

Ph.D. Thesis
Doctor of Philosophy

 **DTU Compute**
Department of Applied Mathematics and Computer Science

An adventure in data structures

Søren Vind

Kongens Lyngby 2015



DTU Compute

Department of Applied Mathematics and Computer Science

Technical University of Denmark

Matematiktorvet

Building 303B

2800 Kongens Lyngby, Denmark

Phone +45 4525 3031

compute@compute.dtu.dk

www.compute.dtu.dk

Abstract

Write English Abstract

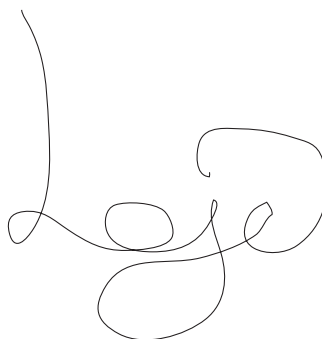
Resume

Write Danish Resume

Preface

This xxx thesis was prepared at the department of Applied Mathematics and Computer Science at the Technical University of Denmark in fulfillment of the requirements for acquiring a yyy degree in zzz.

Kongens Lyngby, January 11, 2015

A handwritten signature in black ink, appearing to read 'Søren Vind'. The signature is fluid and cursive, with a large loop at the end.

Søren Vind

Acknowledgements

People should go here

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Contents

Abstract	i
Resume	iii
Preface	v
Acknowledgements	vii
Contents	ix
1 Introduction	1
1.1 Reading Guide	1
2 Fingerprints in Compressed Strings	3
2.1 Introduction	4
2.2 Preliminaries	6
2.3 Basic fingerprint queries in SLPs	8
2.4 Faster fingerprints in SLPs	8
2.5 Faster fingerprints in Linear SLPs	10
2.6 Finger fingerprints in Linear SLPs	11
2.7 Longest Common Extensions in Compressed Strings	13
3 Output-Sensitive Pattern Extraction in Sequences	19
3.1 Introduction	19
3.2 Preliminaries	22
3.3 Motif Tries and Pattern Extraction	24
3.4 Building Motif Tries	26
3.5 Implementing $\text{GENERATE}(u)$	28
4 Compressed Data Structures for Range Searching	35
4.1 Introduction	35
4.2 Canonical Range Searching Data Structures	37
4.3 Compressed Canonical Range Searching	37
4.4 Similarity Clustering	41
4.5 Open Problems	42

5	Colored Range Searching in Linear Space	45
5.1	Introduction	45
5.2	Colored Range Searching in Almost-Linear Space	51
5.3	2D Colored Range Searching in Linear Space	53
5.4	Dynamic Data Structures	54
5.5	Open Problems	56
6	Indexing Motion Detection Data for Surveillance Video	57
6.1	Introduction	57
6.2	The Index	58
6.3	Experiments	59
6.4	Main Results	61
6.5	Other Results	63
6.6	Conclusion	65
7	Conclusion	67
A	An Appendix	69
	Bibliography	71

CHAPTER 1

Introduction

- The Study of Algorithms - Models (Pointer Machine, Word Ram, Streaming) - Theory vs Practice
- Basics of Data Structures - Trees and Tries - Predecessor
- Strings - Suffix Tree - Fingerprints
- Geometric - Orthogonal Range Searching - With Colors - And Motion
- Compression - DAG compression - SLPs and derivatives

1.1 Reading Guide

- List of papers
- Publication information
- About my project

Dynamic Relative Compression paper

CHAPTER 2

Fingerprints in Compressed Strings

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.

Authors Philip Bille¹, Patrick Hagge Cording¹, Inge Li Gørtz^{1,*}, Benjamin Sach², Hjalte Wedel Vildhøj¹, and Søren Vind^{1,†}.

Publication An extended abstract of this paper appeared at the 13th Algorithms and Data Structures Symposium (WADS 2013).

¹Technical University of Denmark

²University of Warwick

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

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

2.1 Introduction

Given a string S of size N and a Karp-Rabin fingerprint function ϕ , the answer to a $\text{FINGERPRINT}(i, j)$ 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 [KR87] 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., [AFM92; AI06; CH03; CM05; CM07; FT98; Gas+96; Kal02; PP09]).

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 $i = 1 \dots N$ 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 Gasieniec 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 $\text{FINGERPRINT}(i, j)$ 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 et al. [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 [ZL78] and its variations. Our data structures give the following theorem.

Theorem 2.1.1. *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 2.1.1(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 2.1.2. *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 [HT84a]. Other trade-offs are obtained by doing an exponential search over the fingerprints of strings starting in i and j [Bil+12]. 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 2.1.3.

Theorem 2.1.3. *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 [ZL77] consisting of n phrases can be transformed to an SLP with $O(n \lg \frac{N}{n})$ production rules [Cha+05; Ryt03].

Corollary 2.1.3.1. *We can solve the LCE problem in $O(n \lg \frac{N}{n})$ space and query time $O(\lg \ell \lg \lg N)$ 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, \dots, 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 2.1.3.2. *We can solve the LCE problem in $O(n)$ space and query time $O(\lg \ell \lg \lg \ell + \lg \lg N)$ for LZ78 compression.*

2.2 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 $\text{size}(v) = |S(v)|$ is *produced* by a depth-first left-to-right traversal of $v \in G$ and consist of the characters on the leaves in the order they are visited. We let $\text{root}(G)$ denote the root of G , and $\text{left}(v)$ and $\text{right}(v)$ 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 $\text{root}(G_L)$ to have more than two children. All other internal nodes $v \in G_L$ have a leaf as $\text{right}(v)$. Although similar, this is not the same definition as given for the Relaxed SLP by Claude and Navarro [CN11]. 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 [HT84a] 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 $\text{size}(u) \leq \text{size}(v)/2$ 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 $\text{leaf}(H(v))$, is well-defined for each node $v \in G$.

A *predecessor data structure* supports predecessor and successor queries on a set $R \subseteq U = \{0, \dots, N-1\}$ of n integers from a universe U of size N . The answer to a *predecessor query* $\text{PRED}(q)$ is the largest integer $r^- \in R$ such that $r^- \leq q$, while the answer to a *successor query* $\text{SUCC}(q)$ 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 $O(n)$ [EKZ76; MN90; Wil83].

Given a rooted tree T with n vertices, we let $\text{depth}(v)$ 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* $\text{LA}(v, i)$, asking for the ancestor u of $v \in T$ such that $\text{depth}(u) = \text{depth}(v) - i$. There is a level ancestor data structure answering queries in $O(1)$ time using $O(n)$ space [Die91] (see also [BV94; AH00; BF04]).

Fingerprinting

The Karp-Rabin fingerprint [KR87] 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 2.2.1. *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:*

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

In order to calculate the fingerprints of Theorem 2.2.1 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.

2.3 Basic fingerprint queries in SLPs

We now describe a simple data structure for answering $\text{FINGERPRINT}(1, i)$ 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 $\text{size}(v)$ 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 $p + \text{size}(\text{left}(v')) < i$, and follow a left edge otherwise. If following a right edge, update $f = f \oplus \phi(S(\text{left}(v')))$ such that the fingerprint of the full string generated by the left child of v' is added to f , and set $p = p + \text{size}(\text{left}(v'))$. 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 $O(n)$.

2.4 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 $\text{FINGERPRINT}(1, i)$ queries.

