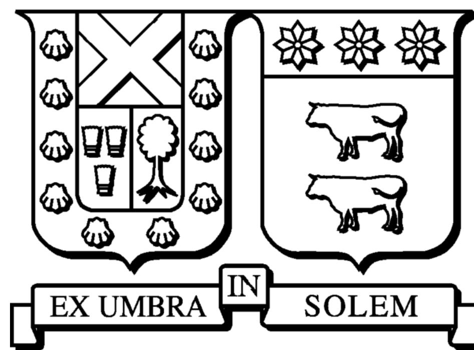


UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA

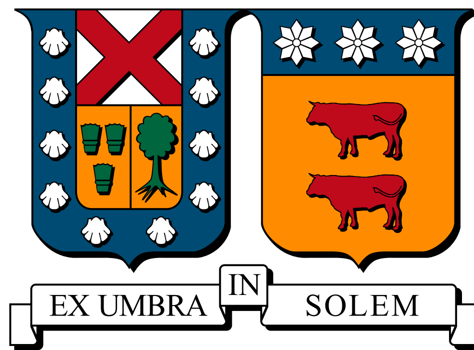
DEPARTAMENTO DE INFORMÁTICA

TRIE-COMPRESSED INTERSECTABLE SETS

Juan Pablo Castillo González

MAGÍSTER EN CIENCIAS DE LA INGENIERÍA INFORMÁTICA

AGOSTO 2023



UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA
DEPARTAMENTO DE INFORMÁTICA

TRIE-COMPRESSED INTERSECTABLE SETS

Tesis de Grado presentada por
JUAN PABLO CASTILLO GONZÁLEZ

como requisito parcial para optar al grado de
MAGÍSTER EN CIENCIAS DE LA INGENIERÍA INFORMÁTICA

PROFESOR GUÍA
DIEGO ARROYUELO

AGOSTO 2023

TÍTULO DE LA TESIS:

TRIE-COMPRESSED INTERSECTABLE SETS

AUTOR:

JUAN PABLO CASTILLO GONZÁLEZ

Trabajo de Tesis, presentado en cumplimiento parcial de los requisitos para el Grado de **Magíster en Ciencias de la Ingeniería Informática** de la Universidad Técnica Federico Santa María.

Diego Arroyuelo ______________________
Profesor Guía

Roberto Asín Achá ______________________
Profesor Correferente Interno

Antonio Fariña ______________________
Profesor Correferente Externo

Agosto 2023.

Valparaíso, Chile.

Resumen

En esta tesis, se presentan algoritmos y estructuras de datos eficientes en espacio y tiempo para el problema offline set intersection. Mostramos que un conjunto de enteros ordenados $S \subseteq [0..u)$ de n elementos se puede representar usando espacio comprimido, mientras se soportan intersecciones de k conjuntos en tiempo adaptativo $O(k\delta log(u/\delta))$, siendo δ la medida de alternancia introducida por Barbay y Kenyon. De esta manera, se mostrará que el desempeño adaptativo de algoritmos como el de Demaine, López-Ortiz y Munro [SODA, 2000], y el de Barbay y Kenyon [TALG, 2008] se puede lograr usando espacio comprimido, introduciendo nuevos conocimientos sobre este problema algorítmico fundamental. Los resultados experimentales sugieren que nuestros enfoques son competitivos en la práctica, superando las alternativas más eficientes (Partitioned Elias-Fano, Roaring Bitmaps, y Recursive Universe Partitioning (RUP)) en varios escenarios, ofreciendo en general trade-offs espacio-tiempo relevantes.

Abstract

In this thesis, we introduce space- and time-efficient algorithms and data structures for the offline set intersection problem. We show that a sorted integer set $S \subseteq [0..u)$ of n elements can be represented using compressed space while supporting k-way intersections in adaptive $O(k\delta \log(u/\delta))$ time, δ being the alternation measure introduced by Barbay and Kenyon. In this way, we show that the adaptive performance of algorithms such as the one by Demaine, López-Ortiz and Munro [SODA, 2000], and the one by Barbay and Kenyon [TALG, 2008] can be achieved using compressed space, introducing new insights into this fundamental algorithmic problem. Our experimental results suggest that our approaches are competitive in practice, outperforming the most efficient alternatives (Partitioned Elias-Fano indexes, Roaring Bitmaps, and Recursive Universe Partitioning (RUP)) in several scenarios, offering in general relevant space-time trade-offs.

Contents

Resumen iii

Abstract iv

List of Tables ix

List of Figures xi

1 Introduction and Problem Definition 1

1.1 The Offline Set Intersection Problem 2

1.2 Objectives 4

1.2.1 General Objective 4

1.2.2 Specific Objectives 4

2 Preliminary Concepts 6

2.1 Tries 6

2.1.1 Binary Tries 7

2.2 Operations over Integer Sets 9

2.2.1 Characteristic Bit Vectors 9

2.3 Compression Measures 10

2.3.1 The Information-Theoric Worst-Case Lower Bound 10

2.3.2 The $\mathtt{gap}(S)$ Compression Measure 10

2.3.3 The $\mathtt{rle}(S)$ Compression Measure 11

2.3.4 The $\mathtt{trie}(S)$ Compression Measure 11

3 Previous Work **13**

3.1 Succinct Set Representations 13

3.2 Elias Codes 13

3.2.1 Elias-γ 14

3.2.2 Elias-δ 14

3.3 Binary Interpolative Coding 15

3.4 Byte-aligned Codes 16

3.4.1 Varint 16

3.4.2 Varint-GB 17

3.4.3 Varint-G8UI 17

3.5 Word-aligned Codes or Simple Family 19

3.5.1 Simple 9 19

3.5.2 Simple 16 20

3.5.3 Simple 8b 21

3.6 Binary Packing 21

3.7 Patched Coding 22

3.8 Elias-Fano 22

3.8.1 Partitioned Elias-Fano 24

3.9 Roaring Bitmaps 25

3.10 Recursive Universe Partitioning 26

3.11 Adaptive Set Intersection Algorithms 26

3.11.1 Barbay and Kenyon 27

3.11.2 Demaine, López-Ortiz and Munro Algorithm 29

3.11.3 Trabb-Pardo's Algorithm 30

3.12 How to Intersect Compressed Sets? 32

4 Compressed Intersectable Sets using Binary Tries 33

4.1 Trie Intersection Certificates 33

4.2 Compressed Representation of Binary Tries 36

4.2.1 Space Analysis 36

4.3 Adaptive and Compressed Intersection 37

5 Compressing Runs on Binary Tries 41

5.1 Encoding Runs on Binary Tries 41

5.2 The `rTrie`(S) Measure 43

5.3 Adaptive Runs Compressed Intersection 44

5.4 Run-partition Certificate 45

6 Implementation and Experimental Results **48**

6.1 Implementation 48

6.1.1 A Multi-threaded Implementation 49

6.2 Experimental Setup 50

6.3 Experimental Results 52

Conclusions and Future Work **57**

Bibliography **58**

List of Tables

6.1 Dataset summary and average space usage (in bits per integer, bpi) for different compression measures and baseline representations. 51

6.2 Average intersection time and space usage (in bits per integer) for all alternatives tested. For `trie` and `rTrie` alternatives, we show running times using the format X/Y, where X is the average intersection time using a single thread, whereas Y is the average running time for 16 threads. 54

List of Figures

2.1 Trie for the strings set {`car, cat, done, trie, try`}. 7

2.2 Binary tries $\mathsf{bintrie}(S_1)$ and $\mathsf{bintrie}(S_2)$ encoding sets $S_1 = \{1, 3, 7, 8, 9, 10, 11, 12\}$ and $S_2 = \{2, 5, 7, 12, 15\}$. 8

3.1 `Varint` encoding the set $S = \{1, 15, 511, 131071\}$. 17

3.2 `Varint-GB` encoding the set $S = \{1, 15, 511, 131071\}$. 18

3.3 `Varint-G8UI` encoding the set $S = \{1, 15, 511, 131071\}$. 18

3.4 `S9` encoding the set $S = \{98, 112, 117, 121\}$, which represents 4 integers in 7-bit chunks within a 32-bit machine word. The case is C4 with header **0011**. 20

3.5 Elias-Fano encoding the sequence $S = \langle 5, 8, 9, 15, 31 \rangle$, bit array H store upper bits and L store $l = \lfloor \log(32/5) \rfloor = 2$ lower bits. 23

3.6 Vertical lines show the smallest partition certificate $\mathcal{P} = \{[0..1], [2..2], [3..4], [5..6], [7..7], [8..11], [12..12], [13..15]\}$ of size $\delta = 8$ of the universe $[0..16)$ for the intersection of sets $S_1 = \{1, 3, 7, 8, 9, 10, 11, 12\}$ and $S_2 = \{2, 5, 7, 12, 15\}$. Gray intervals are the intersection elements, and the uncolored are the ones left out. 29

4.1 Trie certificate for the intersection $\{1, 3, 7, 8, 9, 10, 11, 12\} \cap \{2, 5, 7, 12, 15\}$. This trie shows the nodes that must be checked to determine that the result is, in this case, $\{7, 12\}$. This corresponds to the intersection of the binary tries representing these sets. 34

4.2 Above $\mathsf{bintrie}(S)$ for $S = \{1, 3, 7, 8, 9, 10, 11, 12\}$. Below level-wise bit vector representation of $\mathsf{bintrie}(S)$, dotted lines are implicit, as they are computed using operation $\mathsf{rank_1}$ on the bit vectors. 37

5.1 Above, the binary trie representing set $\{1, 3, 7, 8, 9, 10, 11, 12\}$. Notice that the subtree whose leaves correspond to elements $8, 9, 10, 11$ is a full subtree. Below, our compact representation removing full subtrees and encoding their roots with **00**. 42

5.2 A query instance $Q = \{S_{i_1}, S_{i_2}, S_{i_3}, S_{i_4}\}$ and its smallest run-partition certificate $\mathcal{P}^r_{\mathsf{AC}}(Q) = \{[0..7], [8..9], [10..10], [11..14], [15..15]\}$ of size $\xi = 5$. Gray intervals are the intersection elements, and the uncolored are the ones left out. 46

6.1 Multi-threaded binary trie intersection of T_1 and T_2 for $t = 4$, blue dashed rectangle represent the level $c = \lfloor \log t \rfloor = 2$ in T_1 and T_2. Dashed triangle with $\times$ denotes not part of the intersection. 50

6.2 Space vs. time trade-off for all alternative tested on the 3 datasets. 56

Chapter 1

Introduction and Problem Definition

Technology evolution is no longer keeping pace with the growth of data. In the era of big data, we are currently facing problems storing and processing the huge amounts of data produced every day. People rely on data-intensive applications. Thus, the need to store and query data in devices whose capacity is surpassed by the data volume is routine today, ranging from astronomy data to be processed by super computers, to personal data to be processed by cellphones. This is why, representing integer sets and their basic operations (e.g., intersections, unions and differences) is a fundamental problem for data management applications.

Sets are one of the fundamental mathematical concepts related to data storage. Operations such as intersections, unions, and set differences are key to querying them. For example, the use of `AND` and `OR` logical operators in web search engines translate into intersections and unions, respectively. Representing sets to support their basic operations efficiently has been major concern for many decades. In several applications, such as query processing in information retrieval (IR) and database management systems (DBMS), sets are known in advance to queries, hence data structures can be build to speed up query processing. With this motivation, in this work we focus on the following problem. Specifically, this work is interested in the compact representation of integers to resolve the *Offline Set Intersection Problem* (OSIP), that we define below.

1.1 The Offline Set Intersection Problem

Let $\mathcal{S} = \{S_1, \ldots, S_N\}$ be a family of N sets over the universe $[0, u)$, each one of size $|S_i| = n_i$. Each set of $\mathcal{S}$ is sorted, that is, $S_i[j-1] < S_i[j]$ for $i = [1..N]$, $j = [1..n_i]$ On this family $\mathcal{S}$ we want to queries of the form $Q = \{i_1, \ldots, i_k\} \subseteq [1..N]$, which need to compute:

$$\mathcal{I}(Q) = \bigcap_{i \in Q} S_i.$$

This problem will be called the *Offline Set Intersection Problem* (OSIP, for short).

We assume $u = 2^k$ in this work, for $k \geq 0$. Unless explicitly otherwise stated, we also assume $\log x = \lceil \log_2 x \rceil$ and $\log 0 = 0$. Typical applications of this problem include the efficient support of join operations in DBMS [1, 2], query processing using inverted indexes in IR [3, 4], and computational biology [5], among others. Building a data structure to speed up intersections, however, increases the space usage. Today, data-intensive applications encourage not only time- but also space-efficient solutions [6]. Being able to process big datasets entirely in main memory is the main motivation. Compact, succinct, and compressed data structures are important to achieve this [7]. We study here compressed data structures to efficiently support the OSIP. We assume the word RAM model of computation with word size $w = \Theta(\log u)$. Arithmetic, logic, and bitwise operations, as well as accesses to w-bit memory cells, take O(1) time.

The literature on this problem is vast. For the online version of the problem, where sets to be intersected are given at query time, so there is no time to preprocess them. Algorithms like the ones by Baeza-Yates [8], Demaine et al. [9], and Barbay and Kenyon [10] are among the most efficient and well-known approaches. In particular, the two latter algorithms are adaptive, meaning that they are able to perform faster on "easier" query instances. The algorithm by Barbay and Kenyon runs in optimal $O(\delta \sum_{i \in Q} \log(n_i/\delta))$ time, where δ is the so-called alternation measure that quantifies the query difficulty [10]. The algorithm by Demaine et al. [9] has running time $O(k\delta \log(n/\delta))$, for $n = \sum_{i \in Q} n_i$, which is optimal when $\max_{i \in Q} \{\log n_i\} = O(\min_{i \in Q} \{\log n_i\})$ [11]. These algorithms require sets to be stored in plain form, e.g. using a sorted array or a B-tree [9], requiring $\Theta(mw)$ bits of space, for $m = \sum_{i=1}^{N} n_i$.

This can be excessive when dealing with large databases.

For the OSIP, we have the extensive literature on inverted indexes [12, 13, 3, 14], whose main focus is on practical space-efficient set representations supporting intersections. Approaches like Optimized PForDelta [12], Roaring Bitmaps [13], SIMD-BP128 [3], and Recursive Universe Partitioning [14] shine in practical scenarios, yet without appealing theoretical guarantees of space usage and intersection computation time. Another relevant approach on these lines is Partitioned Elias-Fano (PEF) [15], able to exploit the distribution and clustering of set elements to improve space usage. Barbay and Kenyon's algorithm can be implemented on PEF, taking $O(\delta \sum_{i \in Q} \log(u/n_i))$ time. Regarding space usage, there is no known bound (although it performs well in practice). On a more theoretical track, Bille et al. [16] introduce a data structure that uses $O(mw)$ bits of space and supports intersections in $O(n \log^2(w)/w + k|\mathcal{I}(Q)|)$ time. Cohen and Porat [17] data structure also uses $O(mw)$ bits of space and allows one to compute the intersection between any two sets in $\mathcal{S}$ in $O(\sqrt{N|\mathcal{I}(Q)|} + \mathcal{I}(Q)|)$ time. Besides using linear space, this approach only works for pairwise intersections (and is hard to efficiently extend to multiway intersections). Finally, Ding and Konig [18] introduce a data structure able to compute intersections in $O(n/\sqrt{w} + k|\mathcal{I}(Q)|)$ expected time, and uses linear $O(m)$ space. The space can be improved in practice to use about 1.88 times the space of an Elias γ/δ compressed inverted index [18], yet with no theoretical guarantees. Later, Gagie et al. [19] showed that wavelet trees [20] can support intersections in $O(k\delta \log(u/\delta))$ time, using *uncompressed* $mw(1 + o(1))$ bits of space.

In this work we show that $O(k\delta \log(u/\delta))$ intersection time using compressed space is possible to resolve OSIP. In particular, in Chapter 2 we review the key concepts for this work. In Chapter 3 we revisit previous work and the classic (and neglected) algorithm by Trabb-Pardo [21] (former Knuth's student). In Chapter 4 we introduce the trie intersection certificates concept to prove that running time of Trabb-Pardos's algorithm is actually $O(k\delta \log(u/\delta))$, so it is likely the first adaptive intersection algorithm that ever existed; then, we show that Trabb-Pardo's algorithm can be implemented in compressed space, yielding an adaptive and compressed set intersection algorithm. In Chapter 5 we show how to exploit the presence of runs of successive elements, typical in some applications to formally improve both space

usage of input sets and the intersection computation time by introducing an intersection algorithm that runs in time $O(k\xi \log(u/\xi))$, where $\xi \leq \delta$ is an adaptability measure we introduce. In Chapter 6 we implement our proposals and show preliminary experimental results that indicate that our approaches are appealing not only in theory, but also in practice, outperforming the most competitive state-of-the-art approaches in some practical inverted-index datasets we use in our tests. Overall, we conclude that both theoretical guarantees and practicality can be achieved with a single approach, which is a step forward in bridging the gap between theory and practice in this important line of research.

1.2 Objectives

1.2.1 General Objective

The main objective of this thesis is to design, analyze, implement and evaluate (experimentally and theoretically) a compressed data structure, to compress set of sorted integers. The structure supports set intersections efficiently over compressed space.

1.2.2 Specific Objectives

- To design a compressed data structure based on binary tries.
- To design an intersection algorithm for the compressed data structure of the previous item.
- To evaluate theoretically the space used by the compressed data structure, using compression measures.
- To evaluate theoretically the execution time of the intersection algorithm.
- To implement the compressed data structure using binary tries.
- To implement an intersection algorithm for the compressed data structure.

- To evaluate experimentally the space used by the data structure in comparison to the state of the art.
- To evaluate experimentally the intersection algorithm time compared to the state of the art.
- To make the implementation available for public use.

Chapter 2

Preliminary Concepts

In this chapter, we will review the main previous concepts needed to understand this thesis.

2.1 Tries

Giving a set of strings S, a trie [22, 23] for S is a tree where every string in S is represented by a node in the tree. Let v be the node representing string $s_i \in S$. This means that the trie edges in the root-to-v path are labeled with the symbols of s_i, in order (every edge is labeled with only one symbol). Notice that all nodes descending from v in the trie represent strings which have s_i as prefix. The trie root represents the empty string.

