N suffixes in total, the algorithm will terminate in $O(N)$ time.

Type 2 Comparisons. We adopt an approach similar to that for SLPs and verify ϕ in $O(\lg N)$ rounds. In round p we store the fingerprints of the substrings of length 2^p that start or end at a phrase boundary in a dictionary. We then slide a window of length 2^p over S to find the substrings whose fingerprint equals one of those in the dictionary. Suppose the dictionary in round p contains the fingerprint $\phi(y)$, and we detect a substring x such that $\phi(x) = \phi(y)$. To verify that $x = y$, assume that y starts at a phrase boundary (the case when it ends in a phrase boundary is symmetric). As before, we first check that the first half of x is equal to the first half of y using fingerprints of length 2^{p-1}, which we know are collision-free. Let $x' = S[a_1, b_1]$ and $y' = S[a_2, b_2]$ be the second half of x and y. Contrary to before, we can not directly compare $\phi(x') = \phi(y')$, since neither x' nor y' is guaranteed to start or end at a phrase boundary. Instead, we compare them indirectly using a type 1 and type 2 comparison as follows: Let $k < 2^{p-1}$ be the minimum integer such that $b_1 - k$ or $b_2 - k$ is a start position. If there is no such k then we can compare x' and y' directly as a type 1 comparison. Otherwise, it holds that $x' = y'$ if and only if $\phi(S[b_1 - k, b_1]) = \phi(S[b_2 - k, b_2])$ (type 1) and $\phi(S[a_1 - k - 1, b_1 - k - 1]) = \phi(S[a_2 - k - 1, b_2 - k - 1])$ (type 2), since we have already verified that ϕ is collision-free for type 1 comparisions and type 2 comparisions of length 2^{p-1}.

The analysis is similar to that for SLPs. The sliding window can be implemented in $O(N)$ time, but for each window position we now need $O(\lg \lg N)$ time to retrieve the fingerprints, so the total time to verify ϕ for type 2 collisions becomes $O(N \lg N \lg \lg N)$. The space is $O(n)$ since in each round the dictionary stores at most $O(n)$ fingerprints.

4 Dynamic Relative Compression and Dynamic Partial Sums

Philip Bille* Patrick Hagge Cording Inge Li Gørtz*
Frederik Rye Skjoldjensen† Hjalte Wedel Vildhøj Søren Vind‡

Technical University of Denmark

Abstract

We initiate the study of *dynamic relative compression*, which is the problem of maintaining a compression of a dynamically changing string S. In this model S is compressed as a sequence of pointers to substrings of a static reference string R of length r. Let n be the size of the *optimal compression* of S with regards to R. We give a data structure that maintains an asymptotically optimal compression of S with regards to R using $O(n+r)$ space and $O(\lg n/\lg\lg n+\lg\lg r)$ time to access, replace, insert or delete a character in S. We can improve the update time to $O(\lg n/\lg\lg n)$ at the cost of increasing the space to $O(n+r\lg^{\varepsilon} r)$, for any $\varepsilon>0$. Our result can be generalized to storing multiple compressed strings with regards to the same reference string.

Our main technical contribution is a new linear-space data structure for *dynamic partial sums* on a sequence of w-bit integers with support for insertions and deletions, where w is the word size. Previous data structures assumed $\lg w$-bit or even $O(1)$-bit integers. We support all operations in optimal time $O(\lg s/\lg(w/\delta))$, matching a lower bound by Pătraşcu and Demaine [SODA 2004]. Here s is the length of the sequence and $\delta\leq w$ is the maximum number of bits allowed in updates.

*Supported by the Danish Research Council and the Danish Research Council under the Sapere Aude Program (DFF 4005-00267).

†Supported by the FNU project AlgoDisc (DFF 1323-00178).

‡Supported by a grant from the Danish National Advanced Technology Foundation.

4.1 Introduction

In this paper we study the problem of maintaining a compressed representation of a dynamically changing string S. Our solutions support efficient updates on S while guaranteeing that the compression is always asymptotically optimal.

The model of compression we consider is *relative compression*. Given a static reference string R, a *relative compression of a string S with regards to R* is an encoding of S as a sequence of references to substrings of R. Relative compression is a classic and general model of compression introduced by Storer and Szymanski [Sto+78; Sto+82] in 1978. Variations of this model, such as *relative Lempel-Ziv compression*, have been widely studied and shown to be very useful in practice [Kur+10; Kur+11a; Che+12; Do+14; Hoo+11].

We initiate the study of *dynamic relative compression* (DRC). More specifically, the dynamic relative compression problem is to maintain a relative compression of S under the following operations:

access(i): return the character $S[i]$,

replace(i, α): change $S[i]$ to α,

insert(i, α): insert character α before position i in S,

delete(i): delete the character at position i in S,

where i is a position in S and α is a character that occurs in R. Note that operations insert and delete changes the length of S by a single character. In all our results we also support decompressing an arbitrary substring of length ℓ in time $O(t_{\text{acc}} + \ell)$, where t_{acc} is the time to perform a single access(i) query.

A *relative compression of S with regards to R* is a sequence $C = ((i_1, j_1), ..., (i_{|C|}, j_{|C|}))$ such that $S = R[i_1, j_1] \cdots R[i_{|C|}, j_{|C|}]$. The size of the compression is $|C|$, and we say that C is *optimal* if it has minimum size over all relative compressions of S with regards to R. Throughout the paper, let r be the length of the reference string R, N be the length of the (uncompressed) string S, and n be the size of an optimal relative compression of S with regards to R.

4.1.1 Our Results

We present solutions to the DRC problem that efficiently support updating S while maintaining a relative compression of S of size at most $2n - 1$. All of our results are valid in the Word RAM model with word length $w \geq \lg(n + r)$ bits.

Theorem 4.1.1. *We can solve the DRC problem*

(i) in $O(n + r)$ space and $O\left(\frac{\lg n}{\lg \lg n} + \lg \lg r\right)$ time per operation, or

(ii) in $O(n + r \lg^{\varepsilon} r)$ space and $O\left(\frac{\lg n}{\lg \lg n}\right)$ time per operation, for any $\varepsilon > 0$.

Together, these bounds are optimal for most natural parameter combinations. In particular, any data structure for a string of length N supporting access, insert, and delete must use $\Omega(\lg N/\lg\lg N)$ time in the worst-case regardless of the space [Fre$^+$89] (this is called the *list representation problem*). Since $n \leq N$, we can view $O(\lg n/\lg\lg n)$ as a compressed optimal time bound that is always $O(\lg N/\lg\lg N)$ and better when S is compressible. Hence, Theorem 4.1.1(i) provides a linear-space solution that achieves the compressed optimal time bound except for an $O(\lg\lg r)$ additive term. Note that whenever $n \geq (\lg r)^{\lg^\varepsilon \lg r}$, for any $\varepsilon > 0$, the $O(\lg n/\lg\lg n)$ term dominates the query time and we match the compressed time bound. Hence, Theorem 4.1.1(i) is only suboptimal in the special case when n is almost exponentially smaller than R. In this case, we can use Theorem 4.1.1(ii) which always provides a solution achieving the compressed time bound at the cost of increasing the space to $O(n + r\lg^\varepsilon r)$.

Our solution can be generalized to support storing multiple strings with regards to a single reference string R in the natural way. In this case, n is the sum of the optimal compression sizes with regards to R for each string.

4.1.2 Technical Contributions

We reduce the DRC problem to the following two central problems, which we provide new and improved solutions for. Both of these problems are of independent interest.

4.1.2.1 The Dynamic Partial Sums Problem

Let $Z = Z[1], ..., Z[s]$ be a sequence of w-bit integers. The problem is to support the operations:

sum(i): return $\sum_{j=1}^{i} Z[j]$,

update(i, Δ): set $Z[i] = Z[i] + \Delta$,

search(t): return $1 \leq i \leq s$ such that sum($i-1$) $< t \leq$ sum(i),

insert(i, Δ): insert integer Δ before $Z[i]$,

delete(i): delete $Z[i]$.

We require $Z[i] \geq 0$ for all i at all times to ensure well-defined answers to search(t). The possibly negative Δ argument in update is restricted to be at most δ bits, so $|\Delta| \leq 2^\delta$. This (sometimes implicit) data structure parameter δ is standard and included in the best lower bound. Note that the update restriction implies that only elements of size $\leq 2^\delta$ can be inserted or deleted (as these operations can be used to implement update).

The non-dynamic partial sums problem (without the operations insert and delete) is well-studied [Die89; Ram$^+$01; Hus$^+$03; Fre$^+$89; Hon$^+$11; Hus$^+$96; Fen94; Păt$^+$04]. Pătraşcu and Demaine [Păt$^+$04] presented a linear-space data structure with query time $O(\lg s/\lg(w/\delta))$ per operation and showed a matching lower bound.

We give a new data structure for dynamic partial sums, which also supports the non-standard operations: divide(i,t), which replaces $Z[i]$ by $Z[i]' = t$ and $Z[i]'' = Z[i] - t$, where $0 \leq t \leq Z[i]$; and merge(i), which replaces $Z[i]$ and $Z[i+1]$ with $Z[i]' = Z[i] + Z[i+1]$.

Theorem 4.1.2. *The dynamic partial sums problem can be solved in linear space and $O(\lg s / \lg(w/\delta))$ worst-case time per operation. The data structure also supports the operations* divide *and* merge *in the same time.*

The only known solutions to dynamic partial sums are by Hon et al. [Hon$^+$11] and Navarro and Sadakane [Nav$^+$14]. Their solutions are succinct and use $O(\lg s / \lg\lg s)$ time per operation, but only work for sequences of small integers of $\leq \lg w$ bits.

Our dynamic partial sums data structure is the first to support full w-bit integers. Moreover, we match the lower bound for non-dynamic partial sums of $\Omega(\lg s / \lg(w/\delta))$ time per operation by Pătraşcu and Demaine [Păt$^+$04]. The dynamic operations are supported in optimal time as update can be implemented with insert and delete. Any solution to dynamic partial sums also solves list representation (we can solve access using two sum queries), and we match the lower bound of $\Omega(\lg s / \lg\lg s)$ for elements of $\delta \leq \lg^\varepsilon s$ bits where $\varepsilon < 1$. Pătraşcu and Demaine [Păt$^+$04] show that this is optimal when also supporting sum.

The main difficulty in supporting insertions and deletions is that indices may change, which causes elements in the sequence to shift left or right. We build on the solution to non-dynamic partial sums by Pătraşcu and Demaine [Păt$^+$04], extending it to the dynamic case. They precompute sums and store them in an array coupled with a small data structure for updates. This supports all operations in constant time for a polylogarithmically size sequence, and a standard reduction (that we will also use) is used to store the full sequence in a B-tree.

In our construction we first simplify their approach, only storing (values and indices) of some representative elements in a dynamic integer set data structure by Pătraşcu and Thorup [Păt$^+$14]. All other elements are stored in the update data structure. We then show that this representation can be changed to efficiently support insertions and deletions by modifying the way we store representative indices (and how to support the dynamic operations in the update data structure).

4.1.2.2 The Substring Concatenation Problem

A *substring concatenation query* on a string R takes two pairs of indices (i,j) and (i',j') and returns the start position in R of an occurrence of $R[i,j]R[i',j']$, or `NO` if the string is not a substring of R. The *substring concatenation problem* is to preprocess R into a data structure that supports substring concatenation queries.

Let r be the length of R. Amir et al. [Ami$^+$07] gave a solution using $O(r\sqrt{\lg r})$ space with query time $O(\lg\lg r)$, and very recently Gawrychowski et al. [Gaw$^+$14] showed how to solve the problem in $O(r \lg r)$ space and $O(1)$ time. We propose two new solutions that improve each of their bounds.

Theorem 4.1.3. *The substring concatenation problem can be solved in*

(i) in $O(r \lg^{\varepsilon} r)$ space and $O(1)$ time, for any $\varepsilon > 0$, or

(ii) in $O(r)$ space and $O(\lg\lg r)$ time,

The two solutions lead to the two branches of Theorem 4.1.1.

4.1.3 Extensions

Our results have the following interesting extensions, the details are in Section 4.5.

Theorem 4.1.4. *We can solve the DRC problem, only supporting* access *and* replace

(i) in space $O(n + r)$ and time $O(\lg\lg N)$ for access *and time $O(\lg\lg N + \lg\lg r)$ for* replace*, or*

(ii) in space $O(n + r \lg^{\varepsilon} r)$ and time $O(\lg\lg N)$ for both operations.

This implies an improved solution to the following dynamic pattern matching problem: Given a static pattern P and a text T, maintain the set $\mathcal{L}$ of occurrences of P in T under an update operation that replaces the i^{th} character in T with another character from the alphabet. For this problem, Amir et al. [Ami+07] gave a data structure using $O(|T| + |P|\sqrt{\lg|P|})$ space and supporting updates in $O(\lg\lg|P|)$ time. By using Theorem 4.1.4(i) as a black-box in their work, we obtain a data structure using $O(|T| + |P|)$ space with $O(\lg\lg|P|)$ update time.

Multiple Strings with Split and Concatenate

The *dynamic relative compression problem on multiple strings* is the problem of maintaining a dynamic set of strings, $\mathcal{S} = \{S_1, \ldots, S_k\}$, all compressed in the dynamic relative compression model, relative to the same reference string R. In this problem the operations access, replace, insert and delete take an extra parameter indicating the string from $\mathcal{S}$ to perform the operation on. We additionally wish to support the operations split(i, j), which updates $\mathcal{S}$ to $(\mathcal{S} \setminus S_i) \cup \{S_i[1, j-1], S_i[j, |S_i|]\}$, and concat$(i, j)$ updating $\mathcal{S}$ to $(\mathcal{S} \setminus \{S_i, S_j\}) \cup \{S_i S_j\}$.

Theorem 4.1.5. *We can solve the DRC problem on multiple strings, supporting* access*,* replace*,* insert*,* delete*,* split*, and* concat*,*

(i) in space $O(n + r)$ and time $O(\lg n)$ for access *and time $O(\lg n + \lg\lg r)$ for* replace*,* insert*,* delete*,* split*, and* concat*, or*

(ii) in space $O(n + r \lg^{\varepsilon} r)$ and time $O(\lg n)$ for all operations.

4.1.4 Related Work

Relative Compression The non-recursive external pointer macro compression scheme was introduced by Storer and Szymanski in 1978 [Sto+78; Sto+82]. They provided seminal analytical work, showing that finding the best reference string given some uncompressed string is NP-complete, and showed that $n + r = \Omega(\sqrt{N})$ for any reference string when only compressing a single string. This is the same compression ratio as obtained by the LZ78/LZW schemes [Wel84; Ziv+78] when compressing a single string. However, if compressing multiple strings with regards to the same reference string we can do much better. Relative compression is a special case of their scheme which also supports compressing multiple strings. The general scheme suggested by Storer and Szymanski is also sometimes called LZSS, used by popular archivers such as PKZip, ARJ and RAR.

More recently, Kuruppu et al. [Kur+10] suggested the Relative Lempel-Ziv (RLZ) scheme for compressing highly repetitive data such as genomes. The RLZ scheme defines its parse identically to the greedy LZ77 [Ziv+77] parse when using all substrings of R as the dictionary, meaning that it produces an optimal compression for a given and fixed reference string (shown in [Coh+96; Sto+82; Cro+14]). Multiple papers show impressive compression ratios for both RLZ and LZSS in various practical settings, storing a dictionary of similar strings [Fer+13; Kur+11a; Che+12; Do+14; Hoo+11; Kur+11b; Wan+13; Deo+11; Wan+12; Och+14]. The relative compression scheme considered in this paper is closely related to RLZ, and we may in fact create our initial compression as in the RLZ scheme. However, we maintain an asymptotically optimal compression under updates.

Dynamic Compression Several schemes for dynamic compression exist in the literature. The overall idea is to dynamically maintain a compressed string under a set of operations. In particular, Grossi et al. [Gro+13] presents a scheme for compressing a string S in entropy bounds while dynamically maintaining it under the operations: access a substring of S; replace, insert or delete a character of S; support rank and select queries on S. This work stem from earlier work on string compression in entropy bounds. Both Grossi et al. [Gro+03] and Ferragina et al. [Fer+04] present schemes for providing random access to entropy compressed strings. These ideas have gradually been refined and varied in [Fer+04; Fer+05; Sad+06; Gon+07; Fer+07; Jan+12; Nav+13].

All of the above schemes either use succinct space or compressed space depending on the empirical entropy defined relative to fixed-length contexts. In contrast, relative compression can take advantage of long contexts and hence may achieve much better compression on such strings.

4.2 Dynamic Relative Compression

In this section we show how Theorems 4.1.2 and 4.1.3 lead to Theorem 4.1.1. The proofs of Theorems 4.1.2 and 4.1.3 appear in Section 4.3 and Section 4.4, respectively.

Let $C = ((i_1, j_1), ..., (i_{|C|}, j_{|C|}))$ be the compressed representation of S. From now on, we refer to C as the *cover of* S, and call each element (i_l, j_l) in C a *block*. Recall that a

block (i_l, j_l) refers to a substring $R[i_l, j_l]$ of R. A cover C is *maximal* if concatenating any two consecutive blocks $(i_l, j_l), (i_{l+1}, j_{l+1})$ in C yields a string that does not occur in R, i.e., the string $R[i_l, j_l]R[i_{l+1}, j_{l+1}]$ is not a substring of R. We need the following lemma.

Lemma 4.2.1. *If C_{MAX} is a maximal cover and C is an arbitrary cover of S, then $|C_{\text{MAX}}| \leq 2|C| - 1$.*

Proof. In each block b of C there can start at most two blocks in C_{MAX}, because otherwise two adjacent blocks in C_{MAX} would be entirely contained in the block b, contradicting the maximality of C_{MAX}. In the last block of C there can start at most one block in C_{MAX}. Hence, $|C_{\text{MAX}}| \leq 2|C| - 1$. □

Recall that n is the size of an optimal cover of S with regards to R. The lemma implies that we can maintain a compression with size at most $2n - 1$ by maintaining a maximal cover of S. The remainder of this section describes our data structure for maintaining and accessing such a cover. Initially, we construct a maximal cover of S in $O(|S| + r)$ time by greedily traversing the suffix tree of R.

4.2.1 Data Structure

The high level idea for supporting the DRC operations on S is to store the sequence of block lengths $(j_1 - i_1 + 1, \ldots, j_{|C|} - i_{|C|} + 1)$ in a dynamic partial sums data structure. This allows us, for example, to identify the block that encodes the k^{th} character in S by performing a $\mathsf{search}(k)$ query.

Updates to S are implemented by splitting a block in C, which may break the maximality property. The key idea is to use substring concatenation queries on R to detect consecutive blocks that break maximality. When two such blocks are found, we merge the two blocks in C. We only need a constant number of substring concatenation queries to restore maximality. To maintain the correct sequence of block lengths we use update, divide and merge operations on the dynamic partial sums data structure.

Our data structure consist of the string R, a substring concatenation data structure of Theorem 4.1.3 for R, a maximal cover C for S stored in a doubly linked list, and the dynamic partial sums data structure of Theorem 4.1.2 storing the block lengths of C (we require the data structure to support divide and merge). We also store auxiliary links between a block in the doubly linked list and the corresponding block length in the partial sums data structure, and a list of alphabet symbols in R with the location of an occurrence for each symbol. By Lemma 4.2.1 and since C is maximal we have $|C| \leq 2n - 1 = O(n)$. Hence, the total space for C and the partial sums data structure is $O(n)$. The space for R is $O(r)$ and the space for substring concatenation data structure is either $O(r)$ or $O(r \lg^{\varepsilon} r)$ depending on the choice in Lemma 4.1.3. Hence, in total we use either $O(n + r)$ or $O(n + r \lg^{\varepsilon} r)$ space.

4.2.2 Answering Queries

To answer access(i) queries we first compute search(i) in the dynamic partial sums structure to identify the block $b_l = (i_l, j_l)$ containing position i in S. The local index in $R[i_l, j_l]$ of the i^{th} character in R is $\ell = i - \mathsf{sum}(l-1)$, and thus the answer to the query is the character $R[i_l + \ell]$.

We perform replace and delete by first identifying $b_l = (i_l, j_l)$ and ℓ as above. Then we partition b_l into three new blocks $b_l^1 = (i_l, i_l + \ell - 1)$, $b_l^2 = (i_l + \ell)$, $b_l^3 = (i_l + \ell + 1, j_l)$ where b_l^2 is the single character block for index i in S that we must change. In replace we change b_l^2 to an index of an occurrence in R of the new character (which we can find from the list of alphabet symbols), while we remove b_l^2 in delete. The new blocks and their neighbors, that is, b_{l-1}, b_l^1, b_l^2, b_l^3, and b_{l+1} may now be non-maximal. To restore maximality we perform substring concatenation queries on each consecutive pair of these 5 blocks, and replace non-maximal blocks with merged maximal blocks. A similar idea is used by Amir et al. [Ami$^+$07]. We perform update, divide and merge operations to maintain the corresponding lengths in the dynamic partial sums data structure. The insert operation is similar, but inserts a new single character block between two parts of b_j before restoring maximality. Observe that using $\delta = O(1)$ bits in update is sufficient to maintain the correct block lengths.

In total, each operation require a constant number of substring concatenation queries and dynamic partial sums operations; the latter with time complexity $O(\lg n / \lg(w/\delta)) = O(\lg n / \lg\lg n)$ as $w \geq \lg n$. Hence, the total time for each access, replace, insert, and delete operation is either $O(\lg n / \lg\lg n + \lg\lg r)$ or $O(\lg n / \lg\lg n)$ depending on the substring concatenation data structure used. In summary, this proves Theorem 4.1.1.

4.3 Dynamic Partial Sums

In this section we prove Theorem 4.1.2. Recall that $Z = Z[1], ..., Z[s]$ is a sequence of w-bit integer keys. We show how to implement the two non-standard operations divide(i, t) and merge(i), besides operations sum(i), search(t), and update(i, Δ). We support the operations insert(i, Δ) and delete(i) (insert a new integer Δ before element i or delete element i, respectively) by implementing them using update and a divide or merge operation, respectively. This means that we support inserting or deleting keys with value at most 2^δ.

We first solve the problem for small sequences. The general solution uses a standard reduction, storing Z at the leaves of a B-tree of large outdegree. We use the solution for small sequences to navigate in the internal nodes of the B-tree.

Dynamic Integer Sets We need the following recent result due to Pătraşcu and Thorup [Păt$^+$14] on maintaining a set of integer keys X under insertions and deletions. The queries are as follows, where q is an integer. The membership query member(q) returns true if $q \in X$, predecessor $\mathsf{pred}_X(q)$ returns the largest key $x \in X$ where $x < q$, and successor $\mathsf{succ}_X(q)$ returns the smallest key $x \in X$ where $x \geq q$. The rank $\mathsf{rank}_X(q)$ returns the number of keys in X smaller than q, and select(i) returns the i^{th} smallest key in X.

Lemma 4.3.1 (Pătraşcu and Thorup [Păt+14])**.** *There is a data structure for maintaining a dynamic set of $w^{O(1)}$ w-bit integers that supports insert, delete, membership, predecessor, successor, rank and select in constant time per operation.*

4.3.1 Dynamic Partial Sums for Small Sequences

Let Z be a sequence of at most $B \leq w^{O(1)}$ integer keys. We will show how to store Z in linear space such that all dynamic partial sums operations can be performed in constant time. We let Y be the sequence of prefix sums of Z, defined such that each key $Y[i]$ is the sum of the first i keys in Z, i.e., $Y[i] = \sum_{j=1}^{i} Z[j]$. Observe that $\mathsf{sum}(i) = Y[i]$ and $\mathsf{search}(t)$ is the index of the successor of t in Y. Our goal is to store and maintain a representation of Y subject to the dynamic operations update, divide and merge in constant time per operation.

4.3.1.1 The Scheme by Pătraşcu and Demaine.

We first review a version of the solution to the non-dynamic partial sums problem by Pătraşcu and Demaine [Păt+04], simplified and improved due to Lemma 4.3.1. Our dynamic solution builds on this.

The entire data structure is rebuilt every B operations as follows. We first partition Y greedily into *runs*. Two neighboring elements in Y are in the same run if their difference is at most $B2^{\delta}$, and we call the first element of each run a *representative* for all elements in the run. We use $\mathcal{R}$ to denote the sequence of representative values in Y and $rep(i)$ to be the index of the representative for element $Y[i]$ among the elements in $\mathcal{R}$.

We store Y by splitting representatives and other elements into separate data structures: $\mathcal{I}$ and $\mathcal{R}$ store the representatives at the time of the last rebuild, while $\mathcal{U}$ stores each element in Y as an offset to its representative value as well as updates since the last rebuild. We ensure $Y[i] = \mathcal{R}[rep(i)] + \mathcal{U}[i]$ for any i and can thus reconstruct the values of all elements in Y.

The representatives are stored as follows. $\mathcal{I}$ is the sequence of indices in Y of the representatives and $\mathcal{R}$ is the sequence of representative values in Y. Both $\mathcal{I}$ and $\mathcal{R}$ are stored using the data structure of Lemma 4.3.1. We then define $\mathsf{rep}(i) = \mathsf{rank}_{\mathcal{I}}(\mathsf{pred}_{\mathcal{I}}(i))$ as the index of the representative for i among all representatives, and use $\mathcal{R}[\mathsf{rep}(i)] = \mathsf{select}_{\mathcal{R}}(\mathsf{rep}(i))$ to get the value of the representative for i.

We store in $\mathcal{U}$ the current difference from each element to its representative, $\mathcal{U}[i] = Y[i] - \mathcal{R}[\mathsf{rep}(i)]$ (i.e. updates between rebuilds are applied to $\mathcal{U}$). The idea is to pack $\mathcal{U}$ into a single word of B elements. Observe that $\mathsf{update}(i, \Delta)$ adds value Δ to all elements in Y with index at least i. We can support this operation in constant time by adding to $\mathcal{U}$ a word that encodes Δ for those elements. Since each difference between neighbours in a run is at most $B2^{\delta}$ and $|Y| = O(B)$, the maximum value in $\mathcal{U}$ after a rebuild is $O(B^2 2^{\delta})$. As B updates of size 2^{δ} may be applied before a rebuild, the changed value at each element due to updates is $O(B2^{\delta})$. So each element in $\mathcal{U}$ requires $O(\lg B + \delta)$ bits (including an overflow bit per element). Thus, $\mathcal{U}$ requires $O(B(\lg B + \delta))$ bits in total and can be packed in a single word for $B = O(\min\{w/\lg w, w/\delta\})$.

Between rebuilds the stored representatives are potentially outdated because updates may have changed their values. However, observe that the values of two neighboring representatives differ by more than $B2^\delta$ at the time of a rebuild, so the gap between two representatives cannot be closed by B updates of δ bits each (before the structure is rebuilt again). Hence, an answer to $\mathsf{search}(t)$ cannot drift much from the values stored by the representatives; it can only be in a constant number of runs, namely those with a representative value $\mathsf{succ}_{\mathcal{R}}(t)$ and its two neighbours. In a run with representative value v, we find the smallest j (inside the run) such that $\mathcal{U}[j]+v-t>0$. The smallest j found in all three runs is the answer to the $\mathsf{search}(t)$ query. Thus, by rebuilding periodically, we only need to check a constant number of runs when answering a $\mathsf{search}(t)$ query.

On this structure, Pătraşcu and Demaine [Păt+04] show that the operations sum, search and update can be supported in constant time each as follows:

$\mathsf{sum}(i)$: return the sum of $\mathcal{R}[\mathsf{rep}(i)]$ and $\mathcal{U}[i]$. This takes constant time as $\mathcal{U}[i]$ is a field in a word and representatives are stored using Lemma 4.3.1.

$\mathsf{search}(t)$: let $r_0=\mathsf{rank}_{\mathcal{R}}(\mathsf{succ}_{\mathcal{R}}(t))$. We must find the smallest j such that $\mathcal{U}[j]+R[r]-t>0$ for $r\in\{r_0-1,r_0,r_0+1\}$, where j is in run r. We do this for each r using standard word operations in constant time by adding $R[r]-t$ to all elements in $\mathcal{U}$, masking elements not in the run (outside indices $\mathsf{select}_{\mathcal{I}}(r)$ to $\mathsf{select}_{\mathcal{I}}(r+1)-1$, and counting the number of negative elements.

$\mathsf{update}(i,\Delta)$: we do this in constant time by copying Δ to all fields $j\geq i$ by a multiplication and adding the result to $\mathcal{U}$.

To count the number of negative elements or find the least significant bit in a word in constant time, we use the technique by Fredman and Willard [Fre+93].

Notice that rebuilding the data structure every B operations takes $O(B)$ time, resulting in amortized constant time per operation. We can instead do this incrementally by a standard approach by Dietz [Die89], reducing the time per operation to worst case constant. The idea is to construct the new replacement data structure incrementally while using the old and complete data structure. More precisely, during update j we rebuild the data structure for index $(j \bmod B)$.

4.3.1.2 Efficient Support for Modifications

We now show how to maintain the structure described above while supporting operations $\mathsf{divide}(i,t)$ and $\mathsf{merge}(i)$.

Observe that the operations are only local: Splitting $Z[i]$ into two parts or merging $Z[i]$ and $Z[i+1]$ does not influence the precomputed values in Y (besides adding/removing values for the divided/merged elements). We must update $\mathcal{I}$, $\mathcal{R}$ and $\mathcal{U}$ to reflect these local changes accordingly. Because a divide or merge operation may create new representatives between rebuilds with values that does not fit in $\mathcal{U}$, we change $\mathcal{I}$, $\mathcal{R}$ and $\mathcal{U}$ to reflect these new representatives by rebuilding the data structure locally. This is done as follows.