Lemma 2.4.1 ([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 $O(\lg(\text{size}(v)))$ 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

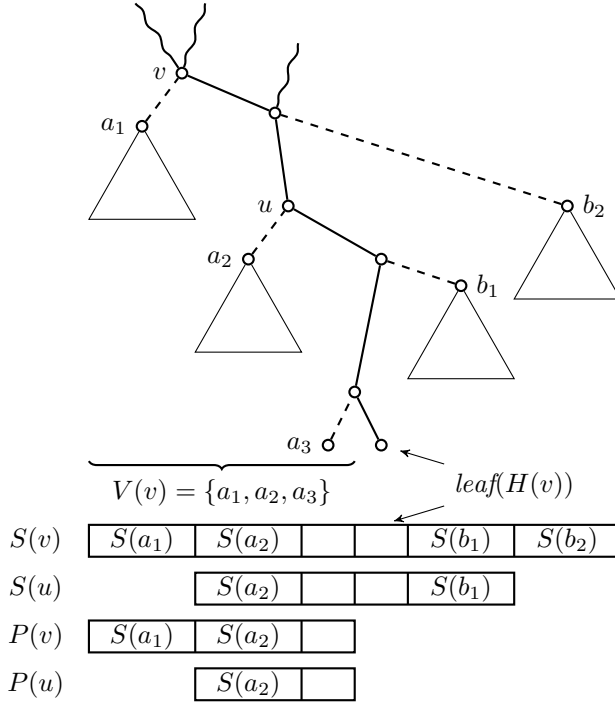


Figure 2.1: Figure showing how $S(v)$ and its prefix $P(v)$ is composed of substrings generated by the left children a_1, a_2, a_3 and right children b_1, b_2 of the heavy path $H(v)$. Also illustrates how this relates to $S(u)$ and $P(u)$ for a node $u \in H(v)$.

the strings produced by nodes in $V(v)$ yields the prefix $P(v) = S(v)[1, L(v)]$, where $L(v) = \sum_{u \in V(v)} \text{size}(u)$. Observe that $P(u)$ is a suffix of $P(v)$ if $u \in H(v)$. See Figure 2.1 for the relationship between u, v and the defined strings.

Let each node $v \in G$ store its unique outgoing heavy path $H(v)$, the length $L(v)$, $\text{size}(v)$, 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 2.4.1. 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 $f = f \oplus \phi(S(\text{left}(u)))$. If u is a leaf, set $f = f \oplus \phi(S(u))$ to include the fingerprint of the terminal character.

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 2.1.1(i).

2.5 Faster fingerprints in Linear SLPs

In this section we show how to quickly answer $\text{FINGERPRINT}(1, i)$ queries on a Linear SLP G_L . In the following we denote the sequence of k children of $\text{root}(G_L)$ from left to right by r_1, \dots, r_k . Also, let $R(j) = \sum_{m=1}^j \text{size}(r_m)$ for $j = 0, \dots, k$. 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 $u = \text{left}(v)$ and $c = S(\text{right}(v))$. If v is a leaf, there is an edge $(\text{root}(F), v^F)$ 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 $\text{root}(F)$ to v^F . Thus, the prefix of length a of $S(v)$ may be produced by reporting the edge labels on the path from $\text{root}(F)$ 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 $j = 0, \dots, k$. 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 $\text{FINGERPRINT}(1, i)$ is answered as follows. Let $R(m)$ be the predecessor of i among $R(0), R(1), \dots, R(k)$. Compose the answer to $\text{FINGERPRINT}(1, i)$ 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

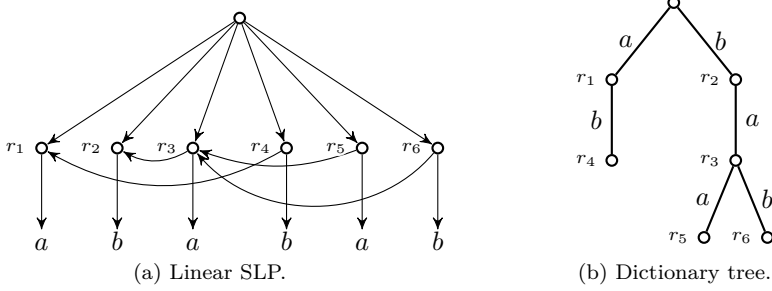


Figure 2.2: A Linear SLP compressing the string `abbaabbaabab` and the dictionary tree obtained from the Linear SLP..

using the doubly linked list. We use a level ancestor query $u^F = \text{LA}(r_{m+1}^F, i - R(m))$ to determine the ancestor of r_{m+1}^F 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 $\text{root}(G_L)$) 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 2.1.1(ii).

2.6 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 $R(0), R(1), \dots, R(k)$. 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.

Finger Predecessor

Let $R \subseteq U = \{0, \dots, N - 1\}$ 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. [BBV12] 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; AT07]. Here, we use a simple reduction for solving the finger predecessor problem using any predecessor data structure as a black box.

Lemma 2.6.1. *Let $R \subseteq U = \{0, \dots, N-1\}$ 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 $\text{SUCC}(q)$ and $\text{PRED}(q)$ 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 $\text{SUCC}(q)$ 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 $\text{SUCC}(q)$ and its left neighbour in T as $\text{PRED}(q)$.

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 [FW93; 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.

¹The 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 .

Given a finger $f \in R$ and a query point $q \in U$, we describe how to determine $\text{PRED}(q)$ and $\text{SUCC}(q)$ 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 $p = \text{PRED}(q)$ and $s = \text{SUCC}(q)$ 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 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 $O(s(N, \frac{n}{\lg N}) \lg N)$. \square

Using the van Emde Boas predecessor data structure [EKZ76; MN90; Wil83] with $t(N, n) = O(\lg \lg N)$ query time using $s(N, n) = O(n)$ space, we obtain the following corollary.

Corollary 2.6.1.1. *Let $R \subseteq U = \{0, \dots, N-1\}$ 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 $O(n)$.*

Finger Fingerprints

We can now prove Theorem 2.1.2. 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 2.6.1.1 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 $O(\lg \lg |i - j|)$.

2.7 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 $\text{LCE}(i, j)$. In this section we show how to use our fingerprint data structures as a tool for doing LCE queries and hereby obtain Theorem 2.1.3.

Computing Longest Common Extensions with Fingerprints

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

Lemma 2.7.1. *For any string S and any partition $S = s_1 s_2 \cdots s_t$ of S into k non-empty substrings called phrases, $\ell = \text{LCE}(i, j)$ 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 $p = 0, \dots, \lg(\ell) + 1$ 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). \square \square

Theorem 2.1.3 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 2.1.2 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.

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).

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 2.7.1). We thus only have to check that ϕ is collision-free among all substrings of length 2^p , $p = 0, \dots, \lg N$. 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.

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 2.7.1 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 comparisons and type 2 comparisons 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.

Annotated Data Streaming paper

Output-Sensitive Pattern Extraction in Sequences

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.

3.1 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 $\{\mathbf{A}, \mathbf{C}, \mathbf{G}, \mathbf{T}\}$ 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; AU07; 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, $\text{TA} \star \text{C} \star \text{ACA} \star \text{GTG}$ 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 $\text{occ}(m)$ 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 in level-wise, using an oracle $\text{GENERATE}(u)$ 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 $\text{occ}(m)$ of occurrences of each maximal motif m .

Theorem 3.1.1. *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 $O\left(n(k + \lg \Sigma) + (k + 1)^3 \times \sum_{m \in \mathcal{M}} \text{occ}(m)\right)$ 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 $O(n \lg \Sigma + \sum_{m \in \mathcal{M}} \text{occ}(m))$ 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 time¹. 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. [LP01] 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 [AU07]. 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 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.

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

²The set intersection problem (SIP) in appendix A of [LP01] 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.

