The Trie Measure, Revisited

Jarno N. Alanko
University of Helsinki, Finland

Ruben Becker
Ca' Foscari University of Venice, Italy

Davide Cenzato
Ca' Foscari University of Venice, Italy

Travis Gagie
Dalhousie University, Halifax, Canada

Sung-Hwan Kim
Ca' Foscari University of Venice, Italy

Bojana Kodric
Ca' Foscari University of Venice, Italy

Nicola Prezza
Ca' Foscari University of Venice, Italy

Abstract

In this paper, we study the following problem: given n subsets $S_1, \ldots, S_n$ of an integer universe $U = \{0, \ldots, u-1\}$, having total cardinality $N = \sum_{i=1}^{n} |S_i|$, find a prefix-free encoding $\mathrm{enc} : U \to \{0,1\}^+$ minimizing the so-called *trie measure*, i.e., the total number of edges in the n binary tries $\mathcal{T}_1, \ldots, \mathcal{T}_n$, where $\mathcal{T}_i$ is the trie packing the encoded integers $\{\mathrm{enc}(x) : x \in S_i\}$. We first observe that this problem is equivalent to that of merging u sets with the cheapest sequence of binary unions, a problem which in [Ghosh et al., ICDCS 2015] is shown to be NP-hard. Motivated by the hardness of the general problem, we focus on particular families of prefix-free encodings. We start by studying the fixed-length *shifted encoding* of [Gupta et al., Theoretical Computer Science 2007]. Given a parameter $0 \le a < u$, this encoding sends each $x \in U$ to $(x + a) \bmod u$, interpreted as a bit-string of $\log u$ bits. We develop the first efficient algorithms that find the value of a minimizing the trie measure when this encoding is used. Our two algorithms run in $O(u + N \log u)$ and $O(N \log^2 u)$ time, respectively. We proceed by studying *ordered encodings* (a.k.a. *monotone* or *alphabetic*), and describe an algorithm finding the optimal such encoding in $O(N + u^3)$ time. Within the same running time, we show how to compute the best *shifted ordered encoding*, provably no worse than *both* the optimal shifted and optimal ordered encodings. We provide implementations of our algorithms and discuss how these encodings perform in practice.

2012 ACM Subject Classification Theory of computation → Data compression; Theory of computation → Design and analysis of algorithms; Theory of computation → Dynamic programming; Theory of computation → Discrete optimization

Keywords and phrases Succinct data structures, degenerate strings, integer encoding

Digital Object Identifier 10.4230/LIPIcs.CPM.2025.19

Related Version *ArXiv Version*: `https://arxiv.org/abs/2504.10703`

Supplementary Material *Software*: `https://github.com/regindex/trie-measure`

Funding *Ruben Becker, Davide Cenzato, Sung-Hwan Kim, Bojana Kodric, Nicola Prezza*: Funded by the European Union (ERC, REGINDEX, 101039208). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council Executive Agency. Neither the European Union nor the granting authority can be held responsible for them.
Jarno N. Alanko: Funded by Academy of Finland grant 339070.
Travis Gagie: Funded by NSERC Discovery Grant RGPIN-07185-2020.

36th Annual Symposium on Combinatorial Pattern Matching (CPM 2025).
Editors: Paola Bonizzoni and Veli Mäkinen; Article No. 19; pp. 19:1–19:20
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Consider the problem of encoding a set of integers $S \subseteq U = \{0, \ldots, u-1\}$ (without loss of generality, we assume u to be a power of two), so as to minimize the overall number of bits used to represent S. In their seminal work on data-aware measures, Gupta et al. [10] proposed and analyzed an encoding for sets of integers based on the idea of packing the integers (seen as strings of $\log u$ bits) into a binary trie: this allows to avoid storing multiple times shared prefixes among the encodings of the integers. The number of edges in such a trie is known as the *trie measure* of the set. See Figure 1 for an example.

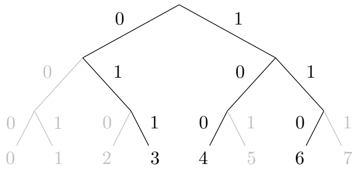


Figure 1 Example of trie encoding the set of integers $\{3, 4, 6\} \subseteq \{0, 1, \ldots, 7\}$ over universe of size $u = 8$. Black edges belong to the trie. Gray edges do not belong to the trie and are shown only for completeness. Each integer is encoded using $\log 8 = 3$ bits (logarithms are in base 2). The trie has 8 edges, so the *trie measure* for this set using the standard integer encoding is 8.

Gupta et al. also showed that this measure approaches worst-case entropy on expectation when the integers are shifted by a uniformly-random quantity modulo u, i.e. when building the trie over the set $S + a := \{x + a \bmod u : x \in S\}$ for a uniform $a \in U$. See Figure 2.