For example, if we need to store the strings set {`car`, `cat`, `done`, `trie`, `try`} we can use a trie as shown in Figure 2.1. Notice that the trie store the common prefixes only one time, when the shared prefix ends the tree path branches into possible characters that follow from that prefix. For `cat` and `car`, the data structure only stores `c` $\rightarrow$ `a` just once, and then the path branches in `c` $\rightarrow$ `a` $\rightarrow$ `r` and `c` $\rightarrow$ `a` $\rightarrow$ `t`. The same occurs for `trie` and `try`. Finally, `done` shares no common prefix with any string in the set, therefore, the corresponding root-to-leaf path.

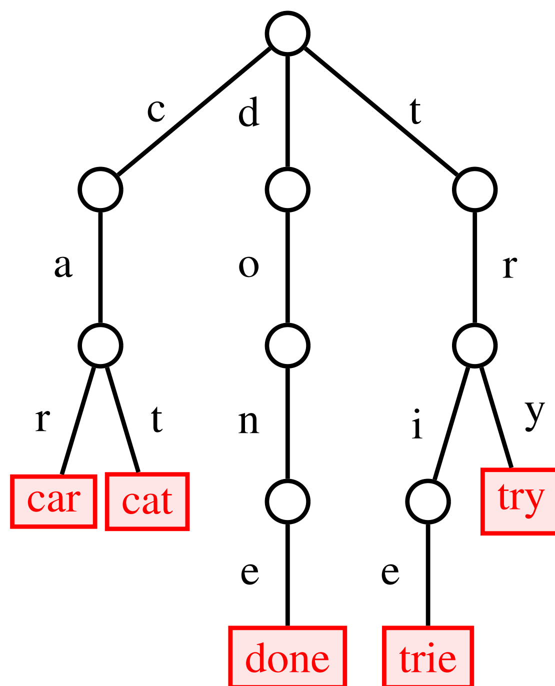


Figure 2.1: Trie for the strings set {`car, cat, done, trie, try`}.

2.1.1 Binary Tries

Let us consider now representing a set $S \in C^{(n)}$ using a binary trie denoted $\mathsf{bintrie}(S)$, where the $\ell = \lceil \log u \rceil$-bit binary encoding of every element is added. Each internal node in $\mathsf{bintrie}(S)$ has two children, the left one corresponding to bit **0** and the right one to bit **1**. The external nodes of $\mathsf{bintrie}(S)$ have no children, as usual. In our case, we distinguish two kinds of external nodes. A *void external node* is one whose depth is either $d < \ell$, or alternatively $d = \ell$ yet it represents no element in S. A *valid external node* (or, simply, *external node*, or alternatively a *leaf*), on the other hand, is one whose depth is exactly ℓ and corresponds to an element in S. Thus, $\mathsf{bintrie}(S)$ has $|S|$ valid external nodes, all at depth ℓ. For a leaf v corresponding to element $x_i \in S$, the root-to-v path is hence labeled with the binary encoding of x_i. This approach has been used for representing sets since at least the late 70s by Trabb-Pardo [21].

For example, consider the sets $S_1 = \{1, 3, 7, 8, 9, 11, 12\}$, and $S_2 = \{2, 5, 7, 12, 15\}$ over universe $[0..16)$, that we shall use as running examples. Figure 2.2 shows the corresponding tries $\mathsf{bintrie}(S_1)$ and $\mathsf{bintrie}(S_2)$, with external nodes shown as red squares and void external nodes with dotted lines.

Definition 2.1. *We say that a node v in* $\mathsf{bintrie}(S)$ covers *all leaves that descend from it. In such a case, we call v a* cover node *of the corresponding leaves.*

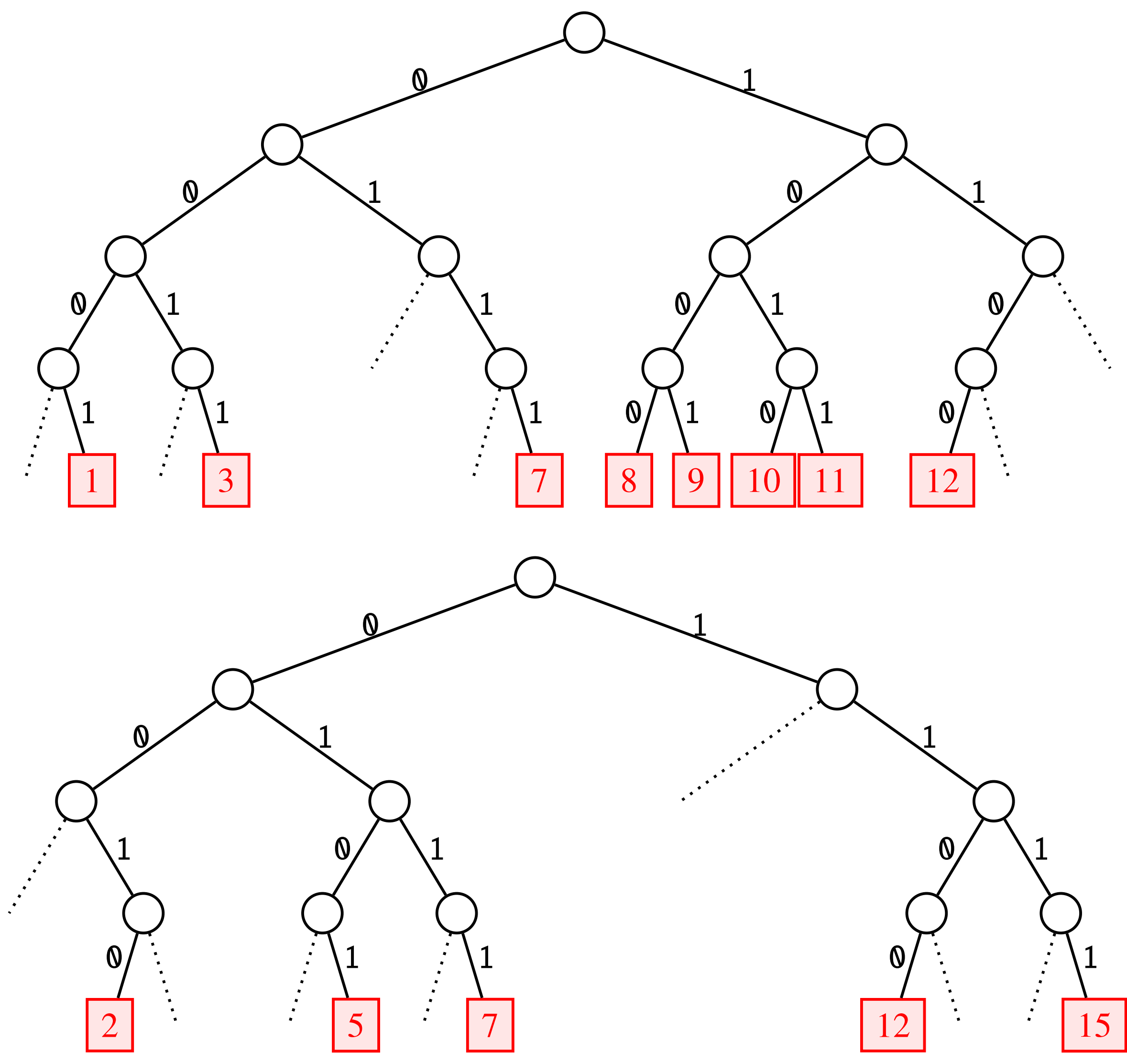


Figure 2.2: Binary tries bintrie(S_1) and bintrie(S_2) encoding sets $S_1 = \{1, 3, 7, 8, 9, 10, 11, 12\}$ and $S_2 = \{2, 5, 7, 12, 15\}$.

The following lemma summarizes several results that shall be important for our work:

Lemma 2.1 ([19], Lemmas 2–5)**.** *For* $\mathsf{bintrie}(S)$*, the following results hold:*

1. *Any contiguous range of L leaves in* $\mathsf{bintrie}(S)$ *is covered by* $O(\log L)$ *nodes.*
2. *Any set of r nodes in* $\mathsf{bintrie}(S)$ *has* $O(r \log \frac{u}{r})$ *ancestors.*
3. *Any set of r nodes in* $\mathsf{bintrie}(S)$ *covering a contiguous range of leaves in the trie has* $O(r + \log u)$ *ancestors.*
4. *Any set of r nodes in* $\mathsf{bintrie}(S)$ *covering subsets of L contiguous leaves has* $O(\log u + r \log \frac{L}{r})$ *ancestors.*

2.2 Operations over Integer Sets

The following operations on a sorted integer set S are of interest:

- $\mathsf{rank}(S, x)$: for $x \in [0..u)$, yields $|\{y \in S,\ y \leq x\}|$.
- $\mathsf{select}(S, j)$: for $1 \leq j \leq |S|$, yields $x \in S$ s.t. $\mathsf{rank}(S, x) = j$.
- $\mathsf{successor}(S, x)$: for $x \in [0..u)$, yields $\min\{y \in S,\ y \geq x\}$.

2.2.1 Characteristic Bit Vectors

For set S, let $C_S[0..u)$ denote its *characteristic bit vector* (cbv for short), such that $C_S[x] = \mathbf{1}$ if and only if $x \in S$ or $C_S[x] = \mathbf{0}$ otherwise.

Using the cbv C_S representation, we are interesting in the following operations:

- $C_S.\mathsf{rank}_{\mathbf{1}}(x)$: for $x \in [0..u)$, yields the number of $\mathbf{1}$s in $C_S[0..x]$.
- $C_S.\mathsf{select}_{\mathbf{1}}(k)$: for $1 \leq j \leq |S|$, yields the smallest position $0 \leq x < u$ such that $C_S.\mathsf{rank}_{\mathbf{1}}(x) = j$.

2.3 Compression Measures

A *compression measure* quantifies the amount of bits needed to encode data using a particular compression model. For an integer universe $U = [0..u)$, let $C^{(n)} \subseteq 2^U$, $n \in U$ denote the class of all sets $S \subseteq U$ such that $|S| = n$. Next, we review the typical compression measures for integer sets $S \in C^{(n)}$, which are important for this work.

2.3.1 The Information-Theoric Worst-Case Lower Bound

A worst-case lower bound on the number of bits needed to represent any $S \in C^{(n)}$ can be obtained using the following information-theoric argument. As $|C^{(n)}| = \binom{u}{n}$, at least $\mathcal{B}(n, u) = \left\lceil \log \binom{u}{n} \right\rceil$ are needed to differentiate a given set $S \in C^{(n)}$ among all other possible sets. This bound is tight, as there are set representations achieving this bound [24]. If $n \ll u$, then:

$$\mathcal{B}(n, u) = n \log e + n \log \frac{u}{n} - O(\log u) \text{ bits.} \tag{2.1}$$

We obtain this value for $\mathcal{B}(n, u)$ using the Stirling approximation of $n!$. This is just a worst case lower bound, as some sets in $C^{(n)}$ can be represented using less space, as we will see next.

2.3.2 The $\mathsf{gap}(S)$ Compression Measure

The $\mathsf{gap}(S)$ measure is related with the space needed to represent a set S using the differences between consecutive elements of S. That is, how much space in bits we needed to store the sequence $G = \langle g_1, \ldots, g_n \rangle$, where $g_1 = x_1$ and, for $i = 2, \ldots, n$, $g_i = x_i - x_{i-1} - 1$. We will call these differences gaps. Thus, in the gap model we have $C_S[0..u) = \mathbf{0}^{g_1}\mathbf{1}\mathbf{0}^{g_2}\mathbf{1} \cdots \mathbf{0}^{g_n}\mathbf{1}$ (assuming wlog that C_S ends with $\mathbf{1}$). Then we define $\mathsf{gap}(S)$ as:

$$\mathsf{gap}(S) = \sum_{i=1}^{n} (\lfloor \log g_i \rfloor + 1) \tag{2.2}$$

This measure exploits the variation in the gaps between consecutive set elements. That is, the closer the elements, the smallest this measure is. It holds that, $\mathsf{gap}(S) \leq \mathcal{B}(n, u)$, with equality only when $g_i = \frac{u}{n}$ (for $i = 1, \ldots, n$).

2.3.3 The $\texttt{rle}(S)$ Compression Measure

When set elements tend to be clustered into runs of successive elements, a (usually) better way to model its cbv is $C_S[0..u) = \mathbf{0}^{z_1}\mathbf{1}^{\ell_1}\mathbf{0}^{z_2}\mathbf{1}^{\ell_2}\cdots\mathbf{0}^{z_r}\mathbf{1}^{\ell_r}$, where the sequences $\mathcal{Z} = \langle z_1, \ldots, z_r\rangle$ and $\mathcal{O} = \langle \ell_1, \ldots, \ell_r\rangle$ are the lengths of the alternating $\mathbf{0}/\mathbf{1}$-runs in C_S (assume wlog that C_S begins with $\mathbf{0}$ and ends with $\mathbf{1}$). Then we define $\texttt{rle}(S)$ as:

$$\texttt{rle}(S) = \sum_{i=1}^{r} (\lfloor \log (z_i - 1) \rfloor + 1) + \sum_{i=1}^{r} (\lfloor \log (\ell_i - 1) \rfloor + 1) \tag{2.3}$$

Unfortunately $\mathsf{gap}(S)$ and $\texttt{rle}(S)$ are not comparable measures, But we can compare it with $\mathcal{B}(n, u)$. Foschini et al. [25], proved that: if $n < u/2$ then $\texttt{rle}(S) < \mathcal{B}(n, u) + n + \mathrm{O}(1)$.

2.3.4 The $\texttt{trie}(S)$ Compression Measure

The following compression measure can be derived from the $\mathsf{bintrie}(S)$ representation (see Section 2.1.1). Given two bit strings x and y of ℓ bits each, let $x \ominus y$ denote the bit string obtained after removing the longest common prefix among x and y from x. For instance, for $x = \mathtt{0110100}$ and $y = \mathtt{0111011}$, we have $x \ominus y = \mathtt{0100}$. The prefix omission method by Klein and Saphira [26] represents a sorted set S as a binary sequence $\mathcal{T} = \langle x_1; x_2 \ominus x_1; \ldots; x_n \ominus x_{n-1}\rangle$. If we denote $|x_i \ominus x_{i-1}|$ the length of bit string $x_i \ominus x_{i-1}$, then the whole sequence uses:

$$\texttt{trie}(S) = |x_1| + \sum_{i=2}^{n} |x_i \ominus x_{i-1}|. \tag{2.4}$$

It turns out that $\texttt{trie}(S)$ [27] is the number of edges in $\mathsf{bintrie}(S)$. Notice that $\texttt{trie}(S)$ decreases as more trie paths are shared among set elements: consider two integers x and

y, the trie represents their longest common prefix just once (then saving space), and then represents both $x \ominus y$ and $y \ominus x$. Extreme cases are as follows:

1. All set elements form a single run of consecutive elements, which maximizes the number of trie edges shared among set elements, hence minimizing the space usage; and
2. The n elements are uniformly distributed within $[0..u)$ (i.e., the gap between successive elements is $g_i = u/n$), which minimizes the number of trie edges shared among elements, and hence maximizes space usage.

Notice that case 2 is similar to the case that maximizes the $\mathtt{gap}(S)$ measure.

Definition 2.2. *Given a set $S = \{x_1, \ldots, x_n\} \subseteq [0..u)$, let $S + a$, for $a \in [0..u)$, denote a shifted version of S: $S + a = \{(x_1 + a) \bmod u, (x_2 + a) \bmod u, \ldots, (x_n + a) \bmod u\}$.*

The following result of Gupta et al. [27] of $\mathtt{trie}(S)$ measure is relevant this work:

Lemma 2.2 ([27], Section 2)**.** *Given a set $S \subseteq [0..u)$ of n elements, it holds that:*

1. $\mathtt{trie}(S) \leq \min\{2\mathtt{gap}(S), n\log(u/n) + 2n - 2\}$.
2. $\exists a \in [0..u)$, *such that* $\mathtt{trie}(S + a) \leq \mathtt{gap}(S) + 2n - 2$.
3. *For any $a \in [0..u)$ chosen uniformly at random,* $\mathtt{trie}(S + a) \leq \mathtt{gap}(S) + 2n - 2$ *on average over all values of $a \in [0..u)$.*

Chapter 3

Previous Work

3.1 Succinct Set Representations

A sorted set $S \subseteq [0..u)$ can be succinctly represented as follows:

- Okanohara and Sadakane [28] introduced the `SDarrays`, to represent set S using $n \log \frac{u}{n} + 2n + o(n)$ bits of space, supporting rank in $O(1 + \log \frac{u}{n})$ time and select in $O(1)$ time.
- Raman et al. [29] proposed a data structure to represent set S using $\mathcal{B}(n, u) + o(u)$ bits of space, supporting rank and select in $O(1)$ time.

3.2 Elias Codes

We explore next Elias' codes [24] for integer sets.

3.2.1 Elias-γ

To encode an integer x, this method performs the followings steps. Let $P = \lfloor \log_2 x \rfloor$, hence $2^P \leq x < 2^{P+1}$. Then, we encode P in unary, that is, we write P zeros followed by a one that indicates the end of the code of P. Finally, the difference $x - 2^P$ is coded in binary (using P bits). In summary:

1. Encode P in unary; that is, write P **0**s followed by **1**.
2. Encode $x - 2^P$ in binary using P bits.

This coding uses $2\lfloor \log x \rfloor + 1$ bits to encode each element $x \in S$. Then, to encode the set S we need:

$$\sum_{i=1}^{n} 2\lfloor \log x_i \rfloor + 1 = 2\mathtt{gap}(S) - n.$$

For example, to encode $x = 17$ using Elias-γ, we have $P = 4$ as $2^4 \leq 17 < 2^5$. Then, we encode $P = 4$ in unary (four **0**s followed by **1**), so we have **00001**. Next, we encode $17 - 2^4 = 1$ in binary using 4 bits, that is, **0001**. Finally, the complete encoding is as follows:

$$\text{Elias-}\gamma\,(17) = \underbrace{\mathbf{0000}}_{P}\ \mathbf{1}\ \underbrace{\mathbf{0001}}_{x-2^P}$$

3.2.2 Elias-δ

Unlike the previous method, we will encode $P+1$ instead of P, this time using Elias-γ rather than unary. Let $L = \lfloor \log_2 (P+1) \rfloor$, hence $2^L \leq P + 1 < 2^{L+1}$. Using that, Elias-δ carries out the following steps:

1. Write L **0**s followed by one **1**, where $L = \lfloor \log (P+1) \rfloor$.
2. Write $(P+1) - 2^L$ in binary using L bits.