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 [AG10] for further references). In computational biology, motif discovery in biological sequences identifies areas of interest [Sag98; Ukk09; Gro+11; AG10]. Computer security researches use patterns in log files to perform intrusion detection and find attack signatures based on their frequencies [DDW99], while commercial anti-spam filtering systems use pattern extraction to detect and block SPAM [RH04]. In the data mining community pattern extraction is used extensively [ME10] as a core method in web page content extraction [CHL03] and time series analysis [PYK06; SR06]. 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 [BDG95] 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 [ACP06].

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 3.3 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 3.4 and 3.5 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.

3.2 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 $\text{prefset}(S)$ be the set of all prefixes of S . The *longest common prefix* $\text{lcp}(x, y)$ is the longest string such that $x[1, |\text{lcp}(x, y)|] = y[1, |\text{lcp}(x, y)|]$ for any two strings $x, y \in \Sigma^*$.

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

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 3.1: Example 1: Maximal Motifs found in string..

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 $\text{dc}(m)$ 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 $\text{occ}(m)$. Note that appending \star to either end of a motif m does not change $\text{occ}(m)$, 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 3.1.

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 $\text{dc}(m) \leq k$, and occurs frequently $\text{occ}(m) \geq q$. A *maximal motif* $m \in M_{qk}$ cannot be extended or specialized into another motif $m' \in M_{qk}$ such that $\text{occ}(m') = \text{occ}(m)$. 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.*

3.3 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. Due to space constraints, all proofs have been omitted in the present version.

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; FP09] that when a motif m is maximal, its solid blocks correspond to nodes in the suffix tree for S , matching their substrings from the root³. 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 $\Pi \subseteq [1, \dots, 2n]$, 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 3.3.1. *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 $1 \leq i \leq j \leq |\Pi|$.*

³The proofs in [Ukk09; FP09] can be easily extended to our notion of maximality.

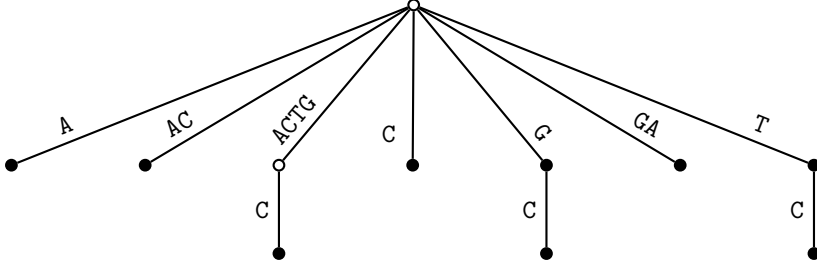


Figure 3.2: Motif trie for Example 1. The black nodes are maximal motifs (with their occurrence lists shown in Fig. 3.1(b)).

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 $x, y \in \Pi$, $\text{lcp}(x, y) \in \Pi$ and it may be computed in constant time after augmenting the suffix tree for S with a lowest common ancestor data structure [HT84b].

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.

Motif Tries

We now exploit the machinery on alphabets described in Section 3.3. For the input sequence S , consider the family M_{qk} defined in Section 3.2, 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 $\ell = \text{dc}(m) + 1 \leq k + 1$. 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 1 (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 trie⁴ 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 $k + 1$.

⁴As 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.

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 $p \in L_u$.

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).

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 3.3.1, 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.

3.4 Building Motif Tries

The goal of this section is to show how to efficiently build the motif trie T discussed in Section 3.3. 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 3.1.1.

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 $L_u = \{1, \dots, n\}$, 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 $\text{GENERATE}(u)$ procedure that generates the children u_1, \dots, u_d of u . We ensure that (i) $\text{GENERATE}(u)$ operates on the $\leq k+1$ length motifs from Π , and (ii) $\text{GENERATE}(u)$ 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 3.5 we will show how to implement $\text{GENERATE}(u)$ and prove:

Lemma 3.4.1. *Algorithm $\text{GENERATE}(u)$ 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 $\text{GENERATE}(u)$ 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 3.4.1, 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 3.3 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 3.3, we obtain:

Theorem 3.4.2. *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 $\text{occ}(m)$ in S for $m \in \mathcal{M}$ can be built in time and space $O \left(n(k + \lg \Sigma) + (k+1)^3 \times \sum_{m \in \mathcal{M}} \text{occ}(m) \right)$.*

3.5 Implementing Generate(u)

We now show how to implement $\text{GENERATE}(u)$ within the time bounds stated by Lemma 3.4.1. 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 .

Nodes of the Motif Trie as Maximal Intervals

The key point in the efficient implementation of the oracle $\text{GENERATE}(u)$ is to relate each node u and its future children u_1, \dots, u_d labeled by solid blocks b_1, \dots, b_d , respectively, to some suitable intervals that represent their occurrence lists $L_u, L_{u_1}, \dots, L_{u_d}$. Though the idea of using intervals for representing trie nodes is not new (e.g. in [AKO04]), 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, \dots, 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 $\text{ext}(p, u) = S[p + |u| + 1, n]$ 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 $\text{ext}(p)$, omitting the motif u if it is clear from the context. We also say that the *skipped characters* $\text{skip}(p)$ at position $p \in L_u$ are the $d = \text{dc}(u) + 2$ characters in S that specialize u into its occurrence p : formally, $\text{skip}(p) = \langle c_0, c_1, \dots, c_{d-1} \rangle$ where $c_0 = S[p - 1]$,

$c_{d-1} = S[p + |u|]$, and $c_i = S[p + j_i - 1]$, for $1 \leq i \leq d - 2$, where $u[j_i] = \star$ is the i th don't care in u .

We denote by E_u the list L_u sorted using as keys the integers for $\text{ext}(p)$ where $p \in L_u$. (We recall from Section 3.3 that the suffixes are represented in the alphabet Π , and thus $\text{ext}(p)$ can be seen as an integer in Π .) By Lemma 3.3.1 consecutive positions in E_u share common prefixes of their extensions. Lemma 3.5.1 below states that these prefixes are the candidates for being correct edge labels for expanding u in the trie.

Lemma 3.5.1. *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 $\text{ext}(p, u)$.*

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

Intervals. Lemma 3.5.1 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 $\text{LCP}(I) = \min_{p_1, p_2 \in I} \text{lcp}(\text{ext}(p_1), \text{ext}(p_2))$, which is a solid block in Π as discussed at the end of Section 3.3. By Lemma 3.3.1, $\text{LCP}(I) = \text{lcp}(\text{ext}(E_u[i]), \text{ext}(E_u[j]))$ 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 [AKO04].

Lemma 3.5.2. *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. \square

⁵In the full version we show that condition (2) is needed to avoid the enumeration of either motifs from $M_{qk} \setminus \mathcal{M}$ or duplicates from \mathcal{M} .

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 3.5.3. *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 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 3.5.1, 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 $\text{LCP}(I) = b_i$, $|P_I| \geq 2$ and $L_{u_i} \subseteq I$. There can be no $p' \in I - L_{u_i}$, as $p' \in L_u$ and $b_i \in \text{prefset}(\text{ext}(p', u))$ means that $p' \in L_{u_i}$. \square

Exploiting the Properties of Maximal Intervals

We now use the properties shown above to implement the oracle $\text{GENERATE}(u)$, resulting in Lemma 3.4.1. Observe that the task of $\text{GENERATE}(u)$ can be equivalently seen by Lemma 3.5.3 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 $O(\text{sort}(L_u) + (k + 1) \cdot |L_u|)$ 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 $\text{dc}(u) < k$, 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 $\text{skip}(E_u[j])$ 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 3.5.4. *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, $\ell_p = \max_{I' \in D(p)} |\text{LCP}(I')|$. For a maximal interval $I \subseteq \mathcal{I}_u$, if $|\text{LCP}(I)| = \ell_p$ for some $p \in I$ we call p a *handle* on I . Handles are relevant for the following reason.