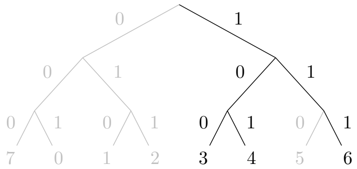


Figure 2 A trie that stores the same set of integers $\{3, 4, 6\} \subseteq \{0, 1, \ldots, 7\}$ of Figure 1, but with the shifted integer encoding mapping each $x \in U$ to (the binary string of $\log u$ bits) $(x + 1) \bmod 8$. The trie has 6 edges, so the *shifted trie measure* with shift $a = 1$ is 6.

In this paper, we revisit this problem in two natural directions: (1) we move from one set to a sequence of sets, and (2) we study more general prefix-free integer encodings. More formally, given n subsets $S_1, \ldots, S_n$ of U, having total cardinality $N = \sum_{i=1}^{n} |S_i|$, we study the problem of finding a prefix-free encoding $\mathrm{enc} : U \rightarrow \{0, 1\}^+$ minimizing the *trie measure* of these sets, i.e. the total number of edges in the n binary tries $\mathcal{T}_1, \ldots, \mathcal{T}_n$, where $\mathcal{T}_i$ is the trie packing the encoded integers $\{\mathrm{enc}(x) : x \in S_i\}$. Solving the problem on set sequences finds applications in data structures for rank/select queries on set sequences – also known in the literature as *degenerate strings* – , an important toolbox in indexes for labeled graphs [8, 6, 5, 7]. Such applications stem from the recent work of Alanko et al. [1], who showed how to solve rank and select queries on set sequences with a data structure – the *subset wavelet tree* – implicitly encoding sets as tries. While [1] only considered the standard binary (balanced) encoding, their technique works for any prefix-free encoding. As a result, our work can directly be applied to optimize the space of their data structure. Another application is the offline set intersection problem where the goal to preprocess subsets of the universe, so that later given a set of indices of the sets, one can compute the intersection of the sets space-time efficiently. Arroyuelo and Castillo [2] presented an adaptive approach that uses a trie representation of sets and they showed that it can benefit from its space-efficient representation.

We begin by showing that finding the optimal such prefix-free encoding is equivalent to the following natural optimization problem: find the minimum-cost sequence of binary unions that merges u given sets, where the cost of a union is the sum of the two sets' cardinalities. As this problem was shown to be NP-hard by Ghosh et al. [9], we focus on particular (hopefully easier) families of encodings. We start by studying the shifted encoding of Gupta et al., who did not discuss the problem of finding the optimal value of $a \in U$ minimizing the corresponding trie measure. We describe an algorithm solving the problem in $O(u + N \log u)$ time. The algorithm is based on the fact that, under this encoding, the trie measure has a highly periodic sub-structure as a function of the shift a. We then remove the linear dependency on the universe size and achieve running time $O(N \log^2 u)$ by storing this periodic structure with a DAG. $O(N \log u)$ running time is also possible by merging the two ideas, but for space constraints we will describe it in the extended version of this article. We also want to remark that the general prefix-free encoding problem is not only difficult to compute, but also it requires to store the ordering of the elements in addition to the trie representation. By a simple enumeration, it needs $O(N' \log u)$ bits of space where N' is the cardinality of the union of the sets (i.e., the number of distinct elements over all sets). On the other hand, the shifted encoding only requires $O(\log u)$ bits since it is sufficient to store a single integer $a \in U$.

We then move to *ordered* encodings: here, the encoding must preserve lexicographically the order of the universe U (i.e. the standard integer order). In this case, we show that the textbook solution of Knuth [12] based on dynamic programming, running in $O(N + u^3)$-time, can be adapted to our scenario. We observe that essentially the same solution allows also shifting the universe (like in the shifted encoding discussed above) at no additional cost. As a result, we obtain that the best *shifted ordered encoding* can be computed in $O(N + u^3)$-time as well. This encoding is never worse than the best shifted and the best ordered encodings.

We conclude our paper with an experimental evaluation of our algorithms on real datasets, showing how these encodings perform with respect to the worst and average-case scenario.

2 Preliminaries and Problem Formulation

A bit-string is a finite sequence of bits, i.e., an element from $\{0,1\}^*$. We count indices from 1, so that for $\beta \in \{0,1\}^+$, $\beta[1]$ is the first element. With $\prec$ we denote the lexicographic order among bit-strings and with $|\cdot|$ we denote the length of bit-strings. For two strings α and β of possibly different length, each not being a prefix of the other, we use $\oplus$ to denote the (non-commutative) operator defined as $\alpha \oplus \beta = \beta[j \dots |\beta|]$, where $j-1$ is the length of the longest common prefix of α and β, i.e., $\alpha[i] = \beta[i]$ for $1 \leq i < j$ and $\alpha[j] \neq \beta[j]$.