3. Finally, write $x - 2^P$ in binary using P bits.

Note that Elias-δ uses $\lfloor \log x \rfloor + 2\lfloor \log(\lfloor \log x \rfloor + 1) \rfloor + 1$ bits to represent an element x. Then, to encode the set S we need:

$$\sum_{i=1}^{n} \lfloor \log x_i \rfloor + 1 + 2\lfloor \log(\lfloor \log x_i \rfloor + 1) \rfloor = \texttt{gap}(S) + o(\texttt{gap}(S)).$$

For example, to encode $x = 17$ we know that $P = 4$ (remember the previous Elias-γ example), therefore, we must encode $P + 1 = 5$ using Elias-γ. So, we look for an L such that $2^L \leq 5 < 2^{L+1}$, for this case $L = 2$ (i.e., $\lfloor \log(4 + 1) \rfloor = 2$). Then, L is encoded in unary followed by **1** and $5 - 2^2$ in binary. That is, **001** and **01** respectively. Finally, we add $17 - 2^4 = 1$ in binary **0001**. The complete encoding is as follows:

$$\text{Elias-}\delta\,(17) = \underbrace{\mathbf{00}}_{L}\ \mathbf{1}\ \underbrace{\mathbf{01}}_{(P+1)-2^L}\ \underbrace{\mathbf{0001}}_{x-2^P}$$

3.3 Binary Interpolative Coding

Moffat and Stuiver [30] proposed the method Binary Interpolative Coding (therefore IPC). This approach takes advantage of sorted integer sequences, dividing recursively the sequence in two halves, and encoding the central element (i.e., the median). Let us define the following quantities $lo \leq S[1]$ and $hi \geq S[n]$. Given these quantities we can encode the central element $S[m]$ (where $m = \lfloor n/2 \rfloor$) in some more appropriate way knowing that $lo \leq S[m] \leq hi$. In fact, we can represent $S[m] - lo - m$ using exactly $\lceil \log(hi - lo - n + 1) \rceil$ bits. We can subtract m because in the worst case there are m consecutive elements between $S[1]$ and $S[m]$. Then, we can apply this method recursively to encode the halves $S[1..m-1]$ and $S[m+1..n]$. Algorithm 1 shows the pseudocode of this method. This approach has shown to be a very space-efficient alternative in practice. Its main drawback is that the decoding process (which is key when operating on a compressed set) is slow in practice.

Algorithm 1: `Interpolative-Coding`(set $S[1..n], hi, lo$)

```
1 begin
2   if n < 3 then
3     return
4   m ← n/2
5   x ← S[m]
    // Encode x − lo − m using ⌈log(hi − lo − n + 1)⌉ bits
6   encode(x − lo − m, hi − lo − n + 1)
7   Interpolative-Coding(S[1..m − 1], lo, x − 1)
8   Interpolative-Coding(S[m + 1..n], x + 1, hi)
```

3.4 Byte-aligned Codes

One of the most popular methods for compressing integer sets is that of byte-aligned codes, which is popular for its simplicity and decoding speed (which is important for intersections). The main idea is to encode in binary each integer using a variable number of bytes. The simplest method using this approach is `Varint` which we see next.

3.4.1 Varint

Thiel and Heaps in 1972 were the first to describe `Varint` [31] (also called `Vbyte`, `Vint`, `variable-byte`, `var-byte`). This method consist in dividing the binary code of each element into 7-bit fragments , using the least significant 7 bits of a byte to store the fragments. Thus, the fragments are stored in a sequence of bytes. Then, the remaining most significant bit (called the *descriptor*) is used to indicate if the encoding of the element ends or continues in the next byte. A **1** indicates that the encoding continues in the next byte, a **0** indicates the current is the last byte in the encoding.

For example, consider the next set $S = \{1, 15, 511, 131071\}$. For 131071 the binary code is **11111111111111111** , which can be divided into the following 7-bit fragments: ⟨**1111111**, **1111111**, **111**⟩. Note that the last fragment is only 3 bits long, thus we only store those bits in the least significant 3 bits of the last byte. Then, for 131071 we have the next sequence of bytes: ⟨**11111111**, **11111111**, **00000111**⟩, where bits in red are the descriptors. The same

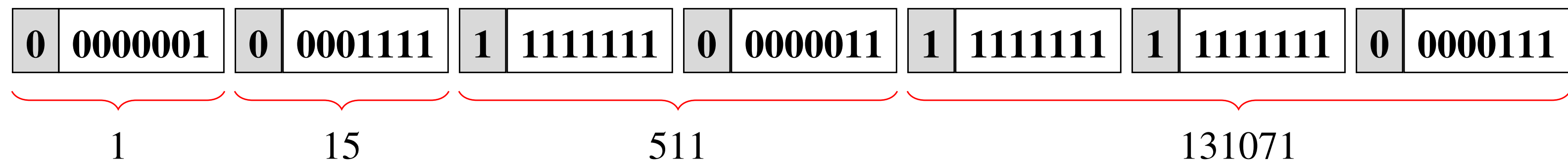


Figure 3.1: `Varint` encoding the set $S = \{1, 15, 511, 131071\}$.

procedure is followed for the remaining elements of S, as shown in the Figure 3.1.

3.4.2 Varint-GB

Although `Varint` in practice is fast at decoding time, it needs many bit mask operations to decode. Dean [32] noticed this and introduced a variant called Group Varint (`VarintGB`). This variant use only 2 bits per integer for descriptors, that is, we only need 2 bits to encode the number of bytes where an integer is encoded. The descriptors of 4 integers are grouped within one byte for fast decoding. Thus, we can store 4 integers using between 5 (4 ints of 1 byte plus 1 byte for descriptors) to 17 bytes (4 ints of 4 bytes plus one byte for descriptors) in total. The next codes are used for encoding the descriptors: **00** for integers of 1 byte, **01** for 2 bytes, **10** for 3 bytes and **11** for 4 bytes. This representation allows to use less bit mask operations to decode as the descriptors are grouped by bytes.

For example, consider the next set $S = \{1, 15, 511, 131071\}$. First, $1 = \mathbf{00000001}$ then, we will encode it in one byte with descriptor **00**. Second, $15 = \mathbf{00001111}$ therefore it can be stored in one byte with descriptor **00**. Third, $511 = \mathbf{111111111}$ then, we can divide the binary code into two bytes $\langle \mathbf{11111111}, \mathbf{00000001} \rangle$ with descriptor **01**. Fourth, $131071 = (\mathbf{11111111111111111})_2$, then we can divide the binary code into three bytes $\langle \mathbf{11111111}, \mathbf{11111111}, \mathbf{00000001} \rangle$ with descriptor **10**. Finally, we group the descriptors of four integers in one byte $\langle \mathbf{00}, \mathbf{00}, \mathbf{01}, \mathbf{10} \rangle$, as shown in Figure 3.2.

3.4.3 Varint-G8UI

Stepanov et al. [33] proposed a `Varint` variant called `Varint-G8UI`, which take advantage of the parallelism of SIMD instructions, because the descriptors are stored and grouped

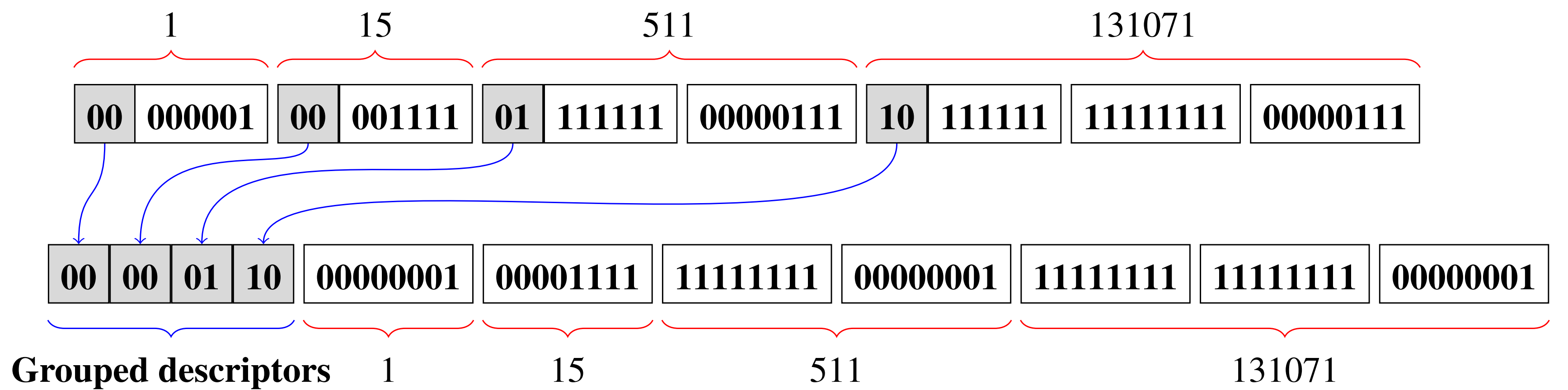


Figure 3.2: `Varint-GB` encoding the set $S = \{1, 15, 511, 131071\}$.

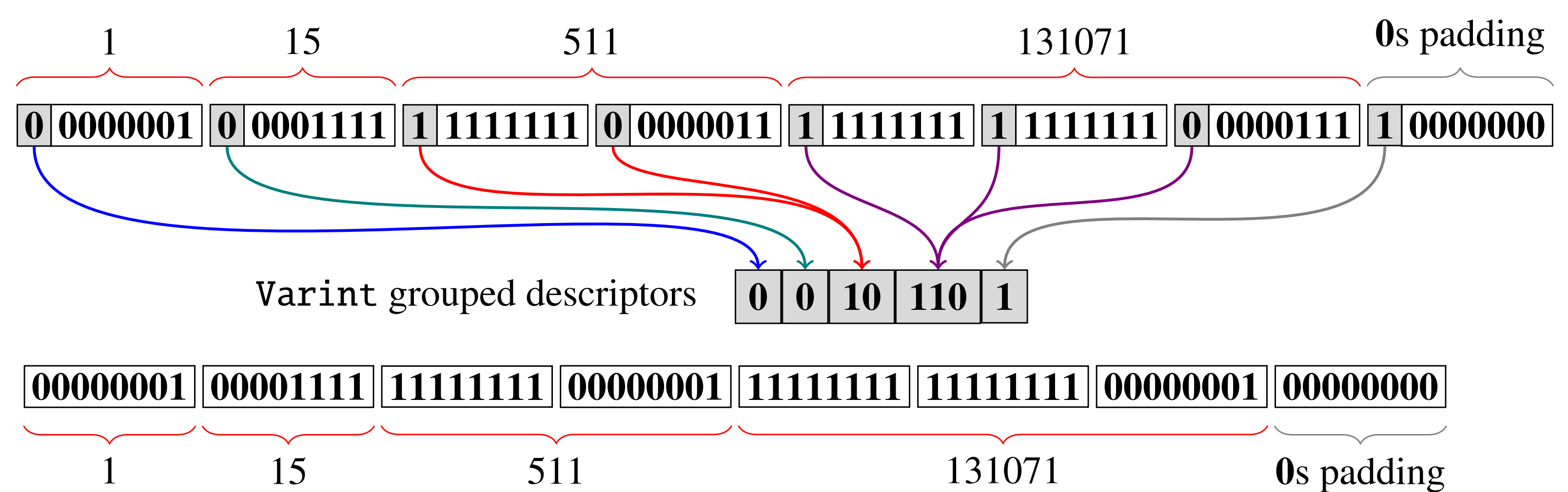


Figure 3.3: `Varint-G8UI` encoding the set $S = \{1, 15, 511, 131071\}$.

separately. This variant encodes the maximum possible number of integers possible in 8 bytes. The descriptors are the same as for `Varint` but now grouped in one byte, to describe the group of 8 bytes. If all 8 bytes are not fully used, the remaining are padded with **0**s and its descriptor with **1**. With this encoding, the number of **0**s in a grouped descriptors will tell us the number of elements encoded in a word.

For example, consider the next set $S = \{1, 15, 511, 131071\}$. As we saw previously the `Varint` encoding for for 131071 is ⟨**11111111**, **11111111**, **00000111**⟩ (see Figure 3.1). Then, we group its descriptor as **110**. And do the same with the remaining elements, obtaining the following grouped descriptors: ⟨**0**, **0**, **10**, **110**, **1**⟩, where the last descriptor is **1** as the last byte is not used. For this representation, we use 8-bit fragments, unlike `Varint`. Finally, the last byte (out of a total of 8) is padded with **0**s because it is not used in the machine word, as shown in Figure 3.3.

3.5 Word-aligned Codes or Simple Family

Note that `Varint` in the best case can encode 4 integers in a 32-bit word or 8 integers in a 64-bit word. However, if the elements to be encoded are small, a considerable amount of space is wasted because at least one byte is needed to encode an integer. To solve this, the idea is to encode a consecutive group of integers within a machine word (e.g., in a word of 16, 32, or 64 bits). This approach is known as word-aligned codes.

3.5.1 Simple 9

The first method that follows the word-aligned approach is `Simple9` (`S9`, for short) proposed by Anh and Moffat [34]. `S9` take several consecutive integers (the maximum possible), and try to fit them within a 32-bit machine word. The integers are encoding within of chunks of fixed equal-size. The size of the chunk is defined as the number of bits needed to store the largest integer in the group. In each word, 4 bits are used as a header to recognize the size and number of chunks in the word, that is, the number of integers and bits to represent each one in the word. Hence, the word consists of two main parts: the header of 4 bits and the data part of 28 bits to store the integers.

There are 9 possible ways to divide 28 bits into chunks of equal size. The meaning of the cases is as follows:

(C1) Header **0000**, 1 chunk of 28 bits.

(C2) Header **0001**, 2 chunks of 14 bits.

(C3) Header **0010**, 3 chunks of 9 bits plus 1 unused bit.

(C4) Header **0011**, 4 chunks of 7 bits.

(C5) Header **0100**, 5 chunks of 5 bits plus 3 unused bits.

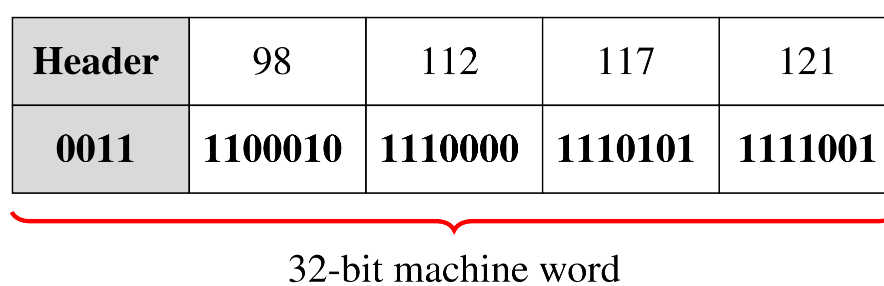
(C6) Header **0101**, 7 chunks of 4 bits.

(C7) Header **0110**, 9 chunks of 3 bits plus 1 unused bit.

(C8) Header **0111**, 14 chunks of 2 bits.

(C9) Header **1000**, 28 chunks of 1 bit.

For example, consider the set $S = \{98, 112, 117, 121\}$. Note that for this sequence, $\lfloor \log_2(121) \rfloor + 1 = 7$ bits are the minimum number bits to represent 121 (in binary). Therefore, we need to encode each element in chunks of 7 bits, that is, C4 of `S9` with header **0011**. As shown in Figure 3.4.



Header	98	112	117	121
0011	**1100010**	**1110000**	**1110101**	**1111001**

Figure 3.4: `S9` encoding the set $S = \{98, 112, 117, 121\}$, which represents 4 integers in 7-bit chunks within a 32-bit machine word. The case is C4 with header **0011**.

3.5.2 Simple 16

Note that `S9` wastes space in 3 of its cases (C3, C5 and C7). Mainly this occurs because `S9` identifies only 9 cases versus the 16 cases can be expressed using 4 bits in the header, and these 9 cases only consider chunks with the same size. Keeping this in mind, Zhang et al. [35] proposed `Simple16` (`S16`, for short) with the aim of saving space introducing further cases to the original 9. For example, for the case of 5 chunks of 5 bits and 3 unused bits (C5 of `S9`), can be replaced by the following two cases:

1. Divide the 28 bits into 3 chunks of 6 bits followed by 2 chunks of 5 bits.
2. Divide the 28 bits into 2 chunks of 5 bits followed by 3 chunks of 6 bits.

With this idea, they identify the cases that replace C3, C5 and C7 of `S9` such that no space is wasted in the word, reaching a total of 16 cases (hence the name, `Simple16`). Other variants have been proposed to `S9` and `S16`, such as `S18` [36].

3.5.3 Simple 8b

Anh and Moffat [37] proposed a method that stores groups of integers in 64-bit machine words. They called this method `Simple8b` (`8b` comes from 8 bytes) or `S8b` for short. Just like `S9`, `S8b` uses a 4-bit header to identify the cases, leaving 60 bits to store the integers into fixed equal-size chunks. The division of 60 bits into equal-size chunks can be done in 14 different ways, where only two cases space is wasted. Therefore, `S8b` defines 14 cases to divide 60 bits into fixed equal-size chunks, plus 2 additional cases to encode runs of 120 or 240 consecutive 1s.

3.6 Binary Packing

This approach consist in partitioning the input set by cardinality, that is, the set is divided in groups of integers of fixed cardinality. For example, we can partitioning a 1 million set of integers in groups of 128 integers, called *blocks*. This idea is similar to *Frame of Reference* proposed by Goldstein et al. [38]. The method encodes the range of values to be stored in blocks, and then the elements are written in blocks in reference to the range. For example, if the values to store in a block are in the range $[1000, 1255]$, they can be stored using 8 bits, e.g., if the value to encode is 1128, it can be represented as $1128 - 1000 = 1128$ using just 8 bits. In summary, this method encodes the range $[l, r]$ (the interval containing the values) and subtracts l to each element, representing each element with $h = \lceil \log (l - r + 1) \rceil$ bits, finally h is stored as a block header within a byte. In general, this method is combined with others (e.g., Elias codes or `IPC`) to improve the space usage and decoding time.