Lemma 3.5.5. *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 3.5.2, 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 $\ell_{p'} \neq |\text{LCP}(I)|$ (since otherwise $p' \in I$ and $\ell_{p'} = |\text{LCP}(I)|$ and thus p' was a handle on I). Clearly for all $p' \in I$, $|\text{LCP}(I)|$ is a lower bound for $\ell_{p'}$. Thus, to get a contradiction it must be the case that $\ell_{p'} > |\text{LCP}(I)|$ 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 3.5.2, a single interval I' is not enough because I' is fully contained (or completely disjoint) in I if $|\text{LCP}(I')| \geq |\text{LCP}(I)|$. \square

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 $\text{LCP}(I')$ 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 3.5.4, \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 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 $(|\text{LCP}(I')|, -i)$ (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 \mathcal{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 3.5.2), 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 3.5.5, 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 3.5.2 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 3.5.6 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 3.5.6. (*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. \square

Lemma 3.5.7. (Completeness) *If $m \in \mathcal{M}$, T stores m and its suffixes.*

Proof. We summarize the proof that $\text{GENERATE}(u)$ is correct and the correct motif trie is produced. From Lemma 3.5.5, we create all intervals in $\text{GENERATE}(u)$ by expanding those with handles, and expanding all composite intervals from these. By Lemma 3.5.3 the intervals found correspond exactly to the children of u in the motif trie. Thus, as $\text{GENERATE}(u)$ 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) $\text{occ}(m') \geq \text{occ}(m)$ and (2) $\text{dc}(m') \leq \text{dc}(m)$. \square

CHAPTER 4

Compressed Data Structures for Range Searching

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.

4.1 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; BS80; Lue78; LW80; Gut; Cla83; KS99; KO91; GG98; BM72; Arg+08; Rob; Pro+03; Com79; EGS08] (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 [TV01; PW00; DSW09], GIS applications [Sch+08; Zhu+02; Hae+10; DSW09], and in compactly representing sparse matrices and web graphs [GSS98; BLN09; 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., [MN07; Bos+09; BCN10; FGN14]. 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 [BLN09; Gar+14; Ber+13]. They consider how to efficiently represent a binary (or tertiary) quad tree by encoding it as bitstrings. That is, their approach 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 4.2. 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 4.3. Finally, we present a *similarity clustering* algorithm in Section 4.4, guaranteeing that geometric repetitions are clustered such that the resulting canonical range searching data structure is compressible.

4.2 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 [KS99].

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 4.1 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*.

4.3 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.

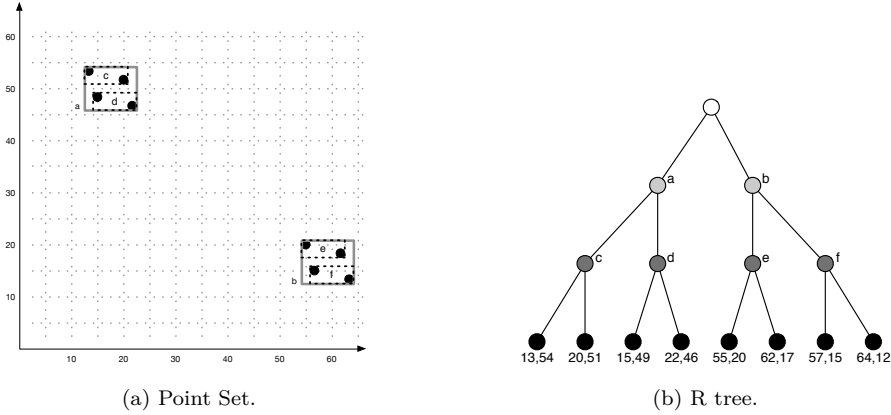


Figure 4.1: A two-dimensional point set with R tree ranges overlaid, and the resulting R tree. Blue ranges are children of the root in the tree, red ranges are at the second level. A vertex label ($a - h$) in the R tree identifies the range. We have omitted the precise coordinates for the ranges, but e.g. range a spans the range $[13, 22] \times [46, 54]$.

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 $r = [x_1, x'_1] \times \dots \times [x_k, x'_k]$ as two k -dimensional vectors $position(r) = (x_1, \dots, x_k)$ and $extent(r) = (x'_1 - x_1, \dots, x'_k - x_k)$ 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 4.3.1. *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

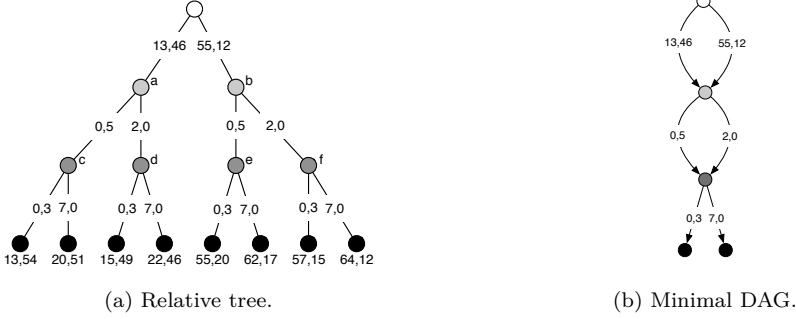


Figure 4.2: The relative tree obtained from the R tree from Figure 4.1 and the resulting minimal DAG G generating the tree. Only coordinates of the lower left corner of the ranges in the R tree are shown. In the relative tree, the absolute coordinates for the points are only shown for illustration, in order to see that the relative coordinates sum to the absolute coordinate along the root-to-leaf paths. .

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, $\text{extent}(r_v) = \text{extent}(r_{v_R})$ and $\text{label}(v) = \text{label}(v_R)$. The offset vector for the (w_R, v_R) edge is $\text{offset}(w_R, v_R) = \text{position}(r_v) - \text{position}(r_w)$. We assume the offset for the root is the 0-vector. Observe that summing up all the offset vectors on $\text{path}(v)$ is exactly $\text{position}(r_v)$, and so $\text{position}(r_{v_R}) = \text{position}(r_v)$.

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. \square

The Data Structure

The compressed canonical data structure is the minimal DAG G of the relative tree R for T . By Lemma 4.3.1 and [DST80] 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 4.2.

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 $v \in T$ 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 4.3.2. *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 4.3.2.1. *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.*

Extension to the I/O Model

We now show that Theorem 4.3.2 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 4.3.1 and [DST80], 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 .

4.4 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 4.3.2.

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

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

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 $\text{points}(v)$ the points in cluster node $v \in C(P)$. Two cluster nodes $v, w \in C(P)$ are equivalent if $\text{points}(v)$ is equivalent to $\text{points}(w)$ and if the subtrees rooted at the nodes are isomorphic such that each isomorphic pair of nodes are equivalent.

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 $i = 0, 1, \dots, \lg D + 1$, where $D = \text{diameter}(P)$. 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 $\maxdist(\text{points}(C_{i-1}), p) \leq d_i$. The clusters at level 0 contain individual points and the cluster at level $\lg D + 1$ contains all points.

Lemma 4.4.1. *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 $d = \text{diameter}(P_1) = \text{diameter}(P_2)$. 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 $\text{points}(c)$. 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 $\text{points}(C_1)$ and $\text{points}(C_2)$ 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. \square

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 4.4.2. *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.

4.5 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.

Colored Range Searching in Linear Space

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.