First, consider the run representatives. Both $\mathsf{divide}(i,t)$ and $\mathsf{merge}(i)$ may require us to create a new run, combine two existing runs or remove a run. In any of those cases, we can find a replacement representative for each run affected. As the operations are only local, the replacement is either a divided or merged element, or one of the neighbours of the replaced representative. Replacing representatives may cause both indices and values for the stored representatives to change. We use insertions and deletions on $\mathcal{R}$ to update representative values.

Since the new operations change the indices of the elements, these changes must also be reflected in $\mathcal{I}$. For example, a $\mathsf{merge}(i)$ operation decrement the indices of all elements with index larger than i compared to the indices stored at the time of the last rebuild (and similarly for $\mathsf{divide}(i,t)$). We should in principle adjust the $O(B)$ changed indices stored in $\mathcal{I}$. The cost of adjusting the indices accordingly when using Lemma 4.3.1 to store $\mathcal{I}$ is $O(B)$. Instead, to get our desired constant time bounds, we represent $\mathcal{I}$ using a resizable data structure with the same number of elements as Y that supports this kind of update. We must support $\mathsf{select}_{\mathcal{I}}(i)$, $\mathsf{rank}_{\mathcal{I}}(q)$, and $\mathsf{pred}_{\mathcal{I}}(q)$ as well as inserting and deleting elements in constant time. Because $\mathcal{I}$ has few and small elements, we can support the operations in constant time by representing it using a bitstring $\mathcal{B}$ and a structure $\mathcal{C}$ which is the prefix sum over $\mathcal{B}$ as follows.

Let $\mathcal{B}$ be a bitstring of length $|Y| \leq B$, where $\mathcal{B}[i] = 1$ iff there is a representative at index i. $\mathcal{C}$ has $|Y|$ elements, where $\mathcal{C}[i]$ is the prefix sum of $\mathcal{B}$ including element i. Since $\mathcal{C}$ requires $O(B \lg B)$ bits in total we can pack it in a single word. We answer queries as follows: $\mathsf{rank}_{\mathcal{I}}(q)$ equals $\mathcal{C}[q-1]$, we answer $\mathsf{select}_{\mathcal{I}}(i)$ by subtracting i from all elements in $\mathcal{C}$ and return one plus the number of elements smaller than 0 (as done in $\mathcal{U}$ when answering search), and we find $\mathsf{pred}_{\mathcal{I}}(q)$ as the index of the least significant bit in $\mathcal{B}$ after having masked all indices larger than q. Updates are performed as follows. Using mask, shift and concatenate operations, we can ensure that $\mathcal{B}$ and $\mathcal{C}$ have the same size as Y at all times (we extend and shrink them when performing divide and merge operations). Inserting or deleting a representative is to set a bit in $\mathcal{B}$, and to keep $\mathcal{C}$ up to date, we employ the same ± 1 update operation as used in $\mathcal{U}$.

We finally need to adjust the relative offsets of all elements with a changed representative in $\mathcal{U}$ (since they now belong to a representative with a different value). In particular, if the representative for $\mathcal{U}[j]$ changed value from v to v', we must subtract $v' - v$ from $\mathcal{U}[j]$. This can be done for all affected elements belonging to a single representative simultaneously in $\mathcal{U}$ by a single addition with an appropriate bitmask (update a range of $\mathcal{U}$). Note that we know the range of elements to update from the representative indices. Finally, we may need to insert or delete an element in $\mathcal{U}$, which can be done easily by mask, shift and concatenate operations on the word $\mathcal{U}$.

We present an example structure where $B2^{\delta} = 4$ in Figures 4.1, 4.2 and 4.3. Figure 4.1 presents the data structure immediately after a rebuild, Figure 4.2 shows the result of performing $\mathsf{divide}(8,3)$ on the structure of Figure 4.1, and Figure 4.3 shows the result of performing $\mathsf{merge}(12)$ on the structure of Figure 4.2.

Concluding, divide and merge can be supported in constant time, so:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Z	5	1	4	7	1	1	6	5	1	1	2	2	1	3	5	10	5	10	2
Y	5	6	10	17	18	19	25	30	31	32	34	36	37	40	45	55	60	70	72
$\mathcal{R}$	$\{5, 17, 25, 30, 45, 55, 60, 70\}$																		
$\mathcal{U}$	0	1	5	0	1	2	0	0	1	2	4	6	7	10	0	0	0	0	2
$\mathcal{B}$	1	0	0	1	0	0	1	1	0	0	0	0	0	0	1	1	1	1	0
$\mathcal{C}$	1	1	1	2	2	2	3	4	4	4	4	4	4	4	5	6	7	8	8

Figure 4.1: The initial data structure constructed from Z..

New index 9 Old index 9

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Z	5	1	4	7	1	1	6	3	2	1	1	2	2	1	3	5	10	5	10	2
Y	5	6	10	17	18	19	25	28	30	31	32	34	36	37	40	45	55	60	70	72
$\mathcal{R}$	$\{5, 17, 25, 45, 55, 60, 70\}$																			
$\mathcal{U}$	0	1	5	0	1	2	0	3	5	6	7	9	11	12	15	0	0	0	0	2
$\mathcal{B}$	1	0	0	1	0	0	1	0	0	0	0	0	0	0	0	1	1	1	1	0
$\mathcal{C}$	1	1	1	2	2	2	3	3	3	3	3	3	3	3	3	4	5	6	7	7

Figure 4.2: The result of divide$(8, 3)$ on the structure of Figure 4.1. Representative value 30 was removed from $\mathcal{R}$. We shifted and updated $\mathcal{U}$, $\mathcal{B}$ and $\mathcal{C}$ to remove the old representative and accommodate for a new element with value 2.

Index containing the sum of the merged indices.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Z	5	1	4	7	1	1	6	3	2	1	1	4	1	3	5	10	5	10	2
Y	5	6	10	17	18	19	25	28	30	31	32	36	37	40	45	55	60	70	72
$\mathcal{R}$	$\{5, 17, 25, 45, 55, 60, 70\}$																		
$\mathcal{U}$	0	1	5	0	1	2	0	3	5	6	7	11	12	15	0	0	0	0	2
$\mathcal{B}$	1	0	0	1	0	0	1	0	0	0	0	0	0	0	1	1	1	1	0
$\mathcal{C}$	1	1	1	2	2	2	3	3	3	3	3	3	3	3	4	5	6	7	7

Figure 4.3: The result of merge(12) on the structure of Figure 4.2.

Theorem 4.3.2. *There is a linear space data structure for dynamic partial sums supporting each operation* search*,* sum*,* update*,* insert*,* delete*,* divide*, and* merge *on a sequence of length* $O(\min\{w/\lg w, w/\delta\})$ *in worst-case constant time.*

4.3.2 Dynamic Partial Sums for Large Sequences

Willard [Wil00] (and implicitly Dietz [Die89]) showed that a leaf-oriented B-tree with out-degree B of height h can be maintained in $O(h)$ worst-case time if: 1) searches, insertions and deletions take $O(1)$ time per node when no splits or merges occur, and 2) merging or splitting a node of size B require $O(B)$ time.

We use this as follows, where Z is our integer sequence of length s. Create a leaf-oriented B-tree of degree $B = \Theta(\min\{w/\lg w, w/\delta\})$ storing Z in the leaves, with height $h = O(\lg_B n) = O(\lg n/\lg(w/\delta))$. Each node $v$uses Theorem 4.3.2 to store the $O(B)$ sums of leaves in each of the subtrees of its children. Searching for t in a node corresponds to finding the successor $Y[i]$ of t among these sums. Dividing or merging elements in Z corresponds to inserting or deleting a leaf. This concludes the proof of Theorem 4.1.2.

4.4 Substring Concatenation

In this section, we give the proof of Theorem 4.1.3. Recall that we must store a string R to answer substring concatenation queries: given two strings x and y return the location of an occurrence of xy in R or `NO` if no such occurrence exist.

To prove (i) we need the following definitions. For a substring x of R, let $S(x)$ denote the suffixes of R that have x as a prefix, and let $S'(x) = \{i+|x| \mid i \in S(x) \wedge i+|x| \leq n\}$, i.e., $S'(x)$ are the suffixes of R that are immediately preceded by x. Hence for two substrings x and y, the suffixes that have xy as a prefix are exactly $S'(x) \cap S(y)$. This is a 2D range reporting problem, but we can reduce to it to a 1D range emptiness problem in *rank space* as follows.

Let $\text{rank}(i)$ be the position of suffix $R[i...]$ in the lexicographic ordering of all suffixes of R, and let $\text{rank}(A) = \{\text{rank}(i) \mid i \in A\}$ for $A \subseteq \{1, n\}$. Then xy is a substring of R if and only if $\text{rank}(S'(x)) \cap \text{rank}(S(y)) = \emptyset$. Note that $\text{rank}(S(y))$ is a range $[a, b] \subseteq [1, n]$, and we can determine this range in constant time for any substring y using a constant-time weighted ancestor query on the suffix tree of R [Gaw$^+$14]. Consequently, we can decide if xy is a substring of R by a 1D range emptiness query on the set $\text{rank}(S'(x))$.

Belazzougui et al. [Bel$^+$10] (see also [Gos$^+$15]) recently gave a 1D range emptiness data structure for a set $A \subseteq [1, r]$ using $O(|A| \lg^\varepsilon r)$ *bits* of space, for any $\varepsilon > 0$, and answering queries in constant time. We will build this data structure for $\text{rank}(S'(x))$, but doing so for all substrings would require space $\tilde{\Omega}(r^2)$.

To arrive at the space bound of $O(r \lg^\varepsilon r)$ (words), we employ the standard technique of a heavy path decomposition [Har$^+$84] on the suffix tree of R, and only build the data structure for substrings of R that correspond to the top of a heavy path. In this way, each suffix will appear in at most $\lg r$ such data structures, leading to the claimed $O(r \lg^\varepsilon r)$ space bound (in words).

To answer a substring concatenation query with substrings x and y, we first determine how far y follows the heavy path in the suffix tree from the location where x stops. This can be done in $O(1)$ time by a standard constant-time LCP query between two suffixes of R. We then proceeed to the top of the next heavy path, where we query the 1D range reporting data structure with the range $\text{rank}(S(y'))$ where y' is the remaining unmatched suffix of y. This completes the query, and the proof of (i).

The second solution (ii) is an implication of a result by Bille et al. [Bil$^+$14]. Given the suffix tree ST_R of R, an *unrooted LCP query* [Col$^+$04] takes a suffix y and a location ℓ in ST_R (either a node or a position on an edge) and returns the location in ST_S that is reached after matching y starting from location ℓ. A substring concatenation query is straightforward to implement using two unrooted LCP queries, the first one starting at the root, and the second starting from the location returned by the first query. It follows from Bille et al. [Bil$^+$14] that we can build a linear space data structure that supports unrooted LCP queries in time $O(\lg \lg r)$ thus completing the proof of (ii).

4.5 Extensions to DRC

In this section we show how to solve two other variants of the dynamic relative compression problem. We first prove Theorem 4.1.4, showing how to improve the query time if only supporting operations access and replace. We then show Theorem 4.1.5, generalising the problem to support multiple strings. These data structures use the same substring concatenation data structure of Theorem 4.1.3 as before but replace the dynamic partial sums data structure.

4.5.1 DRC Restricted to Access and Replace

In this setting we constrain the operations on S to $\mathsf{access}(i)$ and $\mathsf{replace}(i, \alpha)$. Then, instead of maintaining a dynamic partial sums data structure over the lengths of the substrings in C, we only need a dynamic predecessor data structure over the prefix sums. The operations are implemented as before, except that for $\mathsf{access}(i)$ we obtain block b_j by computing the predecessor of i in the predecessor data structure, which also immediately gives us access to ℓ. For $\mathsf{replace}(i, \alpha)$, a constant number of updates to the predecessor data structure is needed to reflect the changes. We use substring concatenation queries to restore maximality as described in Section 4.2. The prefix sums of the subsequent blocks in C are preserved since $|b_j| = |b_j^1| + |b_j^2| + |b_j^3|$. This results in Theorem 4.1.4.

4.5.2 DRC of Multiple Strings with Split and Concatenate

Consider the variant of the dynamic relative compression problem where we want to maintain a relative compression of a set of strings $S_1, \ldots, S_k$. Each string S_i has a cover C_i and all strings are compressed relative to the same string R. In this setting $n = \sum_{i=1}^{k} |C_i|$. In addition to the operations access, replace, insert, and delete, we also want to support split and concatenation of strings. Note that the semantics of the operations change to indicate the string(s) to perform a given operation on.

We build a height-balanced binary tree T_i (e.g. an AVL tree or red-black tree) over the blocks $C_i[1], \ldots, C_i[|C_i|]$ for each string S_i. In each internal node v, we store the sum of the block sizes represented by its leaves. Since the total number of blocks is n, the trees use $O(n)$ space. All operations rely on the standard procedures for searching, inserting, deleting, splitting and joining height-balanced binary trees. All of these run in $O(\lg n)$ time for a tree of size n. See for example [Cor+01] for details on how red-black trees achieve this.

The answer to an $\mathsf{access}(i, j)$ query is found by doing a top-down search in T_i using the sums of block sizes to navigate. Since the tree is balanced and the size of the cover is at most n, this takes $O(\lg n)$ time. The operations $\mathsf{replace}(i, j, \alpha)$, $\mathsf{insert}(i, j, \alpha)$, and $\mathsf{delete}(i, j)$ all initially require that we use $\mathsf{access}(i, j)$ to locate the block containing the j-th character of S_i. To reflect possible changes to the blocks of the cover, we need to modify the corresponding tree to contain more leaves and restore the balancing property. Since the number of nodes added to the tree is constant these operations each take $O(\lg n)$ time. The $\mathsf{concat}(i, j)$ operation requires that we join two trees in the standard way and restore the balancing property of the resulting tree. For the $\mathsf{split}(i, j)$ operation we first split the block that contains position j such that the j-th character is the trailing character of a block. We then split the tree into two trees separated by the new block. This takes $O(\lg n)$ time for a height-balanced tree.

To finalize the implementation of the operations, we must restore the maximality property of the affected covers as described in Section 4.2. At most a constant number of blocks are non-maximal as a result of any of the operations. If two blocks can be combined to one, we delete the leaf that represents the rightmost block, update the leftmost block to reflect the change, and restore the property that the tree is balanced. If the tree subsequently contains an internal node with only one child, we delete it and restore the balancing. Again, this takes $O(\lg n)$ time for balanced trees, which concludes the proof of Theorem 4.1.5.

Acknowledgements

We thank Pawel Gawrychowski for helpful discussions.

5 Compressed Pattern Matching in the Annotated Streaming Model

Markus Jalsenius[1] Benjamin Sach[1] Søren Vind[2*]

[1] University of Bristol
[2] Technical University of Denmark

Abstract

The *annotated streaming model* was originally introduced by Chakrabarti, Cormode and McGregor [ICALP 09, ACM TALG 14]. In this extension of the conventional streaming model, a single query on the stream has to be answered by a *client* with the help from an untrusted *annotator* that provides an annotated data stream (to be used with the input stream). Only one-way communication is allowed. We extend this model by considering multiple queries. In particular, our primary focus is on on-going queries where a new output must be given every time a stream item arrives.

In this model, we show the existence of a data structure that enables us to store and recover information about past items in the stream in very little space on the client. We first use this technique to give a space-annotation trade-off for the annotated multi-indexing problem, which is a natural generalisation of the previously studied annotated indexing problem.

Our main result is a space-annotation trade-off for the classic exact pattern matching problem in phrase-compressed strings. In particular, we show the existence of a $O(\lg n)$ time per phrase, $O(\lg n + m)$ client space solution which uses $O(\lg n)$ words of annotation per phrase. If the client space is increased to $O(n^\epsilon + m)$ then $O(1)$ words of annotation and $O(1)$ time per phrase suffices. Here n is the length of the stream and m is the length of the pattern. Our result also holds for the well-known LZ78 compression scheme which is a special case of phrase-compression.

All of the problems we consider have $\Omega(n)$ randomised space lower bounds in the standard (unannotated) streaming model.

*Supported by a grant from the Danish National Advanced Technology Foundation.

5.1 Introduction

In the *streaming model* [Mun+80; Fla+85; Alo+99], an *input stream* of elements arrive one at a time, and we must solve problems using sub-linear (typically polylogarithmic) space and time per element with a single pass over the data. Throughout the paper, we let n denote the length of the input stream, and assume that input elements require $O(1)$ words of $w \geq \lg n$ bits each.

In order to model the current state of affairs in computing with easy and cheap access to massive computational power over the internet, the *annotated streaming model* introduced by Chakrabarti, Cormode and McGregor [Cha+09; Cha+14] expands the normal streaming model by introducing an untrustworthy *annotator*. This annotator is assumed to have infinite computational resources and storage, and it assists a *client* in solving some problem by providing an *annotated data stream* that is transmitted along with the normal input stream (i.e. the client-annotator communication is one-way). Software and hardware faults, as well as intentional attempts at deceit by the annotator (as could reasonably be the case), are modeled by assuming that the annotator cannot be trusted. Consequently, for a given problem we must create a client algorithm and associated annotation protocol that allows the client to either solve the problem if the annotator is honest, or alternatively to detect a protocol inconsistency. As we are designing an algorithm-protocol pair, we allow the annotator to know the online algorithm used by the client (but, crucially, not the random choices made by the client at runtime). This means that the annotator can simulate the client and its behaviour, up to random choices.

In this paper we introduce a variant of the annotated streaming model which is suited to on-going, high-throughput streaming pattern matching problems. In particular our main result in this paper is a randomized annotated streaming algorithm which detects occurrences of a pattern in a compressed text stream as they occur with high probability. In contrast to the standard annotated streaming model, our result returns an answer (to the implicit query 'is there a new match?') every time a stream element arrives. We also provide worst case bounds on the amount of annotation received by the client between any two text stream elements. This is important in high-throughput applications. A particularly notable feature of our annotated streaming algorithm is that in uses $o(n)$ space which is impossible (even randomized) for this problem in the standard unannotated streaming model. In-fact at one point on the space-annotation trade-off that we present we use only $O(\lg n + m)$ space (where m is the pattern length) while still maintaining $O(\lg n)$ worst case time and annotation per stream element. Our result also holds for the widely used LZ78 compression scheme which is a special case of phrase-compression.

Annotated Data Structures. We also introduce the notion of an *annotated data structure* that is run by the client and the annotator in cooperation. The annotator "answers queries" in the annotated data stream, and the client maintains a small data structure for checking the validity of annotated query answers. Answers can be given by the annotator if the queries are specified only by the client algorithm and the input, meaning that the annotator can predict required answers though no two-way communication takes place. The data structure relies on using Karp-Rabin fingerprints on the client to ensure that

the annotator remains honest. As a result, all answers are correct with high probability. As the annotator has unbounded computational power, it is crucial that they do not have access to the random choices of the client.

5.1.1 The model

Before we give our results, we give a more detailed overview of the key features of the variant of the annotated streaming model that we consider. In each case we highlight how this compares to the standard annotated streaming model.

Multiple Queries. The standard annotated streaming model supports a single query - which occurs after the full input stream arrives. For streaming pattern matching problems, it is conventional and natural to require that occurrences of a pattern in the text are reported as they happen. That is, we consider each new element in the stream as a new query that must be answered before the next stream element arrival. Thus, to support solving pattern matching in the annotated model, we extend the model by allowing *multiple queries*, where queries and their answers may be interleaved with the input stream.

Annotations. In both the standard annotated streaming model and our variant, the annotations can be modeled as additional words which arrive between consecutive elements of the input stream. In the standard model, annotation is measured by the total number of words of annotation received by the client while processing the entire stream. In contrast we will give bounds on the worst-case annotation per element in the input stream. I.e. the maximum number of words of annotation received by the client between any two input elements. These annotation per element guarantees are important in high-throughput applications where queries are highly time sensitive as may well be the case for pattern matching problems. It is also important in applications where the arrival rate of the original stream cannot be controlled. There may simply not be time to receive many words of annotation before the next stream element arrives.

Prescience. In our variant of the annotated streaming model, we assume that the annotator is also *prescient* (as in parts of [Cha$^+$09; Cha$^+$14]), meaning that it has full access to the input stream in advance. In the case that the annotator also provides the input stream this is a very reasonable assumption, and the model we will use in this paper. Our motivation is that the client may require an annotated stream when receiving some input stream to be able to verify if the input stream is valid. For example, our scheme allows the client to perform pattern matching in a compressed input text. That is, the client can for example perform a streamed virus scan (using virus signatures as patterns) on a streamed input file. The result is that a malicious prescient annotator trying to infect the client can not make the client receive an infected file.

An interesting detail is that *any* prescient algorithm in the annotated streaming model can be made non-prescient relatively straightforwardly by blowing up the time before detecting a protocol inconsistency.

5.1.2 Our Results

As our main result, we solve the pattern matching problem in a phrase-compressed text stream where phrases arrive in order, one by one. The compression model is classic, with each phrase being either a single character or an extension of a previous phrase by a single character. This model subsumes for example the widely used LZ78 compression scheme.

This is the first result to show the power of one-way annotation in solving classic problems on strings, proving that the annotator allows us to reduce the space required by the client from linear in the stream length to logarithmic by using a logarithmic amount of annotation per phrase received. We give the following smooth trade-off for the problem:

Theorem 5.1.1. *Let $2 \leq B \leq n$. Given a text compressed into n phrases arriving in a stream and a pattern p of length m. We can maintain a structure in $O(B \lg_B n + m)$ space that allow us to determine if an occurrence of p ends in a newly arrived phrase in $O(\lg_B n)$ words of annotation and $O(\lg_B n)$ time per phrase. Our algorithm is randomised and all matches are output correctly with high probability (at least $1 - 1/n$).*

That is, we can solve the problem in $O(\lg n + m)$ space and $O(\lg n)$ time and annotation per phrase; or if spending $O(n^\varepsilon + m)$ space then $O(1)$ time and annotation per phrase suffices. In the standard streaming model, the problem has a $\Omega(n)$ randomised space lower bound, which show that one-way communication from an untrusted annotator can help in solving classic string problems.

The result is a careful application of techniques for pattern matching in compressed strings, providing a simple initial solution in linear space and constant time per phrase with no annotation. We reduce the space required using a new annotated data structure that allows us to store and access arbitrary information in logarithmic client space with logarithmic overhead.

Before giving the solution to Theorem 5.1.1, we give an interesting warm up with a solution to streamed multi-indexing, motivating and illustrating our solution scheme. From a high level, we solve the problems in three steps:

1. Construct a protocol for the annotation that must be sent by the annotator when an element arrives in the stream.

2. Give a client algorithm that uses the annotation to either solve the problem or to detect a protocol inconsistency before the next input arrives.

3. Store information about the current element for the future, and consistently retrieve required information about the past.

We show the existence of the following new data structure for the *streamed recovery* problem that generally allows us to trade client space for annotation when storing information about the stream. We believe this data structure to be of independent interest. We associate with each input element an automatically incremented timestamp t, and the problem is to maintain an array R with an entry for each timestamp. The operations are `attach`(i, x), which sets $R[i] = x$; and `recover`() returns the data associated with the current timestamp t. We show the following theorem:

Theorem 5.1.2. *Let $2 \leq B \leq n$. There is an annotated data structure for streamed recovery that requires $O(B \lg_B |R|)$ words of space, and executes operations in $O(\lg_B |R|)$ words of worst case annotation and $O(\lg_B |R|)$ time. The result is randomized and all operations are completed correctly with high probability (at least $1 - 1/n$).*

5.1.3 Related Work

Chakrabarti, Cormode and McGregor [Cha+09; Cha+14] introduced the annotated streaming model and gave solutions to a number of natural problems, showing the utility of the annotator in solving classic problems on streams, such as selecting the median element, calculating frequency moments and various graph problems. Assuming a helpful annotator, Cormode, Mitzenmacher and Thaler [Cor+13] show how to solve various graph problems such as connectivity, triangle-freeness, DAG determination, matchings and shortest paths. Semi-streaming solutions requiring superlinear space and annotation to solve triangle counting and computing maximal matchings was given by Thaler [Tha14].

Multiple papers [Cor+12; Kla+14; Cha+15; Kla+13] have considered a variant allowing interactive communication (where the client can query the annotator in a number of *rounds*). Chakrabarti et al. [Cha+15] showed that very little two-way communication is sufficient to solve problems such as nearest neighbour search, range counting and pattern matching, restricting the amount of interaction to be only a constant number of rounds. Klauck and Prakash [Kla+13] consider two-way communication, but similarly to our model variant restrict the amount of annotation spent per input element, giving solutions to the longest increasing subsequence problem (and others). To the best of our knowledge, there are no proposed solutions to any classic string problems where only one-way communication is allowed.

The annotated streaming model is related to a myriad of models from other fields where an untrusted annotator helps a client in solving problems (see e.g. [Göö+15; Bab85; Gol+86; Gol+85; Gol+91]). For example, in communication complexity an Arthur-Merlin Protocol [Bab85; Gol+86] model an all-powerful but untrustworthy Merlin that help Arthur to solve some problem probabilistically (using only public randomness). In cryptology, a related notion is that of Zero Knowledge Proofs [Gol+85; Gol+91], where a client must verify a proof by the annotator without obtaining any knowledge about the actual proof (here, private randomness is permitted).

Karp-Rabin fingerprints. Our work makes use of Karp and Rabin [Kar+87] fingerprints which were originally used to design a randomized string matching algorithm and

since have been used as a central tool to design algorithms for a wide range of problems (see e.g., [Col+03; Kal02; Por+09]). The Karp-Rabin fingerprint of a string is given by the following definition.

Definition 1 (Karp-Rabin fingerprint.)**.** *Let p be a prime and r a random integer in $\{1, 2, 3, ..., p-1\}$. The fingerprint function ϕ for a string S is given by:*

$$\phi(S) = \sum_{i=0}^{|S|-1} S[i]r^i \ mod \ p.$$

We will make extensive use of the following well-known properties. Given $\phi(S)$ and $\phi(S')$, we can compute the fingerprint of the concatentation $\phi(S \circ S')$ in $O(1)$ time. Given $\phi(S)$ and $\phi(S \circ S')$, we can compute the fingerprint $\phi(S')$ in $O(1)$ time. If $p > n^4$ then $\phi(S) = \phi(S')$ iff $S = S'$ with probability at least $1 - 1/n^3$. This is the only source of randomness in our results. For convenience in our algorithm descriptions and correctness we will assume that whenever a comparison between some $\phi(S)$ and $\phi(S')$ is made that $\phi(S) = \phi(S')$ iff $S = S'$. As our client algorithms run in sub-quadratic total time, by applying the union bound, we have that this assumption holds for all comparisons with probability at least $1 - 1/n$ when $p \geq n^4$.

5.2 Multi-Indexing

As an interesting warm-up before showing our main results, we show how to use Theorem 5.1.2 to solve the *multi-indexing* problem. The input stream consist of a sequence of input elements X and queries Q. The answer to an $\texttt{index}(i)$ query is the input element $X[i]$ (where i is an index in the input element sequence). Input elements and queries may be mixed (but queries must refer to the past). At any time $n = |X| + |Q|$ is the stream length so far.

In the standard streaming model (without annotation) it can be shown that even a randomized algorithm must use $\Omega(|X|)$ bits of space on the client. This follows almost immediately by the observation that a streaming algorithm for multi-indexing which uses $o(|X|)$ bits of space would give an $o(|X|)$ bit one-way communication protocol for the indexing problem. It is folklore that this is impossible. In the annotated streaming model without prescience, Chakrabarti et al. [Cha+09] gave a lower bound in the form of a space-annotation product of $\Omega(|X|)$ bits if $|Q| = 1$ (and an upper bound with $O(\sqrt{|X|})$ space and annotation). There are no better lower bounds for $|Q| > 1$ or when having access to prescience.

There are two simple solutions, one of which is using $O(1)$ space to store a Karp-Rabin Fingerprint of X. To answer an $\texttt{index}(i)$ query, the annotator must then replay all of X in $O(|X|)$ annotation, which allows the client to answer the query and verify that X was correctly replayed.

If the client instead stores the entire stream in $O(|X|)$ space, it is easy to answer a query in $O(1)$ time, by simply looking up the answer. This is the solution we will build on, using the data structure of Theorem 5.1.2 as black box storage of X to reduce the space use, resulting in the following trade-off:

Theorem 5.2.1. *Let $2 \leq B \leq n$. We can solve the multi-indexing problem in $O(B \lg_B n)$ space, using $O(\lg_B n)$ words of annotation and $O(\lg_B n)$ time per stream element. All queries are answered correctly with high probability (at least $1 - 1/n$).*

Clearly, we can answer a query $\texttt{index}(i)$ if we have access to element $X[i]$. Assume that on arrival of element $X[i]$ we know that at time t query $\texttt{index}(i)$ will arrive. This allows us to perform an `attach` operation for $X[i]$ at time t, and we can then use a `recover` query to get the element. If we have these timestamps for all queries, we clearly have $|R| = O(n)$ when using Theorem 5.1.2. As the annotator is prescient, it has all the timestamps for queries to $X[i]$ and can send them to the client when the element arrives in the stream. The resulting annotation takes $O(|Q|)$ words in total (but may be unevenly distributed). The remaining difficulty is to force the annotator to send the correct timestamps, and to only send one annotation timestamp per element.