Generally binary packing encodes the gaps of the corresponding set, however, for decoding we need to carry out several operations in order to obtain the actual set elements. Lemire et al. [3] noticed this and proposed a method based on binary packing, which is able to encode and decode 4 integers in parallel using SIMD instructions. This method divides the sequence into blocks of 128 integers, which are then grouped into meta blocks of 16 blocks each , that is, we have metablocks that contain 2048 integers.

3.7 Patched Coding

Note that binary packing does not compress efficiently when large elements are present in the blocks, even though the majority of elements are small. For example, the sequence $\langle 1000, 1003, 1064, 1164, 1255 \rangle$ can be stored using 8 bits per integer, but if there is one element bigger than 1255 (e.g., 65535) we needed to use more bits for each integer of the sequence (e.g., 16 bits if 65535 lies within the block in the above example) disregarding the fact that most of the elements lie in the range $[1000, 1255]$.

To alleviate this problem, Zukowski et al. [12] introduce the idea of *patching* with a method called `PFOR` or `PForDelta` (`PFD` for short) This method consists of defining a value h of bits, which are sufficient to encode a large percentage of integers in each block (e.g., 80%, 90%, etc). Those elements that cannot be represented using h bits (i.e., values $> 2^h$) are encoded in a separate location (out the blocks). We call these values *exceptions*. `PFD` is faster decoding when the block size is multiple of 32 (or 64, depending on the underlying architecture). So, the decompression process involves extracting groups of 32 h-bit, and finally patching the result by decoding the smaller number of exceptions.

Later, Yan et al. [39] introduced a variant called Optimized `PForDelta` (`OptPFD` for short). In this variant, the choice of h to every block is modeled as an optimization problem, in this way obtaining a better trade-off between space and decoding speed.

3.8 Elias-Fano

Elias [40] and Fano [41] independently introduced a method to represent increasing monotone sequences. This encoding divide the binary encoding of each element $x_i \in S$ in two parts:

1. **Lower bits**: we will call *lower bits* the least $l = \lfloor \log(u/n) \rfloor$ significant bits of the binary representation of x_i. For each $x_i \in S$, for $i = 1, \ldots, n$, we explicitly store the l lower bits in an array that we will call L.

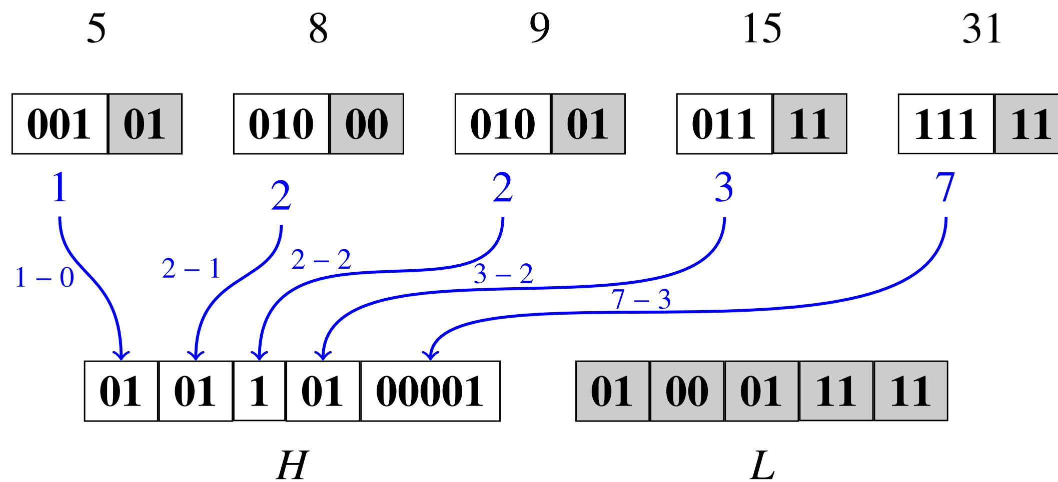


Figure 3.5: Elias-Fano encoding the sequence $S = \langle 5, 8, 9, 15, 31\rangle$, bit array H store upper bits and L store $l = \lfloor \log(32/5) \rfloor = 2$ lower bits.

2. **Upper bits**: we will call *upper bits* $\lceil \log u \rceil - l$ most-significant bits of the binary representation of x_i. These upper bits are coded representing in unary the difference $\lfloor x_i/2^l \rfloor - \lfloor x_{i-1}/2^l \rfloor$, for $i = 1, \ldots, n$, assuming $x_0 = 0$. We store this unary sequence in bit array H.

The Elias-Fano scheme consists of arrays L and H. Notice that this representation in total uses at most $n \lceil \log(u/n) \rceil + 2n$ bits of space.

For example, consider the following sequence: $S = \langle 5, 8, 9, 15, 31\rangle$ over the universe $U = [0, 32)$. In binary the sequence is: $\langle \mathbf{00101}, \mathbf{01000}, \mathbf{01001}, \mathbf{01111}, \mathbf{11111}\rangle$, as shown in Figure 3.5. Then, $l = \lfloor \log(32/5) \rfloor = 2$, so we will store explicitly the 2 least-significant bits in L of each $x_i \in S$ (gray bits in Figure 3.5). For the remaining 3 bits, we will encode their gaps in unary (as explained before) and they will be stored in H (see the blue arrows of Figure 3.5).

Later, Vigna [42] using Elias-Fano coding proposed a data structure to represent inverted indexes, called it `quasi-succint indexes`. This representation allows access to any element of the set in $O(1 + \log u/n)$ time, provided H is represented with a bit vector data structure that supports the rank and select operations in $O(1)$ time.

3.8.1 Partitioned Elias-Fano

Notice that the Elias-Fano representation does not take advantage of extreme cases that may occur. For example, the presence of huge runs of consecutive elements in the set. For these cases we expect to use less space, but $\lceil \log(u/n) \rceil + 2$ bits are still needed to represent each element. Ottaviano and Venturini [15] noticed this, and proposed two variants called `Partitioned Elias-Fano Uniform` and ϵ-`optimal`. Both variants partition the input set S into B blocks (just like binary packing). These variants differ in how to choose the size and number of blocks, as we see next.

Partitioned Elias-Fano Uniform

`Partitioned Elias-Fano Uniform` (`PEF Unif` for short) consists in dividing the set S in $B = n/b$ blocks of b elements. Divide the set by cardinality into blocks of fixed size. The size of the universe of each block is given by $S[b \cdot j] - S[b \cdot (j-1)]$, for $j \in [1..B]$. Finally, each block can be represented choosing one of the following ways:

1. Encode the block using the *cbv* representation (e.g., using a bit vector), whenever $b > u_j/4$, where $j \in [1..B]$. That is, when the Elias-Fano representation use more space than u_j bits.
2. If a run of b consecutive elements is present, we only need to indicate this and nothing else needs to be stored.
3. Otherwise, we use the Elias-Fano coding over the universe u_j, using $\lceil \log(u_j/b) \rceil + 2$ bits per element in the j-th block.

Optimal Partitioned Elias-Fano

Notice that by dividing the set into fixed-size blocks is highly probable to produce sub-optimal compression, as it is hard to align dense groups into fixed-size blocks. Therefore, a

better alternative is to partition S into blocks of variable size to minimize the space. The optimal partition of set S can be calculated in time $\Theta(n^2)$, resolving a variant of the recurrence introduced in [43] using dynamic programming. However, that is very expensive for large sets (e.g., a few thousands of elements). As a result, they introduce an algorithm that approximates the optimal partition. This algorithm calculate the partition in time $O(n \log_{1+\epsilon}(1/\epsilon))$, where $\epsilon \in [0, 1]$, and the partition found uses $1 + \epsilon$ times the space of the optimal partition. This approach is known as partitioned Elias-Fano ϵ-optimal (`PEF opt`, for short).

3.9 Roaring Bitmaps

Lemire et al. [13] proposed a method called `Roaring`, which partitions the universe $[0..2^{32})$ into chunks of ranges of 2^{16} integers, that is, $\langle [0..2^{16}), [2^{16}..2^{17}), \ldots, [2^{31}..2^{32}) \rangle$. For each value in the set, its least significant 16 bits are stored in its corresponding partition, using one the following ways to represent a partition:

1. Encode the partition using the *cbv* representation, that is, a bit vector of 2^{16} bits (elements).
2. Store the elements of a partition in an array of 4096 sorted integers of 16 bits.
3. Store the values $\langle s, l \rangle$, indicating that all elements in interval $[s, s + l]$ are present. That is, represent efficiently runs present in a chunk.

At high level, `Roaring` stores a list of 16-bit numbers (most significant 16 bits of elements), each of which is coupled with a reference to a partition holding the 16 least significant bits of the elements that share the same prefix. The authors choose dynamically the partition type to minimize the space usage.

3.10 Recursive Universe Partitioning

The method *Recursive Universe Partitioning* (RUP for short) was proposed by Pibiri [14]. This method consists in partitioning the universe $U = [0..u)$ of set S. This partitioning is done recursively to represent S logically as a tree of height 3. These partitions initially divide the universe in blocks of size b_1, which are classified into four types depending on their cardinality: *full*, *dense*, *sparse* and *empty*. The *full* blocks are those that contain exactly b_1 elements, and the *empty* blocks are those that contain no element. The dense blocks are those that contain at least $b_1/2$ elements and are encoded using a *cbv*, so this type of block is represented using at most two bits per element. Then, at the root of tree, *full* and *empty* blocks do not have children, that is, we only need to indicate their types. For *dense* blocks, a child is stored with *cbv* encoding (i.e., a bit vector of b_1 bits). *Sparse* blocks will be partitioned into blocks again, this time of size $b_2 < b_1$, and only distinguish between two types: *dense* and *sparse*. The *dense* blocks are stored using a *cbv* of b_2 bits, and the *sparse* blocks are stored in array of $\lceil \log b_2 \rceil$-bit integers. Headers are used to store the types for each block, so we will have H_1 headers for the first partitioning (i.e, for the partitions of universe U of blocks of size b_1) and H_2 for the second (i.e., for the partitions of sparse blocks of size b_2). Note that at the end of the method described above, a tree of height 3 will be obtained, such that, at the root we have the H_1, at the second level the encoding for *dense* blocks of size b_1 and the headers H_2 of *sparse* blocks, and at the third level we have the encoding of blocks of size b_2. Finally the author chooses the values $b_1 = 2^{16}$ and $b_2 = 2^8$, as $u < 2^{32}$ was assumed.

3.11 Adaptive Set Intersection Algorithms

An adaptive algorithm is one whose running time is a function not only of the instance size (as usual), but also of a difficulty measure of the instance. In this way, “easy” instances are solved faster than “difficult” ones, allowing for a more refined analysis than typical worst-case approaches. Next, we review the adaptive intersection algorithms that are of interest for this work.

3.11.1 Barbay and Kenyon

Barbay and Kenyon [11] proposed an adaptive algorithm for the set intersection problem. This algorithm performs a complete exponential search (or also called galloping search) at once on each set, when searching for the current candidate x (possible element of the intersection). That is, the algorithm chooses a candidate x (initially $S_1[1]$) and look for that element using an exponential (or galloping) search in all sets of the query Q. If x is found in the k sets, it is added to the intersection, otherwise another candidate is chosen as the successor of x in the current set (i.e., choose the new candidate in the set where the search failed). The algorithm ends when any of the sets of Q is consumed (i.e., we cannot pick a new candidate in the current set). The pseudocode of this algorithm is provided in the Algorithm 2. Barbay and Kenyon [11, 10] proved that the running time of this algorithm is given by:

$$O\left(\delta \sum_{i \in Q} \log \frac{n_i}{\delta}\right),$$

where δ is the *alternation* measure that to be introduced next.

Alternation Measure

The *alternation measure* defined by Barbay and Kenyon [10] is a simple and intuitive way to measure the difficulty of the instance to intersect. Somehow, any intersection algorithm must certify (or give a proof) that the output is correct, that is, must certify the following:

- That any element in $\mathcal{I}(Q)$ belongs to the k sets $S_{i_1}, \ldots, S_{i_k}$, for $i_1, \ldots, i_k \in Q$, and
- no element in the intersection has been left out of the result.

Barbay and Kenyon defined the *partition certificate* concept as a proof of the two previous points, as defined below.

Definition 3.1. *Given a query* $Q = \{i_1, \ldots, i_k\} \subseteq [1..N]$*, a* partition certificate *is a partition of the universe* $[0..u)$ *into a set of of intervals* $\mathcal{P}_{\text{BK}}(Q) = \{I_1, I_2, \ldots, I_p\}$*, such that:*

Algorithm 2: `BK-Intersection`(sets $S_1, \ldots, S_k$)

```
   Result: The set intersection I = S_1 ∩ ··· ∩ S_k
 1 begin
 2   I ← ∅
 3   x ← S_1[1]
 4   i ← 2
 5   occ ← 1
 6   while x ≠ ∞ do
 7     pos ← GallopingSearch(S_i, x)
 8     y ← S_i[pos]
 9     if x = y then
10       occ ← occ + 1
11     if occ = k ∨ x ≠ y then
12       if occ = k then
13         I ← I ∪ {x}
14       x ← successor(S_i, x)
15       occ ← 1
16     i ← (i + 1) mod k
17   return I
```

1. $\forall x \in \mathcal{I}(Q)$, $[x..x] \in \mathcal{P}_{\text{BK}}(Q)$;

2. $\forall x \notin \mathcal{I}(Q)$, $\exists I_j \in \mathcal{P}_{\text{BK}}, x \in I_j \ \wedge \ \exists q \in Q, S_q \bigcap I_k = \emptyset$.

In the above definition, the first intervals certify the elements reported in the intersection $\mathcal{I}(Q)$. Intervals from item 2, on the other hand, certify those elements that do not belong to $\mathcal{I}$. For a given query Q, several valid partition certificates could be given. However we are interested in the smallest partition certificate of Q, as it takes the least time to be computed.

Definition 3.2. *For a given query instance $Q = \{i_1, \ldots, i_k\} \subseteq [1..N]$, let δ denote the size of the smallest partition certificate of Q.*

Measure δ is known as the *alternation* of the query instance [10], and it measures its difficulty (henceforth, *alternation measure*). Notice $|\mathcal{I}(Q)| \leq \delta$ holds. Figure 3.6 shows the smallest partition certificate (of size $\delta = 8$) for sets S_1 and S_2 of our running example. Barbay and Kenyon [11, 10] proved a lower bound of $\Omega(\delta \sum_{i \in Q} \log(n_i/\delta))$ comparisons for the set

intersection problem. Then, they proved that their intersection algorithm is optimal, running in $O(\delta \sum_{i \in Q} \log(n_i/\delta))$ time.

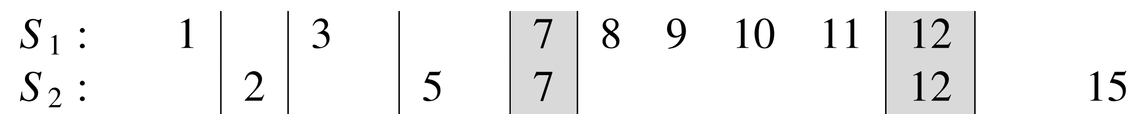


Figure 3.6: Vertical lines show the smallest partition certificate $\mathcal{P} = \{[0..1], [2..2], [3..4], [5..6], [7..7], [8..11], [12..12], [13..15]\}$ of size $\delta = 8$ of the universe $[0..16)$ for the intersection of sets $S_1 = \{1, 3, 7, 8, 9, 10, 11, 12\}$ and $S_2 = \{2, 5, 7, 12, 15\}$. Gray intervals are the intersection elements, and the uncolored are the ones left out.

3.11.2 Demaine, López-Ortiz and Munro Algorithm

One of the first adaptive intersection algorithms proposed was the one developed by Demaine, López-Ortiz and Munro [9] (henceforth DLM algorithm). Algorithm 3 shows the pseudocode. The DLM algorithm is also based on exponential search, that is, search an element in a set from left to right doubling the size of the interval to be searched. Initially the algorithm chooses a candidate element x as the first (smallest) element of the first set of the query (line 3). Then, we repeat the following routine through the sets in a cyclical way: (1) an exponential search step is performed, only doubles once the size of the interval to be searched in current set (line 8); (2) if in the current interval the upper bound is less than x, perform a binary search (lines 9-10); (3) then, increment the counter of occurrences if x is in the current set, and if the counter reaches k we report x as part of the intersection (lines 11-15); (4) finally, if x is not in the set, we choose a new candidate and restart the occurrence count (lines 16-17). The algorithm ends when we can not choose a new candidate (line 16 takes ∞ value).

The running time of this algorithm is given by:

$$O\left(k\delta \log \frac{n}{\delta}\right),$$

where $n = \sum_{i \in Q} n_i$ and δ is the *alternation measure* of Barbay and Kenyon.

Algorithm 3: `DLM-Intersection`(sets $S_1, \ldots, S_k$)

```
    Result: The set intersection I = S_1 ∩ ··· ∩ S_k
 1  begin
 2      I ← ∅
 3      x ← S_1[1]
 4      i ← 2
 5      occ ← 1
 6      pos ← {pos_1, ..., pos_k} // All initialized in 1
 7      while x ≠ ∞ do
 8          pos_i ← 2 · pos_i // One step of exponential search in S_i
 9          if x < S_i[pos_i] then
                // Search x in S_i and return its position
10              pos_i ← BinarySearch(S_i[pos_i..n_i], x)
11              if S_i[pos_i] = x then
12                  occ ← occ + 1
13              if occ = k ∨ S_i[pos_i] ≠ x then
14                  if occ = k then
15                      I ← I ∪ {x}
16                  x ← successor(S_i, x)
17                  occ ← 1
18          i ← (i + 1) mod k
19      return I
```

3.11.3 Trabb-Pardo's Algorithm

Now we revisit an old divide-and-conquer intersection algorithm by Trabb-Pardo [21] in 1978. Algorithm 4 shows the pseudocode. Given a query instance $Q = \{i_1, \ldots, i_k\} \subseteq [1..N]$, the algorithm must be invoked as `TP-Intersection`$(S_{i_1}, \ldots, S_{i_k}, [0..u))$. The main idea is to divide the universe into two halves, to then split each set according to this universe division. The *Divide* steps (lines 10 and 11) can be implemented using binary search. At the first level of recursion, the most-significant bit of every element in sets $S_{i,l}$ is `0`, as they belong to the left half of the universe. Similarly, for $S_{i,r}$ the most-significant bit is `1` as all elements belong to the right half. At each node of the recursion tree, the current universe is divided into two halves, to then recurse on the sets split accordingly.