We denote with $U := \{0, 1, \dots, u-1\}$ the integer universe and we assume that u is a power of two. Logarithms are in base 2, so $\log(x)$ indicates $\log_2(x)$. In particular, $\log u$ is an integer. Notation $[n]$ indicates the set $\{1, 2, \dots, n\}$. For integers $\ell < r$, we denote with $[\ell, r)$ an interval of integers $\{\ell, \ell+1, \cdots, r-2, r-1\}$ and if $\ell = r$, then $[\ell, r) = \emptyset$ is the empty set. We use double curly braces $\{\!\{\dots\}\!\}$ to represent multisets. Given a multiset $\mathcal{I}$ containing subsets of U and an integer $a \in U$, the *depth* of $\mathcal{I}$ at position a is defined as the number of elements of $\mathcal{I}$ that contain a. For integer a, b and p with $p \geq 1$, we say $a \equiv_p b$ if and only if $a \bmod p = b \bmod p$. For a predicate P, we use $\mathbb{1}[P]$ to denote the indicator function that is 1 if P holds and zero otherwise.

A prefix-free encoding of U is a function $\mathrm{enc} : U \to \{0,1\}^+$ that satisfies that for no two distinct $x, y \in U$, it holds that $\mathrm{enc}(x)$ is a prefix of $\mathrm{enc}(y)$. For a set $S \subseteq U$, we let $\mathrm{enc}(S) := \{\mathrm{enc}(x) : x \in S\}$ be the set of bitstrings obtained from encoding S via enc. We say that enc is *ordered* if and only if $x < y$ implies $\mathrm{enc}(x) \prec \mathrm{enc}(y)$ for all $x, y \in U$.

We study the following *trie measure*, generalized from the particular cases studied in [10]:

▶ **Definition 1.** *Let* $\mathrm{enc} : U \to \{0,1\}^+$ *be a prefix-free encoding and let* $S = \{x_1, \dots, x_m\} \subseteq U$ *such that* $i < j$ *implies* $\mathrm{enc}(x_i) \prec \mathrm{enc}(x_j)$ *in lexicographic order. We define*

$$\mathrm{trie}(\mathrm{enc}(S)) = |\,\mathrm{enc}(x_{i_1})| + \sum_{j=2}^{m} |\,\mathrm{enc}(x_{i_{j-1}}) \oplus \mathrm{enc}(x_{i_j})|.$$

In this article we work, more generally, with sequences of sets. The above definitions generalize naturally:

▶ **Definition 2.** *Let* $\mathrm{enc} : U \to \{0,1\}^+$ *be a prefix-free encoding and let* $\mathcal{S} = \langle S_1, \dots, S_n \rangle$ *be a sequence of subsets of* U*. We define* $\mathrm{enc}(\mathcal{S})$ *to be the sequence* $\langle \mathrm{enc}(S_1), \dots, \mathrm{enc}(S_n) \rangle$*, and:*

$$\mathrm{trie}(\mathrm{enc}(\mathcal{S})) = \sum_{i=1}^{n} \mathrm{trie}(\mathrm{enc}(S_i))$$

Throughout the article, we will denote with $N = \sum_{i=1}^{n} |S_i|$ the total cardinality of the input sets $S_1, \dots, S_n$. Definition 2 leads to the central problem explored in this paper, tackled in Sections 3 and 4: given a sequence $\mathcal{S} = \langle S_1, \dots, S_n \rangle$ of n subsets of U, find the encoding enc minimizing $\mathrm{trie}(\mathrm{enc}(\mathcal{S}))$, possibly focusing on particular sub-classes of prefix-free encodings.

A particularly interesting such sub-class, originally introduced by Gupta et al. [10], is that of *shifted encodings*:

▶ **Definition 3.** *Let* $S \subseteq U$ *and* $a \in U$*. Notation* $S + a$ *denotes the application to* S *of the prefix-free encoding sending each* $x \in U$ *to* $(x + a) \bmod u$*, interpreted as a bit-string of* $\log u$ *bits. Similarly, for a sequence* $\mathcal{S} = \langle S_1, \dots, S_n \rangle$ *of subsets of* U *we denote by* $\mathcal{S} + a$ *the sequence* $\langle S_1 + a, \dots, S_n + a \rangle$.

We call $\mathrm{trie}(\mathcal{S} + a)$ the *shifted trie measure.* In Section 3 we study the problem of finding the value $a \in U$ minimizing $\mathrm{trie}(\mathcal{S} + a)$, a problem left open by Gupta et al. [10].

2.1 An Equivalent Problem Formulation

Observe that our trie-encoding problem is equivalent to the following natural encoding problem, which we will resort to in Section 4. Let $\mathcal{S} = \langle S_1, \dots, S_n \rangle$ be a sequence of subsets of U. For each $x \in U$, denote with $A_x = \{i \in [n] \ : \ x \in S_i\}$ the set collecting all indices i of the subsets S_i containing x. Without loss of generality, in this reformulation we assume that $A_x \neq \emptyset$ for all $x \in U$ and $S_i \neq \emptyset$ for all $i \in [n]$ (otherwise, simply re-map integers and ignore empty sets S_i). We furthermore do not require u to be a power of two (this will be strictly required only for the optimal shifted encoding in Section 3).