5.1 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 [JL93] 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 5.1–5.2, 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)

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

Dim.	Query Time	Space Usage	Dyn.	Ref.
$d = 1$	$O(\lg n / \lg \lg n)$	$O(n)$	\times	[JMS05]
$d = 2$	$O(\lg^2 n)$	$O(n^2 \lg^2 n)$		[GJS95]
	$O(\lg^2 n)$	$O(n^2 \lg^2 n)$		[Kap+07]
	$O(X \lg^7 n)$	$O((n/X)^2 \lg^6 n + n \lg^4 n)$		[Kap+07]
	$O\left(\left(\frac{\sigma}{\lg n} + \frac{n}{\lg^c n}\right)(\lg \lg^c n)^{2+\epsilon}\right)$	$O(n)$		New
$d > 2$	$O(\lg^{2(d-1)} n)$	$O(n^d \lg^{2(d-1)} n)$		[Kap+07]
	$O(X \lg^{d-1} n)$	$O((n/X)^{2d} + n \lg^{d-1} n)$		[Kap+07]
	$O\left(\left(\frac{\sigma}{\lg n} + \frac{n}{\lg^c n}\right)(\lg \lg^c n)^{d-1} \lg \lg \lg^c n\right)$	$O(n(\lg \lg^c n)^{d-1})$		New
$d \geq 2$	$O\left(\left(\frac{\sigma}{\lg n} + \frac{n}{\lg^c n}\right)(\lg \lg^c n)^d\right)$	$O(n(\lg \lg^c n)^{d-1})$	\times	New

Table 5.1: 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..

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.

Dim.	Query Time	Space Usage	Dyn.	Ref.
$d = 1$	$O(1 + k)$	$O(n)$	\times	[NV13]
$d = 2$	$O(\lg n + k)$	$O(n \lg n)$	\times	[SJ05]
	$O(\lg^2 n + k \lg n)$	$O(n \lg n)$		[GJS95; Boz]
	$O\left(\left(\frac{\sigma}{\lg n} + \frac{n}{\lg^c n}\right)(\lg \lg^c n)^{2+\epsilon} + k\right)$	$O(n)$		New
$d \geq 2$	$O\left(\left(\frac{\sigma}{\lg n} + \frac{n}{\lg^c n}\right)(\lg \lg^c n)^d + k\right)$	$O(n(\lg \lg^c n)^{d-1})$	\times	New
$d = 3$	$O(\lg^2 n + k)$	$O(n \lg^4 n)$		[GJS95]
$d > 3$	$O(\lg n + k)$	$O(n^{1+\epsilon})$		[Kre92; GJ]
$d \geq 3$	$O\left(\left(\frac{\sigma}{\lg n} + \frac{n}{\lg^c n}\right)(\lg \lg^c n)^{d-1} \lg \lg \lg^c n + k\right)$	$O(n(\lg \lg^c n)^{d-1})$		New

Table 5.2: 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..

Our results

We observe (see Section 5.1) 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 $o(n^{1+\epsilon})$ words of space and answering queries in $o(n)$ worst case time. For colored range counting, no solutions with $o(n \text{ poly} \lg n)$ 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 5.1 and Tables 5.1–5.2). Specifically, we obtain

- $o(n)$ query time and $O(n)$ space in two dimensions,
- $o(n)$ query time and $o(n \text{ poly} \lg n)$ 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 $O(\text{poly} \lg n)$ 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 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 5.1.1. *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 5.1.2. *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 $O(\lg^\epsilon n)$ per point. We thus obtain a query time of $O(\lg^{2+\epsilon} m)$ 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 5.1.3. *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 5.1.3.1. *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 $O(\sigma \lg^{d-2} n \lg n)$. 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 5.1.3.2. *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 $O(\sigma + k \lg n \lg \lg n)$.*

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 [LW13; GJS95] 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. [JMS05] which matches a lower bound by Patrascu [Pat07] is also optimal for one-dimensional colored range counting.

In two dimensions, Gupta et al. [GJS95] 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 $O(n \text{ poly} \lg n)$ 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 relatively well-studied [JL93; Nek12; NV13; Nek14; Gag+12b; SJ05; GJS95; GJS97; Kre92; Mor03; Boz+95; LP12], 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 [NV13], 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 [SJ05] 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 in [GJS95; Boz+95], answering queries with a logarithmic penalty per answer. If the points are located on an $N \times N$ grid, Agarwal et al. [AGM02] 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 $O(\lg^2 n + k)$ [GJS95] 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 $O(n^{1+\epsilon})$ space [Kre92; GJS97]. Other recent work on the problem include external memory model solutions when the points lie on a grid [Nek12; LP12; Nek14].

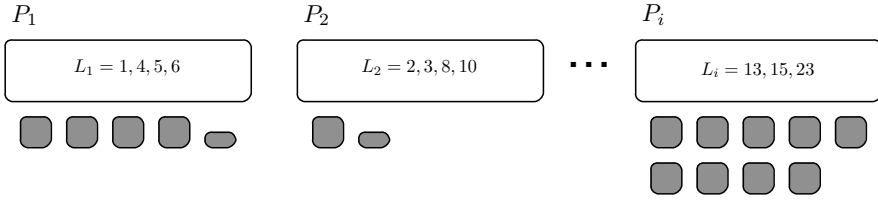


Figure 5.3: Grouping and bucketing of some point set with $f = 4$. The white boxes are the groups and the grey boxes below each group are buckets each storing $O(\lg^c n)$ points..

5.2 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 5.1.2.

Color grouping and bucketing

We partition the points of P into a number of groups P_i , where $i = 1, \dots, \frac{\sigma}{\lg n}$, 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 5.3 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 j th 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.

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 $1, 2, \dots, 2^{\lg j}$ 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 [EKZ76; MN90; Wil83] for the representatives, we thus obtain a query time of $O(\lg \lg m)$ and linear space.

5.3 2D Colored Range Searching in Linear Space

To obtain linear space in two dimensions and the proof of Theorem 5.1.1, we use the same grouping and bucketing approach as in Section 5.2. 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 by 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 5.2, 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 5.2, 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 5.1.1.

5.4 Dynamic Data Structures

We now prove Theorem 5.1.3 by discussing how to support operations $\text{INSERT}(p, c)$ and $\text{DELETE}(p)$, 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.

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 $O(\lg^d m)$.

Updating color grouping and point bucketing

When supporting $\text{INSERT}(p, c)$, 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 $\text{DELETE}(p)$, 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 5.4.

Fixing bucket answers during a query

As mentioned in Section 5.4, 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, \dots, 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, \dots, b_h , which are the answers for the old merged groups P'_1, P'_2, \dots, 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, \dots, 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 $h \leq g \leq f$.

5.5 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.

CHAPTER 6

Indexing Motion Detection Data for Surveillance Video

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.

6.1 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* $\text{MD}(T, A, v, p)$ 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; TH; Hu+04; CD00; Hua11]. Previous papers [Du+14; KCW] 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.

6.2 The Index

A $\text{MD}(T, A, v, p)$ 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, \dots, 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 $\text{MD}(T, A, v, p)$ 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.

6.3 Experiments

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).

Data sets

We tested our index on the following three typical video surveillance scenarios, encoded at 29.97fps using H264/MP4 (reference []). We use different video resolutions (1920×1080 , 1280×720 and 852×480 pixels). See Table 6.1.

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.

Table 6.1: Surveillance video recording samples used for testing. Videos were encoded at 29.97fps using H264/MP4..

Scenario	Length (s)	Motion amount	Size (MB)		
			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

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 occurring 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 across the entire frame.