As sets are known in advance to queries and set divisions carried out by Algorithm 4 are

Algorithm 4: `TP-Intersection`(sets $S_1, \ldots, S_k$; universe $[L..R)$)

```
   Result: The set intersection S_1 ∩ ··· ∩ S_k
1  begin
       // Base cases
2      for i ← 1 to k do
3          if S_i = ∅ then
4              return ∅
5      if L = R then
6          return {L}        // Universe of size 1, all sets are the same
           singleton
7      else
           // Divide
8          M ← ⌊(R + L)/2⌋
9          for i ← 1 to k do
10             S_{i,l} ← {x ∈ S_i | x ∈ [L..M)}
11             S_{i,r} ← {x ∈ S_i | x ∈ [M..R)}
           // Conquer
12         R_1 ← TP-Intersection(S_{1,l}, ..., S_{k,l}, [L..M))
13         R_2 ← TP-Intersection(S_{1,r}, ..., S_{k,r}, [M..R))
           // Combine
14         return R_1 ∪ R_2                            // Disjoint set union
```

query independent (because just depend on the universe), the Divide step of Algorithm 4 can be implemented efficiently by using a suitable set representation that not only stores the set values, but also precomputes the set divisions carried out recursively by the algorithm. Trabb-Pardo proposes to represent each $S_i \in \mathcal{S}$ using bintrie(S_i), mimicking the way set S_i is recursively divided by Algorithm 4. The left child of the root represents all elements whose most-significant bit is 0, i.e., elements in set $S_{i,l}$ of Algorithm 4 (line 10) in the first level of recursion; similarly for $S_{i,r}$, containing all elements in S_i whose most-significant bit is 1. To simulate the recursive execution of Algorithm 4 on the binary tries, one must carry out a DFS traversal in synchronization on all tries involved in the query, following the same path in all of them and stopping (and backtracking if needed) as soon as we reach a dead end in one of the tries (which correspond to dotted lines in Figure 2.2), or we reach a leaf node in all the tries (in which case we have found an element belonging to the intersection). In this way, (1) we stop as soon as we detect a universe interval that does not have any element in the intersection, and (2) we find the relevant elements when we arrive at the leaves.

3.12 How to Intersect Compressed Sets?

Generally, for the intersection to be computed quickly, the integer-set compression approaches seen in previous sections must be fast at decoding time. This is important because any intersection algorithm relies on the successor(S, x) operator (also non-adaptive algorithms), as we saw in Section 3.11. Computing successor(S, x) over compressed sets is expensive, as we must decode a big group of elements to access (in the worst case) a single element. For example, consider we have a set encoded using binary packing. To compute successor(S_i, x) we must find the block that contains the possible successor and decode it. Thus, we have to pay linear time (or logarithmic if we use binary search) to search the block and also a time that is linear on the block size. The time to decode a block can be expensive depending on coding approach (e.g., IPC is slow to decode).

Chapter 4

Compressed Intersectable Sets using Binary Tries

In this chapter we introduce the concept of *trie intersection certificates*, and prove that Trabb-Pardo's algorithm is actually an adaptive algorithm. Also, we devise a space-efficient representation of $\mathsf{bintrie}(S)$, for a set $S = \{x_1, \ldots, x_n\} \subseteq [0..u)$ of n elements such that $0 \le x_1 < \cdots < x_n < u$. This representation will also allow for efficient intersections, supporting Trabb-Pardo's [21] algorithm.

4.1 Trie Intersection Certificates

The concept of *trie intersection certificates*, denoted $\mathsf{cert}(Q)$, as an alternative to existing certificates [9, 10]. Figure 4.1 shows a possible $\mathsf{cert}(Q)$ for the intersection of S_1 and S_2 from Figure 2.2. Let $\mathtt{path}(v)$ denote the binary string labelling the root-to-v path, and $\mathtt{depth}(v) = |\mathtt{path}(v)|$. For a query Q, a binary trie $\mathsf{cert}(Q)$ is a trie partition certificate if:

1. For any internal node v of $\mathsf{cert}(Q)$ such that $\mathtt{path}(v) = b$, there exists an internal node v_i with $\mathtt{path}(v_i) = b$ in every $\mathsf{bintrie}(S_i)$, $i \in Q$;
2. For any void external node v with $\mathtt{depth}(v) = d \le \ell = \lceil \log u \rceil$ and $\mathtt{path}(v) = b$,

there exists at least a set S_i ($i \in Q$) such that there is no node v_i with $\mathtt{path}(v_i) = b$ in $\mathsf{bintrie}(S_i)$. So, the universe interval $[\mathsf{dec}(b \cdot \mathbf{0}^{\ell-d})..\mathsf{dec}(b \cdot \mathbf{1}^{\ell-d})]$ has no element in the intersection, where $\mathsf{dec}(x)$ denotes the decimal representation of a binary string x. We call them *fail nodes*, shown as '×' in Figure 4.1;

3. And for any valid external node v with $\mathtt{depth}(v) = \ell$ corresponding to $\mathtt{path}(v) = b$, there exists a valid external node v_i with $\mathtt{path}(v_i) = b$ in every $\mathsf{bintrie}(S_i)$, $i \in Q$. We call them *success nodes* as they correspond to elements in $\mathcal{I}(Q)$.

Figure 4.1: Trie certificate for the intersection $\{1, 3, 7, 8, 9, 10, 11, 12\} \cap \{2, 5, 7, 12, 15\}$. This trie shows the nodes that must be checked to determine that the result is, in this case, $\{7, 12\}$. This corresponds to the intersection of the binary tries representing these sets.

Notice that $\mathsf{cert}(Q)$ is the trie obtained by intersecting $\mathsf{bintrie}(S_i)$, for all $i \in Q$, and that it is the smallest trie that allows us to prove the correctness of the intersection. For instance, Figure 4.1 shows $\mathsf{cert}(Q)$ for a given query. Interestingly, the recursion tree of Trabb-Pardo's algorithm (Algorithm 4) is exactly $\mathsf{cert}(Q)$, as the algorithm stops as soon as one arrives at a fail node. The external nodes of $\mathsf{cert}(Q)$ cover the universe $[0, u)$ with intervals, similar to Barbay and Kenyon partition certificates, as stated the following definition.

Definition 4.1. *Given a query* $Q = \{i_1, \ldots, i_k\} \subseteq [1..N]$, *its* trie partition certificate *is a partition of the universe* $[0..u)$ *into a set of intervals, we say that* $\mathsf{cert}(Q)$ *induces the following partition of* $[0..u)$ *into a set of intervals that we call* trie partition certificate*:*

$$\mathcal{P}_{\mathtt{AC}}(Q) = \bigcup_{l \in \mathcal{E}(\mathsf{cert}(Q))} \left\{[\mathsf{dec}(\mathtt{path}(l) \cdot \mathbf{0}^{\ell-\mathtt{depth}(l)})..\mathsf{dec}(\mathtt{path}(l) \cdot \mathbf{1}^{\ell-\mathtt{depth}(l)})]\right\},$$

where $\mathcal{E}(\mathsf{cert}(Q))$ *denotes the set of external nodes of* $\mathsf{cert}(Q)$.

For instance, the trie certificate of Figure 4.1 induces the following trie partition certificate of the universe [0..16):

$$\{[0..1], [2..2], [3..3], [4..5], [6..6], [7..7], [8..11], [12..12], [13..13], [14..15]\}.$$

Since $\mathsf{cert}(Q)$ is the recursion tree of Algorithm 4, its running time is $O(k|\mathsf{cert}(Q)|)$. As in the worst-case one must traverse completely all tries $\mathsf{bintrie}(S_i)$, $i \in Q$, we have:

$$k|\mathsf{cert}(Q)| \leq \sum_{i\in Q} \mathtt{trie}(S_i) \leq \sum_{i\in Q} n_i \log \frac{u}{n_i} + 2n_i - 2,$$

where the last bound is from Lemma 2.2 (1). Next we prove an adaptive bound for $k|\mathsf{cert}(Q)|$.

Theorem 4.1. *Given a query instance* $Q = \{i_1, \ldots, i_k\} \subseteq [1..N]$ *with alternation measure* δ *and over sets with universe* $[0..u)$*, algorithm* `TP-Intersection` *computes* $\mathcal{I}(Q) = \cap_{i\in Q} S_i$ *in time* $O(k\delta \log(u/\delta))$.

Proof. Consider a smallest partition certificate $\mathcal{P}_{BK}(Q) = \{I_1, \ldots, I_\delta\}$ of universe $[0..u)$, such that $|I_i| = L_i$ for $i = 1, \ldots, \delta$. Let us think now of the worst-case smallest $\mathsf{cert}(Q)$ we could have, by covering the δ intervals in $\mathcal{P}_{BK}(Q)$ with as many external nodes of $\mathsf{cert}(Q)$ as possible. For any $I_j \in \mathcal{P}_{BK}(Q)$ formed by elements not in $\mathcal{I}(Q)$, there exists a set of external fail nodes in $\mathsf{cert}(Q)$ that cover I_j. This is because when traversing the tries $\mathsf{bintrie}(S_i)$ in coordination, $i \in Q$, the algorithm stops as long as one gets into one of the cover nodes of I_j, since it does not belong to at least one of the tries. According to Lemma 2.1 (1), a contiguous range of L leaves (corresponding to the values in I_j) can be covered with up to $O(\log L)$ nodes. Thus, in the worst-case, $\mathsf{cert}(Q)$ has $O(\sum_{i=1}^{\delta} \log L_i)$ external nodes that overall cover $[0..u)$. Now, recall that the external nodes of $\mathsf{cert}(Q)$ cover the contiguous range of leaves corresponding to $[0..u)$. Hence, according to Lemma 2.1 (3), these external nodes have $O(\sum_{i=1}^{\delta} \log L_i + \log u)$ ancestors, so overall $\mathsf{cert}(Q)$ has $O(\sum_{i=1}^{\delta} \log L_i + \log u)$ nodes. The sum is maximized when $L_i = u/\delta$, for all $1 \leq i \leq \delta$, hence $\mathsf{cert}(Q)$ has $O(\delta \log(u/\delta))$ nodes. The result follows from the fact that for each node in $\mathsf{cert}(Q)$ the algorithm runs in time $O(k)$. □

4.2 Compressed Representation of Binary Tries

We represent $\mathsf{bintrie}(S)$ level-wise, similar to succinct representation of binary trees of Jacobson [44]. Let $B_1[1..2l_1], \ldots, B_\ell[1..2l_\ell]$ be bit vectors such that B_i represents the l_i nodes at level i of $\mathsf{bintrie}(S)$ ($1 \leq i \leq \ell$), from left to right. Each node is encoded using 2 bits, indicating the presence (using bit **1**) or absence (bit **0**) of the left and right children, respectively. In this way, the feasible codewords for trie nodes are **01**, **10**, and **11**, whereas **00** is not a valid codeword. The codewords of all nodes at level $i \geq 1$ in the trie are concatenated from left to right to form B_i. The j-th node at level i (from left to right) is stored at positions $2j - 1$ and $2j$. We say that $2j - 1$ is the position of such node in B_i bit vector.

Let p be the position in B_i corresponding to a node v at level i of $\mathsf{bintrie}(S)$. As the nodes are stored level-wise and from left to right, the number of **1**s before position p in B_i equals the number of nodes in level $i + 1$ that are before the child(ren) of node v. So, $2 \times B_i.\mathsf{rank_1}(p - 1) + 1$ yields the position of B_{i+1} where the first child of node v. Figure 4.2 illustrates our representation.

4.2.1 Space Analysis

The total number of **1**s in the bit vectors of our representation equals the number of edges in the trie. That is, there are $\mathtt{trie}(S)$ **1**s. Besides, the trie has $\mathtt{trie}(S) + 1$ internal nodes and leaves: n of them are leaves, so $\mathtt{trie}(S) - n + 1$ are internal. In our representation we only need to represent the internal trie nodes. As we encode each node using 2 bits, the total space usage for $B_1, \ldots, B_\ell$ is $2(\mathtt{trie}(S) - n + 1)$ bits. On top of them we use Clark's data structure [45] to support rank in O(1) time, adding $o(\mathtt{trie}(S))$ extra bits overall. Then, in total the data structure uses:

$$2(\mathtt{trie}(S) - n + 1) + o(\mathtt{trie}(S))$$

bits of space.

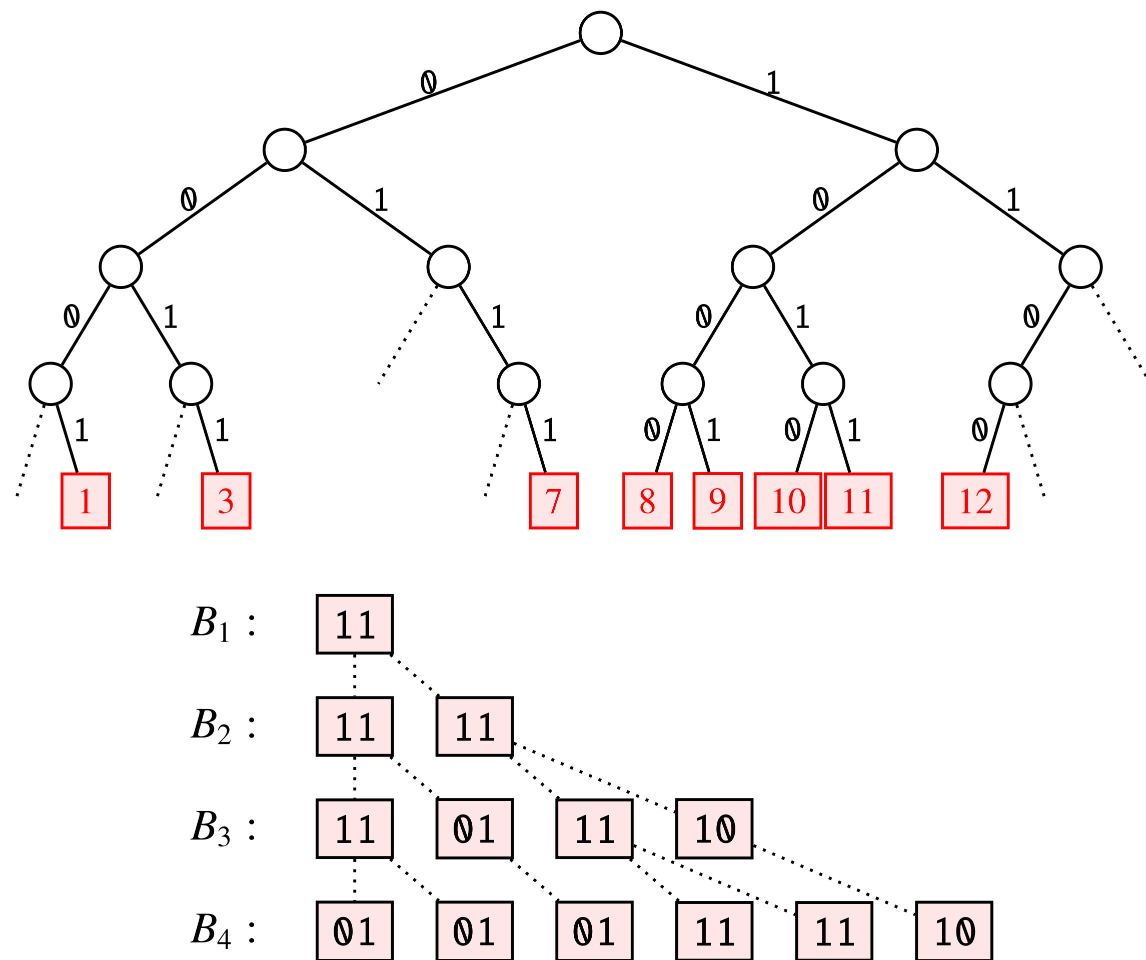


Figure 4.2: Above bintrie(S) for $S = \{1, 3, 7, 8, 9, 10, 11, 12\}$. Below level-wise bit vector representation of bintrie(S), dotted lines are implicit, as they are computed using operation $\mathsf{rank_1}$ on the bit vectors.

4.3 Adaptive and Compressed Intersection

Given a query $Q = \{i_1, \ldots, i_k\} \subseteq [1..N]$, to compute the intersection we traverse bintrie(S_{i_1}), $\ldots$, bintrie(S_{i_k}) using a recursive DFS traversal as in Trabb-Pardo's algorithm (Algorithm 4). Besides the query itself, our algorithm receives:

1. An integer value, *level*, indicating the current recursion level, and
2. Integer values $r_1, \ldots, r_k$, indicating the current nodes in each trie, represented as the positions of these nodes within B_{level}.

Algorithm 5 shows the pseudo-code of our *adaptive* and *compressed* algorithm to compute the compact representation for bintrie($\mathcal{I}(Q)$) (denoted T_I in the pseudocode). The algorithm uses a binary variable s, initialized with **11**, which stores the bitwise-and of all current node codewords (line 4). So, s = **00** means that recursion must stop, s = **10** indicates to go down only to the left, s = **01** just to the right, and s = **11** to both children.