For a given prefix-free encoding enc, let T^{enc} be the binary trie storing the encodings $\mathrm{enc}(0), \dots, \mathrm{enc}(u-1)$ such that, for every $x \in U$, the leaf of T^{enc} reached by $\mathrm{enc}(x)$ is labeled with x. For any binary trie T with leaves labeled by integers let moreover: $r(T)$ be the root of T, T_v denote the subtree of T rooted at node v, and $L(T_v)$ be the set of (integers labeling the) leaves of subtree T_v (i.e. the leaves below node v). We overload notation and identify with T also the set of T's nodes (the use will be always clear by the context). Observe that the definition of the sets A_x allows us to reformulate the measure $\mathrm{trie}(\mathrm{enc}(\mathcal{S}))$ as follows:

$$\mathrm{trie}(\mathrm{enc}(\mathcal{S})) = \sum_{i \in [n]} \mathrm{trie}(\mathrm{enc}(S_i)) = \sum_{v \in T^{\mathrm{enc}} \setminus \{r(T^{\mathrm{enc}})\}} \Big| \bigcup_{x \in L(T_v^{\mathrm{enc}})} A_x \Big|.$$

In other words, the cost of node v is equal to the cardinality of the union of the sets $A_{x_1}, \ldots, A_{x_t}$ corresponding to the leaves $x_1, \ldots, x_t$ below v. The overall cost of the tree is the sum of the costs of its nodes, excluding the root. To see why this formulation is equivalent to the previous one observe that, among the n tries for $\mathrm{enc}(S_1), \ldots, \mathrm{enc}(S_n)$, (a copy of) the incoming edge of v is present in the trie for $\mathrm{enc}(S_i)$ for all $i \in \bigcup_{x \in L(T_v)} A_x$. See Figure 3.

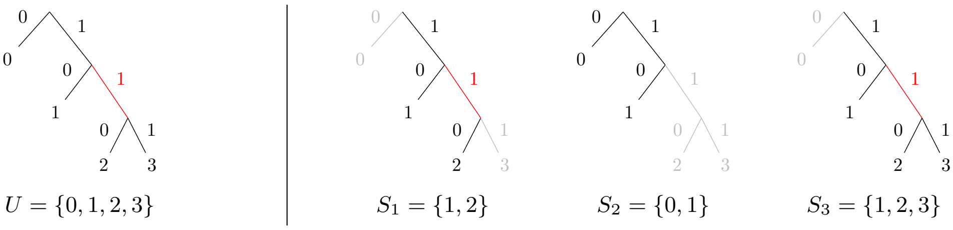


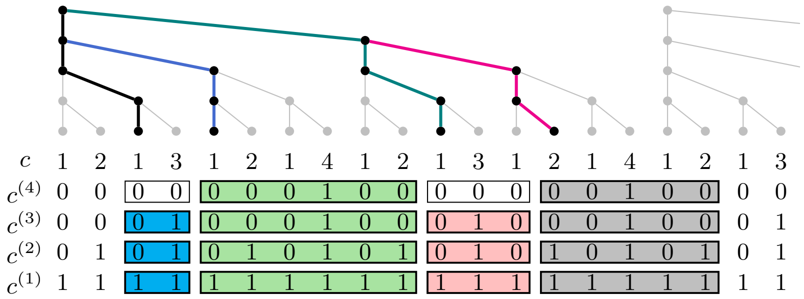
Figure 3 Left. An ordered prefix-free encoding enc of the universe $U = \{0,1,2,3\}$, represented as a binary trie T^{enc}. This encoding is used (right part) to encode sets $S_1 = \{1,2\}, S_2 = \{0,1\}, S_3 = \{1,2,3\}$. The corresponding sets A_x are: $A_0 = \{2\}, A_1 = \{1,2,3\}, A_2 = \{1,3\}, A_3 = \{3\}$. Highlighted in red, an edge leading to a node v with $\cup_{x \in T_v^{\mathrm{enc}}} A_x = A_2 \cup A_3 = \{1,3\}$, meaning that the tries for (the encoded) S_1 and S_3 will contain a copy of the same edge. **Right.** Using the prefix-free encoding enc to encode S_1, S_2, S_3 by packing their codes into three tries (gray edges do not belong to the tries and are shown only for completeness). The tries for S_1, S_2, S_3 contain in total 12 edges, so $\mathrm{trie}(\mathrm{enc}(\langle S_1, S_2, S_3 \rangle)) = 12$. In red: the two copies of the red edge on the left part of the figure, highlighting the equivalence of the two formulations of our trie-encoding problem. As a matter of fact, this is an optimal ordered code.

As there is a one-to-one relation between the set of all prefix-free binary encodings of the integers U and the set of all binary trees with leaves U, we can conclude that the problem of finding a prefix-free encoding enc that minimizes $\mathrm{trie}(\mathrm{enc}(\mathcal{S}))$ can equivalently be seen as the problem of finding a binary tree T with u leaves (labeled with the universe elements $0, \ldots, u-1$, not necessarily in this order) that minimizes the cost function $c(T) := \sum_{v \in T \setminus \{r(T)\}} |\bigcup_{x \in L(T_v)} A_x|$. We note that this problem is similar to the standard problems of (i) finding optimal binary search trees on a set of keys with given frequencies and (ii) finding the optimal prefix-free encoding for a source of symbols with given frequencies. As a matter of fact, if the sets $A_0, \ldots, A_{u-1}$ are disjoint then our problem is equivalent to (i-ii) and Huffman's algorithm finds the optimal solution.