We send the timestamps for queries to $X[i]$ one at a time by stringing together queries, only receiving the timestamp of the *next* query to an element each time we access it. The annotation protocol for a new element in the stream is:

- Let the new stream element be $x = X[i]$ or a query $\texttt{index}(i)$. Then the annotator must send the timestamp j of the next query[1] to item i. The client saves element x for timestamp j by performing an $\texttt{attach}(j, i \circ x)$ operation, where $\circ$ denotes concatenation of words.
- If the stream element is a query $\texttt{index}(i)$, the client first performs a $i' \circ x = \texttt{recover}()$ query to retrieve element x. We check if i' and i match and return x if so; otherwise we report a protocol inconsistency.

Note that when an item x is received in the stream, the client can correctly `attach` it, providing us with the ground truth in the chain of queries to the element. Observe that if the recovered i' does not match i for a query, the protocol was broken, as either the annotator told us a wrong future timestep for the next query to i or the annotated recovery data structure gave a wrong answer. In either case, we have an inconsistency.

5.3 Streamed Recovery

As previously defined, the *streamed recovery* problem is to maintain a data structure that allow us to $\texttt{attach}(i, x)$ some bitstring x to the arrival of a stream element at time i, and to `recover` the bitstring attached to the current stream element. We now give the proof for Theorem 5.1.2.

The overall idea in our solution is to maintain a small data structure on the client that is updated when performing `attach` operations and used to ensure that the answers to `recover` queries provided by the annotator are correct.

To simplify our presentation, we initially assume that all `attach` updates are performed first, followed by all `recover` queries. This assumption can be removed as shown

[1] The next query is the future query with the smallest timestamp.

later without increasing the time and space. Remember that R is the list of items to attach or recover, indexed by the timestamp of the items.

From a high level, we build a balanced B-tree T with out-degree B and R at the leaves, where each internal node have a data structure that allow consistency checking the leafs in its subtree. We let T_i refer to the leaf corresponding to $R[i]$. When an $\texttt{attach}(i, x)$ operation is performed all nodes on the path from T_i to the root are updated. By using the consistency checking data structures when a `recover` query is answered, we can check if the answer fit the expectation.

Since T is balanced with out degree B, it has height $O(\lg_B |R|)$. Each node $u \in T$ covers some interval of R. We use Karp-Rabin Fingerprints to check the consistency of subtrees, storing for each node $u \in T$ the $O(B)$ fingerprints for all prefixes of its children. Clearly, storing T and the fingerprints takes $O(|R|)$ space. However, using the annotator we can reduce the space required to $O(B \lg_B |R|)$ as `recover` queries only move forward in time (so there is no reason to store fingerprints to check the past or the distant future). This is done as follows.

At time t the client stores the fingerprints on a single root-to-leaf path from the root to T_t as well as all immediate children of that path. This path is called the *active fingerprint path* and denoted A_t. The active fingerprint path consist of $O(\lg_B |R|)$ layers of fingerprints with $O(B)$ fingerprints stored in each layer. Thus, the total space required is $O(B \lg_B |R|)$ at any time t.

Since time moves forward, the active fingerprint path starts at the leftmost root-to-leaf path of the tree and moves right through the leaves, each of which correspond to a single `recover` query. The fingerprints in T are constructed by the client when performing $\texttt{attach}(i, x)$ operations. The details are as follows. Note that for any t the active fingerprint path moves from T_t to T_{t+1} through a diamond transition path. Since it moves left to right we can find $p = lca(T_t, T_{t+1})$ on A_t. Let v_t be the child of p that A_t passes through. We know there is a right neighbour v_{t+1} of v_t that A_{t+1} must pass through. At time t the client stores all fingerprints for children of p and thus the full fingerprint for v_{t+1}. Thus, we can force the annotator to send us the correct list of leaves below v_{t+1}, checking the received items with our stored fingerprint. Furthermore, at the same time as receiving these items, we can build up the leftmost fingerprint path in the subtree rooted by v_{t+1}. That is, when we move the active fingerprint path, we can make sure that the annotator sends us the correct list of leaves below v_{t+1}.

The annotation as described transmits each leaf $O(\lg_B |R|)$ times, as it is sent once for each ancestor node in T. However, a lot of annotation may be sent per stream element. We can ensure $O(\lg_B |R|)$ annotated words per element with the following deamortization. The annotator must repeatedly send the items that should be recovered by the right neighbour node on each level of the tree. Transmission of the leaves in the subtree rooted by v is timed such that it ends when the active fingerprint path moves to v. That is, transmission of the leaves below v_{t+1} begin when the path moves to node v_t, where v_{t+1} is on the right of v_t and in the same level. The result of this is that we at each timestamp transmit $O(1)$ leaves for each of the $O(\lg_B |R|)$ levels in the tree.

Our final concern is to remove the requirement of non-interleaved attach and recover queries. In this case, we build T incrementally. This means that we may have already

transferred the leaf T_i (as an empty leaf) before an $\texttt{attach}(i, x)$ operation is performed. It is up to the client to later correct the fingerprint for the ancestors of T_i when an $\texttt{attach}(i, x)$ is made (so we can check recover queries correctly). This can be done by the client assuming the client can already decide where to $\texttt{attach}(i, x)$ items: it involves updating the fingerprints already stored and in transfer that covers item i, of which there are at most $O(\lg_B |R|)$. □

5.4 Compressed Pattern Matching

We now show how to use the power of the annotator to solve the pattern matching problem in a phrase-compressed text stream. The compressed phrases are on the form $i = (j, \sigma)$, which either extends a previous phrase j by an alphabet character σ, or starts a new chain of extending phrases if $j = -1$[2]. The problem is to find occurrences of an (uncompressed) pattern P of m alphabet characters in the uncompressed text as follows: for each arriving phrase we must output true iff there is an occurrence of P ending in the latest phrase. At any time n denotes the number of phrases we have seen so far.

In compressed pattern matching, the output when phrase n arrives can depend on phrases arbitrarily far in the past. This is is in contrast to well-studied uncompressed streaming pattern matching problems in which the output typically only depends on a window of length $O(m)$. More formally, we have that in the standard, unannotated streaming model, there is a space lower bound of $\Omega(n)$ for our problem. This follows via a reduction to the indexing problem with the pattern $P = 1$. We can use the first n phrases to encode a bit string of length n (by starting a new chain with every phrase). We can then 'query' any of these bits by appending the phrase $(j, 0)$ where j is the index to be queried. This lower bound also holds (with a little bit of fiddling) if the phrase scheme is restricted to be LZ78 compression [Ziv+78]. The details are given in the appendix.

We first give an algorithm which determines whether P is a substring of the latest phrase, we will then then extend this to find new pattern occurrences that cross phrase boundaries (but still end in the latest phrase). Before we do so, we briefly discuss some (mostly) standard pattern matching data structures that we store on the client for use in for our solution.

Additional client-side data structures. The following standard data structures use $O(m)$ space on the client and can be constructed during preprocessing in $O(m)$ time using standard methods[3]. We build a suffix tree for P [Wei73] with edges stored in nodes using perfect hashing [Fre+84].

We also build a set of m perfect static dictionaries. Each dictionary D_j is associated with a pattern prefix $P[0, j-1]$. The role of these dictionaries is to provide an analogue of the classic KMP prefix table. However, unlike the classic prefix table, this approach will lead to worst case constant processing times. Each entry in D_j is keyed by an alphabet symbol σ and associated with a length ℓ. Here ℓ is largest non-negative integer such

[2] This models most phrase-based compressors, such as LZ78.

[3] We assume a linear alphabet size.

that $P[0, \ell - 1] = P[j - \ell + 1, j - 1]$ and $P[\ell] = \sigma \neq P[j]$. The dictionary D_j contains every symbol for which ℓ is well-defined. This construction was also used in [Cli+12] (see Lemma 1) which is in turn a rephrasing of the original approach from [Sim93]. It was proven in [Sim93] that, surprisingly, the total size of all D_j summed over all $j \in [m]$ is only $O(m)$. Our perfect static dictionaries are constructed according to the FKS scheme [Fre+84] so lookup operations take $O(1)$ worst-case time.

Occurrences in the latest phrase. We first give a simple $O(n + m)$ client space and $O(1)$ time solution which does not use annotation. When each phrase $i = (j, \sigma)$ arrives, the output is $\texttt{match}(i)$ which is TRUE iff there is a match in phrase i. To determine this we also calculate the length of the longest suffix of phrase i that matches a prefix of P. This length is denoted $\texttt{pref}(i)$. We store both $\texttt{pref}(i)$ and $\texttt{match}(i)$ for each phrase seen so far in $O(n)$ space.

We compute $\texttt{pref}(i)$ from $\texttt{pref}(j)$ and σ using the dictionary D_j, in a similar way to the KMP algorithm. In particular if $\sigma = P[\texttt{pref}(j) + 1]$ then $\texttt{pref}(i) = \texttt{pref}(j) + 1$. Otherwise, we look up σ in D_j in $O(1)$ time. If σ is in the dictionary then $\texttt{pref}(i) = \ell + 1$. Otherwise, $\texttt{pref}(i) = 0$. This follows because both $P[0, \texttt{pref}(j)]$ and $P[0, \texttt{pref}(i)]$ are pattern prefixes.

To decide whether a match occurs in phrase i, we make the observation that,

$$\texttt{match}(i) = \text{TRUE} \text{ if and only if } (\texttt{pref}(i) = m \text{ or } \texttt{match}(j) = \text{TRUE}).$$

This follows because a match in phrase i is either completely contained in phrase j (in which case $\texttt{match}(j) = \text{TRUE}$) or ends at the final character of phrase j (in which case $\texttt{pref}(i) = m$). This completes the basic algorithm description.

To reduce the space to $O(B \lg_B n + m)$ we use the annotated recovery data structure that we gave in Theorem 5.1.2. From the description, we spend $O(n)$ space storing the facts $\texttt{pref}(i)$ and $\texttt{match}(i)$ for each phrase, while the client side data structures for processing phrases take $O(m)$ space in total. We reduce the first term to $O(B \lg_B n)$ by the observation that for a new phrase $i = (j, \sigma)$ we only ever require access to $\texttt{pref}(j)$ and $\texttt{match}(j)$, each of which can be obtained from the data structure of Theorem 5.1.2 using `recover` queries. Furthermore, when receiving phrase i the can send the timestamps of all phrases directly extending i in the annotated stream for use in attaching facts to these timestamps. Since a phrase i may be extended multiple times, the trick is to avoid sending the timestamps for all phrases that extend i at once.

We force the annotator to string together the extension timestamps as in the annotation scheme for multi-indexing as follows. The annotator must send two indices for phrase $i = (j, \sigma)$: the index j' of the next phrase extending j and the index i' of the first phrase extending i. We attach $\texttt{pref}(j)$ and $\texttt{match}(j)$ to phrase j', calculate $\texttt{pref}(i)$ and $\texttt{match}(i)$ as previously shown and attached those facts to phrase i'. As $O(n)$ attach and recover queries are made in total and all phrase processing besides the operations of Theorem 5.1.2 take constant time and $O(m)$ space, we obtain Theorem 5.4.1 below.

Theorem 5.4.1. *In a stream of n phrases of compressed text, one can find the phrases that contain a pattern of length m in $O(B \lg_B n + m)$ space, $O(\lg_B n)$ worst case words of annotation per phrase and $O(\lg_B n)$ time per phrase.*

We now give our main result by showing how to extend this to find matches that cross phrase boundaries. To simplify exposition, matches within phrase boundaries are still found as shown above. We will give our full algorithm in two versions. First we give a time efficient, $O(n+m)$ space solution which is based on existing techniques for the classic offline, unannotated version of the problem. Then we improve the space to be logarithmic by using our annotated recovery data structure.

Occurrences across phrase boundaries. In our first solution when phrase i arrived we computed $\texttt{pref}(i)$, the longest suffix of phrase i that matches a prefix of P. In addition we will also compute $\texttt{tpref}(i)$, the longest prefix of the pattern matching a suffix of the *text* up to and including phrase i. This is distinct from $\texttt{pref}(i)$ because it allows for prefix matches that cross phrase boundaries. In particular it is (conceptually) more difficult to compute because the prefix may cross many phrase boundaries. It is important to observe that for any phrase $i = (j, \sigma)$, in contrast to our previous in-phrase approach, a cross-boundary match in phrase i is not generally implied by $\texttt{tpref}(j) = m$. This is because $\texttt{tpref}(j)$ can extend to the left of phrase j and into a region of the text unrelated to phrase i.

To enable us to find cross-boundary matches efficiently, we will also compute $\texttt{sub}(i)$, the length of the longest substring of the pattern which matches a *prefix* of phrase i. We also store the location of an occurrence of this substring in P. In the first version we explicitly store $\texttt{pref}(j)$, $\texttt{tpref}(j)$ and $\texttt{sub}(j)$ (and its location) on the client for all phrases seen so far in $O(n)$ space.

The key observation is that when $i = (j, \sigma)$ arrives any new cross-boundary matches are completely contained within the portion of the text given by concatenating the strings corresponding to $\texttt{tpref}(i-1)$ and $\texttt{sub}(i)$. This follows immediately from the definitions of $\texttt{tpref}(i-1)$ and $\texttt{sub}(i)$. These are both substrings of P. Further, we can compute $\texttt{sub}(i)$ from $\texttt{sub}(j)$ (and its location) and σ. This can be done in $O(1)$ time using the suffix tree for the pattern. The details are straightforward and are given in the appendix for completeness.

As observed above, any new cross-boundary matches are contained within the concatenation of two pattern substrings. Therefore, we can apply Lemma 5.4.2 to find the substring cross-boundary matches in $O(1)$ time.

Lemma 5.4.2 (Gawrychowski [Gaw13], Lemma 3.1 rephrased)**.** *The pattern P can be preprocessed in $O(m)$ time and $O(m)$ client space to allow queries of the following form in $O(1)$ time: Given two substrings $P[i_1, j_1]$ and $P[i_2, j_2]$ decide whether P occurs within $P[i_1, j_1] \circ P[i_2, j_2]$.*

We now explain how to use our annotated recovery data structure to again improve the space used on the client to $O(B \lg_B n + m)$. As previously shown, we can attach and recover the required facts $\texttt{pref}$ and $\texttt{match}$ for each new phrase $i = (j, \sigma)$ by stringing

together the facts using annotated knowledge about the next references to i and j. We store the new facts `tpref` and `sub` in the exact same way, attaching the them to the next references to i and j. As we spend $O(n)$ space only on the facts and $O(m)$ on the remaining data structure, this concludes the full algorithm description and our proof of Theorem 5.1.1.

Appendix

LZ78 lower bound In LZ78 compression, the phrases given for the lower bound are an incorrect compression of the underlying string. In particular in LZ78 compression there cannot be two phrases which correspond to the same substring (except if one of them is the final phrase). This is because LZ78 phrases are defined to be maximal. That is when compressing a text the next phrase will represent the *longest* (valid) prefix of the uncompressed suffix of the text.

With an unbounded alphabet, a suitable lower bound for LZ78 is given as follows. The overall technique of reduction to indexing is the same. Start with n phrases each representing a single different symbol $x_1 \dots x_n$. Then encode the bit string B one bit at a time, with the i-th bit encoded as the phrase $2i = (i, B[i])$. The pattern is $P = 1$. A query to bit j is modeled by a phrase given by $(3j, 0)$. If there is a match we know the bit was a 1, otherwise 0. Similar (but more involved) constructions work for binary alphabets.

Computing $\texttt{sub}(i)$ from $\texttt{sub}(j)$. If $\texttt{sub}(j)$ is shorter than the length of phrase j then $\texttt{sub}(i) = \texttt{sub}(j)$[4]. Otherwise we must decide whether phrase i is a substring of the pattern. This can be achieved as follows. The location of the string corresponding to $\texttt{sub}(j)$ is stored as the location in the suffix tree for P. We can then extend this in constant time by looking for σ on an edge. Note that $\texttt{sub}(i)$ does not contribute new matches besides $\texttt{pref}(i)$ by itself, as if $|\texttt{sub}(i)| = m$ then we must have $\texttt{pref}(i) = m$ as well.

[4]No new symbol make a longer substring of P match as there is a previous mismatch.

6 Output-Sensitive Pattern Extraction in Sequences

Roberto Grossi[1]* Giulia Menconi[1] Nadia Pisanti[1]
Roberto Trani[1] Søren Vind[2†]

[1] Università di Pisa
[2] Technical University of Denmark

Abstract

Genomic Analysis, Plagiarism Detection, Data Mining, Intrusion Detection, Spam Fighting and Time Series Analysis are just some examples of applications where extraction of recurring patterns in sequences of objects is one of the main computational challenges. Several notions of patterns exist, and many share the common idea of strictly specifying some parts of the pattern and to *don't care* about the remaining parts. Since the number of patterns can be exponential in the length of the sequences, *pattern extraction* focuses on statistically relevant patterns, where any attempt to further refine or extend them causes a loss of significant information (where the number of occurrences changes). Output-sensitive algorithms have been proposed to enumerate and list these patterns, taking polynomial time $O(n^c)$ per pattern for constant $c > 1$, which is impractical for massive sequences of very large length n.

We address the problem of extracting maximal patterns with at most k don't care symbols and at least q occurrences. Our contribution is to give the first algorithm that attains a *stronger* notion of output-sensitivity, borrowed from the analysis of data structures: the cost is proportional to the *actual* number of occurrences of each pattern, which is at most n and practically much smaller than n in real applications, thus avoiding the aforementioned cost of $O(n^c)$ per pattern.

*Partially supported by Italian MIUR PRIN project AMANDA.
†Supported by a grant from the Danish National Advanced Technology Foundation.

6.1 Introduction

In *pattern extraction*, the task is to extract the "most important" and frequently occurring patterns from sequences of "objects" such as log files, time series, text documents, datasets or DNA sequences. Each individual object can be as simple as a character from $\{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$ or as complex as a `json` record from a log file. What is of interest to us is the potentially very large set of all possible different objects, which we call the *alphabet* Σ, and sequence S built with n objects drawn from Σ.

We define the occurrence of a pattern in S as in *pattern matching* but its importance depends on its statistical relevance, namely, if the number of occurrences is above a certain threshold. However, pattern extraction is not to be confused with pattern matching. The problems may be considered inverse of each other: the former gets an input sequence S from the user, and extracts patterns P and their occurrences from S, where both are unknown to the user; the latter gets S and a given pattern P from the user, and searches for P's occurrences in S, and thus only the pattern occurrences are unknown to the user.

Many notions of patterns exist, reflecting the diverse applications of the problem [Gro+11; Ari+07; Sag98; Ukk09]. We study a natural variation allowing the special don't care character $\star$ in a pattern to mean that the position inside the pattern occurrences in S can be ignored (so $\star$ matches any single character in S). For example, $\mathtt{TA}\star\mathtt{C}\star\mathtt{ACA}\star\mathtt{GTG}$ is a pattern for DNA sequences.

A *motif* is a pattern of *any* length with *at most k don't cares* occurring *at least q times* in S. In this paper, we consider the problem of determining the *maximal* motifs, where any attempt to extend them or replace their $\star$'s with symbols from Σ causes a loss of significant information (where the number of occurrences in S changes). We denote the family of all motifs by M_{qk}, the set of maximal motifs $\mathcal{M} \subseteq M_{qk}$ (dropping the subscripts in $\mathcal{M}$) and let $\mathrm{occ}(m)$ denote the number of occurrences of a motif m inside S. It is well known that M_{qk} can be exponentially larger than $\mathcal{M}$ [Par+00].

Our Results We show how to efficiently build an index that we call a *motif trie* which is a trie that contains all prefixes, suffixes and occurrences of $\mathcal{M}$, and we show how to extract $\mathcal{M}$ from it. The motif trie is built level-wise, using an oracle $\textsc{Generate}(u)$ that reveals the children of a node u efficiently using properties of the motif alphabet and a bijection between new children of u and intervals in the ordered sequence of occurrences of u. We are able to bound the resulting running time with a strong notion of *output-sensitive* cost, borrowed from the analysis of data structures, where the cost is proportional to the *actual* number $\mathrm{occ}(m)$ of occurrences of each maximal motif m.

Theorem 6.1.1. *Given a sequence S of n objects over an alphabet Σ, and two integers $q > 1$ and $k \geq 0$, there is an algorithm for extracting the maximal motifs $\mathcal{M} \subseteq M_{qk}$ and their occurrences from S in $O\left(n(k + \lg\Sigma) + (k+1)^3 \times \sum_{m\in\mathcal{M}} \mathrm{occ}(m)\right)$ time.*

Our result may be interesting for several reasons. First, observe that this is an optimal listing bound when the maximal number of don't cares is $k = O(1)$, which is true in many practical applications. The resulting bound is $O(n \lg\Sigma + \sum_{m\in\mathcal{M}} \mathrm{occ}(m))$ time, where the

first additive term accounts for building the motif trie and the second term for discovering and reporting all the occurrences of each maximal motif.

Second, our bound provides a strong notion of output-sensitivity since it depends on how many times each maximal motif occurs in S. In the literature for enumeration, an output-sensitive cost traditionally means that there is polynomial cost of $O(n^c)$ per pattern, for a constant $c > 1$. This is infeasible in the context of big data, as n can be very large, whereas our cost of $\text{occ}(m) \leq n$ compares favorably with $O(n^c)$ per motif m, and $\text{occ}(m)$ can be actually much smaller than n in practice. This has also implications in what we call "the `CTRL-C` argument," which ensures that we can safely stop the computation for a *specific* sequence S if it is taking too much time[1]. Indeed, if much time is spent with our solution, too many results to be really useful may have been produced. Thus, one may stop the computation and refine the query (change q and k) to get better results. On the contrary, a non-output-sensitive algorithm may use long time without producing any output: It does not indicate if it may be beneficial to interrupt and modify the query.

Third, our analysis improves significantly over the brute-force bound: M_{qk} contains pattern candidates of lengths p from 1 to n with up to $\min\{k, p\}$ don't cares, and so has size $\sum_p |\Sigma|^p \times (\sum_{i=1}^{\min\{k,p\}} \binom{p}{i}) = O(|\Sigma|^n n^k)$. Each candidate can be checked in $O(nk)$ time (e.g. string matching with k mismatches), or $O(k)$ time if using a data structure such as the suffix tree [Sag98]. In our analysis we are able to remove both of the nasty exponential dependencies on $|\Sigma|$ and n in $O(|\Sigma|^n n^k)$. In the current scenario where implementations are fast in practice but skip worst-case analysis, or state the latter in pessimistic fashion equivalent to the brute-force bound, our analysis could explain why several previous algorithms are fast in practice. (We have implemented a variation of our algorithm that is very fast in practice.)

Related Work Although the literature on pattern extraction is vast and spans many different fields of applications with various notation, terminology and variations, we could not find time bounds explicitly stated obeying our stronger notion of output-sensitivity, even for pattern classes different from ours. Output-sensitive solutions with a polynomial cost per pattern have been previously devised for slightly different notions of patterns. For example, Parida et al. [Par+01] describe an enumeration algorithm with $O(n^2)$ time per maximal motif plus a bootstrap cost of $O(n^5 \lg n)$ time. [2] Arimura and Uno obtain a solution with $O(n^3)$ delay per maximal motif where there is no limitations on the number of don't cares [Ari+07]. Similarly, the MADMX algorithm [Gro+11] reports dense motifs, where the ratio of don't cares and normal characters must exceed some threshold, in time $O(n^3)$ per maximal dense motif. Our stronger notion of output-sensitivity is borrowed from the design and analysis of data structures, where it is widely employed. For example, searching a pattern P in S using the suffix tree [McC76] has cost proportional to P's length and its number of occurrences. A one-dimensional query in a sorted

[1] Such an algorithm is also called an anytime algorithm in the literature.

[2] The set intersection problem (SIP) in appendix A of [Par+01] requires polynomial time $O(n^2)$: The recursion tree of depth $\leq n$ can have unary nodes, and each recursive call requires $O(n)$ to check if the current subset has been already generated.

array reports all the wanted keys belonging to a range in time proportional to their number plus a logarithmic cost. Therefore it seemed natural to us to extend this notion to enumeration algorithms also.

Applications Although the pattern extraction problem has found immediate applications in stringology and biological sequences, it is highly multidisciplinary and spans a vast number of applications in different areas. This situation is similar to the one for the edit distance problem and dynamic programming. We here give a short survey of some significant applications, but others are no doubt left out due to the difference in terminology used (see [Abo+10] for further references). In computational biology, motif discovery in biological sequences identifies areas of interest[Sag98; Ukk09; Gro+11; Abo+10]. Computer security researches use patterns in log files to perform intrusion detection and find attack signatures based on their frequencies [Deb+99], while commercial anti-spam filtering systems use pattern extraction to detect and block SPAM [Rig+04]. In the data mining community pattern extraction is used extensively [Mab+10] as a core method in web page content extraction [Cha+03] and time series analysis [Pic+06; She+06]. In plagiarism detection finding recurring patterns across a (large) number of documents is a core primitive to detect if significant parts of documents are plagiarized [Bri+95] or duplicated [Bak95; Che+04]. And finally, in data compression extraction of the common patterns enables a compression scheme that competes in efficiency with well-established compression schemes [Apo+06].

As the motif trie is an index, we believe that it may be of independent interest for storing similar patterns across similar strings. Our result easily extends to real-life applications requiring a solution with two thresholds for motifs, namely, on the number of occurrences in a sequence and across a minimum number of sequences.

Reading Guide Our solution has two natural parts. In Section 6.3 we define the *motif trie*, which is an index storing all maximal motifs and their prefixes, suffixes and occurrences. We show how to report only the maximal motifs in time linear in the size of the trie. That is, it is easy to extract the maximal motifs from the motif trie – the difficulty is to build the motif trie without knowing the motifs in advance. In Section 6.4 and 6.5 we give an efficient algorithm for constructing the motif trie and bound its construction time by the number of occurrences of the maximal motifs, thereby obtaining an output-sensitive algorithm.

6.2 Preliminaries

Strings We let Σ be the alphabet of the input string $S \in \Sigma^*$ and $n = |S|$ be its length. For $1 \leq i \leq j \leq n$, $S[i,j]$ is the substring of S between index i and j, both included. $S[i,j]$ is the empty string ε if $i > j$, and $S[i] = S[i,i]$ is a single character. Letting $1 \leq i \leq n$, a prefix or suffix of S is $S[1,i]$ or $S[i,n]$, respectively. We let prefset(S) be the set of all prefixes of S. The *longest common prefix* $lcp(x,y)$ is the longest string such that $x[1,|lcp(x,y)|] = y[1,|lcp(x,y)|]$ for any two strings $x, y \in \Sigma^*$.

String	TACTGACACTGCCGA
Quorum	$q = 2$
Don't cares	$k = 1$

(a) Input and parameters for example.

Maximal Motif	**Occurrence List**
A	2, 6, 8, 15
AC	2, 6, 8
ACTG⋆C	2, 8
C	3, 7, 9, 12, 13
G	5, 11, 14
GA	5, 14
G⋆C	5, 11
T	1, 4, 10
T⋆C	1, 10

(b) Output: Maximal motifs found (and their occurrence list) for the given input.

Figure 6.1: Example 1: Maximal Motifs found in string.

Tries A trie T over an alphabet Π is a rooted, labeled tree, where each edge (u, v) is labeled with a symbol from Π. All edges to children of node $u \in T$ must be labeled with distinct symbols from Π. We may consider node $u \in T$ as a string generated over Π by spelling out characters from the root on the path towards u. We will use u to refer to both the node and the string it encodes, and $|u|$ to denote its string length. A property of the trie T is that for any string $u \in T$, it also stores all prefixes of u. A compacted trie is obtained by compacting chains of unary nodes in a trie, so the edges are labeled with substrings: the suffix tree for a string is special compacted trie that is built on all suffixes of the string [McC76].

Motifs A motif $m \in \Sigma(\Sigma \cup \{\star\})^* \Sigma$ consist of symbols from Σ and *don't care characters* $\star \notin \Sigma$. We let the length $|m|$ denote the number of symbols from $\Sigma \cup \{\star\}$ in m, and let $\mathrm{dc}(m)$ denote the number of $\star$ characters in m. Motif m *occurs* at position p in S iff $m[i] = S[p + i - 1]$ or $m[i] = \star$ for all $1 \leq i \leq |m|$. The number of occurrences of m in S is denoted $\mathrm{occ}(m)$. Note that appending $\star$ to either end of a motif m does not change $\mathrm{occ}(m)$, so we assume that motifs starts and ends with symbols from Σ. A *solid block* is a maximal (possibly empty ε) substring from Σ^* inside m.

We say that a motif m can be *extended* by adding don't cares and characters from Σ to either end of m. Similarly, a motif m can be *specialized* by replacing a don't care $\star$ in m with a symbol $c \in \Sigma$. An example is shown in Figure 6.1.

Maximal Motifs Given an integer quorum $q > 1$ and a maximum number of don't cares $k \geq 0$, we define a family of motifs M_{qk} containing motifs m that have a limited number of don't cares $\mathrm{dc}(m) \leq k$, and occurs frequently $\mathrm{occ}(m) \geq q$. A *maximal motif* $m \in M_{qk}$ cannot be extended or specialized into another motif $m' \in M_{qk}$ such that $\mathrm{occ}(m') =$

occ(m). Note that extending a maximal motif m into motif $m'' \notin M_{qk}$ may maintain the occurrences (but have more than k don't cares). We let $\mathcal{M} \subseteq M_{qk}$ denote the *set of maximal motifs*.