Lines 9–13 carry out the needed computation to go down to the left child. In particular, we compute the positions of the left-subtrie roots using rank_1 operation. Then, in line 13 we recursively go down to the left. The result of that recursion in stored in variable *lChild*, indicating with a **1** that the left recursion yielded a non-empty intersection, **0** otherwise. A similar procedure is carried out for the right child in lines 14–21. Line 17 determines whether we have already computed the rank_1s corresponding to the left child. If that is not the case, we compute them in line 18. In this way, we compute only one rank_1 operation per traversed node in the tries, which is important in practice. Just as for the left child, we store the result of the right-child recursion in variable *rChild* in line 21. Finally, in line 22 we determine whether the left and right recursions yielded an empty intersection or not. If both $lChild = rChild = 0$, the intersection was empty on both children, so we return **0**. Otherwise, we append *lChild* and *rChild* to $T_I.B_{level}$, as that is the codeword of the corresponding node in T_I. Note how we actually generate the output trie T_I in postorder, after we visited both children of the current nodes, despite the input is traversed in preorder. Thus, we write the output in time proportional to its size, saving time in practice.

Besides computing $\mathcal{I}(Q)$, a distinctive feature of our algorithm is that it also allows one to obtain the sequence $\langle \mathsf{rank}(S_{i_1}, x), \ldots, \mathsf{rank}(S_{i_k}, x)\rangle$, for all $x \in \mathcal{I}(Q)$, for free (in asymptotic terms). The idea is to compute $\langle \mathsf{bintrie}(S_{i_1}).B_\ell.\mathsf{rank}_1(r_1), \ldots, \mathsf{bintrie}(S_{i_k}).B_\ell.\mathsf{rank}_1(r_k)\rangle$ every time the recursion reaches level ℓ (i.e., just before the **return** of line 7 in Algorithm 5). Outputting this information is important for several applications, such as cases where set elements have satellite data associated to them. For an element $x_j \in S_i$, the associated data d_j is stored in an auxiliary array $D_i[1..n_i]$ such that $D[\mathsf{rank}(S_i, x_j)] = d_j$. Typical applications are inverted indexes in IR (where ranking information, such as frequencies, is associated to inverted list elements), and the Leapfrog Triejoin algorithm [2] (where at each step we must compute the intersection of sets, and for each element in the intersection we must go down following a pointer associated to it).

We have proved the following theorem:

Theorem 4.2. *Let $\mathcal{S} = \{S_1, \ldots, S_N\}$ be a family of N integer sets, each of size $|S_i| = n_i$ and universe $[0..u)$. There exists a data structure able to represent each set S_i using $2(\mathtt{trie}(S_i) - n_i + 1) + o(\mathtt{trie}(S_i))$ bits, such that given a query $Q = \{i_1, \ldots, i_k\} \subseteq [1..N]$, the intersection*

Algorithm 5: `AC-Intersection`(query Q; roots $r_1, \ldots, r_k$; *level*)

Result: The binary trie T_I representing $\mathcal{I}(Q) = \cap_{i \in Q} S_i$

1 **begin**
  // Preorder DFS
2   $s \leftarrow \mathbf{11}$ // binary encoding
3   **for** $i \in Q$ **do**
4     $s \leftarrow s$ & (bintrie$(S_i).B_{level}[r_i] \cdot$ bintrie$(S_i).B_{level}[r_i + 1]$)
5   **if** $level = \ell$ **then**
6   append s to $T_I.B_\ell$
7   **return 1**
8   $lChild \leftarrow \mathbf{0}$; $rChild \leftarrow \mathbf{0}$
  // Go down to the left in the tries
9   **if** s is **10** or **11** **then**
10    $lRoots \leftarrow \emptyset$
11   **for** $i \in Q$ **do**
12      $lRoots \leftarrow lRoots \cup \{2 \times$ bintrie$(S_i).B_{level}.\mathrm{rank}_\mathbf{1}(r_i - 1) + 1\}$
13    $lChild \leftarrow$ `AC-Intersection`$(Q, lRoots, level + 1)$
  // Go down to the right in the tries
14 **if** s is **01** or **11** **then**
15    $rRoots \leftarrow \emptyset$
16   **for** $i \in Q$ **do**
17   **if** $s = \mathbf{01}$ **then**
18        $rRoots \leftarrow rRoots \cup \{2 \times$ bintrie$(S_i).B_{level}.\mathrm{rank}_\mathbf{1}(r_i - 1) + 2\}$
19   **else**
20        $rRoots \leftarrow rRoots \cup \{lRoots_i + 2\}$
21    $rChild \leftarrow$ `AC-Intersection`$(Q, rRoots, level + 1)$
  // Output written in postorder
22 **if** $lChild \neq \mathbf{0}$ **or** $rChild \neq \mathbf{0}$ **then**
23   append $lChild \cdot rChild$ to $T_I.B_{level}$
24   **return 1**
25 **else**
26   **return 0**

$\mathcal{I}(Q) = \cap_{i \in Q} S_i$ *can be computed in* $O(k\delta \log(u/\delta))$ *time, where* δ *is the alternation measure of* Q*. Besides, for every* $x \in \mathcal{I}(Q)$*, the data structure also allows one to obtain the sequence* $\langle \mathsf{rank}(S_{i_1}, x), \ldots, \mathsf{rank}(S_{i_k}, x) \rangle$ *asymptotically for free.*

Chapter 5

Compressing Runs on Binary Tries

In this chapter, we exploit the presence of runs of successive elements in the input sets to reduce both the space usage of the binary trie representation, as well as intersection time.

5.1 Encoding Runs on Binary Tries

Runs tend to be captured by full subtrees in the corresponding binary tries. See, e.g., the full subtree whose leaves correspond to elements $8, 9, 10, 11$ in the binary trie of Figure 5.1. Let v be a $\mathsf{bintrie}(S)$ node whose subtree is full. Let $\mathsf{depth}(v) = d$. If $b = \mathsf{path}(v)$,the $2^{\ell-d}$ leaves covered by v correspond to the integer interval $[\mathsf{dec}(b \cdot \mathbf{0}^{\ell-d})..\mathsf{dec}(b \cdot \mathbf{1}^{\ell-d})]$. So, the subtree of v can be removed, keeping just v, saving space and still being able to recover the removed elements. In our compact representation, we encode a full-subtree cover node using **00**. Recall that **00** is an invalid codeword, so we use it as a special mark. See Figure 5.1 for an illustration.

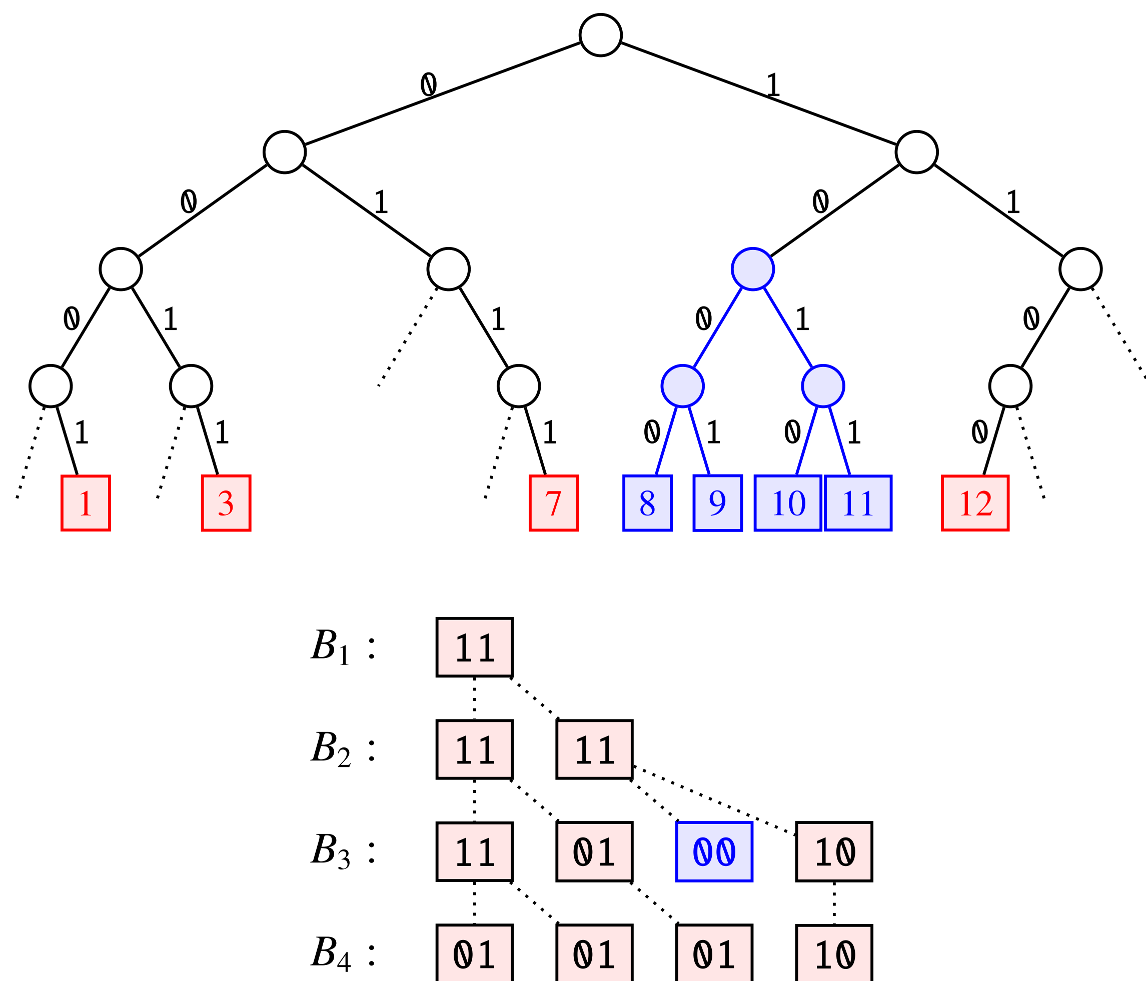


Figure 5.1: Above, the binary trie representing set $\{1, 3, 7, 8, 9, 10, 11, 12\}$. Notice that the subtree whose leaves correspond to elements 8, 9, 10, 11 is a full subtree. Below, our compact representation removing full subtrees and encoding their roots with **00**.

5.2 The $\texttt{rTrie}(S)$ Measure

Definition 5.1. *Let $S \subseteq [0..u)$ be a set of n elements. We define $\texttt{rTrie}(S)$ as the number of edges in $\mathsf{bintrie}(S)$ after removing the maximal full subtrees.*

This immediately implies $\texttt{rTrie}(S) \le \texttt{trie}(S) \le 2\texttt{gap}(S)$, yet we can prove tighter bounds. Assume a set S with r runs of $\ell_1, \ldots, \ell_r$ successive elements each, respectively. The ℓ_i elements of a given run correspond to ℓ_i contiguous leaves in $\mathsf{bintrie}(S)$ which, according to Lemma 2.1 (item 1), are covered by at most $2\lfloor\log(\ell_i/2)\rfloor$ nodes. This is a pessimistic case that removes the least edges, so we analyze it. Among the cover nodes, there are 2 whose subtrees have 0 edges, 2 whose subtrees have 2 edges, 2 whose subtrees have 6 edges, and so on. In general, for each $i = 1, \ldots, \lfloor\log(\ell_i/2)\rfloor$, there are 2 cover nodes whose subtrees have $2^i - 2$ edges. If we remove them all, the total number of edges removed is:

$$2 \sum_{i=1}^{\lfloor\log(\ell_i/2)\rfloor} (2^i - 2) = 2\ell_i - 4\log \ell_i.$$

This removes the least edges belonging to full subtrees, so we can bound

$$\texttt{rTrie}(S) \le \texttt{trie}(S) - \sum_{i=1}^{r} (2\ell_i - 4\log \ell_i). \tag{5.1}$$

We can also prove the following bounds.

Lemma 5.1. *Given a set $S \subseteq [0..u)$ of n elements, it holds that*

1. $\texttt{rTrie}(S) \le 2 \cdot \min\{\texttt{rle}(S) + \sum_{i=1}^{r} \log \ell_i,\ \texttt{gap}(S)\}$.

2. $\exists a \in [0..u)$, *such that:*

$$\texttt{rTrie}(S + a) \le \min\{\texttt{rle}(S) - \sum_{i=1}^{r} \ell_i + 3\sum_{i=1}^{r} \log \ell_i,\ \texttt{gap}(S)\} + 2n - 2.$$

3. $\texttt{rTrie}(S+a) \leq \min\{\texttt{rle}(S) - \sum_{i=1}^{r} \ell_i + 3\sum_{i=1}^{r} \log \ell_i,\ \texttt{gap}(S)\} + 2n - 2$ *on average, assuming* $a \in [0..u)$ *is chosen uniformly at random.*

Proof. Since S has r runs of $\ell_1, \ldots, \ell_r$ elements, we can rewrite $\texttt{gap}(S)$ as:

$$\texttt{gap}(S) = \sum_{i=1}^{r}(\lfloor \log (z_i - 1) \rfloor + 1) + \sum_{i=1}^{r}(\ell_i - 1).$$

As $\texttt{rTrie}(S) \leq \texttt{trie}(S) \leq 2\texttt{gap}(S)$, and $\texttt{rTrie}(S) \leq \texttt{trie}(S) - \sum_{i=1}^{r}(2\ell_i - 4\log \ell_i)$ (Equation 5.1), it holds that

$$\begin{aligned}
\texttt{rTrie}(S) &\leq \texttt{trie}(S) - \sum_{i=1}^{r}(2\ell_i - 4\log \ell_i) \\
&\leq 2(\sum_{i=1}^{r}(\lfloor \log (z_i - 1) \rfloor + 1) + \sum_{i=1}^{r}(\ell_i - 1)) - \sum_{i=1}^{r}(2\ell_i - 4\log \ell_i) \\
&= 2\sum_{i=1}^{r}(\lfloor \log (z_i - 1) \rfloor + 1) + 4\sum_{i=1}^{r}\log \ell_i \\
&\approx 2(\texttt{rle}(S) + \sum_{i=1}^{r}\log \ell_i),
\end{aligned}$$

proving item 1. Items 2 and 3 can be proved similarly from items 2 and 3 of Lemma 2.2. □

5.3 Adaptive Runs Compressed Intersection

Algorithm 6 shows the pseudo-code of our *adaptive* and *runs compressed* algorithm to compute the compact representation for $\mathsf{bintrie}(\mathcal{I}(Q))$ (denoted T_I in the pseudocode). Given a query $Q = \{i_1, \ldots, i_k\} \subseteq [1..N]$, the procedure to compute $\mathcal{I}(Q)$ is similar to that of Algorithm 5. The only difference is that if in a given trie $\mathsf{bintrie}(S_i)$ we arrive at a node encoded **00**, every possible set element in the subtrie of the node belongs to S_i. In other words, the intersection within the current subtries is independent of S_i, so we can safely temporarily exclude $\mathsf{bintrie}(S_i)$ from the intersection and continue intersecting the remaining tries. To implement this idea, we keep boolean flags $f_1, \ldots, f_k$ such that f_j corresponds to $\mathsf{bintrie}(S_{i_j})$. The idea is that at each point during the synchronized DFS traversal, only tries whose flag is

true participate in the intersection. Initially, we set $f_i \leftarrow$ true, for $1 \le i \le k$. If, during the intersection process, we arrive at a node encoded **00** in bintrie(S_i), we set $f_i \leftarrow$ false. When the recursion at a node encoded **00** in bintrie(S_i) finishes, we set $f_i \leftarrow$ true again. If, at a given point, all tries have been temporarily excluded but one, let us say bintrie(S_j), we only need to traverse the current subtree in S_j, copying it verbatim to the output. If this subtree contains nodes encoded **00**, they will appear in the output. This way, the maximal runs of successive elements in the output will be covered by nodes encoded **00**. This fact is key for the adaptive running time of our algorithm, as we shall see below.

5.4 Run-partition Certificate

We analyze our algorithm introducing the following variant of partition certificates.

Definition 5.2. *Given a query instance* $Q = \{i_1, \ldots, i_k\} \subseteq [1..N]$*, a run-partition certificate for it is a partition of the universe* $[0..u)$ *into a set of intervals* $\mathcal{P}^r_{\mathtt{AC}}(Q) = \{I_1, I_2, \ldots, I_p\}$*, such that the following conditions hold:*

1. $\forall x \in \mathcal{I}(Q), \exists I_j \in \mathcal{P}^r_{\mathtt{AC}}(Q),\ x \in I_j \wedge \mathcal{I}(Q) \cap I_j = I_j$*;*
2. $\forall x \notin \mathcal{I}(Q), \exists I_j \in \mathcal{P}^r_{\mathtt{AC}}(Q), x \in I_j\ \wedge\ \exists q \in Q, S_q \cap I_j = \emptyset$*.*

Let ξ *denote the size of the smallest run-partition certificate* $\mathcal{P}^r_{\mathtt{AC}}(Q)$ *of* Q*. We call* ξ *the* run alternation measure.

Item 2 is the same as for Barbay and Kenyon's partition certificates, corresponding to intervals of elements not in $\mathcal{I}(Q)$. Item 1, on the other hand, corresponds to elements in $\mathcal{I}(Q)$ which, unlike Barbay and Kenyon certificates, are not necessarily covered by singletons: our definition allows one to cover a run of successive elements in $\mathcal{I}(Q)$ using a single interval. Clearly, $\xi \le \delta$ holds. Besides, although $|\mathcal{I}(Q)| \le \delta$ holds, in our case there can be query instances such that $\xi < |\mathcal{I}(Q)|$. Figure 5.2 illustrates our definition for an intersection of 4 sets on the universe $[0..15)$. Notice that $\xi = 5$, whereas $|\mathcal{I}(Q)| = 6$ and $\delta = 9$.

S_{i_1} :				7	8	9	10	11	12	13	14	15
S_{i_2} :		5	6	7	8	9	10	11	12	13	14	
S_{i_3} :	4	5	6	7	8	9		11	12	13	14	
S_{i_4} :					8	9	10	11	12	13	14	15

Figure 5.2: A query instance $Q = \{S_{i_1}, S_{i_2}, S_{i_3}, S_{i_4}\}$ and its smallest run-partition certificate $\mathcal{P}^r_{\mathtt{AC}}(Q) = \{[0..7], [8..9], [10..10], [11..14], [15..15]\}$ of size $\xi = 5$. Gray intervals are the intersection elements, and the uncolored are the ones left out.