Observe that this reformulation of the problem is equivalent, in turn, to the following optimization problem: given u sets $A_0, \ldots, A_{u-1}$, find the minimum-cost sequence of (binary) set unions that merges all sets into $\cup_{i=0}^{u-1} A_i$, where the cost of merging sets A and A' is $|A| + |A'|$. Ghosh et al. in [9] proved this problem to be NP-hard, and provided tight approximations. This motivates us to study less general families of prefix-free encodings, which hopefully can be optimized more efficiently.

3 Optimal Shifted Encoding

Given a sequence $\mathcal{S} = \langle S_1, \cdots, S_n \rangle$ of n subsets of U, in this section we study the problem of finding an optimal shift $a \in U$ that minimizes $\mathrm{trie}(\mathcal{S} + a)$. After finding a useful reformulation of $\mathrm{trie}(\mathcal{S} + a)$, we describe an algorithm for finding an optimal shift a. The algorithm is parameterized on an abstract data structure for integer sequences. Using a simple array, we obtain an $O(u + N \log u)$-time algorithm. A DAG-compressed segment tree, on the other hand, gives an $O(N \log^2 u)$-time algorithm.



c	1	2	1	3	1	2	1	4	1	2	1	3	1	2	1	4	1	2	1	3
$c^{(4)}$	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0
$c^{(3)}$	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1
$c^{(2)}$	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
$c^{(1)}$	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Figure 4 The nodes in the trie of $S = \{x_1, x_2, x_3, x_4\} = \{2, 4, 10, 13\}$ in universe $u = 16$. The i-th box on each row spans range $[x_i, x_{i+1})$, with $x_5 := x_1 + u$. The number of edges in the trie of the set is equal to the number of shaded boxes that contain at least one 1-bit. The shaded boxes correspond to the edges with the same color.

3.1 Trie measure as a function of the shift a

Let $S = \{x_1, x_2, \ldots, x_m\} \subseteq U$ be a non-empty integer set with $0 \leq x_1 < x_2 < \cdots < x_m < u$. We assume $\mathrm{enc}(x)$ is the standard $(\log u)$-bit binary representation of x throughout this section. Observe that for every non-negative integers $0 \leq x < y < u$, it holds that

$$|\,\mathrm{enc}(x) \oplus \mathrm{enc}(y)| = \max_{j \in [x,y)} |\,\mathrm{enc}(j) \oplus \mathrm{enc}(j+1)|. \tag{1}$$

Consider the (infinite) sequence $\langle c_j \rangle_{j \geq 0}$ with $c_j = |\,\mathrm{enc}(j \bmod u) \oplus \mathrm{enc}((j+1) \bmod u)|$. This sequence is the infinite copy of the first $u/2$ elements (see the first row in Figure 4) of the sequence known as "ruler function" (OEIS sequence A001511[1]). This sequence can be decomposed into the sum of $\log u$ periodic binary sequences. For integers $k \in [\log u]$ and $j \geq 0$, let us define $c_j^{(k)}$ as follows.

$$c_j^{(k)} = \begin{cases} 1 & \text{if } j+1 \text{ is a multiple of } 2^{k-1}, \\ 0 & \text{otherwise.} \end{cases}$$

Then, for every $j \geq 0$, it holds that $c_j = \sum_{k=1}^{\log u} c_j^{(k)}$. Together with (1), we obtain

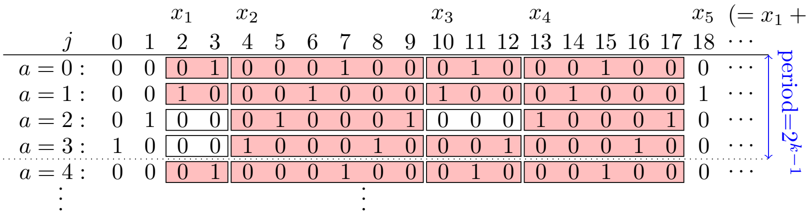
$$|\,\mathrm{enc}(x) \oplus \mathrm{enc}(y)| = \max_{j \in [x,y)} \sum_{k=1}^{\log u} c_j^{(k)} = \sum_{k=1}^{\log u} \max_{j \in [x,y)} c_j^{(k)}, \tag{2}$$

where we used that the maximum and the sum are interchangeable since, for every $k > 1$, if $c_j^{(k)} = 1$ then $c_j^{(k-1)} = 1$ by definition.