Table 6.2: 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

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 6.2. The source code can be found at [\[1\]](#).

6.4 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.

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 6.3 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.

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 6.5). 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 6.4 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.

Table 6.3: Index query time versus video query time for 1 and 1024 regions queried. Size of index compared to the video size..

Scenario	Query Reg.	Time (s)		Speedup	Size
		Index	Video		
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 6.4: Speedup for index query time compared to video query time for students scenario, with varying input video resolutions and number of regions queried..

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

Resolution Comparison

Table 6.4 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 6.5. 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 regions 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).

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 6.6, 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

Table 6.5: Size of indexes for varying input video resolutions..

Scenario	Index Size (MB)			Index Size (%)		
	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 6.6: Index size for varying number of stored regions and resolutions in the students scenario..

Store Reg.	Index Size (MB)			Index Size (%)		
	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%

regions only causes the index to about double ($2\times$) in size. That is, it is relatively cheap to increase the resolution of regions.

6.5 Other Results

In this section, we review the index performance when varying the different parameters listed in Table 6.2. 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.

Compressor

Table 6.7 shows the size of the index when the histograms are compressed using a number of different compressors, and Table 6.8 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 6.5).

Table 6.7: Index size comparison for different compressors..

Scenario	Index Size (MB) / Compressor				
	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 6.8: Index compression time for different compressors..

Scenario	Compression Time (s) / Compressor				
	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

Table 6.9: Fraction of time spent analysing regions of total query time for students scenario (remainder is decompression). .

Resolution	Index Query Time		Video Query Time		Size
	1	1024	1	1024	
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%

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 6.9. 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×) after decompression.

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 write $r_1[0]r_2[0]r_1[1]r_2[1] \dots r_1[255]r_2[255]$ 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.

6.6 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.

CHAPTER 7

Conclusion

Morbi pharetra ligula integer mollis mi nec neque ultrices vitae volutpat leo ullamcorper. In at tellus magna. Curabitur quis posuere purus. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Suspendisse tristique placerat feugiat. Aliquam vitae est at enim auctor ultrices eleifend a urna. Donec non tincidunt felis. Maecenas at suscipit orci.

APPENDIX A

An Appendix

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Bibliography

- [] <https://github.com/sorenvind/phd-motiondetectionindex>.
- [ACP06] 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, pages 793–813.
- [AFM92] Amihoud Amir, Martin Farach, and Yossi Matias. “Efficient randomized dictionary matching algorithms”. In: *Proc. 3rd CPM*. 1992, pages 262–275.
- [AG10] Mohamed Ibrahim Abouelhoda and Moustafa Ghanem. “String Mining in Bioinformatics”. In: *Scientific Data Mining and Knowledge Discovery*. 2010, pages 207–247.
- [AGM02] Pankaj K Agarwal, Sathish Govindarajan, and S Muthukrishnan. “Range searching in categorical data: Colored range searching on grid”. In: *Proc. 10th ESA*. 2002, pages 17–28.
- [AH00] Stephen Alstrup and Jacob Holm. “Improved algorithms for finding level ancestors in dynamic trees”. In: *Proc. 27th ICALP*. 2000, pages 73–84.
- [AI06] Alexandr Andoni and Piotr Indyk. “Efficient algorithms for substring near neighbor problem”. In: *Proc. 17th SODA*. 2006, pages 1203–1212.
- [AKO04] Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. “Replacing suffix trees with enhanced suffix arrays”. In: *JDA 2.1* (2004), pages 53–86.
- [And99] Arne Andersson. “General balanced trees”. In: *J. Algorithms* 30.1 (1999), pages 1–18.
- [Arg+08] Lars Arge et al. “The Priority R-tree: A practically efficient and worst-case optimal R-tree”. In: *ACM TALG* 4.1 (2008), page 9.
- [AT07] Arne Andersson and Mikkel Thorup. “Dynamic ordered sets with exponential search trees”. In: *Journal of the ACM (JACM)* 54.3 (2007), page 13.
- [AU07] Hiroki Arimura and Takeaki Uno. “An efficient polynomial space and polynomial delay algorithm for enumeration of maximal motifs in a sequence”. In: *JCO* (2007).
- [Bak95] Brenda S Baker. “On finding duplication and near-duplication in large software systems”. In: *Proc. 2nd WCRE*. 1995, pages 86–95.

- [BBV12] Djamal Belazzougui, Paolo Boldi, and Sebastiano Vigna. “Predecessor search with distance-sensitive query time”. In: *arXiv:1209.5441* (2012).
- [BCN10] J  r  my Barbay, Francisco Claude, and Gonzalo Navarro. “Compact rich-functional binary relation representations”. In: *Proc. 9th LATIN*. 2010, pages 170–183.
- [BDG95] Sergey Brin, James Davis, and Hector Garcia-Molina. “Copy detection mechanisms for digital documents”. In: *Proc. ACM SIGMOD*. Volume 24. 2. 1995, pages 398–409.
- [Ben75] Jon Louis Bentley. “Multidimensional binary search trees used for associative searching”. In: *Comm. ACM* 18.9 (1975), pages 509–517.
- [Ben79] Jon Louis Bentley. “Multidimensional binary search trees in database applications”. In: *IEEE Trans. Softw. Eng.* 4 (1979), pages 333–340.
- [Ber+08] Mark de Berg et al. *Computational Geometry: Algorithms and Applications*. 3rd. 2008. ISBN: 3540779736, 9783540779735.
- [Ber+13] Guillermo de Bernardo et al. “Compact Queriable Representations of Raster Data”. In: *Proc. 20th SPIRE*. 2013, pages 96–108.
- [BF04] M.A. Bender and M. Farach-Colton. “The level ancestor problem simplified”. In: *Theoret. Comput. Sci.* 321 (2004), pages 5–12.
- [Bil+11] P. Bille et al. “Random access to grammar-compressed strings”. In: *Proc. 22nd SODA*. 2011, pages 373–389.
- [Bil+12] Philip Bille et al. “Time-Space Trade-Offs for Longest Common Extensions”. In: *Proc. 23rd CPM*. 2012, pages 293–305.
- [BLN09] Nieves R Brisaboa, Susana Ladra, and Gonzalo Navarro. “K2-trees for compact web graph representation”. In: *Proc. 16th SPIRE*. 2009, pages 18–30.
- [BM72] R Bayer and EM McCreight. “Organization and maintenance of large ordered indexes”. In: *Acta Informatica* 1.3 (1972), pages 173–189.
- [Bos+09] Prosenjit Bose et al. “Succinct orthogonal range search structures on a grid with applications to text indexing”. In: *Proc. 11th WADS*. 2009, pages 98–109.
- [Boz+95] Panayiotis Bozanis et al. “New Upper Bounds for Generalized Intersection Searching Problems”. In: *Proc. 22nd ICALP*. 1995, pages 464–474.
- [BS80] Jon Louis Bentley and James B Saxe. “Decomposable searching problems I. Static-to-dynamic transformation”. In: *J. Algorithms* 1.4 (1980), pages 301–358.
- [BV94] O. Berkman and U. Vishkin. “Finding level-ancestors in trees”. In: *J. Comput. System Sci.* 48.2 (1994), pages 214–230.