Motifs $m \in M_{qk}$ that are *left-maximal* or *right-maximal* cannot be specialized or extended on the left or right without decreasing the number of occurrences, respectively. They may, however, be prefix or suffix of another (possibly maximal) $m' \in M_{qk}$, respectively.

Fact 1. *If motif $m \in M_{qk}$ is right-maximal then it is a suffix of a maximal motif.*

6.3 Motif Tries and Pattern Extraction

This section introduces the *motif trie*. This trie is not used for searching but its properties are exploited to orchestrate the search for maximal motifs in $\mathcal{M}$ to obtain a strong output-sensitive cost.

6.3.1 Efficient Representation of Motifs

We first give a few simple observations that are key to our algorithms. Consider a suffix tree built on S over the alphabet Σ, which can be done in $O(n \lg |\Sigma|)$ time. It is shown in [Ukk09; Fed$^+$09] that when a motif m is maximal, its solid blocks correspond to nodes in the suffix tree for S, matching their substrings from the root[3]. For this reason, we introduce a new alphabet, the *solid block alphabet* Π of size at most $2n$, consisting of the strings stored in all the suffix tree nodes.

We can write a maximal motif $m \in M_{qk}$ as an alternating sequence of $\leq k+1$ solid blocks and $\leq k$ don't cares, where the first and last solid block must be different from ε. Thus we represent m as a sequence of $\leq k+1$ strings from Π since the don't cares are implicit. By traversing the suffix tree nodes in *preorder* we assign integers to the strings in Π, allowing us to assume that $\Pi \subseteq [1, \ldots, 2n]$, and so each motif $m \in M_{qk}$ is actually represented as a sequence of $\leq k+1$ integers from 1 to $|\Pi| = O(n)$. Note that the order on the integers in Π shares the following grouping property with the strings over Σ.

Lemma 6.3.1. *Let A be an array storing the sorted alphabet Π. For any string $x \in \Sigma^*$, the solid blocks represented in Π and sharing x as a common prefix, if any, are grouped together in A in a contiguous segment $A[i, j]$ for some $1 \leq i \leq j \leq |\Pi|$.*

When it is clear from its context, we will use the shorthand $x \in \Pi$ to mean equivalently a string x represented in Π or the integer x in Π that represents a string stored in a suffix tree node. We observe that the set of strings represented in Π is *closed* under the longest common prefix operation: for any $x, y \in \Pi$, $lcp(x, y) \in \Pi$ and it may be computed in constant time after augmenting the suffix tree for S with a lowest common ancestor data structure [Har$^+$84].

Summing up, the above relabeling from Σ to Π only requires the string $S \in \Sigma^*$ and its suffix tree augmented with lowest common ancestor information.

[3]The proofs in [Ukk09; Fed$^+$09] can be easily extended to our notion of maximality.

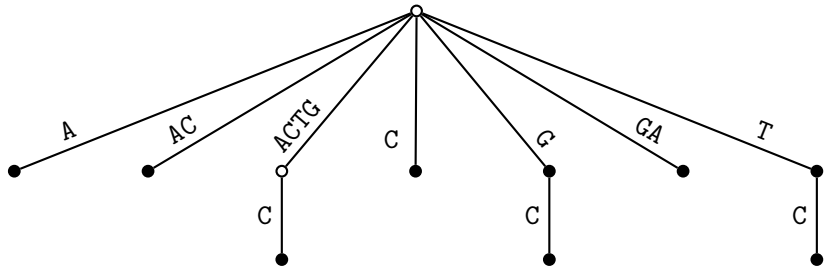


Figure 6.2: Motif trie for Example 1. The black nodes are maximal motifs (with their occurrence lists shown in Figure 6.1(b)).

6.3.2 Motif Tries

We now exploit the machinery on alphabets described in Section 6.3.1. For the input sequence S, consider the family M_{qk} defined in Section 6.2, where each m is seen as a string $m = m[1,\ell]$ of $\ell \leq k+1$ integers from 1 to $|\Pi|$. Although each m can contain $O(n)$ symbols from Σ, we get a benefit from treating m as a short string over Π: unless specified otherwise, the prefixes and suffixes of m are respectively $m[1,i]$ and $m[i,\ell]$ for $1 \leq i \leq \ell$, where $\ell = \mathrm{dc}(m)+1 \leq k+1$. This helps with the following definition as it does not depend on the $O(n)$ symbols from Σ in a maximal motif m but it solely depends on its $\leq k+1$ length over Π.

Definition 2 (Motif Trie)**.** *A* motif trie T *is a trie over alphabet* Π *which stores all maximal motifs* $\mathcal{M} \subseteq M_{qk}$ *and their suffixes.*

As a consequence of being a trie, T implicitly stores all prefixes of all the maximal motifs and edges in T are labeled using characters from Π. Hence, all sub-motifs of the maximal motifs are stored in T, and the motif trie can be essentially seen as a generalized suffix trie[4] storing $\mathcal{M}$ over the alphabet Π. From the definition, T has $O((k+1)\cdot|\mathcal{M}|)$ leaves, the total number of nodes is $O(|T|) = O((k+1)^2\cdot|\mathcal{M}|)$, and the height is at most $k+1$.

We may consider a node u in T as a string generated over Π by spelling out the $\leq k+1$ integers from the root on the path towards u. To decode the motif stored in u, we retrieve these integers in Π and, using the suffix tree of S, we obtain the corresponding solid blocks over Σ and insert a don't care symbol between every pair of consecutive solid blocks. When it is clear from the context, we will use u to refer to *(1)* the node u or *(2)* the string of integers from Π stored in u, or *(3)* the corresponding motif from $(\Sigma \cup \{\star\})^*$. We reserve the notation $|u|$ to denote the length of motif u as the number of characters from $\Sigma \cup \{\star\}$. Each node $u \in T$ stores a list L_u of occurrences of motif u in S, i.e. u occurs at p in S for $p \in L_u$.

[4]As it will be clear later, a compacted motif trie does not give any advantage in terms of the output-sensitive bound compared to the motif trie.

Since child edges for $u \in T$ are labeled with solid blocks, the child edge labels may be prefixes of each other, and one of the labels may be the empty string ε (which corresponds to having two neighboring don't cares in the decoded motif).

6.3.3 Reporting Maximal Motifs using Motif Tries

Suppose we are given a motif trie T but we do not know which nodes of T store the maximal motifs in S. We can identify and report the maximal motifs in T in $O(|T|) = O((k+1)^2 \cdot |\mathcal{M}|) = O((k+1)^2 \cdot \sum_{m \in \mathcal{M}} \text{occ}(m))$ time as follows.

We first identify the set R of nodes $u \in T$ that are right-maximal motifs. A characterization of right-maximal motifs in T is relatively simple: we choose a node $u \in T$ if *(i)* its parent edge label is not ε, and *(ii)* u has no descendant v with a non-empty parent edge label such that $|L_u| = |L_v|$. By performing a bottom-up traversal of nodes in T, computing for each node the length of the longest list of occurrences for a node in its subtree with a non-empty edge label, it is easy to find R in time $O(|T|)$ and by Fact 1, $|R| = O((k+1) \cdot |\mathcal{M}|)$.

Next we perform a radix sort on the set of pairs $\langle |L_u|, \text{reverse}(u) \rangle$, where $u \in R$ and $\text{reverse}(u)$ denotes the reverse of the string u, to select the motifs that are also left-maximal (and thus are maximal). In this way, the suffixes of the maximal motifs become prefixes of the reversed maximal motifs. By Lemma 6.3.1, those motifs sharing common prefixes are grouped together consecutively. However, there is a caveat, as one maximal motif m' could be a suffix of another maximal motif m and we do not want to drop m': in that case, we have that $|L_m| \neq |L_{m'}|$ by the definition of maximality. Hence, after sorting, we consider consecutive pairs $\langle |L_{u_1}|, \text{reverse}(u_1) \rangle$ and $\langle |L_{u_2}|, \text{reverse}(u_2) \rangle$ in the order, and eliminate u_1 iff $|L_{u_1}| = |L_{u_2}|$ and u_1 is a suffix of u_2 in time $O(k+1)$ per pair (i.e. prefix under reverse). The remaining motifs are maximal.

6.4 Building Motif Tries

The goal of this section is to show how to efficiently build the motif trie T discussed in Section 6.3.2. Suppose without loss of generality that enough new symbols are prepended and appended to the sequence S to avoid border cases. We want to store the maximal motifs of S in T as strings of length $\leq k+1$ over Π. Some difficulties arise as we do not know in advance which are the maximal motifs. Actually, we plan to find them *during* the output-sensitive construction of T, which means that we would like to obtain a construction bound close to the term $\sum_{m \in \mathcal{M}} \text{occ}(m)$ stated in Theorem 6.1.1.

We proceed in top-down and level-wise fashion by employing an *oracle* that is invoked on each node u on the last level of the partially built trie, and which reveals the future children of u. The oracle is executed many times to generate T level-wise starting from its root u with $L_u = \{1, \ldots, n\}$, and stopping at level $k+1$ or earlier for each root-to-node path. Interestingly, this sounds like the wrong way to do anything efficiently, e.g. it is a slow way to build a suffix tree, however the oracle allows us to amortize the total cost to construct the trie. In particular, we can bound the total cost by the total number of occurrences of the maximal motifs stored in the motif trie.

The oracle is implemented by the GENERATE(u) procedure that generates the children $u_1, \ldots, u_d$ of u. We ensure that *(i)* GENERATE(u) operates on the $\leq k+1$ length motifs from Π, and *(ii)* GENERATE(u) avoids generating the motifs in $M_{qk} \setminus \mathcal{M}$ that are not suffixes or prefixes of maximal motifs. This is crucial, as otherwise we cannot guarantee output-sensitive bounds because M_{qk} can be exponentially larger than $\mathcal{M}$.

In Section 6.5 we will show how to implement GENERATE(u) and prove:

Lemma 6.4.1. *Algorithm* GENERATE(u) *produces the children of u and can be implemented in time $O(\text{sort}(L_u) + (k+1) \cdot |L_u| + \sum_{i=1}^{d} |L_{u_i}|)$.*

By summing the cost to execute procedure GENERATE(u) for all nodes $u \in T$, we now bound the construction time of T. Observe that when summing over T the formula stated in Lemma 6.4.1, each node exists once in the first two terms and once in the third term, so the latter can be ignored when summing over T (as it is dominated by the other terms)

$$\sum_{u \in T} (\text{sort}(L_u) + (k+1) \cdot |L_u| + \sum_{i=1}^{d} |L_{u_i}|) = O\left(\sum_{u \in T} (\text{sort}(L_u) + (k+1) \cdot |L_u|) \right) .$$

We bound

$$\sum_{u \in T} \text{sort}(L_u) = O\left(n(k+1) + \sum_{u \in T} |L_u| \right)$$

by running a single cumulative radix sort for all the instances over the several nodes u at the same level, allowing us to amortize the additive cost $O(n)$ of the radix sorting among nodes at the same level (and there are at most $k+1$ such levels).

To bound $\sum_{u \in T} |L_u|$, we observe $\sum_i |L_{u_i}| \geq |L_u|$ (as trivially the ε extension always maintains the number of occurrences of its parent). Consequently we can charge each leaf u the cost of its $\leq k$ ancestors, so

$$\sum_{u \in T} |L_u| = O\left((k+1) \times \sum_{\text{leaf } u \in T} |L_u| \right) .$$

Finally, from Section 6.3.2 there cannot be more leaves than maximal motifs in $\mathcal{M}$ and their suffixes, and the occurrence lists of maximal motifs dominate the size of the non-maximal ones in T, which allows us to bound:

$$(k+1) \times \sum_{\text{leaf } u \in T} |L_u| = O\left((k+1)^2 \times \sum_{m \in \mathcal{M}} \text{occ}(m) \right) .$$

Adding the $O(n \lg \Sigma)$ cost for the suffix tree and the LCA ancestor data structure of Section 6.3.1, we obtain:

Theorem 6.4.2. *Given a sequence S of n objects over an alphabet Σ and two integers $q > 1$ and $k \geq 0$, a motif trie containing the maximal motifs $\mathcal{M} \subseteq M_{qk}$ and their occurrences* occ(m) *in S for $m \in \mathcal{M}$ can be built in time and space $O\Big(n(k + \lg \Sigma) + (k+1)^3 \times \sum_{m \in \mathcal{M}} \text{occ}(m)\Big)$.*

6.5 Implementing GENERATE(u)

We now show how to implement GENERATE(u) in the time bounds stated by Lemma 6.4.1. The idea is as follows. We first obtain E_u, which is an array storing the occurrences in L_u, sorted lexicographically according to the suffix associated with each occurrence. We can then show that there is a bijection between the children of u and a set of maximal intervals in E_u. By exploiting the properties of these intervals, we are able to find them efficiently through a number of scans of E_u. The bijection implies that we thus efficiently obtain the new children of u.

6.5.1 Nodes of the Motif Trie as Maximal Intervals

The key point in the efficient implementation of the oracle GENERATE(u) is to relate each node u and its future children $u_1, \ldots, u_d$ labeled by solid blocks $b_1, \ldots, b_d$, respectively, to some suitable intervals that represent their occurrence lists $L_u, L_{u_1}, \ldots, L_{u_d}$. Though the idea of using intervals for representing trie nodes is not new (e.g. in [Abo$^+$04]), we use intervals to expand the trie rather than merely representing its nodes. Not all intervals generate children as not all solid blocks that extend u necessarily generate a child. Also, some of the solid blocks $b_1, \ldots, b_d$ can be prefixes of each other and one of the intervals can be the empty string ε. To select them carefully, we need some definitions and properties.

Extensions. For a position $p \in L_u$, we define its *extension* as the suffix $\mathrm{ext}(p,u) = S[p+|u|+1, n]$ that starts at the position after p with an offset equivalent to skipping the prefix matching u plus one symbol (for the don't care). We may write $\mathrm{ext}(p)$, omitting the motif u if it is clear from the context. We also say that the *skipped characters* $\mathrm{skip}(p)$ at position $p \in L_u$ are the $d = \mathrm{dc}(u)+2$ characters in S that specialize u into its occurrence p: formally, $\mathrm{skip}(p) = \langle c_0, c_1, \ldots, c_{d-1} \rangle$ where $c_0 = S[p-1]$, $c_{d-1} = S[p+|u|]$, and $c_i = S[p+j_i-1]$, for $1 \le i \le d-2$, where $u[j_i] = \star$ is the ith don't care in u.

We denote by E_u the list L_u sorted using as keys the integers for $\mathrm{ext}(p)$ where $p \in L_u$. (We recall from Section 6.3.1 that the suffixes are represented in the alphabet Π, and thus $\mathrm{ext}(p)$ can be seen as an integer in Π.) By Lemma 6.3.1 consecutive positions in E_u share common prefixes of their extensions. Lemma 6.5.1 below states that these prefixes are the candidates for being correct edge labels for expanding u in the trie.

Lemma 6.5.1. *Let u_i be a child of node u, b_i be the label of edge (u,u_i), and $p \in L_u$ be an occurrence position. If position $p \in L_{u_i}$ then b_i is a prefix of $\mathrm{ext}(p,u)$.*

Proof. Assume otherwise, so $p \in L_u \cap L_{u_i}$ but $b_i \notin \mathrm{prefset}(\mathrm{ext}(p,u))$. Then there is a mismatch of solid block b_i in $\mathrm{ext}(p,u)$, since at least one of the characters in b_i is not in $\mathrm{ext}(p,u)$. But this means that u_i cannot occur at position p, and consequently $p \notin L_{u_i}$, which is a contradiction. □

Intervals. Lemma 6.5.1 states a necessary condition, so we have to filter the candidate prefixes of the extensions. We use the following notion of intervals to facilitate this task. We call $I \subseteq E_u$ an *interval* of E_u if I contains consecutive entries of E_u. We write $I = [i, j]$ if I covers the range of indices from i to j in E_u. The *longest common prefix* of an interval is defined as $\text{LCP}(I) = \min_{p_1,p_2 \in I} lcp(\text{ext}(p_1), \text{ext}(p_2))$, which is a solid block in Π as discussed at the end of Section 6.3.1. By Lemma 6.3.1, $\text{LCP}(I) = lcp(\text{ext}(E_u[i]), \text{ext}(E_u[j]))$ can be computed in $O(1)$ time, where $E_u[i]$ is the first and $E_u[j]$ the last element in $I = [i, j]$.

Maximal Intervals. An interval $I \subseteq E_u$ is *maximal* if *(1)* there are at least q positions in I (i.e. $|I| \geq q$), *(2)* motif u cannot be specialized with the skipped characters in $\text{skip}(p)$ where $p \in I$, and *(3)* any other interval $I' \subseteq E_u$ that strictly contains I has a shorter common prefix (i.e. $|\text{LCP}(I')| < |\text{LCP}(I)|$ for $I' \supset I$) [5]. We denote by $\mathcal{I}_u$ the *set of all maximal intervals* of E_u, and show that $\mathcal{I}_u$ form a tree covering of E_u. A similar lemma for intervals over the LCP array of a suffix tree was given in [Abo+04].

Lemma 6.5.2. *Let $I_1, I_2 \in \mathcal{I}_u$ be two maximal intervals, where $I_1 \neq I_2$ and $|I_1| \leq |I_2|$. Then either I_1 is contained in I_2 with a longer common prefix (i.e. $I_1 \subset I_2$ and $|\text{LCP}(I_1)| > |\text{LCP}(I_2)|$) or the intervals are disjoint (i.e. $I_1 \cap I_2 = \emptyset$).*

Proof. Let $I_1 = [i, j]$ and $I_2 = [i', j']$. Assume partial overlaps are possible, $i' \leq i \leq j' < j$, to obtain a contradiction. Since $|\text{LCP}(I_1)| \geq |\text{LCP}(I_2)|$, the interval $I_3 = [j', j]$ has a longest common prefix $|\text{LCP}(I_3)| \geq |\text{LCP}(I_2)|$, and so I_2 could have been extended and was not maximal, giving a contradiction. The remaining cases are symmetric. □

The next lemma establishes a useful bijection between maximal intervals $\mathcal{I}_u$ and children of u, motivating why we use intervals to expand the motif trie.

Lemma 6.5.3. *Let u_i be a child of a node u. Then the occurrence list L_{u_i} is a permutation of a maximal interval $I \subseteq \mathcal{I}_u$, and vice versa. The label on edge (u, u_i) is the solid block $b_i = \text{LCP}(I)$. No other children or maximal intervals have this property with u_i or I.*

Proof. We prove the statement by assuming that T has been built, and that the maximal intervals have been computed for a node $u \in T$.

We first show that given a maximal interval $I \in \mathcal{I}_u$, there is a single corresponding child $u_i \in T$ of u. Let $b_i = \text{LCP}(I)$ denote the longest common prefix of occurrences in I, and note that b_i is distinct among the maximal intervals in $\mathcal{I}_u$. Also, since b_i is a common prefix for all occurrence extensions in I, the motif $u \star b_i$ occurs at all locations in I (as we know that u occurs at those locations). Since $|I| \geq q$ and $u \star b_i$ is an occurrence at all $p \in I$, there must be a child u_i of u, where the edge (u, u_i) is labeled b_i and where $I \subseteq L_{u_i}$. From the definition of tries, there is at most one such node. There can be no $p' \in L_{u_i} - I$, since that would mean that an occurrence of $u \star b_i$ was not stored in I, contradicting the maximality assumption of I. Finally, because $|P_I| \geq 2$ and b_i is the longest common

[5] Condition *(2)* is needed to avoid the enumeration of either motifs from $M_{qk} \setminus \mathcal{M}$ or duplicates from $\mathcal{M}$.

prefix of all occurrences in I, not all occurrences of u_i can be extended to the left using one symbol from Σ. Thus, u_i is a prefix or suffix of a maximal motif.

We now prove the other direction, that given a child $u_i \in T$ of u, we can find a single maximal interval $I \in \mathcal{I}_u$. First, denote by b_i the label on the (u, u_i) edge. From Lemma 6.5.1, b_i is a common prefix of all extensions of the occurrences in E_{u_i}. Since not all occurrences of u_i can be extended to the left using a single symbol from Σ, b_i is the longest common prefix satisfying this, and there are at least two different skipped characters of the occurrences in L_{u_i}. Now, we know that $u_i = u \star b_i$ occurs at all locations $p \in L_{u_i}$. Observe that L_{u_i} is a (jumbled) interval of E_u (since otherwise, there would be an element $p' \in E_u$ which did not match u_i but had occurrences from L_{u_i} on both sides in E_u, contradicting the grouping of E_u). All occurrences of u_i are in L_{u_i} so L_{u_i} is a (jumbled) maximal interval of E_u. We just described a maximal interval with a distinct set of occurrences, at least two different skipped characters and a common prefix, so there must surely be a corresponding interval $I \in \mathcal{I}_u$ such that $\mathrm{LCP}(I) = b_i$, $|P_I| \geq 2$ and $L_{u_i} \subseteq I$. There can be no $p' \in I - L_{u_i}$, as $p' \in L_u$ and $b_i \in \mathrm{prefset}(\mathrm{ext}(p', u))$ means that $p' \in L_{u_i}$. □

6.5.2 Exploiting the Properties of Maximal Intervals

We now use the properties shown above to implement the oracle GENERATE(u), resulting in Lemma 6.4.1. Observe that the task of GENERATE(u) can be equivalently seen by Lemma 6.5.3 as the task of finding all maximal intervals $\mathcal{I}_u$ in E_u, where each interval $I \in \mathcal{I}_u$ corresponds exactly to a distinct child u_i of u. We describe three main steps, where the first takes $O(\mathrm{sort}(L_u) + (k+1) \cdot |L_u|)$ time, and the others require $O(\sum_{i=1}^{d} |L_{u_i}|)$ time. The interval $I = E_u$ corresponding to the solid block ε is trivial to find, so we focus on the rest. We assume $\mathrm{dc}(u) < k$, as otherwise we are already done with u.

Step 1. Sort occurrences and find maximal runs of skipped characters. We perform a radix-sort of L_u using the extensions as keys, seen as integers from Π, thus obtaining array E_u. To facilitate the task of checking condition *(2)* for the maximality of intervals, we compute for each index $i \in E_u$ the smallest index $R(i) > i$ in E_u such that motif u cannot be specialized with the skipped characters in $\mathrm{skip}(E_u[j])$ where $j \in [i, R(i)]$. That is, there are at least two different characters from Σ hidden by each of the skipped characters in the interval. (If $R(i)$ does not exist, we do not create $[i, R(i)]$.) We define $|P_I|$ as the minimum number of different characters covered by each skipped character in interval I, and note that $|P_{[i,R(i)]}| \geq 2$ by definition.

To do so we first find, for each skipped character position, all indices where a maximal run of equal characters end: $R(i)$ is the maximum indices for the given i. This helps us because for any index i inside such a block of equal characters, $R(i)$ must be on the right of where the block ends (otherwise $[i, R(i)]$ would cover only one character in that block). Using this to calculate $R(i)$ for all indices $i \in E_u$ from left to right, we find each answer in time $O(k+1)$, and $O((k+1) \cdot |E_u|)$ total time. We denote by $\mathcal{R}$ the set of intervals $[i, R(i)]$ for $i \in E_u$.

Lemma 6.5.4. *For any maximal interval $I \equiv [i,j] \in \mathcal{I}_u$, there exists $R(i) \leq j$, and thus $[i,R(i)]$ is an initial portion of I.*

Step 2. Find maximal intervals with handles. We want to find all maximal intervals covering each position of E_u. To this end, we introduce *handles*. For each $p \in E_u$, its *interval domain* $D(p)$ is the set of intervals $I' \subset E_u$ such that $p \in I'$ and $|P_{I'}| \geq 2$. We let ℓ_p be the length of the longest shared solid block prefix b_i over $D(p)$, namely, $\ell_p = \max_{I' \in D(p)} |\mathrm{LCP}(I')|$. For a maximal interval $I \subseteq \mathcal{I}_u$, if $|\mathrm{LCP}(I)| = \ell_p$ for some $p \in I$ we call p a *handle* on I. Handles are relevant for the following reason.

Lemma 6.5.5. *For each maximal interval $I \subseteq \mathcal{I}_u$, either there is a handle $p \in E_u$ on I, or I is fully covered by ≥ 2 adjacent maximal intervals with handles.*

Proof. From Lemma 6.5.2, any maximal interval $I \in \mathcal{I}_u$ is either fully contained in some other maximal interval, or completely disjoint from other maximal intervals. Partial overlaps of maximal intervals are impossible.

Now, assume there is no handle $p \in L_u$ on I. If so, all $p' \in I$ has $\ell_{p'} \neq |\mathrm{LCP}(I)|$ (since otherwise $p' \in I$ and $\ell_{p'} = |\mathrm{LCP}(I)|$ and thus p' was a handle on I). Clearly for all $p' \in I$, $|\mathrm{LCP}(I)|$ is a lower bound for $\ell_{p'}$. Thus, to get a contradiction it must be the case that $\ell_{p'} > |\mathrm{LCP}(I)|$ for all $p' \in I$. This can only happen if I is completely covered by ≥ 2 maximal intervals with a larger longest common prefix. From Lemma 6.5.2, a single interval I' is not enough because I' is fully contained (or completely disjoint) in I if $|\mathrm{LCP}(I')| \geq |\mathrm{LCP}(I)|$. □

Let $\mathcal{H}_u$ denote the set of maximal intervals with handles. We now show how to find the set $\mathcal{H}_u$ among the intervals of E_u. Observe that for each occurrence $p \in E_u$, we must find the interval I' with the largest $\mathrm{LCP}(I')$ value among all intervals containing p.

From the definition, a handle on a maximal interval I' requires $|P_{I'}| \geq 2$, which is exactly what the intervals in $\mathcal{R}$ satisfy. As the LCP value can only drop when extending an interval, these are the only candidates for maximal intervals with handles. Note that from Lemma 6.5.4, $\mathcal{R}$ contains a prefix for all of the (not expanded) maximal intervals because it has all intervals from left to right obeying the conditions on length and skipped character conditions. Furthermore, $|\mathcal{R}| = O(|E_u|)$, since only one $R(i)$ is calculated for each starting position. Among the intervals $[i,R(i)] \in \mathcal{R}$, we will now show how to find those with maximum LCP (i.e. where the LCP value equals ℓ_p) for all p.

We use an idea similar to that used in Section 6.3.3 to filter maximal motifs from the right-maximal motifs. We sort the intervals $I' = [i,R(i)] \in \mathcal{R}$ in decreasing lexicographic order according to the pairs $\langle |\mathrm{LCP}(I')|, -i \rangle$ (i.e. decreasing LCP values but increasing indices i), to obtain the sequence $\mathcal{C}$. Thus, if considering the intervals left to right in $\mathcal{C}$, we consider intervals with larger LCP values from left to right in S before moving to smaller LCP values.

Consider an interval $I_p = [i,R(i)] \in \mathcal{C}$. The idea is that we determine if I_p has already been added to $\mathcal{H}_u$ by some previously processed handled maximal interval. If not, we expand the interval (making it maximal) and add it to $\mathcal{H}_u$, otherwise I_p is discarded.

When $\mathcal{C}$ is fully processed, all occurrences in E_u are covered by maximal intervals with handles.

First, since maximal intervals must be fully contained in each other (from Lemma 6.5.2), we determine if $I_p = [i, R(i)] \in \mathcal{C}$ is already fully covered by previously expanded intervals (with larger LCP values) – if not, we must expand I_p. Clearly, if either i or $R(i)$ is not included in any previous expansions, we must expand I_p. Otherwise, if both i and $R(i)$ is part of a single previous expansion $I_q \in \mathcal{C}$, I_p should not be expanded. If i and $R(i)$ is part of two different expansions I_q and I_r we compare their extent values: I_p must be expanded if some $p \in I_p$ is not covered by either I_q or I_r. To enable these checks we mark each $j \in E_u$ with the longest processed interval that contains it (during the expansion procedure below).

Secondly, to expand I_p maximally to the left and right, we use pairwise *lcp* queries on the border of the interval. Let $a \in I_p$ be a border occurrence and $b \notin I_p$ be its neighboring occurrence in E_u (if any, otherwise it is trivial). When $|lcp(a, b)| < |\text{LCP}(I_p)|$, the interval cannot be expanded to span b. When the expansion is completed, I_p is a maximal interval and added to $\mathcal{H}_u$. As previously stated, all elements in I_p are marked as being part of their longest covering handled maximal interval by writing I_p on each of its occurrences.

Step 3. Find composite maximal intervals covered by maximal intervals with handles. From Lemma 6.5.5, the only remaining type of intervals are composed of maximal intervals with handles from the set $\mathcal{H}_u$. A *composite maximal interval* must be the union of a sequence of adjacent maximal intervals with handles. We find these as follows. We order $\mathcal{H}_u$ according to the starting position and process it from left to right in a greedy fashion, letting $I_h \in \mathcal{H}_u$ be one of the previously found maximal intervals with handles. Each interval I_h is responsible for generating exactly the composite maximal intervals where the sequence of covering intervals starts with I_h (and which contains a number of adjacent intervals on the right). Let $I'_h \in \mathcal{H}_u$ be the interval adjacent on the right to I_h, and create the composed interval $I_c = I_h + I'_h$ (where $+$ indicates the concatenation of consecutive intervals). To ensure that a composite interval is new, we check as in Step 2 that there is no previously generated maximal interval I with $|\text{LCP}(I)| = |\text{LCP}(I_c)|$ such that $I_c \subseteq I$. This is correct since if there is such an interval, it has already been fully expanded by a previous expansion (of composite intervals or a handled interval). Furthermore, if there is such an interval, all intervals containing I_c with shorter longest common prefixes have been taken care of, since from Lemma 6.5.2 maximal intervals cannot straddle each other. If I_c is new, we know that we have a new maximal composite interval and can continue expanding it with adjacent intervals. If the length of the longest common prefix of the expanded interval changes, we must perform the previous check again (and add the previously expanded composite interval to $\mathcal{I}_u$).