We must also introduce a fourth type of node to our trie certificate definition of Section 4.1. If for an internal node v of $\mathsf{cert}(Q)$ with $\mathtt{path}(v) = b$, it holds that there is a node v_i with $\mathtt{path}(v_i) = b$ in every $\mathsf{bintrie}(S_i)$, $i \in Q$, and the subtries of all v_is is full, then v is called an *internal success node*. It is important to note that every interval I_j from item 1 of Definition 5.2 is covered only by internal success nodes. Also, internal success nodes only cover intervals from item 1 of Definition 5.2.

The main result is stated in the following theorem:

Theorem 5.1. *Let $\mathcal{S} = \{S_1, \ldots, S_N\}$ be a family of N integer sets, each of size $|S_i| = n_i$ and universe $[0..u)$. There exists a data structure able to represent each set S_i using $2\mathtt{rTrie}(S_i)(1 + o(\mathtt{rTrie}(S_i)))$ bits, such that given a query $Q = \{i_1, \ldots, i_k\} \subseteq [1..N]$, the intersection $\mathcal{I}(Q) = \cap_{i \in Q} S_i$ can be computed in $O(k\xi \log(u/\xi))$ time, where ξ is the run alternation measure of Q.*

Proof. Consider the smallest run-partition certificate $\mathcal{P}^r_{\mathtt{AC}}(Q) = \{I_1, \ldots, I_\xi\}$ of universe $[0..u)$, such that $|I_i| = L_i$ for $i = 1, \ldots, \xi$. Let us cover these ξ intervals with as many nodes of the smallest $\mathsf{cert}(Q)$ as possible. As we already saw in the proof of Theorem 4.1, all intervals I_i such that $I_i \cap \mathcal{I}(Q) = \emptyset$ are covered by at most $O(\log L_i)$ nodes in $\mathsf{cert}(Q)$. We now prove the same for intervals $I_j \subseteq \mathcal{I}(Q)$, which are covered by internal success nodes of $\mathsf{cert}(Q)$. The only thing to note is that our algorithm stops as soon as it arrives to an internal success node. As there can be $O(\log L_j)$ such cover nodes, universe $[0..u)$ can be covered by $O(\sum_{i=1}^{\xi} \log L_i) = O(\sum_{i=1}^{\xi} \log(u/\xi))$ nodes, hence $\mathsf{cert}(Q)$ has $O(\xi \log(u/\xi))$ nodes overall. The result follows from the fact that at each node the algorithm runs in time $O(k)$. □

Algorithm 6: ARC-Intersection(query Q; roots $r_1, \ldots, r_k$; bool $f_1, \ldots, f_k$; $level$)

```
   Result: The binary trie T_I representing I(Q) = ∩_{i∈Q} S_i
 1 begin
     // Preorder DFS
 2   s ← 11 ; t ← 00 // binary encoding
 3   for i ∈ Q do
 4     if f_i = true then
 5       node ← bintrie(S_i).B_level [r_i] · bintrie(S_i).B_level [r_i + 1]
 6       if node = 00 then  f_i ← false
 7       else  s ← s & node
 8       t ← t | node
 9   if s = 00 and t ≠ 00 then  return 0
10   if t = 00 then  append 00 to T_I.B_level and return 1
11   if level = ℓ then  append s to T_I.B_ℓ and return 1
12   lChild ← 0;  rChild ← 0
     // Go down to the left in the tries
13   if s is 10 or 11  then
14     lRoots ← ∅
15     for i ∈ Q do
16       if f_i = true then  lRoots ← lRoots ∪ {2 × bintrie(S_i).B_level.rank_1(r_i) + 1}
17       else  lRoots ← lRoots ∪ {0}
18     lChild ← ARC-Intersection(Q, lRoots, level + 1)
     // Go down to the right in the tries
19   if s is 01 or 11  then
20     rRoots ← ∅
21     for i ∈ Q do
22       if f_i = true then
23         if s = 01  then
             rRoots ← rRoots ∪ {2 × bintrie(S_i).B_level.rank_1(r_i) + 2}
24         else  rRoots ← rRoots ∪ {lRoots_i + 1}
25       else
26         rRoots ← rRoots ∪ {0}
27     rChild ← ARC-Intersection(Q, rRoots, level + 1)
     // Output written in postorder
28   if lChild ≠ 0 or rChild ≠ 0  then
29     append lChild · rChild to T_I.B_level
30     return 1
31   else  return 0
```

Chapter 6

Implementation and Experimental Results

6.1 Implementation

We implemented bit vectors $B_1, \ldots, B_\ell$ in plain form using class `bit_vector<>` from the `sdsl` library [46]. We support rank_1 on them using different data structures to obtain the following schemes.

- (`trie v`, `rTrie v`): the variants defined in Chapter 4 and 5, respectively —using `rank_support_v` for rank_1. It uses 25% extra space on top of the bit vector, supporting rank_1 in O(1) time.
- (`trie v5`, `rTrie v5`): use `rank_support_v5`, requiring 6.25% extra space on top of the bit vectors, supporting rank_1 in O(1) time. This alternative is smaller, yet slower in practice.
- (`trie IL`, `rTrie IL`): use `rank_support_il`, aiming at reducing the number of cache misses to compute rank_1. We use block size 512, requiring ≈12.5% extra space on top of the bit vectors, while supporting rank_1 in O(1) time.

Most state-of-the-art alternatives we compare with do not support operation $\mathsf{rank}(S, x)$. So, to be fair, we do not store any rank_1 data structure for the last-level bit vector B_ℓ. Recall that $\mathsf{rank}(S, x)$ is equivalent to a rank_1 on the corresponding position of B_ℓ. We implemented Algorithm 5 and 6 on our compact trie data structures, following the descriptions from Chapters 4 and 5 very closely. We implemented, however, two alternatives for representing the output: (1) the binary trie representation, and (2) the plain array representation. In our experiments we will use the latter, to be fair: all testes alternatives produce their outputs in plain form.

6.1.1 A Multi-threaded Implementation

We also implemented a simple multi-threaded version of our algorithm. Let t denote the number of available cpu threads. Then, we define $c = \lfloor \log t \rfloor$. Our algorithm proceeds as in Algorithm 5, generating a binary trie of height c (that we will call *top trie*), with at most t leaves. Then, we execute Algorithm 5 again, this time in parallel, with each thread starting from a different leaf of the top trie. Each thread generates its own output in parallel, using our compact trie representation. Once all threads finish, we concatenate these tries to generate the final output. We just need to count, in parallel, how many nodes there are in each level of the trie. Then, we allocate a bit vector of the appropriate size for each level, where each thread will write its own part of the output in parallel. This simple approach does not guarantee load balancing among threads, however it works relatively well in practice.

For example, consider that we have the binary tries T_1 and T_2 shown in Figure 6.1, and we want to compute $T_1 \cap T_2$. Also consider that our cpu has 4 threads available, we call these 4 threads as t_1, t_2, t_3 and t_4. Then, we need to calculate the cut level $c = \lfloor \log 4 \rfloor = 2$. Therefore, we use the Algorithm 5 (or 6 depending on the representation of T_1 and T_2) to resolve the intersection between T_1 and T_2 until level 2 (dashed blue rectangle in Figure 6.1). As it can be noticed, when reaching level 2 we have 3 subtries (red triangles in Figure 6.1) independently, where the threads can work to solve the problem locally. It is easy to see that when the threads t_1, t_2 and t_3 finish, we only need to concatenate the results of the threads. As it can be seen, this method does not guarantee using all available threads, e.g., in this example we only use 3 out of 4 threads available. To alleviate this problem, if there are

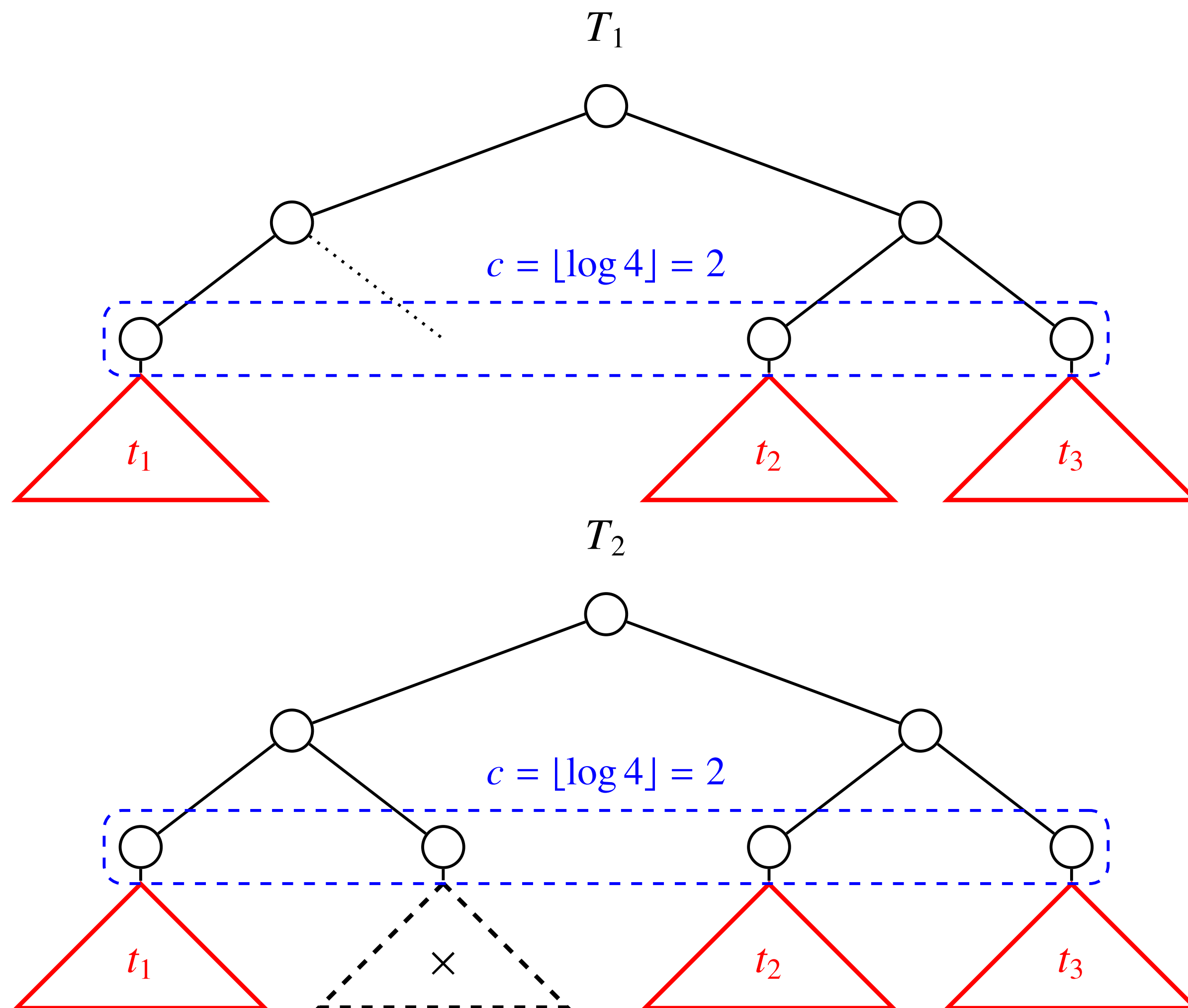


Figure 6.1: Multi-threaded binary trie intersection of T_1 and T_2 for $t = 4$, blue dashed rectangle represent the level $c = \lfloor \log t \rfloor = 2$ in T_1 and T_2. Dashed triangle with $\times$ denotes not part of the intersection.

idle threads after we solve the top trie, we can go down one level in one of the subproblems (subtries in red in Figure 6.1) trying to allocate as many threads as possible.

6.2 Experimental Setup

Hardware and Software

All experiments were run on server with an i7 10700k CPU with 8 cores and 16 threads, with turbo boost disabled, running at 4.70 GHz in all cores and running Ubuntu 20.04 LTS operating system. We have 32GB of RAM(DDR4-3.6GHz) running in dual channel. Our implementation is developed in C++, compiled with `g++` 9.3.0 and optimization flags `-O3` and `-march=native`.

	Gov2	ClueWeb09	CC-News
# Lists	57,225	131,567	79,831
# Integers	5,509,206,378	14,895,136.282	18,415,151,585
u	25,205,179	50,220,423	43,495,426
$\lceil \log u \rceil$	25	26	26
gap(S)	2.25	3.25	3,70
rle(S)	1.99	3.33	4,23
trie(S)	3.48	4.56	5,18
rTrie(S)	2.51	4.00	5,12
Elias γ	3.71	5.74	6.81
Elias δ	3.64	5.40	6.69
Fibonacci	3.90	5.35	6.09
Elias γ 128	4.07	6.10	7.05
Elias δ 128	4.00	5.77	7.17
Fibonacci 128	4.26	5.71	6.45
rrr_vector<>	11.82	19.94	11.29
sd_vector<>	8.45	8.52	7.17

Table 6.1: Dataset summary and average space usage (in bits per integer, bpi) for different compression measures and baseline representations.

Datasets and Queries

In our tests, we used family of sets corresponding to inverted indexes of three standard document collections: Gov2 [1], ClueWeb09 [2], and CC-News [47]. For Gov2 and ClueWeb09 collections, we used the freely-available inverted indexes and query log by D. Lemire [3], corresponding to the URL-sorted document enumeration [48], which tends to yield runs of successive elements in the sets. The query log contains 20,000 random queries from the TREC million-query track (1MQ). Each query has at least 2 query terms. Also, each term is in the top-1M most frequently queried terms. For CC-News we use the freely-available inverted index by Mackenzie et al. [47] in a *Common Index File Format* (CIFF) [49], as well as their query log of 9,666 queries. Table 6.1 shows a summary of statistics of the collections. In all cases, we only keep inverted lists with length at least 4096.

[1] https://www-nlpir.nist.gov/projects/terabyte/
[2] https://lemurproject.org/
[3] See https://lemire.me/data/integercompression2014.html for download details.

Compression Measures and Baselines

Table 6.1 also shows the average bit per integer (bpi) for different compression measures on our tested set collections. We also show the average bpi for different integer compression approaches, namely Elias γ and δ [24], Fibonacci [50], `rrr_vector<>` [29], and `sd_vector<>` [28], all of them from the `sdsl` library [46]. We show a plain version of them, as well as variants with blocks of 128 integers. The latter are needed to speed up decoding. However, it is well known that these codes are relatively slow to be decoded, and hence yield higher intersection times. On the other hand, `sd_vector<>` uses $n \log(u/n) + 2n + o(n)$ bits to encode a set of n elements from the universe $[0..u)$, which is close to the worst-case upper bound $\mathcal{B}(n, u)$. Finally, `rrr_vector<>` uses $\mathcal{B}(n, u) + o(u)$ bits of space. As it can be seen, the $o(u)$-bit term yields a space usage higher than the remaining alternatives. These values will serve as a baseline to compare our results.

6.3 Experimental Results

Next, we compare our approaches with state-of-the-art set compression alternatives available at the project *Performant Indexes and Search for Academia* [4] (PISA) [51]:

- `IPC`: the Binary Interpolative Coding approach by Moffat et al. [30]. This is a highly space-efficient approach, with a relatively slow processing performance (as seen in Section 3.3).
- `PEF Opt`: the highly competitive approach by Ottaviano and Venturini [15] (seen in Section 3.8.1).
- `OptPFD`: The Optimized PForDelta approach by Yan et al. [12] (seen in Section 3.7).
- `SIMD-BP128`: The highly efficient approach by Lemire and Boytsov [3], aimed at decoding billions of integers per second using vectorization capabilities of modern processors (as seen in Section 3.6).

[4] `https://github.com/pisa-engine/pisa`

- `Simple16`: The approach by Zhang at al. [35], a variant of the `Simple9` approach [34] that combines a relatively good space usage and an efficient intersection time (seen in Sections 3.5.1 and 3.5.2).

- `VarintGB`: The approach used in Google and presented by Dean [32] (seen in Section 3.4.2).

- `Varint-G8IU`: by Stepanov et al. [33], using SIMD instructions to speed-up set processing (seen in Section 3.4.3).

We also compared with the following approaches, available from their authors:

- `Roaring`: the compressed bitmap approach by Lemire et al. [13], widely used as indexing tool on several systems and platforms[5]. Roaring bitmaps are highly competitive, leveraging modern CPU hardware architectures (seen in Section 3.9). We use the code from the authors[6].

- `RUP`: The recent recursive universe partitioning approach by Pibiri [14], using also SIMD instructions to speed up processing (seen in Section 3.10). We use the code from the author[7].

Table 6.2 shows the average experimental intersection time and space usage (in bits per integer) for all the alternatives tested. For `trie` and `rTrie` alternatives, we show running times using the format X/Y, where X is the average intersection time using a single thread, whereas Y is the average running time for 16 threads. The results show that the multi-threaded implementation is between ~5 to ~10 times faster than the single thread version. As it can be seen, our approaches introduce relevant trade-offs. We compare next with the most competitive approaches on the different datasets.

- For Gov2, `rTrie` uses 1.166–1.329 times the space of `PEF`, the former being 1.549–2.442 times faster. `rTrie` uses 0.481–0.548 times the space of `Roaring`, the former

[5] https://roaringbitmap.org/
[6] https://github.com/RoaringBitmap/CRoaring
[7] https://github.com/jermp/s_indexes

	Gov2		ClueWeb09		CC-News	
Data Structure	Space	Time	Space	Time	Space	Time
IPC	3.34	8.66	5.15	30.18	5.87	68.98
S16	4.65	2.44	6.72	8.66	6.88	19.74
OptPFD	4.07	2.15	6.28	7.79	6.50	11.80
PEF Opt	3.62	1.88	5.85	6.50	5.80	17.33
VarintGB	10.80	1.43	11.40	7.34	11.04	12.38
Varint-G8IU	9.97	1.38	10.55	5.25	10.24	12.09
SIMD-BP128	6.07	1.29	8.98	4.47	7.36	15.90
Roaring	8.77	1.09	12.62	3.75	9.86	5.56
RUP	5.04	1.10	8.44	4.27	8.41	5.44
trie (v5)	5.18	6.44 / 1.21	7.46	23.68 / 2.81	8.77	78.18 / 8.72
trie (IL)	5.41	6.30 / 1.06	7.83	23.40 / 2.42	9.30	76.08 / 7.46
trie (v)	5.85	3.67 / 0.77	8.50	13.54 / 1.64	9.99	46.99 / 5.21
rTrie (v5)	4.22	6.43 / 1.22	6.95	24.52 / 3.07	8.73	82.46 / 9.74
rTrie (IL)	4.42	6.42 / 1.10	7.31	24.42 / 2.62	9.16	80.63 / 8.13
rTrie (v)	4.81	3.80 / 0.77	7.96	14.62 / 1.96	9.95	51.26 / 6.09

Table 6.2: Average intersection time and space usage (in bits per integer) for all alternatives tested. For trie and rTrie alternatives, we show running times using the format X/Y, where X is the average intersection time using a single thread, whereas Y is the average running time for 16 threads.

being up to 1.415 times faster. Finally, rTrie uses 0.837–0.954 times the space of RUP, the former being up to 1.428 times faster.