Recalling that the value of $c_j^{(k)}$ is either 0 or 1, this representation allows us to represent the cost (i.e. the number of edges) of a trie for $S = \{x_1, \cdots, x_m\}$ as follows. Consider the $\log u$ binary sequences $(c_j^{(k)})_{j \geq 0}$ for $k \in [\log u]$. For each sequence, and for every $1 \leq i \leq m$, consider a range $[x_i, x_{i+1})$ of indices of the sequence $(c_j^{(k)})_{j \geq 0}$ where we define $x_{m+1} = x_1 + u$. For each x_{i+1} with $i \in [m-1]$, observe that adding x_{i+1} to the trie containing the integers $\{x_1, x_2, \cdots, x_i\}$ creates a new edge at level k if and only if $\max_{j \in [x_i, x_{i+1})} c_j^{(k)} = 1$. Notice also that x_1 always creates $\log u$ edges forming the left-most path of the trie; at the same time, it always holds that $\max_{j \in [x_m, x_{m+1})} c_j^{(k)} = 1$ for all $k \in [\log u]$ because $c_{u-1}^{(k)} = 1$ for

---

[1] `https://oeis.org/A001511`



			x_1		x_2						x_3			x_4					x_5	$(= x_1 + u)$
j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	$\cdots$
$a = 0:$	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	$\cdots$
$a = 1:$	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	$\cdots$
$a = 2:$	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	$\cdots$
$a = 3:$	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	$\cdots$
$a = 4:$	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	$\cdots$
$\vdots$								$\vdots$												

Figure 5 Representation of trie$(S + a)$ based on $c_j^{(k)}$ for $S = \{2, 4, 10, 13\}$ at level $k = 3$ and $u = 16$. Each row represents $c_{j+a}^{(3)}$, for $a \in U$. Boxes indicate ranges $[x_i, x_{i+1})$ and red boxes contain at least one 1. As an example, consider the pair x_1, x_2 (leftmost boxes). Among the shifts $0, \ldots, 4$, the only shifts for which we do not pay the cost of $k = 3$ for this pair are $a = 2$ and $a = 3$; i.e., $x_1 + a$ and $x_2 + a$ share an edge at level 3 for $a = 2, 3$ in trie$(S + a)$. Hence, in the figure we have two non-red boxes in the rows corresponding to $a = 2, 3$.

every $k \in [\log u]$. See Figure 4 for an example. More generally, the cost of a trie shifted by a with $a \geq 0$ can be represented as in the following lemma whose proof is deferred to Appendix A.1.

▶ **Lemma 4.** *Let $S = \{x_1, \cdots, x_m\}$ be a set of m integers with $0 \leq x_1 < \cdots < x_m < u$. Let us define $x_{m+1} := x_1 + u$. For every $a \in U$, it holds that*

$$\text{trie}(S + a) = \sum_{k=1}^{\log u} \sum_{i=1}^{m} \max_{j \in [x_i + a, x_{i+1} + a)} c_j^{(k)}$$

Lemma 4 allows us to compute trie$(S + a)$ by summing over $k \in [\log u]$ the number of the shifted ranges $[x_i + a, x_{i+1} + a)$ (for $i \in [m]$) such that $\max_{j \in [x_i + a, x_{i+1} + a)} c_j^{(k)} = 1$; see Figure 5 for an example. The following lemmata give an illustration of which values of a involve a cost of 1 with a range $[x_i, x_{i+1})$.

▶ **Lemma 5.** *Let $k \in [\log u]$, and $x, y \in [0, 2u)$ with $x + 2^{k-1} \leq y$. Then it holds for every $a \in U$ that $\max_{j \in [x+a, y+a)} c_j^{(k)} = 1$.*

Proof. Immediate from that any interval of length 2^{k-1} contains a multiple of 2^{k-1}. ◀

▶ **Lemma 6.** *For $a \in U$, $k \in [\log u]$, and $x, y \in [0, 2u)$ with $x < y$, it holds that $\max_{j \in [x+a, y+a)} c_j^{(k)} = 1$ if and only if $a \equiv_{2^{k-1}} b$ for some $b \in [2u - y, 2u - x)$.*

Proof. Recall that $\max\{c_j^{(k)} : j \in [x+a, y+a)\} = 1$ if and only if there exists $j \in [x+a, y+a)$ such that $j + 1$ is a multiple of 2^{k-1} by definition of $c_j^{(k)}$. Assume that $j \in [x + a, y + a)$ is such that $j + 1$ is a multiple of 2^{k-1}. Equivalently, $j = t \cdot 2^{k-1} - 1$ for some integer t and $x + a < t \cdot 2^{k-1} \leq y + a$. The latter is equivalent to $a \in [t \cdot 2^{k-1} - y, t \cdot 2^{k-1} - x)$. This is then equivalent to $a \equiv_{2^{k-1}} b$ for some $b \in [-y, -x)$. Using that $2u$ is a multiple of 2^{k-1} and $k \in [\log u]$, this is in turn equivalent to $a \equiv_{2^{k-1}} b$ for some $b \in [2u - y, 2u - x)$. ◀

Considering $\max_{j \in [x_i + a, x_{i+1} + a)} c_j^{(k)}$ as a function of a, observe that it has a period of 2^{k-1} because of the periodicity of $c_j^{(k)}$. This means that we do not need to consider all the values of $a \in U$ but we can consider only $a \in [0, 2^{k-1})$. For each pair of consecutive elements x_i and x_{i+1} for $i \in [m]$, let us define $I_k(x_i, x_{i+1})$ as:

$$I_k(x_i, x_{i+1}) = \begin{cases} [0, 2^{k-1}) & \text{if } x_{i+1} - x_i \geq 2^{k-1}, \\ [\ell, r) & \text{if } \ell < r, \\ [0, r) \cup [\ell, 2^{k-1}) & \text{otherwise.} \end{cases} \tag{3}$$

where $\ell := (2u - x_{i+1}) \bmod 2^{k-1}$, and $r := (2u - x_i) \bmod 2^{k-1}$. Note that $I_k(x_i, x_{i+1})$ can be represented with one or two intervals.