By analyzing the algorithm described, one can prove the following two lemmas showing that the motif trie is generated correctly. While Lemma 6.5.6 states that ε-extensions may be generated (i.e. a sequence of $\star$ symbols may be added to suffixes of maximal motifs), a simple bottom-up cleanup traversal of T is enough to remove these.

Lemma 6.5.6. ***(Soundness)*** *Each motif stored in T is a prefix or an ε-extension of some suffix of a maximal motif (encoded using alphabet Π and stored in T).*

Proof. The property to be shown for motif $m \in T$ is: *(1)* m is a prefix of some suffix of a maximal motif $m' \in \mathcal{M}$ (encoded using alphabet Π), or *(2)* m is the suffix of some maximal motif $m' \in \mathcal{M}$ extended by at most k ε solid blocks (and don't cares).

Note that we only need to show that GENERATE(u) can only create children of $u \in T$ with the desired property. We prove this by induction. In the basis, u is the root and GENERATE(u) produce all motifs such that adding a character from Σ to either end decreases the number of occurrences: this is ensured by requiring that there must be more than two different skipped characters in the occurrences considered, using the LCP of such intervals and only extending intervals to span occurrences maintaining the same LCP length. Since there are no don't cares in these motifs they cannot be specialized and so each of them must be a prefix or suffix of some maximal motif.

For the inductive step, we prove the property by construction, assuming $\text{dc}(u) < k$. Consider a child u_i generated by GENERATE(u) by extending with solid block b_i: it must not be the case that, without losing occurrences, *(a)* u_i can be specialized by converting one of its don't cares into a solid character from Σ, or *(b)* u_i can be extended in either direction using only characters from Σ. If either of these conditions is violated, u_i can clearly not satisfy the property (in the first case, the generalization u_i is not a suffix or prefix of the specialized maximal motif). However, these conditions are sufficient, as they ensure that u_i is encoded using Π and cannot be specialized or extended without using don't cares. Thus, if $b_i \neq \varepsilon$, u_i is either a prefix of some suffix of a maximal motif (since u_i ends with a solid block it may be maximal), or if $b_i = \varepsilon$, u_i may be an ε-extension of u (or a prefix of some suffix if some descendant of u_i has the same number of occurrences and a non-ε parent edge).

By the induction hypothesis, u satisfies *(1)* or *(2)* and u is a prefix of u_i. Furthermore, the occurrences of u have more than one different character at all locations covered by the don't cares in u (otherwise one of those locations in u could be specialized to the common character). When generating children, we ensure that *(a)* cannot occur by forcing the occurrence list of generated children to be large enough that at least two different characters is covered by each don't care. That is, u_i may only be created if it cannot be specialized in any location, ensured by only considering intervals covering $L(p)$ and $R(p)$. Condition *(b)* is avoided by ensuring that there are at least two different skipped characters for the occurrences of u_i and forcing the extending block b_i to be maximal under that condition. □

Lemma 6.5.7. ***(Completeness)*** *If $m \in \mathcal{M}$, T stores m and its suffixes.*

Proof. We summarize the proof that GENERATE(u) is correct and the correct motif trie is produced. From Lemma 6.5.5, we create all intervals in GENERATE(u) by expanding those with handles, and expanding all composite intervals from these. By Lemma 6.5.3 the intervals found correspond exactly to the children of u in the motif trie. Thus, as GENERATE(u) is executed for all $u \in T$ when $\text{dc}(u) \leq k - 1$, all nodes in T is created correctly until depth $k + 1$.

Now clearly T contains $\mathcal{M}$ and all the suffixes: for a maximal motif $m \in \mathcal{M}$, any suffix m' is generated and stored in T as *(1)* $\mathrm{occ}(m') \geq \mathrm{occ}(m)$ and *(2)* $\mathrm{dc}(m') \leq \mathrm{dc}(m)$. □

Part III

Points

7 Compressed Data Structures for Range Searching

Philip Bille* Inge Li Gørtz* Søren Vind†

Technical University of Denmark

Abstract

We study the orthogonal range searching problem on points that have a significant number of *geometric repetitions*, that is, subsets of points that are identical under translation. Such repetitions occur in scenarios such as image compression, GIS applications and in compactly representing sparse matrices and web graphs. Our contribution is twofold.

First, we show how to compress geometric repetitions that may appear in standard range searching data structures (such as K-D trees, Quad trees, Range trees, R-trees, Priority R-trees, and K-D-B trees), and how to implement subsequent range queries on the compressed representation with only a constant factor overhead.

Secondly, we present a compression scheme that efficiently identifies geometric repetitions in point sets, and produces a hierarchical clustering of the point sets, which combined with the first result leads to a compressed representation that supports range searching.

*Supported by a grant from the Danish Council for Independent Research | Natural Sciences.
†Supported by a grant from the Danish National Advanced Technology Foundation.

7.1 Introduction

The *orthogonal range searching* problem is to store a set of axis-orthogonal k-dimensional objects to efficiently answer *range queries*, such as reporting or counting all objects inside a k-dimensional query range. Range searching is a central primitive in a wide range of applications and has been studied extensively over the last 40 years [Ben75; Ben79; Ore82; Ben+80; Lue78; Lee+80; Gut84; Cla83; Kan+99; Kre+91; Gae+98; Bay+72; Arg+08; Rob81; Pro+03; Com79; Epp+08] (Samet presents an overview in [Sam90]).

In this paper we study range searching on points that have a significant number of *geometric repetitions*, that is, subsets of points that are identical under translation. Range searching on points sets with geometric repetitions arise naturally in several scenarios such as data and image analysis [Tet+01; Paj+00; Dic+09], GIS applications [Sch+08; Zhu+02; Hae+10; Dic+09], and in compactly representing sparse matrices and web graphs [Gal+98; Bri+09; Gar+14; Ber+13].

Our contribution is twofold. First, we present a simple technique to effectively compress geometric repetitions that may appear in standard range searching data structures (such as K-D trees, Quad trees, Range trees, R-trees, Priority R-trees, and K-D-B trees). Our technique replaces repetitions within the data structures by a single copy, while only incurring an $O(1)$ factor overhead in queries (both in standard RAM model and I/O model of computation). The key idea is to compress the underlying tree representation of the point set into a corresponding minimal DAG that captures the repetitions. We then show how to efficiently simulate range queries directly on this DAG. This construction is the first solution to take advantage of geometric repetitions. Compared to the original range searching data structure the time and space complexity of the compressed version is never worse, and with many repetitions the space can be significantly better. Secondly, we present a compression scheme that efficiently identifies translated geometric repetitions. Our compression scheme guarantees that if point set P_1 is a translated geometric repetition of point set P_2 and P_1 and P_2 are at least a factor 2 times their diameter away from other points, the repetition is identified. This compression scheme is based on a hierarchical clustering of the point set that produces a tree of height $O(\lg D)$, where D is the diameter of the input point set. Combined with our first result we immediately obtain a compressed representation that supports range searching.

Related Work

Several succinct data structures and entropy-based compressed data structures for range searching have recently been proposed, see e.g., [Mäk+07; Bos+09; Bar+10; Far+14]. While these significantly improve the space of the classic range searching data structure, they all require at least a $\Omega(N)$ *bits* to encode N points. In contrast, our construction can achieve exponential compression for highly compressible point sets (i.e. where there is a lot of geometric repetitions).

A number of papers have considered the problem of compactly representing web graphs and tertiary relations [Bri+09; Gar+14; Ber+13]. They consider how to efficiently represent a binary (or tertiary) quad tree by encoding it as bitstrings. That is, their ap-

proach may be considered compact storage of a (sparse) adjacency matrix for a graph. The approach allows compression of quadrants of the quad tree that only contain zeros or ones. However, it does not exploit the possibly high degree of geometric repetition in such adjacency matrices (and any quadrant with different values cannot be compressed).

To the best of our knowledge, the existence of geometric repetitions in the point sets has not been exploited in previous solutions for neither compression nor range searching. Thus, we give a new perspective on those problems when repetitions are present.

Outline

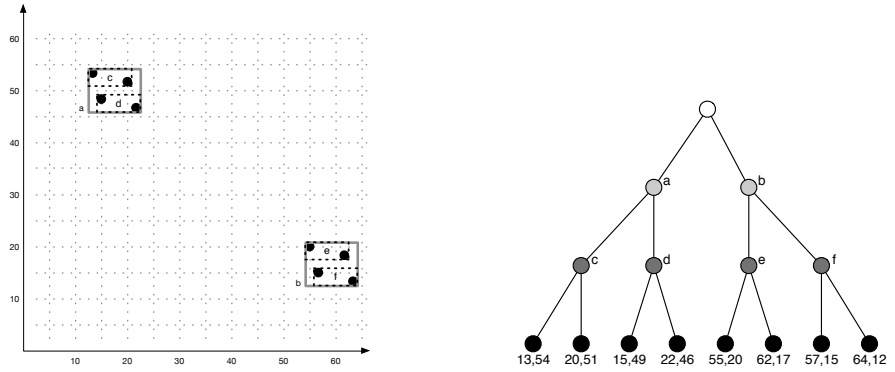
We first present a general model for range searching, which we call a *canonical range searching data structure*, in Section 7.2. We show how to compress such data structures efficiently and how to support range searching on the compressed data structure in the same asymptotic time as on the uncompressed data structure in Section 7.3. Finally, we present a *similarity clustering* algorithm in Section 7.4, guaranteeing that geometric repetitions are clustered such that the resulting canonical range searching data structure is compressible.

7.2 Canonical Range Searching Data Structures

We define a *canonical range searching data structure* T, which is an ordered, rooted and labeled tree with N vertices. Each vertex $v \in T$ has an associated k-dimensional axis-parallel range, denoted r_v, and an arbitrary label, denoted $label(v)$. We let $T(v)$ denote the subtree of T rooted at vertex v and require that ranges of vertices in $T(v)$ are contained in the range of v, so for every vertex $u \in T(v)$, $r_u \subseteq r_v$. Leafs may store either points or ranges, and each point or range may be stored in several leafs. The data structure supports *range queries* that produce their result after evaluating the tree through a (partial) traversal starting from the root. In particular, we can only access a node after visiting all ancestors of the node. Queries can use any information from visited vertices. A similar model for showing lower bounds for range searching appeared was used by Kanth and Singh in [Kan$^+$99].

Geometrically, the children of a vertex v in a canonical range searching data structure divide the range of v into a number of possibly overlapping ranges. At each level the tree divides the k-dimensional regions at the level above into smaller regions. Canonical range searching data structures directly capture most well-known range searching data structures, including Range trees, K-D trees, Quad trees and R-trees as well as B-trees, Priority R-trees and K-D-B trees.

Example: Two-dimensional R tree The two-dimensional R tree is a canonical range searching data structure since a vertex covers a range of the plane that contains the ranges of all vertices in its subtree. The range query is a partial traversal of the tree starting from the root, visiting every vertex having a range that intersects the query range and reporting all vertices with their range fully contained in the query range. Figure 7.1



(a) Point Set. (b) R tree.

Figure 7.1: A two-dimensional point set with R tree ranges overlaid, and the resulting R tree. Blue ranges are children of the root in the tree, red ranges are at the second level. A vertex label (a - h) in the R tree identifies the range. We have omitted the precise coordinates for the ranges, but e.g. range a spans the range $[13,22] \times [46,54]$.

shows an R tree for a point set, where each vertex is labeled with the range that it covers. The query described for R trees can be used on any canonical range searching data structure, and we will refer to it as a *canonical range query*.

7.3 Compressed Canonical Range Searching

We now show how to compress geometric repetitions in any canonical range searching data structure T while incurring only a constant factor overhead in queries. To do so we convert T into a *relative tree* representation, which we then compress into a minimal DAG representation that replaces geometric repetitions by single occurrences. We then show how to simulate a range query on T with only constant overhead directly on the compressed representation. Finally, we extend the result to the I/O model of computation.

7.3.1 The Relative Tree

A *relative tree* R is an ordered, rooted and labeled tree storing a relative representation of a canonical range searching data structure T. The key idea is we can encode a range or a point $r = [x_1, x'_1] \times \ldots \times [x_k, x'_k]$ as two k-dimensional vectors $\mathit{position}(r) = (x_1, \ldots, x_k)$ and $\mathit{extent}(r) = (x'_1 - x_1, \ldots, x'_k - x'_k)$ corresponding to an *origin position* and an *extent* of r. We use this representation in the relative tree, but only store extent vectors at vertices

explicitly. The origin position vector for the range r_v of a vertex $v \in R$ is calculated from offset vectors stored on the path from the root of R to v, denoted $path(v)$.

Formally, each vertex $v \in R$ stores a label, $label(v)$, and a k-dimensional extent vector $extent(r_v)$. Furthermore, each edge $(u, v) \in R$ stores an offset vector $offset(u, v)$. The position vector for r_v is calculated as $position(r_v) = \sum_{(a,b)\in path(v)} offset(a, b)$. We say that two vertices $v, w \in R$ are *equivalent* if the subtrees rooted at the vertices are isomorphic, including all labels and vectors. That is, v and w are equivalent if the two subtrees $R(v)$ and $R(w)$ are equal.

It is straightforward to convert a canonical range searching data structure into the corresponding relative tree.

Lemma 7.3.1. *Given any canonical range searching data structure T, we can construct the corresponding relative tree R in linear time and space.*

Proof. First, note that a relative tree allows each vertex to store extent vectors and labels. Thus, to construct a relative tree R representing the canonical range searching data structure T, we can simply copy the entire tree including extent vectors and vertex labels. So we only need to show how to store offset vectors in R to ensure that the ranges for each pair of copied vertices are equal.

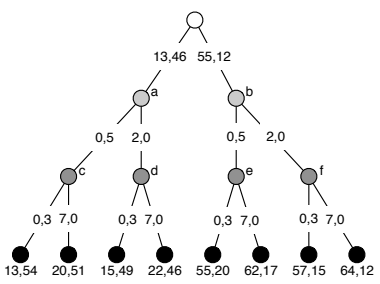
Consider a vertex $v \in T$ and its copy $v_R \in R$ and their parents $w \in T$ and $w_R \in R$. Since the extent vector and vertex labels are copied, $extent(r_v) = extent(r_{v_R})$ and $label(v) = label(v_R)$. The offset vector for edge (w_R, v_R) is $offset(w_R, v_R) = position(r_v) - position(r_w)$. We assume the offset for the root is the 0-vector. Observe that summing up all the offset vectors on $path(v)$ is exactly $position(r_v)$, and so $position(r_{v_R}) = position(r_v)$.

Since each vertex and edge in T is only visited a constant number of times during the mapping, the construction time for R is $O(N)$. The total number of labels stored by R is asymptotically equal to the number of labels stored by T. Finally, the degrees of vertices does not change from T to R. Thus, if $v \in T$ is mapped to $v_R \in R$ and v requires s space, v_R requires $\Theta(s)$ space. □

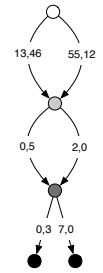
7.3.2 The Data Structure

The compressed canonical data structure is the minimal DAG G of the relative tree R for T. By Lemma 7.3.1 and [Dow$^+$80] we can build it in $O(N)$ time. Since G replaces equivalent subtrees in R by a single subtree, geometric repetitions in T are stored only once in G. For an example, see Figure 7.2.

Now consider a range query Q on the canonical range searching data structure T. We show how to simulate Q efficiently on G. Assuming $v_G \in G$ generates $v_R \in R$, we say that v_G generates $v \in T$ if v_R is the relative tree representation of v. When we visit a vertex $v_G \in G$, we calculate the origin position $position(r_{v_G})$ from the sum of the offset vectors along the root-to-v_G path. The origin position for each vertex can be stored on the way down in G, since we may only visit a vertex after visiting all ancestors (meaning that we can only arrive at v_G from a root-to-v_G path in G). Thus, it takes constant time to maintain the origin position for each visited vertex. Finally, a visit to a child of $v \in T$



(a) Relative tree.



(b) Minimal DAG.

Figure 7.2: The relative tree obtained from the R tree from Figure 7.1 and the resulting minimal DAG G generating the tree. Only coordinates of the lower left corner of the ranges in the R tree are shown. In the relative tree, the absolute coordinates for the points are only shown for illustration, in order to see that the relative coordinates sum to the absolute coordinate along the root-to-leaf paths .

can be simulated in constant additional time by visiting a child of $v_G \in G$. So we can simulate a visit to $v \in T$ by visiting the vertex $v_G \in G$ that generates v and in constant time calculate the origin position for v_G.

Any label comparison takes the same time on G and T since the label must be equal for $v_G \in G$ to generate $v \in T$. Now, since there is only constant overhead in visiting a vertex and comparing labels, it follows that if Q uses t time we can simulate it in $O(t)$ time on G. In summary, we have the following result.

Theorem 7.3.2. *Given a canonical range searching data structure T with N vertices, we can build the minimal DAG representation G of T in linear time. The space required by G is $O(n)$, where n is the size of the minimal DAG for a relative representation of T. We can support any query Q on T that takes time t on G in time $O(t)$.*

As an immediate corollary, we get the following result for a number of concrete range searching data structures.

Corollary 7.3.2.1. *Given a K-D tree, Quad tree, R tree or Range tree, we can in linear time compress it into a data structure using space proportional to the size of the minimal relative DAG representation which supports canonical range searching queries with $O(1)$ overhead.*

7.3.3 Extension to the I/O Model

We now show that Theorem 7.3.2 extends to the I/O model of computation. We assume that each vertex in T require $\Theta(B)$ space, where B is the size of a disk block. To allow

for such vertices, we relax the definition of a canonical range searching data structure to allow it to store B k-dimensional ranges. From Lemma 7.3.1 and [Dow$^+$80], if a vertex $v \in T$ require $\Theta(B)$ space, then so does the corresponding vertex $v_G \in G$. Thus, the layout of the vertices on disk does not asymptotically influence the number of disk reads necessary to answer a query, since only a constant number of vertices can be retrieved by each disk read. This means that visiting a vertex in either case takes a constant number of disk blocks, and so the compressed representation does not asymptotically increase the number of I/Os necessary to answer the query. Hence, we can support any query Q that uses p I/Os on T using $O(p)$ I/Os on G.

7.4 Similarity Clustering

We now introduce the *similarity clustering* algorithm. Even if there are significant geometric repetitions in the point set P, the standard range searching data structures may not be able to capture this and may produce data structures that are not compressible. The similarity clustering algorithm allows us to create a canonical range searching data structure for which we can guarantee good compression using Theorem 7.3.2.

7.4.1 Definitions

Points and point sets We consider points in k-dimensional space, assuming k is constant. The distance between two points p_1 and p_2, denoted $d(p_1, p_2)$, is their euclidian distance. We denote by $P = \{p_1, p_2, \ldots, p_r\}$ a point set containing r points. We say that two point sets P_1, P_2 are *equivalent* if P_2 can be obtained from P_1 by translating all points with a constant k-dimensional offset vector.

The minimum distance between a point p_q and a point set P is the distance between p_q and the closest point in P, $\mathit{mindist}(P, p_q) = \min_{p \in P} d(p, p_q)$. The minimum distance between two point sets P_1, P_2 is the distance between the two closest points in the two sets, $\mathit{mindist}(P_1, P_2) = \min_{p_1 \in P_1, p_2 \in P_2} d(p_1, p_2)$. These definitions extend to maximum distance in the natural way, denoted $\mathit{maxdist}(P, p_q)$ and $\mathit{maxdist}(P_1, P_2)$. The diameter of a point set P is the maximum distance between any two points in P, $\mathit{diameter}(P) = \max_{p_1, p_2 \in P} d(p_1, p_2) = \mathit{maxdist}(P, P)$.

A point set $P_1 \subset P$ is *lonely* if the distance from P_1 to any other point is more than twice $\mathit{diameter}(P_1)$, i.e. $\mathit{mindist}(P_1, P \setminus P_1) > 2 \times \mathit{diameter}(P_1)$.

Clustering A hierarchical clustering of a point set P is a tree, denoted $C(P)$, containing the points in P at the leaves. Each node in the tree $C(P)$ is a cluster containing all the points in the leaves of its subtree. The root of $C(P)$ is the cluster containing all points. We denote by $\mathit{points}(v)$ the points in cluster node $v \in C(P)$. Two cluster nodes $v, w \in C(P)$ are equivalent if $\mathit{points}(v)$ is equivalent to $\mathit{points}(w)$ and if the subtrees rooted at the nodes are isomorphic such that each isomorphic pair of nodes are equivalent.

7.4.2 Hierarchical Clustering Algorithm for Lonely Point Sets

Order P in lexicographically increasing order according to their coordinates in each dimension, and let $\Delta(P)$ denote the ordering of P. The similarity clustering algorithm performs a greedy clustering of the points in P in levels $i = 0, 1, \ldots, \lg D + 1$, where $D = \mathit{diameter}(P)$. Each level i has an associated clustering distance threshold d_i, defined as $d_0 = 0$ and $d_i = 2^{i-1}$ for all other i.

The clustering algorithm proceeds as follows, processing the points in order $\Delta(P)$ at each level. If a point p is not clustered at level $i > 0$, create a new cluster C_i centered around the point p (and its cluster C_{i-1} at the previous level). Include a cluster C_{i-1} from level $i-1$ in C_i if $\mathit{maxdist}(\mathit{points}(C_{i-1}), p) \leq d_i$. The clusters at level 0 contain individual points and the cluster at level $\lg D + 1$ contains all points.

Lemma 7.4.1. *Given a set of points P, the similarity clustering algorithm produces a clustering tree containing equivalent clusters for any pair of equivalent lonely point sets.*

Proof. Let P_1 and P_2 be two lonely point sets in P such that P_1 and P_2 are equivalent, and let $d = \mathit{diameter}(P_1) = \mathit{diameter}(P_2)$. Observe that a cluster formed at level i has at most diameter $2d_i = 2^i$. Thus, since all points are clustered at every level and all points outside P_1 have a distance greater than $2d$ to any point in P_1, there is a cluster $c \in C(P)$ formed around point $a \in P_1$ at level $j = \lceil \lg d \rceil$ containing no points outside P_1. Now, assume some point $p \in P_1$ is not in $\mathit{points}(c)$. As all unclustered points within distance $2^j \geq d$ from a are included in c, this would mean that p was clustered prior to creating c. This contradicts the assumption that P_1 is lonely, since it can only happen if some point outside P_1 is closer than $2d$ to p. Concluding, c contains exactly the points in P_1. The same argument naturally extends to P_2.

Now, let C_1, C_2 be the clusters containing the points from P_1, P_2, respectively. Observe that $\mathit{points}(C_1)$ and $\mathit{points}(C_2)$ are equivalent. Furthermore, because each newly created cluster process candidate clusters to include in the same order, the resulting trees for C_1 and C_2 are isomorphic and have the same ordering. Thus, the clusters C_1 and C_2 are equivalent. □

Because the clustering proceeds in $O(\lg D)$ levels, the height of the clustering tree is $O(\lg D)$. Furthermore, by considering all points and all of their candidates at each level, the clustering can be implemented in time $O(N^2 \lg D)$. Observe that the algorithm allows creation of paths of clusters with only a single child cluster. If such paths are contracted to a single node to reduce the space usage, the space required is $O(N)$ words. In summary, we have the following result.

Theorem 7.4.2. *Given a set of N points with diameter D, the similarity clustering algorithm can in $O(N^2 \lg D)$ time create a tree representing the clustering of height $O(\lg D)$ requiring $O(N)$ words of space. The algorithm guarantees that any pair of equivalent lonely point sets results in the same clustering, producing equivalent subtrees in the tree representing the clustering.*

Since the algorithm produces equivalent subtrees in the tree for equivalent lonely point sets, the theorem gives a compressible canonical range searching data structure for point sets with many geometric repetitions.

7.5 Open Problems

The technique described in this paper for generating the relative tree edge labels only allows for translation of the point sets in the underlying subtrees. However, the given searching technique and data structure generalizes to scaling and rotation (if simply storing a parent-relative scaling factor and rotation angle in each node, along with the nodes parent-relative translation vector). We consider it an open problem to efficiently construct a relative tree that uses such transformations of the point set.

Another interesting research direction is if it is possible to allow for small amounts of noise in the point sets. That is, can we represent point sets that are almost equal (where few points have been moved a little) in a compressed way? An even more general question is how well one can do when it comes to compression of higher dimensional data in general.

Finally, the $O(N^2 \lg D)$ time bound for generating the similarity clustering is prohibitive for large point sets. So an improved construction would greatly benefit the possible applications of the clustering method and is of great interest.

8 Colored Range Searching in Linear Space

Roberto Grossi[1]* Søren Vind[2]†

[1] Università di Pisa
[2] Technical University of Denmark

Abstract

In *colored range searching*, we are given a set of n colored points in $d \geq 2$ dimensions to store, and want to support orthogonal range queries taking colors into account. In the *colored range counting* problem, a query must report the number of distinct colors found in the query range, while an answer to the *colored range reporting* problem must report the distinct colors in the query range.

We give the first linear space data structure for both problems in two dimensions ($d = 2$) with $o(n)$ worst case query time. We also give the first data structure obtaining almost-linear space usage and $o(n)$ worst case query time for points in $d > 2$ dimensions. Finally, we present the first dynamic solution to both counting and reporting with $o(n)$ query time for $d \geq 2$ and $d \geq 3$ dimensions, respectively.

*Partially supported by Italian MIUR PRIN project AMANDA.
†Supported by a grant from the Danish National Advanced Technology Foundation.

8.1 Introduction

In standard range searching a set of points must be stored to support queries for any given orthogonal d-dimensional query box Q (see [Ber+08] for an overview). Two of the classic problems are standard range reporting, asking for the points in Q, and standard range counting, asking for the number of points in Q. In 1993, Janardan and Lopez [Jan+93] introduced one natural generalisation of standard range searching, requiring each point to have a color. This variation is known as *colored range searching*[1]. The two most studied colored problems are *colored range reporting*, where a query answer must list the distinct colors in Q, and *colored range counting*, where the number of distinct colors in Q must be reported.

As shown in Tables 8.1–8.2, there has been a renewed interest for these problems due to their applications. For example, in database analysis and information retrieval the d-dimensional points represent entities and the colors are *classes* (or categories) of entities. Due to the large amount of entities, statistics about their classes is the way modern data processing is performed. A colored range query works in this scenario and reports some *statistical analysis* for the classes using the range on the entities as a filtering method: "which kinds of university degrees do European workers with age between 30 and 40 years and salary between 30,000 and 50,000 euros have?". Here the university degrees are the colors and the filter is specified by the range of workers that are European with the given age and salary. The large amount of entities involved in such applications calls for nearly *linear space* data structures, which is the focus of this paper.

Curiously, counting is considered harder than reporting among the colored range queries as it is not a decomposable problem: knowing the number of colors in two halves of Q does not give the query answer. This is opposed to reporting where the query answer can be obtained by merging the list of colors in two halves of Q and removing duplicates. For the standard problems, both types of queries are decomposable and solutions to counting are generally most efficient.

In the following, we denote the set of d-dimensional points by P and let $n = |P|$. The set of distinct colors is Σ and $\sigma = |\Sigma| \leq n$, with $k \leq \sigma$ being the number of colors in the output. We use the notation $\lg^a b = (\lg b)^a$ and adopt the RAM with word size $w = \Theta(\lg n)$ bits, and the size of a data structure is the number of occupied words.

Observe that for both colored problems, the trivial solution takes linear time and space, storing the points and looking through all of them to answer the query. Another standard solution is to store one data structure for all points of each color that supports standard range emptiness queries ("is there any point inside Q?"). In two dimensions, this approach can answer queries in $O(\sigma \lg n)$ time and linear space using a range emptiness data structure by Nekrich [Nek09]. However, since $\sigma = n$ in the worst case, this does not guarantee a query time better than trivial. Due to the extensive history and the number of problems considered, we will present our results before reviewing existing work.

[1] Also known in the literature as *categorical* or *generalized* range searching.

Dim.	Query Time	Space Usage	Dyn.	Ref.
1	$O(\lg n/\lg\lg n)$	$O(n)$	×	§
2	$O(\lg^2 n)$	$O(n^2\lg^2 n)$		⋆
	$O(\lg^2 n)$	$O(n^2\lg^2 n)$		†
	$O(X\lg^7 n)$	$O((\frac{n}{X})^2\lg^6 n + n\lg^4 n)$		†
	$O\left((\frac{\sigma}{\lg n} + \frac{n}{\lg^c n})(\lg\lg^c n)^{2+\varepsilon}\right)$	$O(n)$		New
> 2	$O(\lg^{2(d-1)} n)$	$O(n^d\lg^{2(d-1)} n)$		†
	$O(X\lg^{d-1} n)$	$O((\frac{n}{X})^{2d} + n\lg^{d-1} n)$		†
	$O\left((\frac{\sigma}{\lg n} + \frac{n}{\lg^c n})(\lg\lg^c n)^{d-1}\lg\lg\lg^c n\right)$	$O\left(n(\lg\lg^c n)^{d-1}\right)$		New
≥ 2	$O\left((\frac{\sigma}{\lg n} + \frac{n}{\lg^c n})(\lg\lg^c n)^d\right)$	$O\left(n(\lg\lg^c n)^{d-1}\right)$	×	New

Table 8.1: Known and new solutions to *colored range counting*, ordered according to decreasing space use in each dimension group. Dyn. column shows if solution is dynamic, d is the dimensionality. References are abbreviated § = [JáJ$^+$05], ⋆ = [Gup$^+$95], † = [Kap$^+$07].