- [CD00] Ross Cutler and Larry S. Davis. “Robust real-time periodic motion detection, analysis, and applications”. In: *IEEE Trans. PAMI* 22.8 (2000), pages 781–796.
- [CH03] Richard Cole and Ramesh Hariharan. “Faster suffix tree construction with missing suffix links”. In: *SIAM J. Comput.* 33.1 (2003), pages 26–42.
- [Cha+05] Moses Charikar et al. “The smallest grammar problem”. In: *IEEE Trans. Inf. Theory* 51.7 (2005), pages 2554–2576.
- [Che+04] Xin Chen et al. “Shared information and program plagiarism detection”. In: *IEEE Trans Inf Theory* 50.7 (2004), pages 1545–1551.
- [CHL03] 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), pages 129–147.
- [Cla83] Kenneth L Clarkson. “Fast algorithms for the all nearest neighbors problem”. In: *Proc. 24th FOCS*. Volume 83. 1983, pages 226–232.
- [CM05] Graham Cormode and S Muthukrishnan. “Substring compression problems”. In: *Proc. 16th SODA*. 2005, pages 321–330.
- [CM07] Graham Cormode and S Muthukrishnan. “The string edit distance matching problem with moves”. In: *ACM Trans. Algorithms* 3.1 (2007), page 2.
- [CN11] F. Claude and G. Navarro. “Self-indexed grammar-based compression”. In: *Fundamenta Informaticae* 111.3 (2011), pages 313–337.
- [Com79] Douglas Comer. “Ubiquitous B-tree”. In: *ACM CSUR* 11.2 (1979), pages 121–137.
- [DDW99] Hervé Debar, Marc Dacier, and Andreas Wespi. “Towards a taxonomy of intrusion-detection systems”. In: *Computer Networks* 31.8 (1999), pages 805–822.
- [Die91] Paul F. Dietz. “Finding Level-Ancestors in Dynamic Trees”. In: *Proc. 2nd WADS*. 1991, pages 32–40.
- [DST80] Peter J Downey, Ravi Sethi, and Robert Endre Tarjan. “Variations on the common subexpression problem”. In: *J. ACM* 27.4 (1980), pages 758–771.
- [DSW09] Christian Dick, Jens Schneider, and Ruediger Westermann. “Efficient Geometry Compression for GPU-based Decoding in Realtime Terrain Rendering”. In: *CGF* 28.1 (2009), pages 67–83.
- [Du+14] Shan Du et al. “Event Detection by Spatio-Temporal Indexing of Video Clips.” In: *IJCTE* 6.1 (2014).
- [EGS08] David Eppstein, Michael T Goodrich, and Jonathan Z Sun. “Skip quadrees: Dynamic data structures for multidimensional point sets”. In: *IJCGA* 18.01n02 (2008), pages 131–160.

- [EKZ76] P. van Emde Boas, R. Kaas, and E. Zijlstra. “Design and implementation of an efficient priority queue”. In: *Theory Comput. Syst.* 10.1 (1976), pages 99–127.
- [FGN14] Arash Farzan, Travis Gagie, and Gonzalo Navarro. “Entropy-bounded representation of point grids”. In: *CGTA* 47.1 (2014), pages 1–14.
- [FP09] Maria Federico and Nadia Pisanti. “Suffix tree characterization of maximal motifs in biological sequences”. In: *Theor. Comput. Sci.* 410.43 (2009), pages 4391–4401.
- [FT98] Martin Farach and Mikkel Thorup. “String Matching in Lempel–Ziv Compressed Strings”. In: *Algorithmica* 20.4 (1998), pages 388–404.
- [FW93] Michael L. Fredman and Dan E. Willard. “Surpassing the information theoretic bound with fusion trees”. In: *J. Comput. System Sci.* 47.3 (1993), pages 424–436.
- [Gag+12a] Travis Gagie et al. “A faster grammar-based self-index”. In: *arXiv:1209.5441* (2012).
- [Gag+12b] Travis Gagie et al. “Colored range queries and document retrieval”. In: *TCS* (2012).
- [Gar+14] Sandra Alvarez Garcia et al. “Interleaved K2-Tree: Indexing and Navigating Ternary Relations”. In: *Proc. DCC*. 2014, pages 342–351.
- [Gas+05] Leszek Gasieniec et al. “Real-time traversal in grammar-based compressed files”. In: *Proc. 15th DCC*. 2005, page 458.
- [Gas+96] Leszek Gasieniec et al. “Randomized efficient algorithms for compressed strings: The finger-print approach”. In: *Proc. 7th CPM*. 1996, pages 39–49.
- [GG98] Volker Gaede and Oliver Günther. “Multidimensional access methods”. In: *ACM CSUR* 30.2 (1998), pages 170–231.
- [GJS95] P Gupta, R Janardan, and M Smid. “Further Results on Generalized Intersection Searching Problems: Counting, Reporting, and Dynamization”. In: *J. Algorithms* 19.2 (1995), pages 282–317.
- [GJS97] Prosenjit Gupta, Ravi Janardan, and Michiel Smid. “A technique for adding range restrictions to generalized searching problems”. In: *Inform. Process. Lett.* 64.5 (1997), pages 263–269.
- [Gro+11] Roberto Grossi et al. “MADMX: A strategy for maximal dense motif extraction”. In: *J. Comp. Biol.* 18.4 (2011), pages 535–545.
- [GSS98] Nicola Galli, Bernhard Seybold, and Klaus Simon. “Compression of Sparse Matrices: Achieving Almost Minimal Table Size”. In: *Proc. ALEX*. 1998, pages 27–33.
- [Gut] Antonin Guttman. “R-trees: A dynamic index structure for spatial searching”. In: *Proc. 1984 ACM SIGMOD*. Volume 14. 2, pages 47–57.

- [Hae+10] Simon Haegler et al. “Grammar-based Encoding of Facades”. In: *CGF* 29.4 (2010), pages 1479–1487.
- [Hag98] Torben Hagerup. “Sorting and Searching on the Word RAM”. In: *Proc. 15th STACS*. 1998, pages 366–398.
- [HT84a] D. Harel and R. E. Tarjan. “Fast algorithms for finding nearest common ancestors”. In: *SIAM J. Comput.* 13.2 (1984), pages 338–355.
- [HT84b] D. Harel and R.E. Tarjan. “Fast algorithms for finding nearest common ancestors”. In: *SIAM J. Comput.* 13.2 (1984), pages 338–355.
- [Hu+04] Weiming Hu et al. “A survey on visual surveillance of object motion and behaviors”. In: *IEEE Trans. SMC* 34.3 (2004), pages 334–352.
- [Hua11] Shih-Chia Huang. “An advanced motion detection algorithm with video quality analysis for video surveillance systems”. In: *IEEE Trans. CSVT* 21.1 (2011), pages 1–14.
- [JL93] Ravi Janardan and Mario Lopez. “Generalized intersection searching problems”. In: *IJCGA* 3.01 (1993), pages 39–69.
- [JMS05] Joseph JáJá, Christian W Mortensen, and Qingmin Shi. “Space-efficient and fast algorithms for multidimensional dominance reporting and counting”. In: *Proc. 15th ISAAC*. 2005, pages 558–568.
- [Kal02] Adam Kalai. “Efficient pattern-matching with don’t cares”. In: *Proc. 13th SODA*. 2002, pages 655–656.
- [Kap+07] Haim Kaplan et al. “Counting colors in boxes”. In: *Proc. 18th SODA*. 2007, pages 785–794.
- [Kap+13] Alexis Kaporis et al. “Improved Bounds for Finger Search on a RAM”. In: *Algorithmica* (2013), pages 1–38.
- [KCW] Wei Chieh Kao, Shih Hsuan Chiu, and Che Y. Wen. “An effective surveillance video retrieval method based upon motion detection”. In: *IEEE ISI 2008* (), pages 261–262.
- [KO91] Marc J van Kreveld and Mark H Overmars. “Divided k-d trees”. In: *Algorithmica* 6.1-6 (1991), pages 840–858.
- [KR87] Richard M Karp and Michael O Rabin. “Efficient randomized pattern-matching algorithms”. In: *IBM J. Res. Dev.* 31.2 (1987), pages 249–260.
- [Kre92] Marc van Kreveld. *New results on data structures in computational geometry*. PhD thesis, Department of Computer Science, University of Utrecht, Netherlands, 1992.
- [KS99] KV Ravi Kanth and Ambuj Singh. “Optimal Dynamic Range Searching in Non-replicating Index Structures”. In: *Proc. 7th ICDT*. 1999, pages 257–276.
- [LP01] I. Rigoutsos L. Parida and D. E. Platt. “An Output-Sensitive Flexible Pattern Discovery Algorithm”. In: *Proc. 12th CPM*. 2001, pages 131–142.