Let $\mathcal{I}_k$ be the multiset $\mathcal{I}_k = \{\!\{ I_k(x_i, x_{i+1}) \ : \ i \in [m] \}\!\}$. Consider the number of edges of the trie at level k for the shifted set $S + a$ for a specific point $a \in U$. By the inner sum of Lemma 4 along with Lemma 5 and Lemma 6, this number is equal to the number of members of $\mathcal{I}_k$ containing a specific point $a \bmod 2^{k-1}$, i.e., to the *depth* of $\mathcal{I}_k$ at position $a \bmod 2^{k-1}$. In the following lemma, we show that $\mathrm{trie}(S + a)$ can be expressed as the sum over $k \in [\log u]$ of the depth of $\mathcal{I}_k$ at position $a \bmod 2^{k-1}$.

▶ **Lemma 7.** *Let $S = \{x_1, x_2, \cdots, x_m\}$ be a set of integers with $0 \le x_1 < x_2 < \cdots < x_m < u$. Let $x_{m+1} := x_1 + u$ and $a \in U$. Then,*

$$\mathrm{trie}(S + a) = \sum_{k=1}^{\log u} \sum_{i=1}^{m} \mathbb{1}[(a \bmod 2^{k-1}) \in I_k(x_i, x_{i+1})].$$

Proof. According to Lemma 4, we have $\mathrm{trie}(S + a) = \sum_{k=1}^{\log u} \sum_{i=1}^{m} \max_{j \in [x_i + a, x_{i+1} + a)} c_j^{(k)}$. Hence it is sufficient to show that $a \bmod 2^{k-1} \in I_k(x_i, x_{i+1})$ iff $\max_{j \in [x_i + a, x_{i+1} + a)} c_j^{(k)} = 1$. We distinguish two cases: (Case 1) If $x_{i+1} - x_i \ge 2^{k-1}$, we have $I_k(x_i, x_{i+1}) = [0, 2^{k-1})$ and thus clearly $a \bmod 2^{k-1} \in I_k(x_i, x_{i+1})$. At the same time $\max_{j \in [x_i + a, x_{i+1} + a)} c_j^{(k)} = 1$ by Lemma 5 for any $a \in U$. (Case 2) Assume $x_{i+1} - x_i < 2^{k-1}$. Let $\ell = (2u - x_{i+1}) \bmod 2^{k-1}$ and $r = (2u - x_i) \bmod 2^{k-1}$ as in Eq. (3). By Lemma 6, it holds that $\max_{j \in [x_i + a, x_{i+1} + a)} c_j^{(k)} = 1$ if and only if $a \equiv_{2^{k-1}} b$ for some $b \in [2u - x_{i+1}, 2u - x_i)$. The latter is equivalent to:

$$a \equiv_{2^{k-1}} b \text{ for some } b \in [\ell, \ell + x_{i+1} - x_i) \tag{4}$$

We have two cases. (Case 2a) Assume that $\ell < r$ holds. Since $x_{i+1} - x_i < 2^{k-1}$, $\ell < r$ if and only if $\ell + x_{i+1} - x_i < 2^{k-1}$. Observing that $r = (\ell + x_{i+1} - x_i) \bmod 2^{k-1}$, we have $\ell + x_{i+1} - x_i = r$. (Case 2b) Now assume that $\ell \ge r$. Then Eq. (4) is equivalent to $a \equiv_{2^{k-1}} b$ with $b \in [\ell, 2^{k-1})$ or $b \in [2^{k-1}, r + 2^{k-1})$, which can be rewritten as $a \equiv_{2^{k-1}} b$ with $b \in [0, r)$ or $b \in [\ell, 2^{k-1})$. This completes the proof. ◀

Lemma 7 can be naturally generalized to a sequence of sets $\mathcal{S} + a = \langle S_1 + a, \cdots, S_n + a \rangle$. Together with Definition 2, we obtain:

$$\mathrm{trie}(\mathcal{S} + a) = \sum_{i=1}^{n} \mathrm{trie}(S_i + a) = \sum_{k=1}^{\log u} \sum_{i=1}^{n} \sum_{j=1}^{|S_i|} \mathbb{1}[(a \bmod 2^{k-1}) \in I_k(x_j^{(i)}, x_{j+1}^{(i)})]. \tag{5}$$

where $x_j^{(i)}$ is the j-th smallest element of S_i. In the following subsection, we develop algorithms to find an optimal shift $a \in [0, u)$ minimizing $\mathrm{trie}(\mathcal{S} + a)$ using this formulation.