- For ClueWeb09, rTrie uses 1.188–1.361 times the space of PEF, the former being 2.117–3.316 times faster. Also, rTrie uses 0.551–0.631 times the space of Roaring, the former being 1.221–1.913 times faster.

- For CC-News, the resulting inverted index has considerably less runs in the inverted lists, hence the space usage of trie and rTrie is about the same. However, trie is faster than rTrie, as the code to handle runs introduces an overhead that does not pay off in this case. So, we will use trie to compare here. It uses 1.512–1.722 times the space of PEF, the former being 1.987–3.326 times faster. trie uses 0.889–1.013 times the space of Roaring, the former being up to 1.067 times faster. Finally, trie uses 1.043–1.188 times the space of RUP, the former being up to 1.044 times faster. Finally,

`rTrie` uses 0.823–0.943 times the space of `RUP`, the former being 1.391–2.178 times faster.

Finally, we can conclude that in all tested datasets, at least one of our trade-offs is the fastest, outperforming the highly-engineered ultra-efficient set compression techniques we tested, as can be seen in Figure 6.2.

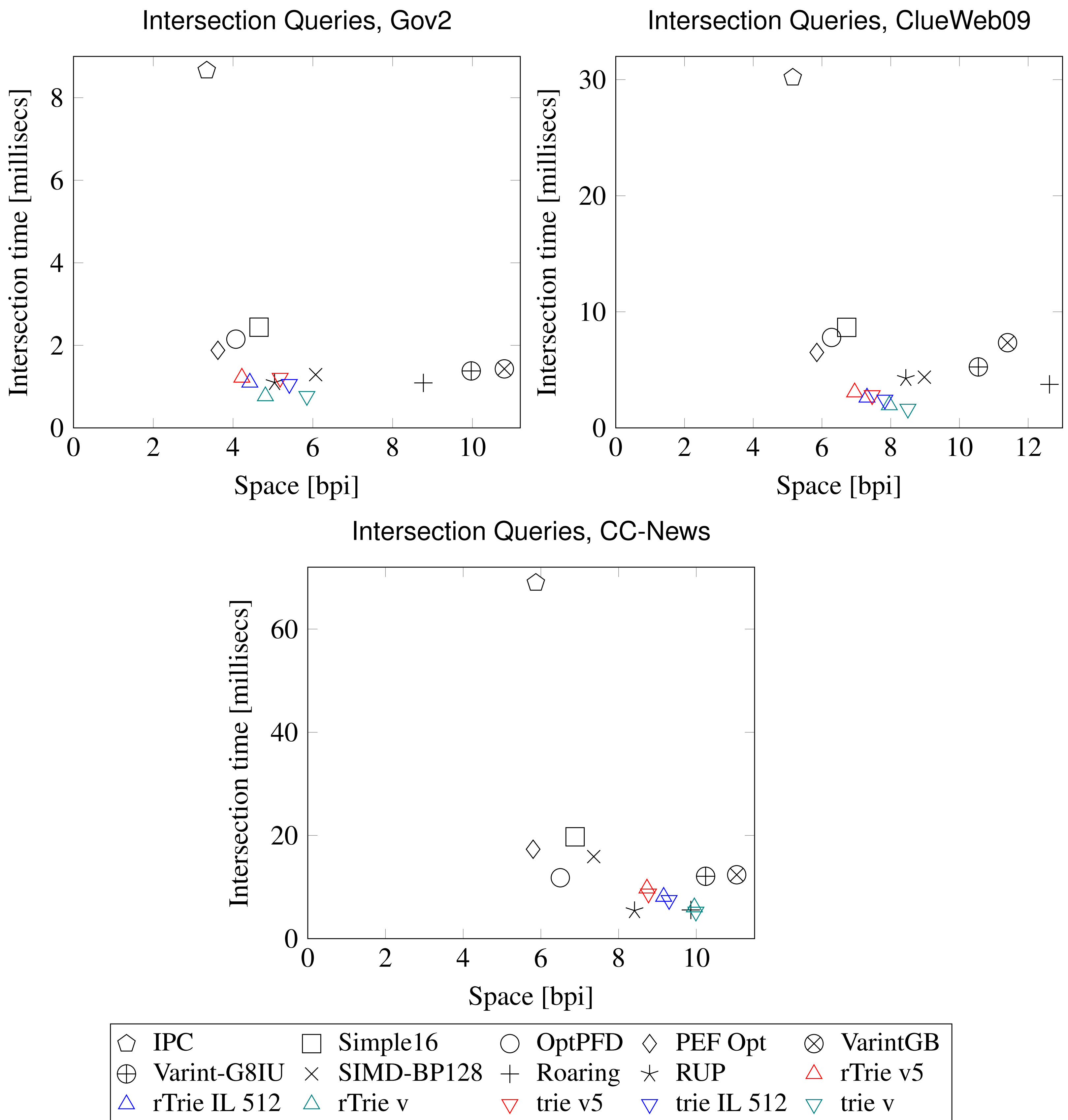


Figure 6.2: Space vs. time trade-off for all alternative tested on the 3 datasets.

Conclusions and Future Work

Trie partition certificates, the main concept we introduced as an alternative to existing certificates by Demaine et al. [9] and Barbay and Kenyon [10], allowed us to introduce our main contributions. In particular, we were able to prove that Trabb-Pardo's intersection algorithm [21] works in $O(k\delta \log(u/\delta))$ time, where δ is the alternation measure of the query instance [10]. Thus, Trabb-Pardo's intersection algorithm was likely the first adaptive intersection algorithm that ever existed, appearing about 22 years before Demaine et al.'s adaptive approach. The lack of analysis on this algorithm (the original author only analyzed his algorithm in the average case) might explain the lack of consideration regarding this algorithm, in particular in practice. Motivated by this result, we introduced compressed representations of integer sets preserving the running time of Trabb-Pardo's algorithm, and even improving it. Summarizing, our proposals: (1) use compressed space usage, (2) have adaptive intersection computation time, and (3) have highly competitive practical performance.

Multiple avenues for future research are open now. For instance, novel data structures supporting operation rank_1 have emerged recently [52]. These offer interesting trade-offs, using less space than then ones we used, with competitive operation times. Another interesting line is that of alternative binary trie compact representations. E.g., a DFS representation (rather than BFS, as the one used in this work), which would potentially reduce the number of cache misses when traversing the tries. Finally, our representation would support dynamic sets (where insertion and deletion of elements are allowed) if we use dynamic binary tries [53].

Bibliography

[1] R. Elmasri and S. B. Navathe, *Fundamentals of Database Systems, 6th Edition*. Pearson, 2011.

[2] T. L. Veldhuizen, "Triejoin: A simple, worst-case optimal join algorithm," in *Proc. 17th International Conference on Database Theory (ICDT)* (N. Schweikardt, V. Christophides, and V. Leroy, eds.), pp. 96–106, OpenProceedings.org, 2014.

[3] D. Lemire, L. Boytsov, and N. Kurz, "Simd compression and the intersection of sorted integers," *Software: Practice and Experience*, vol. 46, no. 6, pp. 723–749, 2016.

[4] I. Witten, A. Moffat, and T. Bell, *Managing Gigabytes: Compressing and Indexing Documents and Images, 2nd Edition*. Morgan Kaufmann, 1999.

[5] R. M. Layer and A. R. Quinlan, "A parallel algorithm for n-way interval set intersection," *Proc. IEEE*, vol. 105, no. 3, pp. 542–551, 2017.

[6] D. Arroyuelo, J. Fuentes-Sepúlveda, and D. Seco, "Three success stories about compact data structures," *Communications of the ACM*, vol. 63, no. 11, pp. 64–65, 2020.

[7] G. Navarro, *Compact Data Structures – A Practical Approach*. Cambridge University Press, 2016.

[8] R. Baeza-Yates, "A fast set intersection algorithm for sorted sequences," in *Proc. 15th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 3109, pp. 400–408, Springer, 2004.

[9] E. D. Demaine, A. López-Ortiz, and J. I. Munro, "Adaptive set intersections, unions, and differences," in *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms, January 9-11, 2000, San Francisco, CA, USA* (D. B. Shmoys, ed.), pp. 743–752, ACM/SIAM, 2000.

[10] J. Barbay and C. Kenyon, "Alternation and redundancy analysis of the intersection problem," *ACM Transations on Algorithms*, vol. 4, no. 1, pp. 4:1–4:18, 2008.

[11] J. Barbay and C. Kenyon, "Adaptive intersection and t-threshold problems," in *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms, January 6-8, 2002, San Francisco, CA, USA* (D. Eppstein, ed.), pp. 390–399, ACM/SIAM, 2002.

[12] M. Zukowski, S. Heman, N. Nes, and P. Boncz, "Super-scalar ram-cpu cache compression," in *22nd International Conference on Data Engineering (ICDE'06)*, pp. 59–59, 2006.

[13] D. Lemire, O. Kaser, N. Kurz, L. Deri, C. O'Hara, F. Saint-Jacques, and G. Ssi-Yan-Kai, "Roaring bitmaps: Implementation of an optimized software library," *Software: Practice and Experience*, vol. 48, pp. 867–895, jan 2018.

[14] G. E. Pibiri, "Fast and compact set intersection through recursive universe partitioning," in *2021 Data Compression Conference (DCC)*, pp. 293–302, 2021.

[15] G. Ottaviano and R. Venturini, "Partitioned elias-fano indexes," in *Proceedings of the 37th International ACM SIGIR Conference on Research & Development in Information Retrieval*, SIGIR '14, (New York, NY, USA), p. 273–282, Association for Computing Machinery, 2014.

[16] P. Bille, A. Pagh, and R. Pagh, "Fast evaluation of union-intersection expressions," in *Proc. 18th International Symposium on Algorithms and Computation (ISAAC)*, LNCS 4835, pp. 739–750, Springer, 2007.

[17] H. Cohen and E. Porat, "Fast set intersection and two-patterns matching," *Theoretical Computer Science*, vol. 411, no. 40-42, pp. 3795–3800, 2010.

[18] B. Ding and A. König, "Fast set intersection in memory," *Proc. VLDB Endowment*, vol. 4, no. 4, pp. 255–266, 2011.

[19] T. Gagie, G. Navarro, and S. J. Puglisi, "New algorithms on wavelet trees and applications to information retrieval," *Theoretical Computer Science*, vol. 426, pp. 25–41, 2012.

[20] R. Grossi, A. Gupta, and J. S. Vitter, "High-order entropy-compressed text indexes," in *Proc. of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pp. 841–850, ACM/SIAM, 2003.

[21] L. T. Pardo, *Set Representation and Set Intersection*. PhD thesis, Stanford University, 1978. D. E. Knuth, advisor.

[22] R. De La Briandais, "File searching using variable length keys," in *Papers Presented at the the March 3-5, 1959, Western Joint Computer Conference*, IRE-AIEE-ACM '59 (Western), (New York, NY, USA), p. 295–298, Association for Computing Machinery, 1959.

[23] E. Fredkin, "Trie memory," *Commun. ACM*, vol. 3, p. 490–499, Sept. 1960.

[24] P. Elias, "Universal codeword sets and representations of the integers," *IEEE Transactions on Information Theory*, vol. 21, no. 2, pp. 194–203, 1975.

[25] L. Foschini, R. Grossi, A. Gupta, and J. S. Vitter, "When indexing equals compression: Experiments with compressing suffix arrays and applications," *ACM Transactions on Algorithms*, vol. 2, no. 4, pp. 611–639, 2006.

[26] S. T. Klein and D. Shapira, "Searching in compressed dictionaries," in *Proc. Data Compression Conference (DCC)*, p. 142, IEEE Computer Society, 2002.

[27] A. Gupta, W.-K. Hon, R. Shah, and J. S. Vitter, "Compressed data structures: Dictionaries and data-aware measures," *Theoretical Computer Science*, vol. 387, no. 3, pp. 313–331, 2007. The Burrows-Wheeler Transform.

[28] D. Okanohara and K. Sadakane, "Practical entropy-compressed rank/select dictionary," in *Proc. of 9th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pp. 60–70, 2007.

[29] R. Raman, V. Raman, and S. R. Satti, "Succinct indexable dictionaries with applications to encoding *k*-ary trees, prefix sums and multisets," *ACM Transactions on Algorithms*, vol. 3, no. 4, p. 43, 2007.

[30] A. Moffat and L. Stuiver, "Binary interpolative coding for effective index compression," *Information Retrieval*, vol. 3, no. 1, pp. 25–47, 2000.

[31] L. Thiel and H. Heaps, "Program design for retrospective searches on large data bases," *Information Storage and Retrieval*, vol. 8, no. 1, pp. 1–20, 1972.

[32] J. Dean, "Challenges in building large-scale information retrieval systems: Invited talk," in *Proceedings of the Second ACM International Conference on Web Search and Data Mining*, WSDM '09, (New York, NY, USA), p. 1, Association for Computing Machinery, 2009.

[33] A. A. Stepanov, A. R. Gangolli, D. E. Rose, R. J. Ernst, and P. S. Oberoi, "Simd-based decoding of posting lists," in *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*, CIKM '11, (New York, NY, USA), p. 317–326, Association for Computing Machinery, 2011.

[34] V. N. Anh and A. Moffat, "Index compression using fixed binary codewords," in *Proceedings of the 15th Australasian Database Conference - Volume 27*, ADC '04, (AUS), p. 61–67, Australian Computer Society, Inc., 2004.

[35] J. Zhang, X. Long, and T. Suel, "Performance of compressed inverted list caching in search engines," in *Proceedings of the 17th International Conference on World Wide Web*, WWW '08, (New York, NY, USA), p. 387–396, Association for Computing Machinery, 2008.

[36] D. Arroyuelo, M. Oyarzún, S. González, and V. Sepulveda, "Hybrid compression of inverted lists for reordered document collections," *Information Processing & Management*, vol. 54, no. 6, pp. 1308–1324, 2018.

[37] V. N. Anh and A. Moffat, "Index compression using 64-bit words," *Software: Practice and Experience*, vol. 40, no. 2, pp. 131–147, 2010.

[38] J. Goldstein, R. Ramakrishnan, and U. Shaft, "Compressing relations and indexes," in *Proceedings 14th International Conference on Data Engineering*, pp. 370–379, 1998.

[39] H. Yan, S. Ding, and T. Suel, "Inverted index compression and query processing with optimized document ordering," in *Proceedings of the 18th International Conference on World Wide Web*, WWW '09, (New York, NY, USA), p. 401–410, Association for Computing Machinery, 2009.

[40] P. Elias, "Efficient storage and retrieval by content and address of static files," *Journal of the ACM (JACM)*, vol. 21, no. 2, pp. 246–260, 1974.

[41] R. M. Fano, "On the number of bits required to implement an associative memory. memorandum 61," *Computer Structures Group, Project MAC, MIT, Cambridge, Mass., nd*, p. 27, 1971.

[42] S. Vigna, "Quasi-succinct indices," in *Proceedings of the Sixth ACM International Conference on Web Search and Data Mining*, WSDM '13, (New York, NY, USA), p. 83–92, Association for Computing Machinery, 2013.

[43] A. L. Buchsbaum, G. S. Fowler, and R. Giancarlo, "Improving table compression with combinatorial optimization," *J. ACM*, vol. 50, p. 825–851, nov 2003.

[44] G. Jacobson, "Space-efficient static trees and graphs," in *Proc. 30th Annual Symposium on Foundations of Computer Science (FOCS)*, pp. 549–554, IEEE Computer Society, 1989.

[45] D. Clark, *Compact PAT trees*. PhD thesis, University of Waterloo, 1997.

[46] S. Gog and M. Petri, "Optimized succinct data structures for massive data," *Software: Practice and Experience*, vol. 44, no. 11, pp. 1287–1314, 2014.

[47] J. Mackenzie, R. Benham, M. Petri, J. R. Trippas, J. S. Culpepper, and A. Moffat, "Cc-news-en: A large english news corpus," in *Proc. CIKM*, p. To Appear, 2020.

[48] F. Silvestri, "Sorting out the document identifier assignment problem," in *Proc. of 29th European Conference on IR Research (ECIR)*, LNCS 4425, pp. 101–112, Springer, 2007.

[49] J. Lin, J. Mackenzie, C. Kamphuis, C. Macdonald, A. Mallia, M. Siedlaczek, A. Trotman, and A. de Vries, "Supporting interoperability between open-source search engines with the common index file format," 2020.

[50] A. S. Fraenkel and S. T. Klein, “Robust universal complete codes for transmission and compression,” *Discrete Applied Mathematics*, vol. 64, no. 1, pp. 31–55, 1996.

[51] A. Mallia, M. Siedlaczek, J. Mackenzie, and T. Suel, “PISA: performant indexes and search for academia,” in *Proc. of the Open-Source IR Replicability Challenge co-located with 42nd International ACM SIGIR Conference on Research and Development in Information Retrieval*, pp. 50–56, 2019.

[52] F. Kurpicz, “Engineering compact data structures for rank and select queries on bit vectors,” in *Proc. 29th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 13617, pp. 257–272, Springer, 2022.

[53] D. Arroyuelo, P. Davoodi, and S. R. Satti, “Succinct dynamic cardinal trees,” *Algorithmica*, vol. 74, no. 2, pp. 742–777, 2016.