8.1.1 Our Results

We observe (see Section 8.1.2) that there are no known solutions to any of the colored problems in two dimensions that uses $O(n)$ words of space and answer queries in $o(n)$ worst case time. Furthermore, for colored range reporting there are no known solutions in $d > 3$ dimensions using $o(n^{1+\varepsilon})$ words of space and answering queries in $o(n)$ worst case time. For colored range counting, no solutions with $o(n\,\mathrm{polylg}\,n)$ words of space and $o(n)$ worst case time exist.

We present the first data structures for colored range searching achieving these bounds, improving almost logarithmically over previously known solutions in the worst case (see Section 8.1.2 and Tables 8.1–8.2). Specifically, we obtain

- $o(n)$ query time and $O(n)$ space in two dimensions,
- $o(n)$ query time and $o(n\,\mathrm{polylg}\,n)$ space for counting in $d \geq 2$ dimensions,
- $o(n)$ query time and $o(n^{1+\varepsilon})$ space for reporting in $d > 3$ dimensions,
- $o(n)$ query time supporting $O(\mathrm{polylg}\,n)$ updates in $d \geq 2$ and $d \geq 3$ dimensions for counting and reporting, respectively.

We note that while our bounds have an exponential dimensionality dependency (as most previous results), it only applies to $\lg\lg n$ factors in the bounds. Our solutions can

Dim.	Query Time	Space Usage	Dyn.	Ref.
1	$O(1+k)$	$O(n)$	×	§
2	$O(\lg n + k)$	$O(n \lg n)$		†
	$O(\lg^2 n + k \lg n)$	$O(n \lg n)$	×	⋆, ‡
	$O\left((\frac{\sigma}{\lg n} + \frac{n}{\lg^c n})(\lg \lg^c n)^{2+\varepsilon} + k\right)$	$O(n)$		New
≥ 2	$O\left((\frac{\sigma}{\lg n} + \frac{n}{\lg^c n})(\lg \lg^c n)^{d} + k\right)$	$O\left(n(\lg \lg^c n)^{d-1}\right)$	×	New
3	$O(\lg^2 n + k)$	$O(n \lg^4 n)$		⋆
> 3	$O(\lg n + k)$	$O(n^{1+\varepsilon})$		⋄, $
≥ 3	$O\left((\frac{\sigma}{\lg n} + \frac{n}{\lg^c n})(\lg \lg^c n)^{d-1} \lg \lg \lg^c n + k\right)$	$O\left(n(\lg \lg^c n)^{d-1}\right)$		New

Table 8.2: Known and new solutions to *colored range reporting*, ordered according to decreasing space use in each dimension group. Dyn. column shows if solution is dynamic, d is the dimensionality. References are abbreviated § = [Nek+13], ⋆ = [Gup+95], † = [Shi+05], ‡ = [Boz+95], ⋄ = [Kre92], $ = [Gup+97].

be easily implemented and parallelized, so they are well-suited for the distributed processing of large scale data sets. Our main results can be summarised in the following theorems, noting that $c > 1$ is an arbitrarily chosen integer, and $\sigma \leq n$ is the number of distinct colors.

Theorem 8.1.1. *There is a linear space data structure for two-dimensional colored range counting and reporting storing n points, each assigned one of σ colors. The data structure answers queries in time $O((\sigma/\lg n + n/\lg^c n)(\lg \lg^c n)^{2+\varepsilon})$, with reporting requiring an additive term $O(k)$.*

Theorem 8.1.2. *There is a $O(n(\lg \lg^c n)^{d-1})$ space data structure storing n d-dimensional colored points each assigned one of σ colors. Each colored range counting and reporting query takes time $O((\sigma/\lg n + n/\lg^c n)(\lg \lg^c n)^{d-1} \lg \lg \lg^c n)$. Reporting requires an additive $O(k)$ time.*

To obtain these results, we partition points into groups depending on their color. Each group stores all the points for at most $\lg n$ specific colors. Because the colors are partitioned across the groups, we can obtain the final result to a query by merging query results for each group (and we have thus obtained a decomposition of the problem along the color dimension). A similar approach was previously used in [Kap+07].

In order to reduce the space usage of our data structure, we partition the points in each group into a number of buckets of at most $\lg^c n$ points each. The number of buckets is $O(\sigma/\lg n + n/\lg^c n)$, with the first term counting all underfull buckets and

the second counting all full buckets. Each bucket stores $m \leq \lg^c n$ points colored with $f \leq \lg n$ different colors. To avoid counting a color several times across different buckets, we use a solution to the d-dimensional colored range reporting problem in each bucket for which answers to queries are given as bitstrings. Answers to the queries in buckets can be merged efficiently using bitwise operations on words. We finally use an $o(n)$ space lookup table to obtain the count or list of colors present in the merged answer.

The solution to d-dimensional colored range reporting for each bucket is obtained by building a d-dimensional range tree for the m points, which uses a new linear space and $O(\lg\lg m)$ time solution to restricted one-dimensional colored range reporting as the last auxiliary data structure. In total, each bucket requires $O(m \lg^{d-1} m)$ space and $O(\lg^{d-1} m \lg\lg m)$ query time. In two dimensions, we reduce the space to linear by only storing representative rectangles in the range tree covering $O(\lg m)$ points each. Using the linear space range reporting data structure by Nekrich [Nek09], we enumerate and check the underlying points for each of the $O(\lg m)$ range tree leaves intersecting the query range, costing us a small penalty of $O(\lg^\varepsilon n)$ per point. We thus obtain a query time of $O(\lg^{2+\varepsilon} m)$ for two-dimensional buckets in linear space.

Using classic results on dynamisation of range trees, we can dynamise the data structure with a little additional cost in query time. Previously, there was no known dynamic data structures with $o(n)$ query time for colored range counting in $d \geq 2$ dimensions and colored range reporting in $d \geq 3$ dimensions. Consequently, this is the first such dynamic data structure.

Theorem 8.1.3. *There is a dynamic $O(n(\lg\lg^c n)^{d-1})$ space data structure storing n d-dimensional colored points each assigned one of σ colors. The data structure answers colored range counting and reporting queries in time $O((\sigma/\lg n + n/\lg^c n)(\lg\lg^c n)^d)$. Reporting requires a additive $O(k)$ time and updates are supported in $O((\lg\lg^c n)^d)$ amortised time.*

Finally, if paying a little extra space, we can get a solution to the problem where the query time is bounded by the number of distinct colors instead of the number of points. This is simply done by not splitting color groups into buckets, giving the following result.

Corollary 8.1.3.1. *There is a $O(n \lg^{d-1} n)$ space data structure for n d-dimensional colored points each assigned one of σ colors. The data structure answers colored range counting and reporting queries in time $O(\sigma \lg^{d-2} n \lg\lg n)$. Reporting requires a additive $O(k)$ time.*

In two dimensions this is a logarithmic improvement over the solution where a range emptiness data structure is stored for each color at the expense of a $\lg n$ factor additional space. The above approach can be combined with the range emptiness data structure by Nekrich [Nek09] to obtain an output-sensitive result where a penalty is paid per color reported:

Corollary 8.1.3.2. *There is a $O(n \lg n)$ space data structure storing n two-dimensional colored points each assigned one of σ colors. The data structure answers colored range counting and reporting queries in time $O(\sigma + k \lg n \lg\lg n)$.*

8.1.2 Previous Results

Colored range counting. Colored range counting is challenging, with a large gap in the known bounds compared to standard range counting, especially in two or more dimensions. For example, a classic range tree solves two-dimensional standard range counting in logarithmic time and $O(n \lg n)$ space, but no polylogarithmic time solutions in $o(n^2)$ space are known for colored range counting.

Larsen and van Walderveen [Lar+13; Gup+95] showed that colored range counting in one dimension is equivalent to two-dimensional standard range counting. Thus, the optimal $O(\lg n / \lg \lg n)$ upper bound for two-dimensional standard range counting by JáJá et al. [JáJ+05] which matches a lower bound by Patrascu [Păt07] is also optimal for one-dimensional colored range counting.

In two dimensions, Gupta et al. [Gup+95] show a solution using $O(n^2 \lg^2 n)$ space that answers queries in $O(\lg^2 n)$ time. They obtain their result by storing n copies of a data structure which is capable of answering three-sided queries. The same bound was matched by Kaplan et al. [Kap+07] with a completely different approach in which they reduce the problem to standard orthogonal range counting in higher dimensions. Kaplan et al. also present a tradeoff solution with $O(X \lg^7 n)$ query time and $O((\frac{n}{X})^2 \lg^6 n + n \lg^4 n)$ space for $1 \leq X \leq n$. Observe that the minimal space use for the tradeoff solution is $O(n \lg^4 n)$.

In $d > 2$ dimensions, the only known non-trivial solutions are by Kaplan et al. [Kap+07]. One of their solutions answers queries in $O(\lg^{2(d-1)} n)$ time and $O(n^d \lg^{2(d-1)} n)$ space, and they also show a number of tradeoffs, the best one having $O(X \lg^{d-1} n)$ query time and using $O((\frac{n}{X})^{2d} + n \lg^{d-1} n)$ space for $1 \leq X \leq n$. In this case, the minimal space required by the tradeoff is $O(n \lg^{d-1} n)$.

Kaplan et al. [Kap+07] showed that answering n two dimensional colored range counting queries in $O(n^{p/2})$ time (including all preprocessing time) yields an $O(n^p)$ time algorithm for multiplying two $n \times n$ matrices. For $p < 2.373$, this would improve the best known upper bound for matrix multiplication [Wil12]. Thus, solving two dimensional colored range counting in polylogarithmic time per query and $O(n \operatorname{polylg} n)$ space would be a major breakthrough. This suggest that even in two dimensions, no polylogarithmic time solution may exist.

Colored range reporting. The colored range reporting problem is well-studied [Jan+93; Nek12; Nek+13; Nek14; Gag+12b; Shi+05; Gup+95; Gup+97; Kre92; Mor03; Boz+95; Lar+12], with output-sensitive solutions almost matching the time and space bounds obtained for standard range reporting in one and two dimensions. In particular, Nekrich and Vitter recently gave a dynamic solution to one dimensional colored range reporting with optimal query time $O(1 + k)$ and linear space [Nek+13], while Gagie et al. earlier presented a succinct solution with query time logarithmic in the length of the query interval [Gag+12b].

In two dimensions, Shi and JaJa obtain a bound of $O(\lg n + k)$ time and $O(n \lg n)$ space [Shi+05] by querying an efficient static data structure for three-sided queries, storing each point $O(\lg n)$ times. Solutions for the dynamic two-dimensional case were developed

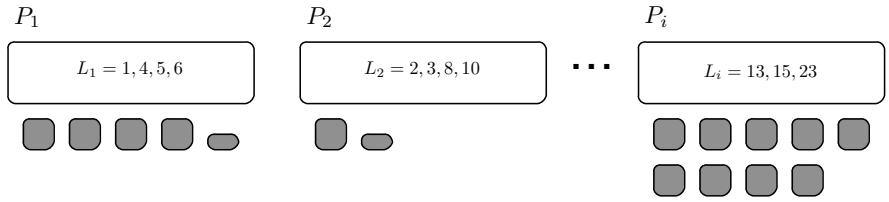


Figure 8.3: Grouping and bucketing of some point set with $f = 4$. The white boxes are the groups and the grey boxes below each group are buckets each storing $O(\lg^c n)$ points.

in [Gup+95; Boz+95], answering queries with a logarithmic penalty per answer. If the points are located on an $N \times N$ grid, Agarwal et al. [Aga+02] present a solution with query time $O(\lg\lg N + k)$ and space use $O(n \lg^2 N)$. Gupta et al. achieve a static data structure using $O(n \lg^4 n)$ space and answering queries in $O(\lg^2 n + k)$ [Gup+95] in the three-dimensional case. To the best of our knowledge, the only known non-trivial data structures for $d > 3$ dimensions are by van Kreveld and Gupta et al., answering queries in $O(\lg n + k)$ time and using $O(n^{1+\varepsilon})$ space [Kre92; Gup+97]. Other recent work on the problem include external memory model solutions when the points lie on a grid [Nek12; Lar+12; Nek14].

8.2 Colored Range Searching in Almost-Linear Space

We present here the basic approach that is modified to obtain our theorems. We first show how to partition the points into $O(\sigma/\lg n + n/\lg^c n)$ buckets each storing $m = O(\lg^c n)$ points of $f = O(\lg n)$ distinct colors, for which the results can be easily combined. We then show how to answer queries in each bucket in time $O(\lg^{d-1} m \lg\lg m)$ and space $O(m \lg^{d-1} m)$, thus obtaining Theorem 8.1.2.

8.2.1 Color Grouping and Bucketing

We partition the points of P into a number of groups P_i, where $i = 1, \ldots, \frac{\sigma}{\lg n}$, depending on their color. Each group stores all points having $f = \lg n$ distinct colors (except for the last group which may store points with less distinct colors). For each group P_i we store an ordered color list L_i of the f colors in the group. That is, a group may contain $O(n)$ points but the points have at most f distinct colors. Since colors are partitioned among groups, we can clearly answer a colored query by summing or merging the results to the same query in each group.

Each group is further partitioned into a number of buckets containing $m = \lg^c n$ points each (except for the last bucket, which may contain fewer points). Since the buckets partition the points and there cannot be more than one bucket with fewer than $\lg^c n$

points in each group, the total number of buckets is $O(\sigma/\lg n + n/\lg^c n)$. See Figure 8.3 for an example of the grouping and bucketing.

We require that each bucket in a group P_i supports answering *restricted colored range reporting queries* with an f-bit long bitstring, where the jth bit indicates if color $L_i[j]$ is present in the query area Q in that bucket. Clearly, we can obtain the whole answer for Q and P_i by using bitwise OR operations to merge answers to the restricted colored range reporting query Q for all buckets in P_i. We call the resulting bitstring $F_{i,Q}$, which indicates the colors present in the query range Q across the entire group P_i.

Finally, we store a lookup table T of size $O(\sqrt{n}\lg n) = o(n)$ for all possible bitstrings of length $\frac{f}{2}$. For each bitstring, the table stores the number of 1s present in the bitstring and the indices where the 1s are present. Using two table lookups in T with the two halves of $F_{i,Q}$, we can obtain both the number of colors present in P_i for Q, and their indices into L_i in $O(1)$ time per index.

Summarising, we can merge answers to the restricted colored range reporting queries in $O(1)$ time per bucket and obtain the full query results for each group P_i. Using a constant number of table lookups per group, we can count the number of colors present in Q. There is $O(1)$ additional cost per reported color.

8.2.2 Restricted Colored Range Reporting for Buckets

Each bucket in a group P_i stores up to $m = \lg^c n$ points colored with up to $f = \lg n$ distinct colors, and must support restricted colored range reporting queries, reporting the colors in query range Q using an f-bit long bitstring. A simple solution is to use a classic d-dimensional range tree R, augmented with an f-bit long bitstring for each node on the last level of R (using the L_i ordering of the f colors). The colors within the range can thus be reported by taking the bitwise OR of all the bitstrings stored at the $O(\lg^d m)$ summary nodes of R spanning the range in the last level. This solution takes total time $O(\frac{f}{w}\lg^d m) = O(\lg^d m)$ and space $O(m\lg^{d-1} m\frac{f}{w}) = O(m\lg^{d-1} m)$, and it can be constructed in time $O(m\lg^{d-1} m)$ by building the node bitstrings from the leaves and up (recall that $w = \Theta(\lg n)$ is the word size).

The above solution is enough to obtain some of our results, but we can improve it by replacing the last level in R with a new data structure for restricted one-dimensional colored range reporting over integers that answer queries in time $O(\lg\lg m)$ and linear space. A query may perform $O(\lg^{d-1} m)$ one-dimensional queries on the last level of the range tree, so the query time is reduced to $O(\lg^{d-1} m\lg\lg m)$ per bucket. The new data structure used at the last level is given in the next section.

Observe that though the points are not from a bounded universe, we can remap a query in a bucket to a bounded universe of size m in time $O(\lg m)$ and linear space per dimension. We do so for the final dimension, noting that we only need to do it once for all $O(\lg^{d-1} m)$ queries in the final dimension.

1D restricted colored range reporting on integers.

Given $O(m)$ points in one dimension from a universe of size m, each colored with one of $f = \lg n$ distinct colors, we now show how to report the colors contained in a query range in $O(\lg\lg m)$ time and linear space, encoded as an f-bit long bitstring. First, partition the points into $j = O(m/\lg m)$ intervals spanning $\Theta(\lg m)$ consecutive points each. Each interval is stored as a balanced binary search tree of height $O(\lg\lg m)$, with each node storing a f-bit long bitstring indicating the colors that are present in its subtree. Clearly, storing all these trees take linear space.

We call the first point stored in each interval a *representative* and store a predecessor data structure containing all of the $O(m/\lg m)$ representatives of the intervals. Also, each representative stores $O(\lg m)$ f-bit long bitstrings, which are summaries of the colors kept in the $1, 2, \ldots, 2^{\lg j}$ neighboring intervals. We store these bitstrings both towards the left and the right from the representative, in total linear space.

A query $[a, b]$ is answered as follows. We decompose the query into two parts, first finding the answer for all intervals fully contained in $[a, b]$, and then finding the answer for the two intervals that only intersect $[a, b]$. The first part is done by finding the two outermost representatives inside the interval (called a', b', where $a \leq a' \leq b' \leq b$) by using predecessor queries with a and b on the representatives. Since we store summaries for all power-of-2 neighboring intervals of the representatives, there are two bitstrings stored with a' and b' which summarises the colors in all fully contained intervals.

To find the answer for the two intervals that contain a or b, we find $O(\lg\lg m)$ nodes of the balanced binary tree for the interval and take the bitwise OR of the bitstrings stored at those nodes in $O(\lg\lg m)$ total time. Using one of the classic predecessor data structures [Emd$^+$76; Meh$^+$90; Wil83] for the representatives, we thus obtain a query time of $O(\lg\lg m)$ and linear space.

8.3 2D Colored Range Searching in Linear Space

To obtain linear space in two dimensions and the proof of Theorem 8.1.1, we use the same grouping and bucketing approach as in Section 8.2. For each group $P_{i'}$, we only change the solution of each bucket B_i in $P_{i'}$, recalling that B_i contains up to $m = \lg^c n$ points with $f = \lg n$ distinct colors, so as to use linear space instead of $O(m \lg m)$ words of space.

We store a linear space 2D standard range reporting data structure A_i due to Nekrich [Nek09] for all points in the bucket B_i. As shown in [Nek09], A_i supports orthogonal standard range reporting queries in $O(\lg m + r \lg^\varepsilon m)$ time and updates in $O(\lg^{3+\varepsilon} m)$ time, where r is the reported number of points and $\varepsilon > 0$.

We also store a simple 2D range tree R_i augmented with f-bit long bitstrings on the last level as previously described in Section 8.2.2, but instead of storing points in R_i, we reduce its space usage by only storing areas covering $O(\lg m)$ points of B_i each. This can be done by first building R_i taking $O(m \lg m)$ space and time, and then cutting off subtrees at nodes at maximal height (called cutpoint nodes) such that at most $c' \lg m$ points are covered by each cutpoint node, for a given constant $c' > 0$. In this way, each cutpoint

node is implicitly associated with $O(\lg m)$ points, which can be succinctly represented with $O(1)$ words as they all belong to a distinct 2D range. Note that the parent of a cutpoint node has $\Omega(\lg m)$ descending points, hence there are $O(m/\lg m)$ cutpoint nodes.

A query is answered by finding $O(\lg^2 m)$ summary nodes in R_i that span the entire query range Q. Combining bitstrings as described in Section 8.2.2, the colors for all fully contained ranges that are not stored in the leaves can thus be found. Consider now one such leaf ℓ covering an area intersecting Q: since the $O(\lg m)$ points spanned by ℓ may not be all contained in Q, we must check those points individually. Recall that the points associated with ℓ are those spanning a certain range Q', so they can be succinctly represented by Q'. To actually retrieve them, we issue a query Q' to A_i, check which ones belong to $Q' \cap Q$, and build a bitstring for the colors in $Q' \cap Q$. We finally merge the bitstrings for all summary nodes and intersecting leaves in constant time per bitstring to obtain the final result.

The time spent answering a query is $O(\lg^2 m)$ to find all bitstrings in non-leaf nodes of R_i and to combine all the bitstrings. The time spent finding the bitstring in leaves is $O(\lg^{1+\varepsilon} m)$ per intersecting leaf as we use Nekrich's data structure A_i with $r = O(\lg m)$. Observe that only two leaves spanning a range of $O(\lg m)$ points may be visited in each of the $O(\lg m)$ second level data structures visited, so the time spent in all leaves is $O(\lg^{2+\varepsilon} m)$, which is also the total time. Finally, since we reduced the size of the range tree by a factor $\Theta(\lg m)$, the total space usage is linear. This concludes the proof of Theorem 8.1.1.

8.4 Dynamic Data Structures

We now prove Theorem 8.1.3 by discussing how to support operations INSERT(p, c) and DELETE(p), inserting and deleting a point p with color c, respectively. Note that the color c may be previously unused. We still use parameters f and m to denote the number of colors in groups and points in buckets, respectively. We first give bounds on how to update a bucket, and then show how to support updates in the color grouping and point bucketing.

8.4.1 Updating a Bucket

Updating a bucket with a point corresponds to updating a d-dimensional range tree using known techniques in dynamic data structures. Partial rebuilding [And99; Ove87] requires amortised time $O(\lg^d m)$, including updating the bitstrings in the partially rebuilt trees and in each node of the last level trees (which takes constant time). Specifically, the bitstrings for the $O(\lg^{d-1} m)$ trees on the last level where a point was updated may need to have the bitstrings fixed on the path to the root on that level. This takes time $O(\lg m)$ per tree, giving a total amortised update time of $O(\lg^d m)$.

8.4.2 Updating Color Grouping and Point Bucketing

When supporting INSERT(p, c), we first need to find the group to which c belongs. If the color is new and there is a group P_i with less than f colors, we must update the color list L_i. Otherwise, we can create a new group P_i for the new color. In the group P_i, we must find a bucket to put p in. If possible, we put p in a bucket with less than m points, or otherwise we create a new bucket for p. Keeping track of sizes of groups and buckets can be done using priority queues in time $O(\lg \lg n)$. Note that we never split groups or buckets on insertions.

As for supporting DELETE(p), we risk making both groups and buckets underfull, thus requiring a merge of either. A bucket is underfull when it contains less than $m/2$ points. We allow at most one underfull bucket in a group. If there are two underfull buckets in a group, we merge them in time $O(m \lg^d m)$. Since merging buckets can only happen after $\Omega(m)$ deletions, the amortized time for a deletion in this case is $O(\lg^d m)$. A group is underfull if it contains less than $f/2$ colors and, as for buckets, if there are any two underfull groups P_i, P_j, we merge them. When merging P_i, P_j into a new group P_r, we concatenate their color lists L_i, L_j into L_r, removing the colors that are no more present while keeping the relative ordering of the surviving colors from L_i, L_j. In this way, a group merge does not require us to merge the underlying buckets, as points are partitioned arbitrarily into the buckets. However, a drawback arises: as the color list L_r for the merged group is different from the color lists L_i, L_j used for answering bucket queries, this may introduce errors in bucket query answers. Recall that an answer to a bucket query is an f-bit long bitstring which marks with 1s the colors in L_i that are in the range Q. So we have a bitstring for L_i, and one for L_j, for the buckets previously belonging to P_i, P_j, but we should instead output a bitstring for L_r in time proportional to the number of buckets in P_r. We handle this situation efficiently as discussed in Section 8.4.3.

8.4.3 Fixing Bucket Answers During a Query

As mentioned in Section 8.4.2, we do not change the buckets when two or more groups are merged into P_r. Consider the f-bit long bitstring b_i that is the answer for one merged group, say P_i', relative to its color list, say L_i. However, after the merge, only a sublist $L_i' \subseteq L_i$ of colors survives as a portion of the color list L_r for P_r. We show how to use L_i and L_i' to contribute to the f-bit long bitstring a that is the answer to query Q for the color list L_r in P_r. The time constraint is that we can spend time proportional to the number, say g, of buckets in P_r.

We need some additional information. For each merged group P_i', we create an f-bit long bitstring v_i with bit j set to 1 if and only if color $L_i[j]$ survives in L_r (i.e. some point in P_r has color $L_i[j]$). We call v_i the *possible answer bitstring* and let o_i be the number of 1s in v_i: in other words, L_i' is the sublist built from L_i by choosing the colors $L_i[j]$ such that $v_i[j] = 1$, and $o_i = |L_i'|$.

Consider now the current group P_r that is the outcome of $h \leq f$ old merged groups, say $P_1', P_2', \ldots, P_h'$ in the order of the concatenation of their color lists, namely, $L_r = L_1' \cdot L_2' \cdots L_h'$. Since the number of buckets in P_r is $g \geq h$, we can spend $O(g)$ time to obtain

the f-bit long bitstrings $b_1, b_2, \ldots, b_h$, which are the answers for the old merged groups $P'_1, P'_2, \ldots, P'_h$, and combine them to obtain the answer a for P_r.

Here is how. The idea is that the bits in a from position $1 + \sum_{l=1}^{i-1} o_l$ to $\sum_{l=1}^{i} o_l$ are reserved for the colors in L'_i, using 1 to indicate which color in L'_i is in the query Q and 0 which is not. Let us call b'_i this o_i-bit long bitstring. Recall that we have b_i, which is the f-bit long bitstring that is the answer for P'_i and refers to L_i, and also v_i, the possible answer bitstring as mentioned before.

To obtain b'_i from b_i, v_i and o_i in constant time, we would like to employ a lookup table $S[b, v]$ for all possible f-bitstrings b and v, precomputing all the outcomes (in the same fashion as the Four Russians trick). However, the size of S would be $2^f \times 2^f \times f$ bits, which is too much (remember $f = \lg n$). We therefore build S for all possible $(f/3)$-bitstrings b and v, so that S uses $o(n)$ words of memory. This table is periodically rebuilt when n doubles or becomes one fourth, following a standard rebuilding rule. We therefore compute b'_i from b_i, v_i by dividing each of them in three parts, looking up S three times for each part, and combining the resulting three short bitstrings, still in $O(1)$ total time.

Once we have found $b'_1, b'_2, \ldots, b'_h$ in $O(h)$ time as shown above, we can easily concatenate them with bitwise shifts and ORs, in $O(h)$ time, so as to produce the wanted answer $a = b'_1 \cdot b'_2 \cdots b'_h$ as a f-bit long bitstring for P_r and its color list L_r. Recall that P_r consists of h buckets where $h \leq g \leq f$. Indeed, if it were $h > f$, there would be some groups with no colors. Since $\Omega(f)$ deletions must happen before two groups are merged, we can clean and remove the groups that have no more colors, i.e, with $o_i = 0$, and maintain the invariant that $h \leq g \leq f$.

8.5 Open Problems

There are a lot of loose ends in colored range searching that deserve to be investigated, and we will shortly outline a few of them. The hardness reduction by Kaplan et al. [Kap+07] gives hope that colored range counting can be proven hard, and we have indeed assumed that this is the case here. If taking instead an upper bound approach as this paper, improved time bounds obtainable in little space, or with some restriction on the number of colors, would be very interesting motivated by the large scale applications of the problem.

9 Indexing Motion Detection Data for Surveillance Video

Philip Bille* Inge Li Gørtz* Søren Vind†

Technical University of Denmark

Abstract

We show how to compactly index video data to support fast *motion detection* queries. A query specifies a time interval T, a area A in the video and two thresholds v and p. The answer to a query is a list of timestamps in T where $\geq p$% of A has changed by $\geq v$ values.

Our results show that by building a small index, we can support queries with a speedup of two to three orders of magnitude compared to motion detection without an index. For high resolution video, the index size is about 20% of the compressed video size.

*Supported by a grant from the Danish Council for Independent Research | Natural Sciences.

†Supported by a grant from the Danish National Advanced Technology Foundation.

9.1 Introduction

Video data require massive amounts of storage space and substantial computational resources to subsequently analyse. For motion detection in video surveillance systems, this is particularly true, as the video data typically have to be stored (in compressed form) for extended periods for legal reasons and motion detection requires time-consuming decompressing and processing of the data. In this paper, we design a simple and compact index for video data that supports efficient motion detection queries. This enables fast motion detection queries on a selected time interval and area of the video frame without the need for decompression and processing of the video file.

Problem & Goal

A *motion detection query* $\texttt{MD}(\texttt{T}, \texttt{A}, \texttt{v}, \texttt{p})$ specifies a time range T, an area A, and two thresholds $v \in [0, 255]$ and $p \in [0, 100]$. The answer to the query is a list of timestamps in T where the amount of motion in A exceeds thresholds v and p, meaning that $\geq p\%$ of the pixels in A changed by $\geq v$ pixel values. Our goal is build an index for video data that supports motion detection queries. Ideally, the index should be small compared to the compressed size of the video data and should support queries significantly faster than motion detection without an index.