- [LP12] Kasper Green Larsen and Rasmus Pagh. “I/O-efficient data structures for colored range and prefix reporting”. In: *Proc. 23rd SODA*. 2012, pages 583–592.
- [Lue78] George S Lueker. “A data structure for orthogonal range queries”. In: *Proc. 19th FOCS*. 1978, pages 28–34.
- [LW13] Kasper Green Larsen and Freek van Walderveen. “Near-Optimal Range Reporting Structures for Categorical Data.” In: *Proc. 24th SODA*. 2013, pages 265–276.
- [LW80] DT Lee and CK Wong. “Quintary trees: a file structure for multidimensional database systems”. In: *ACM TODS* 5.3 (1980), pages 339–353.
- [McC76] Edward M. McCreight. “A Space-Economical Suffix Tree Construction Algorithm”. In: *Journal of the ACM* 23.2 (April 1976), pages 262–272.
- [ME10] Nizar R Mabroukeh and Christie I Ezeife. “A taxonomy of sequential pattern mining algorithms”. In: *ACM CSUR* 43.1 (2010), page 3.
- [MN07] Veli Mäkinen and Gonzalo Navarro. “Rank and select revisited and extended”. In: *TCS* 387.3 (2007), pages 332–347.
- [MN90] K. Mehlhorn and S. Näher. “Bounded ordered dictionaries in $O(\lg \lg N)$ time and $O(n)$ space”. In: *Inform. Process. Lett.* 35.4 (1990), pages 183–189.
- [Mor03] Christian W Mortensen. *Generalized static orthogonal range searching in less space*. Technical report. TR-2003-22, The IT University of Copenhagen, 2003.
- [Nek09] Yakov Nekrich. “Orthogonal Range Searching in Linear and Almost-linear Space”. In: *Comput. Geom. Theory Appl.* 42.4 (2009), pages 342–351.
- [Nek12] Yakov Nekrich. “Space-efficient range reporting for categorical data”. In: *Proc. 31st PODS*. 2012, pages 113–120.
- [Nek14] Yakov Nekrich. “Efficient range searching for categorical and plain data”. In: *ACM TODS* 39.1 (2014), page 9.
- [NV13] Yakov Nekrich and Jeffrey Scott Vitter. “Optimal color range reporting in one dimension”. In: *Proc. 21st ESA*. 2013, pages 743–754.
- [Ore82] Jack A Orenstein. “Multidimensional tries used for associative searching”. In: *Inform. Process. Lett.* 14.4 (1982), pages 150–157.
- [Ove87] Mark H Overmars. *Design of Dynamic Data Structures*. 1987. ISBN: 038712330X.
- [Par+00] Laxmi Parida et al. “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, pages 297–308.

- [Pat07] Mihai Patrascu. “Lower bounds for 2-dimensional range counting”. In: *Proc. 39th STOC*. 2007, pages 40–46.
- [PP09] Benny Porat and Ely Porat. “Exact and approximate pattern matching in the streaming model”. In: *Proc. 50th FOCS*. 2009, pages 315–323.
- [Pro+03] Octavian Procopiuc et al. “Bkd-tree: A dynamic scalable kd-tree”. In: *Proc. 8th SSTD*. 2003, pages 46–65.
- [PW00] Renato Pajarola and Peter Widmayer. “An image compression method for spatial search”. In: *IEEE Trans. Image Processing* 9.3 (2000), pages 357–365.
- [PYK06] Lukáš Pichl, Takuya Yamano, and Taisei Kaizoji. “On the symbolic analysis of market indicators with the dynamic programming approach”. In: *Advances in Neural Networks-ISNN*. 2006, pages 432–441.
- [RH04] Isidore Rigoutsos and Tien Huynh. “Chung-Kwei: a Pattern-discovery-based System for the Automatic Identification of Unsolicited E-mail Messages”. In: *CEAS*. 2004.
- [Rob] John T Robinson. “The KDB-tree: a search structure for large multidimensional dynamic indexes”. In: *Proc. 1981 ACM SIGMOD*, pages 10–18.
- [Ryt03] Wojciech Rytter. “Application of Lempel–Ziv factorization to the approximation of grammar-based compression”. In: *Theoret. Comput. Sci.* 302.1 (2003), pages 211–222.
- [Sac+94] Todd S Sachs et al. “Real-time motion detection in spiral MRI using navigators”. In: *Magnetic resonance in medicine* 32.5 (1994), pages 639–645.
- [Sag98] Marie-France Sagot. “Spelling approximate repeated or common motifs using a suffix tree”. In: *Proc. 3rd LATIN*. 1998, pages 374–390.
- [Sam90] Hanan Samet. *Applications of spatial data structures*. Addison-Wesley, 1990.
- [Sch+08] Grant Schindler et al. “Detecting and matching repeated patterns for automatic geo-tagging in urban environments”. In: *CVPR*. 2008, pages 1–7.
- [SJ05] Qingmin Shi and Joseph JáJá. “Optimal and near-optimal algorithms for generalized intersection reporting on pointer machines”. In: *Inform. Process. Lett.* 95.3 (2005), pages 382–388.
- [SR06] Reza Sherkat and Davood Rafiei. “Efficiently evaluating order preserving similarity queries over historical market-basket data”. In: *Proc. 22nd ICDE*. 2006, pages 19–19.
- [TH] Ying-Li Tian and Arun Hampapur. “Robust salient motion detection with complex background for real-time video surveillance”. In: *WACV 2005*.

- [TV01] Igor V Tetko and Alessandro EP Villa. “A pattern grouping algorithm for analysis of spatiotemporal patterns in neuronal spike trains.” In: *J. Neurosci. Meth.* 105.1 (2001), pages 1–14.
- [Ukk09] Esko Ukkonen. “Maximal and minimal representations of gapped and non-gapped motifs of a string”. In: *Theor. Comput. Sci.* 410.43 (2009), pages 4341–4349.
- [Wil12] Virginia Vassilevska Williams. “Multiplying matrices faster than Coppersmith-Winograd”. In: *Proc. 44th STOC.* 2012, pages 887–898.
- [Wil83] D.E. Willard. “Log-logarithmic worst-case range queries are possible in space $\Theta(N)$ ”. In: *Inform. Process. Lett.* 17.2 (1983), pages 81–84.
- [Zhu+02] Qing Zhu et al. “An Efficient Data Management Approach for Large Cyber-City GIS”. In: *ISPRS Archives* (2002), pages 319–323.
- [ZL77] Jacob Ziv and Abraham Lempel. “A universal algorithm for sequential data compression”. In: *Information Theory, IEEE Trans. Inf. Theory* 23.3 (1977), pages 337–343.
- [ZL78] Jacob Ziv and Abraham Lempel. “Compression of individual sequences via variable-rate coding”. In: *Information Theory, IEEE Trans. Inf. Theory* 24.5 (1978), pages 530–536.