Related Work

Several papers have considered the problem of *online* motion detection, where the goal is to efficiently identify movement in the video in real time, see e.g. [Sac+94; Tia+05; Hu+04; Cut+00; Hua11]. Previous papers [Du+14; Kao+08] mentions indexing movement of objects based on motion trajectories embedded in video encoding. However, to the best of our knowledge, our solution is the first to show a highly efficient index for motion detection queries on the raw video.

Our Results

We design a simple index for surveillance video files, which support motion detection queries efficiently. The performance of the index is tested by running experiments on a number of surveillance videos that we make freely available for use. These test videos capture typical surveillance camera scenarios, with varying amounts of movement in the video.

Our index reduces the supported time- and area-resolution of queries by building summary *histograms* for the number of changed pixels in a number of *regions* of frames succeeding each other. Histograms for a frame are compressed and stored using an off-the-shelf compressor. Queries are answered by decompressing the appropriate histograms and looking up the answer to the query in the histograms.

The space required by the index varies with the amount of motion in the video and the region resolution supported. The query time only varies slightly. Compared to motion detection without an index we obtain:

- A query time speedup of several orders of magnitude, depending on the resolution of the original video. The choice of compressor has little influence on the time required to answer a query by the index.

- A space requirement which is $10 - 90\%$ of the compressed video. The smallest relative space requirement occur for high resolution video. Quadrupling the region resolution roughly doubles the space use.

Furthermore, as the resolution of the video increases, the time advantage of having an index grows while the additional space required by the index decreases compared to the compressed video data. That is, the index performs increasingly better for higher resolution video.

9.2 The Index

A `MD(T,A,v,p)` query spans several dimensions in the video file: The time dimension given by T and two spatial dimensions given by A. However, as high-dimensional data structures for range queries typically incur high space cost, we have decided to not implement our index using such data structures. Instead, we create a large number of two-dimensional data structures for the pixel value difference for each successive pair of frames, called a *difference frame*. Answering a query then involves querying the data structures for all difference frames in T.

We restrict the query area A to always be a collection of *regions*, $r_1, \ldots, r_k$. The height and width of a region is determined by the video resolution and the number of regions in each dimension of the video (if other query areas are needed, the index can be used as a filter). For simplicity, we assume that the pixel values are grey-scale.

The index stores the following. For each region r and difference frame F, we store a histogram $H_{F,r}$, counting for each value $0 \leq c \leq 255$ the number of pixels in the region changed by at least c pixel values. Clearly a histogram can be stored using 256 values only. While this may exceed the number of pixels in a region when storing many regions per frame, it generally does not. However, because modern video encoding is extremely efficient, the raw histograms may take more space than the compressed video (especially for low video resolutions). Thus, we compress the histograms using an off-the-shelf compressor before storing them to disk.

To answer a `MD(T,A,v,p)` query, we decompress and query the histograms for each region in A across all difference frames in T. Let $|r|$ denote the number of pixels in region r. For a specific difference frame F, we calculate $p' = \sum_{r \in A} H_{F,r}[v] / \sum_{r \in A} |r|$, which is exactly the percentage of pixels in A changed by $\geq v$ pixel values. Thus, if $p' \geq p$, frame F is a matching timestamp.

Table 9.1: Surveillance video recording samples used for testing. Videos were encoded at 29.97fps using H264/MP4.

			Size (MB)		
Scenario	**Length (s)**	**Motion amount**	**1080p**	**720p**	**480p**
Office	60	Low	9.0	3.0	1.2
Students	60	Medium	27.3	7.8	3.3
Rain	60	High	67.6	18.1	4.6

9.3 Experiments

9.3.1 Experimental setup

All experiments ran on an Apple Macbook Pro with an Intel Core i7-2720QM CPU, 8GB ram and a 128GB Apple TS128C SSD disk, plugged into the mains power. All reported results (both time and space) were obtained as the average over three executions (we note that the variance across these runs was extremely low).

9.3.2 Data sets

We tested our index on the following three typical video surveillance scenarios, encoded at 29.97fps using H264/MP4 (reference [Vin14]). We use different video resolutions (1920×1080, 1280×720 and 852×480 pixels). See Table 9.1.

Office Recording of typical workday activities in a small well-lit office with three people moving. The image is almost static, only containing small movements by the people. There is very little local motion in the video.

Students Recording of a group of students working in small groups, with trees visible through large windows that give a lot of reflection. People move about, which gives a medium amount of motion across most of the frame.

Rain A camera mounted on the outside of a building, recording activities along the building and looking towards another building. It is windy and raining, which combined with many trees in the frame creates a high amount of motion in the entire frame.

9.3.3 Implementation

The system was implemented in Python using bindings to efficient C/C++ libraries where possible. In particular, OpenCV and NumPy were used extensively, and we used official python bindings to the underlying C/C++ implementations of the compressors.

The implementation uses a number of tunable parameters, see Table 9.2. The source code can be found at [Vin14].

9.4 Main Results

We now show the most significant experimental results for our index compared to the trivial method. We show results on both query time and index space when applicable. Unless otherwise noted, the index was created for a video size of 1080p, storing 1024 regions/frame, 3 frames/second, 1 frame/file, using linear packing and zlib-6 compression. We will only give detailed results for the students scenario, as the index performs relatively worst in this case.

9.4.1 Regions Queried

The first set of experiments show the influence of the number of regions queried in the image on the total query time and also check if one scenario diverts significantly from the others in query time performance. The number of regions queried was both extremes (1 and 1024).

Table 9.3 summarises the results, with the index size shown relative to the video size. The query time of the index does not vary with the video input, while that is the case for the video compression approach. Observe that though the total time spent answering a query using the index scales with the number of regions queried, it never exceeds 1 s, while the video approach spends at least 250 s in all cases. Thus, the index answers queries at least two orders of magnitude faster than the video compression approach.

Table 9.2: Tunable parameters for use in experiments.

Name	Description
Frames/Second	The frame rate of the difference frames to index.
Regions/Frame	The number of regions to divide a frame into.
Compressor	The compressor used to compress the histograms.
Frames/File	The number of frames for which the histograms should be stored in the same file on disk
Packing	Which strategy should be used when storing histograms for more than one frame in same file on disk

Table 9.3: Index query time versus video query time for 1 and 1024 regions queried. Size of index compared to the video size.

		Time (s)			
Scenario	**Query Reg.**	**Index**	**Video**	**Speedup**	**Size**
Office	1	0.17	463.51	2726×	24.4%
	1024	0.81	467.17	576×	2.2 MB
Students	1	0.20	249.76	1248×	20.1%
	1024	0.83	253.86	305×	5.5 MB
Rain	1	0.20	351.40	1757×	8.0%
	1024	0.83	355.98	428×	5.4 MB

Table 9.4: Speedup for index query time compared to video query time for students scenario, with varying input video resolutions and number of regions queried.

	Speedup / Query Reg.			
Resolution	**1**	**64**	**1024**	**Size**
480p	454×	368×	102×	75.8%
720p	903×	745×	219×	47.4%
1080p	1248×	1060×	305×	20.1%

The very small difference in query time for both extremes is surprising, since it directly influences the number of pixels to analyse in each difference frame. The relative increase is much larger for the index query than the video, meaning that the time spent performing the actual query is a larger fraction of the total query time for the index (as shown in Section 9.5). We believe that the reason the office scenario has the worst performance is that the video compression is most efficient here (and thus harder to decompress).

Table 9.4 shows that the relative index query time increases when fewer regions are queried. However, even when querying all regions the index has a performance which is at least two orders of magnitude quicker than the video.

9.4.2 Resolution Comparison

Table 9.4 show the influence of the video resolution on the speedup obtained for the students scenario. The difference in space required for the index with varying resolutions is shown in Table 9.5. Note that the index times are almost always the same (varies between 0.2s and 1s), while the video query times decrease from around 250s at 1080p to 75s at 480p, and we thus only report the relative speedup for different numbers of re-

Table 9.5: Size of indexes for varying input video resolutions.

	Index Size (MB)			Index Size (%)		
Scenario	**1080p**	**720p**	**480p**	**1080p**	**720p**	**480p**
Office	2.2	1.5	1.1	24.4%	50.0%	91.7%
Students	5.5	3.7	2.5	20.1%	47.4%	75.8%
Rain	5.4	3.7	2.2	8.0%	20.4%	47.8%

Table 9.6: Index size for varying number of stored regions and resolutions in the students scenario.

	Index Size (MB)			Index Size (%)		
Store Reg.	**1080p**	**720p**	**480p**	**1080p**	**720p**	**480p**
64	1.1	0.8	0.6	4.0%	10.2%	18.2%
256	2.3	1.7	1.2	8.4%	21.8%	36.4%
1024	5.5	3.7	2.5	20.1%	47.4%	75.8%

gions queried. The relative index performance compared to the video approach improves in both space and time with larger video resolution. The index query time varies very little with the resolution (which is as expected, since the number of histogram values to check does not change).

9.4.3 Regions Stored

Clearly, the number of regions stored by the index has an influence on the index size (as this directly corresponds to the number of histograms to store). However, from Table 9.6, it is clear that the influence is smaller than would be expected. In fact, due to the more efficient compression that is achieved, quadrupling (4×) the number of regions only causes the index to about double (2×) in size. That is, it is relatively cheap to increase the resolution of regions.

9.5 Other Results

In this section, we review the index performance when varying the different parameters listed in Table 9.2. Changing the index parameters had insignificant influence on query times, so we only show results for the index space when varying the parameters. The largest contributor to the index advantage over the video decompression method is the idea of storing an index, and thus we only briefly review the results when varying the parameters.

Table 9.7: Index size comparison for different compressors.

	Index Size (MB) / Compressor				
Scenario	`lz4`	`snappy`	`zlib`	`lzma`	`bzip2`
Office	2.9	6.6	2.2	1.7	1.5
Students	7.5	10.9	5.5	4.1	3.5
Rain	7.4	10.6	5.4	4.2	3.4

Table 9.8: Index compression time for different compressors.

	Compression Time (s) / Compressor				
Scenario	`lz4`	`snappy`	`zlib`	`lzma`	`bzip2`
Office	0.04	0.07	1.01	29.67	30.67
Students	0.07	0.11	1.29	32.69	31.80
Rain	0.07	0.11	1.41	31.89	31.92

Compressor

Table 9.7 shows the size of the index when the histograms are compressed using a number of different compressors, and Table 9.8 shows the time spent compressing the index in total (remember the input is a 60s video file). In all of the tests in the previous section, we have used the `zlib-6` compressor, as it gives a good tradeoff between compression time and space use. If one can spare the computing resources, there is hope for almost halving the index space if switching to `bzip2` compression. Note, however, that the query time increases with a slower decompressor (especially when querying few regions, as shown in Section 9.5).

Query Time Components

To see the influence of the compressor used, we determined how much of the total query time is spent checking the decompressed values, compared to the amount of time spent decompressing the video frames or histograms. The results are in Table 9.9. It is evident that the video resolution and our chosen index compressor only has a small influence on the total query time when the number of regions queried is large: Around 75% of the time is spent on the actual query. As for the video query time, $> 95\%$ of the total time is spent decompressing the video, with the actual query time being very insignificant in all cases. In absolute terms, the query times for the index and the decompressed video approach are comparable (difference around $10\times$) after decompression.

Table 9.9: Fraction of time spent analysing regions of total query time for students scenario (remainder is decompression).

	Index Query Time		Video Query Time		
Resolution	**1**	**1024**	**1**	**1024**	**Size**
480p	2.63%	76.73%	0.01%	1.48%	75.8%
720p	3.27%	76.84%	0.01%	1.84%	47.4%
1080p	3.08%	73.79%	0.02%	3.84%	20.1%

Frames/File & Histogram Packing Strategies

We tested the influence on the index size if storing more than one difference frame per file on disk. We tested four different value packing strategies: linear, binned, reg-linear, reg-binned. Consider a frame F with two region histograms r_1, r_2. In the linear strategy, we just write all values from r_1 followed by all values from r_2. In the binned strategy, we interleave the value for the same histogram index from all regions in a frame, i.e. we wite $r_1[0]r_2[0]r_1[1]r_2[1]\ldots r_1[255]r_2[255]$ on disk. When storing multiple frames F_1, F_2 in a file, assume r_1, r_2 has the same spatial coordinate in both frames. Then the reg-linear strategy writes r_1 followed by r_2, while the reg-binned strategy interleaves the values as before.

The hope is that this would result in more efficient compression, since the histogram for the same region may be assumed to be very similar across neighbouring frames. However, our results show that storing more frames per file and changing the packing strategy had very little effect on the index efficiency for storing many regions. One exception is when storing few regions (less than 64), increasing the number of frames per file decreases the index size due to the added redundancy available for the compressor.

9.6 Conclusion

We have shown an index for motion detection data from surveillance video cameras that provides a speedup of at least two orders of magnitude when answering motion detection queries in surveillance video. The size of the index is small compared to the video files, especially for high resolution video.

Bibliography

[Abo+04] Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. "Replacing suffix trees with enhanced suffix arrays". In: *JDA* 2.1 (2004), pp. 53–86 (cit. on pp. 74, 75).

[Abo+10] Mohamed Ibrahim Abouelhoda and Moustafa Ghanem. "String Mining in Bioinformatics". In: *Scientific Data Mining and Knowledge Discovery*. 2010, pp. 207–247 (cit. on p. 68).

[Aga+02] Pankaj K. Agarwal, Sathish Govindarajan, and S. Muthukrishnan. "Range searching in categorical data: Colored range searching on grid". In: *Proc. 10th ESA*. 2002, pp. 17–28 (cit. on p. 99).

[Alo+99] Noga Alon, Yossi Matias, and Mario Szegedy. "The Space Complexity of Approximating the Frequency Moments". In: *J. Comput. Syst. Sci.* 58.1 (1999), pp. 137–147 (cit. on pp. 11, 54).

[Als+00] Stephen Alstrup and Jacob Holm. "Improved algorithms for finding level ancestors in dynamic trees". In: *Proc. 27th ICALP*. 2000, pp. 73–84 (cit. on p. 27).

[Ami+07] Amihood Amir, Gad M. Landau, Moshe Lewenstein, and Dina Sokol. "Dynamic text and static pattern matching". In: *ACM Trans. Alg.* 3.2 (2007), p. 19 (cit. on pp. 10, 40, 41, 44).

[Ami+92] Amihood Amir, Martin Farach, and Yossi Matias. "Efficient randomized dictionary matching algorithms". In: *Proc. 3rd CPM*. 1992, pp. 262–275 (cit. on pp. 8, 24).

[And+00] Arne Andersson and Mikkel Thorup. "Tight (er) worst-case bounds on dynamic searching and priority queues". In: *Proc. 32nd STOC*. 2000, pp. 335–342 (cit. on p. 18).

[And+06] Alexandr Andoni and Piotr Indyk. "Efficient algorithms for substring near neighbor problem". In: *Proc. 17th SODA*. 2006, pp. 1203–1212 (cit. on pp. 8, 24).

[And+07] Arne Andersson and Mikkel Thorup. "Dynamic ordered sets with exponential search trees". In: *J. ACM* 54.3 (2007), p. 13 (cit. on p. 31).

[And99] Arne Andersson. "General balanced trees". In: *J. Algorithms* 30.1 (1999), pp. 1–18 (cit. on p. 102).

[Apo+06] Alberto Apostolico, Matteo Comin, and Laxmi Parida. "Bridging lossy and lossless compression by motif pattern discovery". In: *General Theory of Information Transfer and Combinatorics*. 2006, pp. 793–813 (cit. on p. 68).

[Apo+11] Alberto Apostolico, Cinzia Pizzi, and Esko Ukkonen. “Efficient algorithms for the discovery of gapped factors”. In: *Algorithms Mol. Biol.* 6 (2011), p. 5 (cit. on p. 13).

[Arg+08] Lars Arge, Mark De Berg, Herman Haverkort, and Ke Yi. “The Priority R-tree: A practically efficient and worst-case optimal R-tree”. In: *ACM Trans. Alg.* 4.1 (2008), p. 9 (cit. on pp. 14, 84).

[Ari+07] Hiroki Arimura and Takeaki Uno. “An efficient polynomial space and polynomial delay algorithm for enumeration of maximal motifs in a sequence”. In: *JCO* (2007) (cit. on pp. 13, 66, 67).

[Bab85] László Babai. “Trading group theory for randomness”. In: *Proc. 17th STOC*. 1985, pp. 421–429 (cit. on p. 57).

[Bak95] Brenda S. Baker. “On finding duplication and near-duplication in large software systems”. In: *Proc. 2nd WCRE*. 1995, pp. 86–95 (cit. on p. 68).

[Bar+10] Jérémy Barbay, Francisco Claude, and Gonzalo Navarro. “Compact rich-functional binary relation representations”. In: *Proc. 9th LATIN*. 2010, pp. 170–183 (cit. on p. 84).

[Bay+72] Rudolf Bayer and Edward M. McCreight. “Organization and maintenance of large ordered indexes”. In: *Acta Informatica* 1.3 (1972), pp. 173–189 (cit. on pp. 14, 84).

[Bel+10] Djamal Belazzougui, Paolo Boldi, Rasmus Pagh, and Sebastiano Vigna. “Fast prefix search in little space, with applications”. In: *Proc. 18th ESA*. 2010, pp. 427–438 (cit. on p. 49).

[Bel+12] Djamal Belazzougui, Paolo Boldi, and Sebastiano Vigna. “Predecessor search with distance-sensitive query time”. In: *arXiv:1209.5441* (2012) (cit. on p. 31).

[Ben+04] Michael A. Bender and Martin Farach-Colton. “The level ancestor problem simplified”. In: *Theor. Comput. Sci.* 321 (2004), pp. 5–12 (cit. on p. 27).

[Ben+80] Jon Louis Bentley and James B. Saxe. “Decomposable searching problems I. Static-to-dynamic transformation”. In: *J. Algorithms* 1.4 (1980), pp. 301–358 (cit. on pp. 14, 84).

[Ben75] Jon Louis Bentley. “Multidimensional binary search trees used for associative searching”. In: *Comm. ACM* 18.9 (1975), pp. 509–517 (cit. on pp. 14, 84).

[Ben79] Jon Louis Bentley. “Multidimensional binary search trees in database applications”. In: *IEEE Trans. Softw. Eng.* 4 (1979), pp. 333–340 (cit. on pp. 14, 84).

[Ber+08] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications*. 3rd. 2008 (cit. on p. 94).

[Ber+13] Guillermo de Bernardo, Sandra Álvarez-Garca, Nieves R. Brisaboa, Gonzalo Navarro, and Oscar Pedreira. “Compact Querieable Representations of Raster Data”. In: *Proc. 20th SPIRE*. 2013, pp. 96–108 (cit. on pp. 14, 84).

[Ber+94] Omer Berkman and Uzi Vishkin. “Finding level-ancestors in trees”. In: *J. Comput. System Sci.* 48.2 (1994), pp. 214–230 (cit. on p. 27).

[Bil+11] Philip Bille, Gad M. Landau, Rajeev Raman, Kunihiko Sadakane, Srinivasa Rao Satti, and Oren Weimann. “Random access to grammar-compressed strings”. In: *Proc. 22nd SODA*. 2011, pp. 373–389 (cit. on pp. 9, 24–26, 28, 29).

[Bil+12a] Philip Bille, Inge Li Gørtz, Benjamin Sach, and Hjalte Wedel Vildhøj. “Time-Space Trade-Offs for Longest Common Extensions”. In: *Proc. 23rd CPM*. 2012, pp. 293–305 (cit. on p. 25).

[Bil+12b] Philip Bille, Inge Li Gørtz, Hjalte Vildhøj, and Søren Vind. “String Indexing for Patterns with Wildcards”. In: *Proc. 13th SWAT*. 2012, pp. 283–294 (cit. on p. 4).

[Bil+13] Philip Bille, Patrick Hagge Cording, Inge Li Gørtz, Benjamin Sach, Hjalte Wedel Vildhøj, and Søren Vind. “Fingerprints in Compressed Strings”. In: *Proc. 13th WADS*. 2013, pp. 146–157 (cit. on p. 3).

[Bil+14] Philip Bille, Inge Li Gørtz, Hjalte Wedel Vildhøj, and Søren Vind. “String indexing for patterns with wildcards”. In: *Theory Comput. Syst.* 55.1 (2014), pp. 41–60 (cit. on pp. 4, 11, 50).

[Bil+15] Philip Bille, Inge Li Gørtz, and Søren Vind. “Compressed Data Structures for Range Searching”. In: *Proc. 9th LATA*. 2015, pp. 577–586 (cit. on p. 4).

[Bos+09] Prosenjit Bose, Meng He, Anil Maheshwari, and Pat Morin. “Succinct orthogonal range search structures on a grid with applications to text indexing”. In: *Proc. 11th WADS*. 2009, pp. 98–109 (cit. on p. 84).

[Boz+95] Panayiotis Bozanis, Nectarios Kitsios, Christos Makris, and Athanasios K. Tsakalidis. “New Upper Bounds for Generalized Intersection Searching Problems”. In: *Proc. 22nd ICALP*. 1995, pp. 464–474 (cit. on pp. 96, 98, 99).

[Bri+09] Nieves R. Brisaboa, Susana Ladra, and Gonzalo Navarro. “K2-trees for compact web graph representation”. In: *Proc. 16th SPIRE*. 2009, pp. 18–30 (cit. on pp. 14, 84).

[Bri+95] Sergey Brin, James Davis, and Hector Garcia-Molina. “Copy detection mechanisms for digital documents”. In: *Proc. ACM SIGMOD*. 1995, pp. 398–409 (cit. on p. 68).

[Cha+03] Chia-Hui Chang, Chun-Nan Hsu, and Shao-Cheng Lui. “Automatic information extraction from semi-structured web pages by pattern discovery”. In: *Decis. Support. Syst.* 34.1 (2003), pp. 129–147 (cit. on p. 68).

[Cha+05] Moses Charikar, Eric Lehman, Ding Liu, Rina Panigrahy, Manoj Prabhakaran, Amit Sahai, and Abbi Shelat. "The smallest grammar problem". In: *IEEE Trans. Inf. Theory* 51.7 (2005), pp. 2554–2576 (cit. on p. 26).

[Cha+09] Amit Chakrabarti, Graham Cormode, and Andrew McGregor. "Annotations in data streams". In: *Proc. 36th ICALP*. 2009, pp. 222–234 (cit. on pp. 11, 54, 55, 57, 58).

[Cha+10] Timothy M. Chan and Mihai Pătraşcu. "Counting inversions, offline orthogonal range counting, and related problems". In: *Proc. 21st SODA*. 2010, pp. 161–173 (cit. on pp. 18, 19).

[Cha+14] Amit Chakrabarti, Graham Cormode, Andrew McGregor, and Justin Thaler. "Annotations in data streams". In: *ACM Trans. Alg.* 11.1 (2014), p. 7 (cit. on pp. 11, 54, 55, 57).

[Cha+15] Amit Chakrabarti, Graham Cormode, Andrew McGregor, Justin Thaler, and Suresh Venkatasubramanian. "Verifiable Stream Computation and Arthur–Merlin Communication". In: *CCC*. 2015 (cit. on p. 57).

[Cha90] Bernard Chazelle. "Lower bounds for orthogonal range searching: I. The reporting case". In: *J. ACM* 37.2 (1990), pp. 200–212 (cit. on p. 16).

[Che+04] Xin Chen, Brent Francia, Ming Li, Brian Mckinnon, and Amit Seker. "Shared information and program plagiarism detection". In: *IEEE Trans. Inf. Theory* 50.7 (2004), pp. 1545–1551 (cit. on p. 68).

[Che+12] B. G. Chern, Idoia Ochoa, Alexandros Manolakos, Albert No, Kartik Venkat, and Tsachy Weissman. "Reference based genome compression". In: *Proc. IEEE ITW*. 2012, pp. 427–431 (cit. on pp. 10, 38, 42).

[Cla+11] Francisco Claude and Gonzalo Navarro. "Self-indexed grammar-based compression". In: *Fund. inform.* 111.3 (2011), pp. 313–337 (cit. on p. 26).

[Cla83] Kenneth L. Clarkson. "Fast algorithms for the all nearest neighbors problem". In: *Proc. 24th FOCS*. Vol. 83. 1983, pp. 226–232 (cit. on pp. 14, 84).

[Cli+12] Raphaël Clifford, Markus Jalsenius, Ely Porat, and Benjamin Sach. "Pattern matching in multiple streams". In: *Proc. 23rd CPM*. 2012, pp. 97–109 (cit. on p. 62).

[Coh+96] Martin Cohn and Roger Khazan. "Parsing with suffix and prefix dictionaries". In: *Proc. 6th DCC*. 1996, pp. 180–180 (cit. on p. 42).

[Col+03] Richard Cole and Ramesh Hariharan. "Faster suffix tree construction with missing suffix links". In: *SIAM J. Comput.* 33.1 (2003), pp. 26–42 (cit. on pp. 8, 24, 58).

[Col+04] Richard Cole, Lee-Ad Gottlieb, and Moshe Lewenstein. "Dictionary matching and indexing with errors and don't cares". In: *Proc. 36th STOC*. 2004, pp. 91–100 (cit. on p. 50).

[Com+13] Matteo Comin and Davide Verzotto. “Filtering Degenerate Patterns with Application to Protein Sequence Analysis”. In: *Algorithms* 6.2 (2013), pp. 352–370 (cit. on p. 13).

[Com79] Douglas Comer. “Ubiquitous B-tree”. In: *ACM CSUR* 11.2 (1979), pp. 121–137 (cit. on pp. 14, 84).

[Cor+01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, second edition*. MIT Press, 2001 (cit. on p. 51).

[Cor+05] Graham Cormode and S. Muthukrishnan. “Substring compression problems”. In: *Proc. 16th SODA*. 2005, pp. 321–330 (cit. on pp. 8, 24).

[Cor+07] Graham Cormode and S. Muthukrishnan. “The string edit distance matching problem with moves”. In: *ACM Trans. Alg.* 3.1 (2007), p. 2 (cit. on pp. 8, 24).

[Cor+12] Graham Cormode, Michael Mitzenmacher, and Justin Thaler. “Practical verified computation with streaming interactive proofs”. In: *Proc. 3rd ITCS*. 2012, pp. 90–112 (cit. on p. 57).

[Cor+13] Graham Cormode, Michael Mitzenmacher, and Justin Thaler. “Streaming graph computations with a helpful advisor”. In: *Algorithmica* 65.2 (2013), pp. 409–442 (cit. on p. 57).

[Cro+14] Maxime Crochemore, Alessio Langiua, and Filippo Mignosi. “Note on the greedy parsing optimality for dictionary-based text compression”. In: *Theor. Comput. Sci.* 525 (2014), pp. 55–59 (cit. on p. 42).

[Cun+12] Fabio Cunial and Alberto Apostolico. “Phylogeny Construction with Rigid Gapped Motifs”. In: *J. Comp. Biol.* 19.7 (2012), pp. 911–927 (cit. on p. 13).

[Cut+00] Ross Cutler and Larry S. Davis. “Robust real-time periodic motion detection, analysis, and applications”. In: *IEEE Trans. PAMI* 22.8 (2000), pp. 781–796 (cit. on p. 106).

[Deb+99] Hervé Debar, Marc Dacier, and Andreas Wespi. “Towards a taxonomy of intrusion-detection systems”. In: *Computer Networks* 31.8 (1999), pp. 805–822 (cit. on p. 68).

[Deo+11] Sebastian Deorowicz and Szymon Grabowski. “Robust relative compression of genomes with random access”. In: *Bioinformatics* 27.21 (2011), pp. 2979–2986 (cit. on pp. 10, 42).

[Dic+09] Christian Dick, Jens Schneider, and Ruediger Westermann. “Efficient Geometry Compression for GPU-based Decoding in Realtime Terrain Rendering”. In: *Comput. Graph. Forum* 28.1 (2009), pp. 67–83 (cit. on pp. 14, 84).

[Die+94] Paul F. Dietz and Rajeev Raman. “A constant update time finger search tree”. In: *Inform. Process. Lett.* 52.3 (1994), pp. 147–154 (cit. on p. 18).

[Die89] Paul F. Dietz. “Optimal algorithms for list indexing and subset rank”. In: *Proc. 1st WADS*. 1989, pp. 39–46 (cit. on pp. 18, 39, 46, 49).

[Die91] Paul F. Dietz. “Finding Level-Ancestors in Dynamic Trees”. In: *Proc. 2nd WADS*. 1991, pp. 32–40 (cit. on p. 27).

[Do+14] Huy Hoang Do, Jesper Jansson, Kunihiko Sadakane, and Wing-Kin Sung. “Fast relative Lempel–Ziv self-index for similar sequences”. In: *Theor. Comput. Sci.* 532 (2014), pp. 14–30 (cit. on pp. 10, 38, 42).

[Dow+80] Peter J. Downey, Ravi Sethi, and Robert Endre Tarjan. “Variations on the common subexpression problem”. In: *J. ACM* 27.4 (1980), pp. 758–771 (cit. on pp. 87, 89).

[Du+14] Shan Du, Choudhury A Rahman, Saika Sharmeen, and Wael Badawy. “Event Detection by Spatio-Temporal Indexing of Video Clips.” In: *IJCTE* 6.1 (2014) (cit. on p. 106).

[Emd+76] Peter van Emde Boas, Rob Kaas, and Erik Zijlstra. “Design and implementation of an efficient priority queue”. In: *Theory Comput. Syst.* 10.1 (1976), pp. 99–127 (cit. on pp. 18, 27, 33, 101).

[Emd77] Peter van Emde Boas. “Preserving order in a forest in less than logarithmic time and linear space”. In: *Inform. Process. Lett.* 6.3 (1977), pp. 80–82 (cit. on p. 18).

[Epp+08] David Eppstein, Michael T. Goodrich, and Jonathan Z. Sun. “Skip quadtrees: Dynamic data structures for multidimensional point sets”. In: *IJCGA* 18.01n02 (2008), pp. 131–160 (cit. on pp. 14, 84).

[Esk04] Eleazar Eskin. “From profiles to patterns and back again: a branch and bound algorithm for finding near optimal motif profiles”. In: *Proc. 8th RECOMB*. 2004, pp. 115–124 (cit. on p. 13).

[Far+14] Arash Farzan, Travis Gagie, and Gonzalo Navarro. “Entropy-bounded representation of point grids”. In: *CGTA* 47.1 (2014), pp. 1–14 (cit. on p. 84).

[Far+98] Martin Farach and Mikkel Thorup. “String Matching in Lempel–Ziv Compressed Strings”. In: *Algorithmica* 20.4 (1998), pp. 388–404 (cit. on pp. 8, 24).

[Fed+09] Maria Federico and Nadia Pisanti. “Suffix tree characterization of maximal motifs in biological sequences”. In: *Theor. Comput. Sci.* 410.43 (2009), pp. 4391–4401 (cit. on p. 70).

[Fed+14] Maria Federico, Pierre Peterlongo, Nadia Pisanti, and Marie-France Sagot. “Rime: Repeat Identification”. In: *Discrete Appl. Math.* 163.3 (2014), pp. 275–286 (cit. on p. 13).

[Fen94] Peter M. Fenwick. “A new data structure for cumulative frequency tables”. In: *Software Pract. Exper.* 24.3 (1994), pp. 327–336 (cit. on pp. 18, 39).

[Fer+04] Paolo Ferragina, Giovanni Manzini, Veli Mäkinen, and Gonzalo Navarro. *Succinct representation of sequences*. Tech. rep. TR/DCC-2004-5, University of Chile, 2004 (cit. on p. 42).

[Fer+05] Paolo Ferragina and Giovanni Manzini. “Indexing compressed text”. In: *J. ACM* 52.4 (2005), pp. 552–581 (cit. on p. 42).

[Fer+07] Paolo Ferragina and Rossano Venturini. “A simple storage scheme for strings achieving entropy bounds”. In: *Theor. Comput. Sci.* 372.1 (2007), pp. 115–121 (cit. on p. 42).

[Fer+13] Paolo Ferragina, Igor Nitto, and Rossano Venturini. “On the Bit-Complexity of Lempel–Ziv Compression”. In: *SIAM J. Comput.* 42.4 (2013), pp. 1521–1541 (cit. on pp. 10, 42).

[Fla+85] Philippe Flajolet and G. Nigel Martin. “Probabilistic counting algorithms for data base applications”. In: *J. Comput. Syst. Sci.* 31.2 (1985), pp. 182–209 (cit. on pp. 11, 54).

[Fre+84] Michael L. Fredman, János Komlós, and Endre Szemerédi. “Storing a sparse table with 0(1) worst case access time”. In: *J. ACM* 31.3 (1984), pp. 538–544 (cit. on pp. 61, 62).

[Fre+89] Michael L. Fredman and Michael Saks. “The cell probe complexity of dynamic data structures”. In: *Proc. 21st STOC*. 1989, pp. 345–354 (cit. on pp. 10, 18, 19, 39).

[Fre+93] Michael L. Fredman and Dan E. Willard. “Surpassing the information theoretic bound with fusion trees”. In: *J. Comput. System Sci.* 47.3 (1993), pp. 424–436 (cit. on pp. 32, 46).

[Gae+98] Volker Gaede and Oliver Günther. “Multidimensional access methods”. In: *ACM CSUR* 30.2 (1998), pp. 170–231 (cit. on pp. 14, 84).

[Gag+12a] Travis Gagie, Pawel Gawrychowski, Juha Kärkkäinen, Yakov Nekrich, and Simon J. Puglisi. “A faster grammar-based self-index”. In: *arXiv:1209.5441* (2012) (cit. on pp. 9, 24).

[Gag+12b] Travis Gagie, Juha Kärkkäinen, Gonzalo Navarro, and Simon J. Puglisi. “Colored range queries and document retrieval”. In: *Theor. Comput. Sci.* (2012) (cit. on p. 98).

[Gal+98] Nicola Galli, Bernhard Seybold, and Klaus Simon. “Compression of Sparse Matrices: Achieving Almost Minimal Table Size”. In: *Proc. ALEX*. 1998, pp. 27–33 (cit. on pp. 14, 84).

[Gar+14] Sandra Alvarez Garcia, Nieves R. Brisaboa, Guillermo de Bernardo, and Gonzalo Navarro. “Interleaved K2-Tree: Indexing and Navigating Ternary Relations”. In: *Proc. DCC*. 2014, pp. 342–351 (cit. on pp. 14, 84).

[Gas+05] Leszek Gasieniec, Roman Kolpakov, Igor Potapov, and Paul Sant. “Real-time traversal in grammar-based compressed files”. In: *Proc. 15th DCC*. 2005, p. 458 (cit. on p. 24).

[Gas+96] Leszek Gasieniec, Marek Karpinski, Wojciech Plandowski, and Wojciech Rytter. “Randomized efficient algorithms for compressed strings: The fingerprint approach”. In: *Proc. 7th CPM*. 1996, pp. 39–49 (cit. on pp. 8, 24).

[Gaw+14] Pawel Gawrychowski, Moshe Lewenstein, and Patrick K. Nicholson. "Weighted Ancestors in Suffix Trees". In: *Proc. 22nd ESA*. 2014, pp. 455–466 (cit. on pp. 40, 49).

[Gaw13] Pawel Gawrychowski. "Optimal pattern matching in LZW compressed strings". In: *ACM Trans. Alg.* 9.3 (2013), p. 25 (cit. on p. 63).

[Gol+85] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. "The knowledge complexity of interactive proof-systems". In: *Proc. 17th STOC*. 1985, pp. 291–304 (cit. on p. 57).

[Gol+86] Shafi Goldwasser and Michael Sipser. "Private coins versus public coins in interactive proof systems". In: *Proc. 18th STOC*. 1986, pp. 59–68 (cit. on p. 57).

[Gol+91] Oded Goldreich, Silvio Micali, and Avi Wigderson. "Proofs that yield nothing but their validity or all languages in NP have zero-knowledge proof systems". In: *J. ACM* 38.3 (1991), pp. 690–728 (cit. on p. 57).

[Gon+07] Rodrigo González and Gonzalo Navarro. "Compressed Text Indexes with Fast Locate". In: *Proc. 18th CPM*. 2007, pp. 216–227 (cit. on p. 42).

[Göö+15] Mika Göös, Toniann Pitassi, and Thomas Watson. "Zero-Information Protocols and Unambiguity in Arthur-Merlin Communication". In: *Proc. 6th ITCS*. 2015, pp. 113–122 (cit. on p. 57).

[Gos+15] Mayank Goswami, Allan Grønlund, Kasper Green Larsen, and Rasmus Pagh. "Approximate Range Emptiness in Constant Time and Optimal Space". In: *Proc. 26th SODA*. 2015. Chap. 52, pp. 769–775 (cit. on p. 49).

[Gro+03] Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. "High-order entropy-compressed text indexes". In: *Proc. 14th SODA*. 2003, pp. 841–850 (cit. on p. 42).

[Gro+11] Roberto Grossi, Andrea Pietracaprina, Nadia Pisanti, Geppino Pucci, Eli Upfal, and Fabio Vandin. "MADMX: A strategy for maximal dense motif extraction". In: *J. Comp. Biol.* 18.4 (2011), pp. 535–545 (cit. on pp. 13, 66–68).

[Gro+13] Roberto Grossi, Rajeev Raman, Satti Srinivasa Rao, and Rossano Venturini. "Dynamic Compressed Strings with Random Access". In: *Proc. 40th ICALP*. 2013, pp. 504–515 (cit. on pp. 10, 42).

[Gro+14a] Roberto Grossi, Giulia Menconi, Nadia Pisanti, Roberto Trani, and Søren Vind. "Output-Sensitive Pattern Extraction in Sequences". In: *Proc. 34th FSTTCS*. Vol. 29. 2014, pp. 303–314 (cit. on p. 3).

[Gro+14b] Roberto Grossi and Søren Vind. "Colored Range Searching in Linear Space". In: *Proc. 14th SWAT*. 2014, pp. 229–240 (cit. on p. 4).

[Gup+95] Prosenjit Gupta, Ravi Janardan, and Michiel Smid. “Further Results on Generalized Intersection Searching Problems: Counting, Reporting, and Dynamization”. In: *J. Algorithms* 19.2 (1995), pp. 282–317 (cit. on pp. 95, 96, 98, 99).

[Gup+97] Prosenjit Gupta, Ravi Janardan, and Michiel Smid. “A technique for adding range restrictions to generalized searching problems”. In: *Inform. Process. Lett.* 64.5 (1997), pp. 263–269 (cit. on pp. 96, 98, 99).

[Gut84] Antonin Guttman. “R-trees: A dynamic index structure for spatial searching”. In: *Proc. ACM SIGMOD*. 1984, pp. 47–57 (cit. on pp. 14, 84).

[Hae+10] Simon Haegler, Peter Wonka, Stefan Mueller Arisona, Luc Van Gool, and Pascal Mueller. “Grammar-based Encoding of Facades”. In: *Comput. Graph. Forum* 29.4 (2010), pp. 1479–1487 (cit. on pp. 14, 84).

[Hag98] Torben Hagerup. “Sorting and Searching on the Word RAM”. In: *Proc. 15th STACS*. 1998, pp. 366–398 (cit. on pp. 7, 32).

[Har+84] Dov Harel and Robert E. Tarjan. “Fast algorithms for finding nearest common ancestors”. In: *SIAM J. Comput.* 13.2 (1984), pp. 338–355 (cit. on pp. 25, 26, 49, 70).

[Hon+11] Wing-Kai Hon, Kunihiko Sadakane, and Wing-Kin Sung. “Succinct data structures for Searchable Partial Sums with optimal worst-case performance”. In: *Theor. Comput. Sci.* 412.39 (2011), pp. 5176–5186 (cit. on pp. 18, 19, 39, 40).

[Hoo+11] Christopher Hoobin, Simon J. Puglisi, and Justin Zobel. “Relative Lempel-Ziv factorization for efficient storage and retrieval of web collections”. In: *Proc. VLDB Endowment* 5.3 (2011), pp. 265–273 (cit. on pp. 10, 38, 42).

[Hu+04] Weiming Hu, Tieniu Tan, Liang Wang, and Steve Maybank. “A survey on visual surveillance of object motion and behaviors”. In: *IEEE Trans. SMC* 34.3 (2004), pp. 334–352 (cit. on p. 106).

[Hua11] Shih-Chia Huang. “An advanced motion detection algorithm with video quality analysis for video surveillance systems”. In: *IEEE Trans. CSVT* 21.1 (2011), pp. 1–14 (cit. on p. 106).

[Hus+03] Thore Husfeldt and Theis Rauhe. “New lower bound techniques for dynamic partial sums and related problems”. In: *SIAM J. Comput.* 32.3 (2003), pp. 736–753 (cit. on pp. 18, 39).

[Hus+96] Thore Husfeldt, Theis Rauhe, and Søren Skyum. “Lower Bounds for Dynamic Transitive Closure, Planar Point Location, and Parentheses Matching”. In: *Proc. 5th SWAT*. 1996, pp. 198–211 (cit. on pp. 18, 39).

[Ili+05] Costas S. Iliopoulos, James A. M. McHugh, Pierre Peterlongo, Nadia Pisanti, Wojciech Rytter, and Marie-France Sagot. “A first approach to finding common motifs with gaps”. In: *Int. J. Found. Comput. Sci.* 16.6 (2005), pp. 1145–1154 (cit. on p. 13).

[JáJ+05] Joseph JáJá, Christian W. Mortensen, and Qingmin Shi. “Space-efficient and fast algorithms for multidimensional dominance reporting and counting”. In: *Proc. 15th ISAAC*. 2005, pp. 558–568 (cit. on pp. 95, 98).

[Jan+12] Jesper Jansson, Kunihiko Sadakane, and Wing-Kin Sung. “CRAM: Compressed Random Access Memory”. In: *Proc. 39th ICALP*. 2012, pp. 510–521 (cit. on p. 42).

[Jan+93] Ravi Janardan and Mario Lopez. “Generalized intersection searching problems”. In: *IJCGA* 3.01 (1993), pp. 39–69 (cit. on pp. 94, 98).

[Joh81] Donald B. Johnson. “A priority queue in which initialization and queue operations take $O(\lg \lg D)$ time”. In: *Math. Syst. Theory* 15.1 (1981), pp. 295–309 (cit. on p. 18).

[Kal02] Adam Kalai. “Efficient pattern-matching with don’t cares”. In: *Proc. 13th SODA*. 2002, pp. 655–656 (cit. on pp. 8, 24, 58).

[Kan+99] Kothuri Venkata Ravi Kanth and Ambuj Singh. “Optimal Dynamic Range Searching in Non-replicating Index Structures”. In: *Proc. 7th ICDT*. 1999, pp. 257–276 (cit. on pp. 14, 84, 85).

[Kao+08] Wei Chieh Kao, Shih Hsuan Chiu, and Che Y. Wen. “An effective surveillance video retrieval method based upon motion detection”. In: 2008, pp. 261–262 (cit. on p. 106).

[Kap+07] Haim Kaplan, Natan Rubin, Micha Sharir, and Elad Verbin. “Counting colors in boxes”. In: *Proc. 18th SODA*. 2007, pp. 785–794 (cit. on pp. 15, 16, 95, 96, 98, 104).

[Kap+13] Alexis Kaporis, Christos Makris, Spyros Sioutas, Athanasios Tsakalidis, Kostas Tsichlas, and Christos Zaroliagis. “Improved Bounds for Finger Search on a RAM”. In: *Algorithmica* (2013), pp. 1–38 (cit. on p. 31).

[Kar+87] Richard M. Karp and Michael O. Rabin. “Efficient randomized pattern-matching algorithms”. In: *IBM J. Res. Dev.* 31.2 (1987), pp. 249–260 (cit. on pp. 8, 24, 27, 57).

[Kla+13] Hartmut Klauck and Ved Prakash. “Streaming computations with a loquacious prover”. In: *Proc. 4th ITCS*. 2013, pp. 305–320 (cit. on p. 57).

[Kla+14] Hartmut Klauck and Ved Prakash. “An improved interactive streaming algorithm for the distinct elements problem”. In: *Proc. 41st ICALP*. 2014, pp. 919–930 (cit. on p. 57).

[Kre+91] Marc van Kreveld and Mark Overmars. “Divided k-d trees”. In: *Algorithmica* 6.1-6 (1991), pp. 840–858 (cit. on pp. 14, 84).

[Kre92] Marc van Kreveld. *New results on data structures in computational geometry*. PhD thesis, University of Utrecht, 1992 (cit. on pp. 96, 98, 99).

[Kur+10] Shanika Kuruppu, Simon J. Puglisi, and Justin Zobel. “Relative Lempel-Ziv compression of genomes for large-scale storage and retrieval”. In: *Proc. 17th SPIRE*. 2010, pp. 201–206 (cit. on pp. 38, 42).

[Kur+11a] Shanika Kuruppu, Simon J. Puglisi, and Justin Zobel. “Optimized relative Lempel-Ziv compression of genomes”. In: *Proc. 34th ACSC*. 2011, pp. 91–98 (cit. on pp. 10, 38, 42).

[Kur+11b] Shanika Kuruppu, Simon J. Puglisi, and Justin Zobel. “Reference sequence construction for relative compression of genomes”. In: *Proc. 18th SPIRE*. 2011, pp. 420–425 (cit. on pp. 10, 42).

[Lar+12] Kasper Green Larsen and Rasmus Pagh. “I/O-efficient data structures for colored range and prefix reporting”. In: *Proc. 23rd SODA*. 2012, pp. 583–592 (cit. on pp. 98, 99).

[Lar+13] Kasper Green Larsen and Freek van Walderveen. “Near-Optimal Range Reporting Structures for Categorical Data.” In: *Proc. 24th SODA*. 2013, pp. 265–276 (cit. on p. 98).

[Lee+80] D. T. Lee and C. K. Wong. “Quintary trees: a file structure for multidimensional database systems”. In: *ACM TODS* 5.3 (1980), pp. 339–353 (cit. on pp. 14, 84).

[Lue78] George S. Lueker. “A data structure for orthogonal range queries”. In: *Proc. 19th FOCS*. 1978, pp. 28–34 (cit. on pp. 14, 84).

[Mab+10] Nizar R. Mabroukeh and Christie I Ezeife. “A taxonomy of sequential pattern mining algorithms”. In: *ACM CSUR* 43.1 (2010), p. 3 (cit. on p. 68).

[Mäk+07] Veli Mäkinen and Gonzalo Navarro. “Rank and select revisited and extended”. In: *Theor. Comput. Sci.* 387.3 (2007), pp. 332–347 (cit. on p. 84).

[McC76] Edward M. McCreight. “A Space-Economical Suffix Tree Construction Algorithm”. In: *J. ACM* 23.2 (April 1976), pp. 262–272 (cit. on pp. 67, 69).

[Meh+90] Kurt Mehlhorn and Stefan Näher. “Bounded ordered dictionaries in $O(\lg\lg N)$ time and $O(n)$ space”. In: *Inform. Process. Lett.* 35.4 (1990), pp. 183–189 (cit. on pp. 18, 27, 33, 101).

[Mor03] Christian W. Mortensen. *Generalized static orthogonal range searching in less space*. Tech. rep. TR-2003-22, ITU Copenhagen, 2003 (cit. on p. 98).

[Mun+80] Ian Munro and Mike S. Paterson. “Selection and sorting with limited storage”. In: *Theor. Comput. Sci.* 12.3 (1980), pp. 315–323 (cit. on pp. 11, 54).

[Nav+13] Gonzalo Navarro and Yakov Nekrich. “Optimal dynamic sequence representations”. In: *Proc. 24th SODA*. 2013, pp. 865–876 (cit. on p. 42).

[Nav+14] Gonzalo Navarro and Kunihiko Sadakane. “Fully Functional Static and Dynamic Succinct Trees”. In: *ACM Trans. Alg.* 10.3 (2014), p. 16 (cit. on pp. 19, 40).

[Nek+13] Yakov Nekrich and Jeffrey Scott Vitter. “Optimal color range reporting in one dimension”. In: *Proc. 21st ESA*. 2013, pp. 743–754 (cit. on pp. 96, 98).

[Nek09] Yakov Nekrich. “Orthogonal Range Searching in Linear and Almost-linear Space”. In: *Comput. Geom. Theory Appl.* 42.4 (2009), pp. 342–351 (cit. on pp. 94, 97, 101).

[Nek12] Yakov Nekrich. “Space-efficient range reporting for categorical data”. In: *Proc. 31st PODS*. 2012, pp. 113–120 (cit. on pp. 98, 99).

[Nek14] Yakov Nekrich. “Efficient range searching for categorical and plain data”. In: *ACM TODS* 39.1 (2014), p. 9 (cit. on pp. 98, 99).

[Och+14] Idoia Ochoa, Mikel Hernaez, and Tsachy Weissman. “iDoComp: a compression scheme for assembled genomes”. In: *Bioinformatics* (2014) (cit. on pp. 10, 42).

[Ore82] Jack A. Orenstein. “Multidimensional tries used for associative searching”. In: *Inform. Process. Lett.* 14.4 (1982), pp. 150–157 (cit. on pp. 14, 84).

[Ove87] Mark Overmars. *Design of Dynamic Data Structures*. 1987 (cit. on p. 102).

[Paj+00] Renato Pajarola and Peter Widmayer. “An image compression method for spatial search”. In: *IEEE Trans. Image Processing* 9.3 (2000), pp. 357–365 (cit. on pp. 14, 84).

[Par+00] Laxmi Parida, Isidore Rigoutsos, Aris Floratos, Daniel E. Platt, and Yuan Gao. “Pattern discovery on character sets and real-valued data: linear bound on irredundant motifs and an efficient polynomial time algorithm”. In: *Proc. 11th SODA*. 2000, pp. 297–308 (cit. on p. 66).

[Par+01] Laxmi Parida, Isidore Rigoutsos, and Dan Platt. “An Output-Sensitive Flexible Pattern Discovery Algorithm”. In: *Proc. 12th CPM*. 2001, pp. 131–142 (cit. on p. 67).

[Păt+04] Mihai Pătraşcu and Erik D. Demaine. “Tight bounds for the partial-sums problem”. In: *Proc. 15th SODA*. 2004, pp. 20–29 (cit. on pp. 18, 19, 39, 40, 45, 46).

[Păt+14] Mihai Pătraşcu and Mikkel Thorup. “Dynamic Integer Sets with Optimal Rank, Select, and Predecessor Search”. In: *Proc. 55th FOCS*. 2014, pp. 166–175 (cit. on pp. 18, 19, 40, 44, 45).

[Păt07] Mihai Pătraşcu. “Lower bounds for 2-dimensional range counting”. In: *Proc. 39th STOC*. 2007, pp. 40–46 (cit. on p. 98).

[Pic+06] Luká Pichl, Takuya Yamano, and Taisei Kaizoji. “On the symbolic analysis of market indicators with the dynamic programming approach”. In: *Proc. 3rd ISNN*. 2006, pp. 432–441 (cit. on p. 68).

[Por+09] Benny Porat and Ely Porat. “Exact and approximate pattern matching in the streaming model”. In: *Proc. 50th FOCS*. 2009, pp. 315–323 (cit. on pp. 8, 24, 58).

[Pro+03] Octavian Procopiuc, Pankaj K. Agarwal, Lars Arge, and Jeffrey Scott Vitter. “Bkd-tree: A dynamic scalable kd-tree”. In: *Proc. 8th SSTD*. 2003, pp. 46–65 (cit. on pp. 14, 84).

[Rah+11] Saladi Rahul, Prosenjit Gupta, Ravi Janardan, and K. S. Rajan. “Efficient top-k queries for orthogonal ranges”. In: *Proc. 5th WALCOM*. 2011, pp. 110–121 (cit. on p. 16).

[Ram+01] Rajeev Raman, Venkatesh Raman, and Satti Srinivasa Rao. “Succinct dynamic data structures”. In: *Proc. 7th WADS*. 2001, pp. 426–437 (cit. on pp. 18, 39).

[Rig+04] Isidore Rigoutsos and Tien Huynh. “Chung-Kwei: a Pattern-discovery-based System for the Automatic Identification of Unsolicited E-mail Messages”. In: *CEAS*. 2004 (cit. on p. 68).

[Rob81] John T. Robinson. “The KDB-tree: a search structure for large multidimensional dynamic indexes”. In: *Proc. ACM SIGMOD*. 1981, pp. 10–18 (cit. on pp. 14, 84).

[Ryt03] Wojciech Rytter. “Application of Lempel–Ziv factorization to the approximation of grammar-based compression”. In: *Theor. Comput. Sci.* 302.1 (2003), pp. 211–222 (cit. on p. 26).

[Sac+94] Todd S. Sachs, Craig H. Meyer, Bob S. Hu, Jim Kohli, Dwight G. Nishimura, and Albert Macovski. “Real-time motion detection in spiral MRI using navigators”. In: *Magn. Reson. Med.* 32.5 (1994), pp. 639–645 (cit. on p. 106).

[Sad+06] Kunihiko Sadakane and Roberto Grossi. “Squeezing Succinct Data Structures into Entropy Bounds”. In: *Proc. 17th SODA*. 2006, pp. 1230–1239 (cit. on p. 42).

[Sag98] Marie-France Sagot. “Spelling approximate repeated or common motifs using a suffix tree”. In: *Proc. 3rd LATIN*. 1998, pp. 374–390 (cit. on pp. 13, 66–68).

[Sam90] Hanan Samet. *Applications of spatial data structures*. Addison-Wesley, 1990 (cit. on pp. 14, 84).

[Sch+08] Grant Schindler, Panchapagesan Krishnamurthy, Roberto Lublinerman, Yanxi Liu, and Frank Dellaert. “Detecting and matching repeated patterns for automatic geo-tagging in urban environments”. In: *CVPR*. 2008, pp. 1–7 (cit. on pp. 14, 84).

[She+06] Reza Sherkat and Davood Rafiei. “Efficiently evaluating order preserving similarity queries over historical market-basket data”. In: *Proc. 22nd ICDE*. 2006, pp. 19–19 (cit. on p. 68).

[Shi+05] Qingmin Shi and Joseph JáJá. “Optimal and near-optimal algorithms for generalized intersection reporting on pointer machines”. In: *Inform. Process. Lett.* 95.3 (2005), pp. 382–388 (cit. on pp. 96, 98).

[Sim93] Imre Simon. “String matching algorithms and automata”. In: *Proc. 1st SPIRE*. 1993, pp. 151–157 (cit. on p. 62).

[Sto+78] James A. Storer and Thomas G. Szymanski. “The Macro Model for Data Compression”. In: *Proc. 10th STOC*. 1978, pp. 30–39 (cit. on pp. 10, 38, 42).

[Sto+82] James A. Storer and Thomas G. Szymanski. “Data compression via textual substitution”. In: *J. ACM* 29.4 (1982), pp. 928–951 (cit. on pp. 10, 38, 42).

[Tet+01] Igor V. Tetko and Alessandro E. P. Villa. “A pattern grouping algorithm for analysis of spatiotemporal patterns in neuronal spike trains.” In: *J. Neurosci. Meth.* 105.1 (2001), pp. 1–14 (cit. on pp. 14, 84).

[Tha14] Justin Thaler. “Semi-Streaming Algorithms for Annotated Graph Streams”. In: *arXiv:1407.3462* (2014) (cit. on p. 57).

[Tia+05] Ying-Li Tian and Arun Hampapur. “Robust salient motion detection with complex background for real-time video surveillance”. In: *WACV*. 2005 (cit. on p. 106).

[Tur36] Alan Mathison Turing. “On computable numbers, with an application to the Entscheidungsproblem”. In: *J. of Math* 58 (1936), pp. 345–363 (cit. on p. 4).

[Ukk09] Esko Ukkonen. “Maximal and minimal representations of gapped and non-gapped motifs of a string”. In: *Theor. Comput. Sci.* 410.43 (2009), pp. 4341–4349 (cit. on pp. 13, 66, 68, 70).

[Ver+13] Elad Verbin and Wei Yu. “Data structure lower bounds on random access to grammar-compressed strings”. In: *Proc. 24th CPM*. 2013, pp. 247–258 (cit. on p. 9).

[Vin+14] Søren Vind, Philip Bille, and Inge Li Gørtz. “Indexing Motion Detection Data for Surveillance Video”. In: *Proc. IEEE ISM*. 2014, pp. 24–27 (cit. on p. 4).

[Vin14] Søren Vind. *Source code for motion detection experiments*. `https://github.com/sorenvind/phd-motiondetectionindex`. 2014 (cit. on pp. 108, 109).

[Wan+12] Sebastian Wandelt and Ulf Leser. “Adaptive efficient compression of genomes.” In: *Algorithms Mol. Biol.* 7.30 (2012), pp. 1–9 (cit. on pp. 10, 42).

[Wan+13] Sebastian Wandelt and Ulf Leser. “FRESCO: Referential compression of highly similar sequences”. In: *IEEE/ACM Trans. Comput. Biol. Bioinf.* 10.5 (2013), pp. 1275–1288 (cit. on pp. 10, 42).

[Wei73] Peter Weiner. “Linear pattern matching algorithms”. In: *Proc. 14th SWAT*. 1973, pp. 1–11 (cit. on p. 61).

[Wel84] Terry A. Welch. “A Technique for High-Performance Data Compression”. In: *IEEE Computer* 17.6 (1984), pp. 8–19 (cit. on p. 42).

[Wil00] Dan E. Willard. “Examining computational geometry, van Emde Boas trees, and hashing from the perspective of the fusion tree”. In: *SIAM J. Comput.* 29.3 (2000), pp. 1030–1049 (cit. on p. 49).

[Wil12] Virginia Vassilevska Williams. “Multiplying matrices faster than Coppersmith-Winograd”. In: *Proc. 44th STOC*. 2012, pp. 887–898 (cit. on p. 98).

[Wil83] Dan E. Willard. “Log-logarithmic worst-case range queries are possible in space Θ(N)”. In: *Inform. Process. Lett.* 17.2 (1983), pp. 81–84 (cit. on pp. 18, 27, 33, 101).

[Zhu+02] Qing Zhu, Xuefeng Yao, Duo Huang, and Yetin Zhang. “An Efficient Data Management Approach for Large Cyber-City GIS”. In: *ISPRS Archives* (2002), pp. 319–323 (cit. on pp. 14, 84).

[Ziv+77] Jacob Ziv and Abraham Lempel. “A Universal Algorithm for Sequential Data Compression”. In: *IEEE Trans. Inform. Theory* 23.3 (1977), pp. 337–343 (cit. on pp. 9, 26, 42).

[Ziv+78] Jacob Ziv and Abraham Lempel. “Compression of Individual Sequences via Variable-Rate Coding”. In: *IEEE Trans. Inform. Theory* 24.5 (1978), pp. 530–536 (cit. on pp. 9, 24, 42, 61).

{2.7182818284}ϑφέρτυθιοπσδφγηξκλ