3.2 Algorithms for the optimal shift

Let $S_1, \cdots, S_n \subseteq U$ be n sets of integers. For $i \in [n]$ and $j \in [|S_i|]$, let $x_j^{(i)}$ denote the j-th smallest element of S_i. For $k \in [\log u]$, let $D_k[0..2^{k-1})$ be a sequence of length 2^{k-1} such that, for $a \in [0, 2^{k-1})$,

$$D_k[a] = \sum_{i=1}^{n} \sum_{j=1}^{|S_i|} \mathbb{1}[(a \bmod 2^{k-1}) \in I_k(x_j^{(i)}, x_{j+1}^{(i)})]. \tag{6}$$

Algorithm 1 General algorithm for finding an optimal shift of $S_1, \cdots, S_n$.

```
1 $\mathcal{D}$.initialize()          // Initialize and start with universe of size $2^0$
2 for $k = 1..\log u$ do
3     for $i = 1..n$ do
4         for $j = 1..|S_i|$ do
5             for each interval $[\ell, r)$ forming $I_k(x^{(i)}_j, x^{(i)}_{j+1})$ do
6                 $\mathcal{D}$.add($\ell, r$)                      // Add interval segments
7     $\mathcal{D}$.extend()                    // Extend the universe size into $[0, 2^k)$
8 return $\mathcal{D}$.argmin()
```

Observe that $D_k[a]$ is defined as the two inner sums of Eq. (5); in other words, $D_k[a]$ is the number of interval segments at level k that contain a specific position $a \in [0, 2^{k-1})$. To consider the cumulative sum of the number of interval segments that contain position a up to level $k \in [\log u]$, let C_k be a sequence of length 2^k that, for $a \in [0, 2^k)$, is defined as

$$C_k[a] = \sum_{k'=1}^{k} D_{k'}[a \bmod 2^{k'-1}]. \tag{7}$$

Then, by Equations (5) and (6) it holds that $\mathrm{trie}(\mathcal{S} + a) = C_{\log u}[a]$ for every $a \in U$. Therefore, if we can compute C_k in an efficient way, this will allow us to find an optimal shift by computing $\arg\min_{a \in U} C_{\log u}[a]$. To do this, now consider an abstract data type $\mathcal{D}$ that supports the following four operations:

1. $\mathcal{D}$.`initialize`(): create a sequence A of length 1 and initialize it as $A[0] \leftarrow 0$.
2. $\mathcal{D}$.`add`(ℓ, r): update $A[a] \leftarrow A[a] + 1$ for $a \in [\ell, r)$.
3. $\mathcal{D}$.`extend`(): duplicate the sequence as $A \leftarrow AA$.
4. $\mathcal{D}$.`argmin`(): return $\arg\min_{0 \le a < |A|} A[a]$.

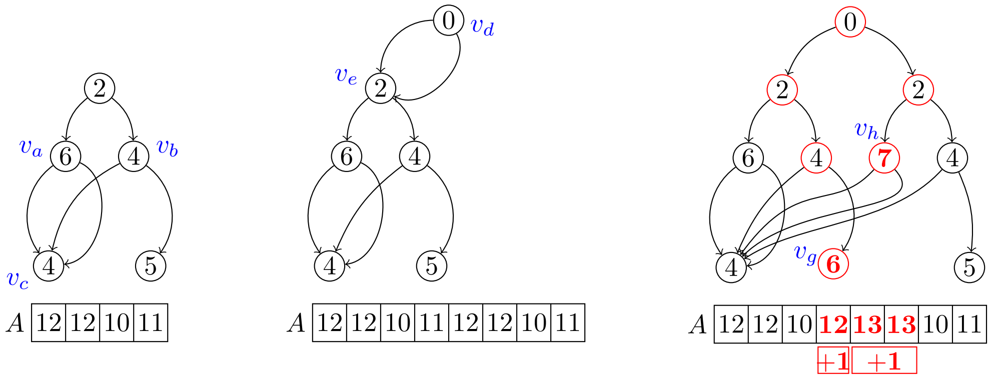
Based on these operations, in Algorithm 1 we describe a general procedure to find an optimal shift. For each level $k \in [\log u]$ and every set S_i, we iterate on every pair of consecutive elements $x^{(i)}_j, x^{(i)}_{j+1}$. In Line 5, we use the fact that $I_k(\cdot)$ can be represented as the union of at most two intervals on U, see Eq. (3). We increment $A[a]$ by 1 for every $a \in [\ell, r)$ by calling $\mathcal{D}$.`add`(ℓ, r) (Line 6). After processing all consecutive pairs, we extend the universe into $[0, 2^k)$ (Line 7). To see that the algorithm is correct, observe that the amount by which $A[a]$ is incremented in the three inner loops equals $D_k[a]$ according to Eq. (6). Eq. (7) yields that, for $k \in [\log u]$ and $a \in [0, 2^{k-1})$, the number C_k can be written recursively as

$$C_k[a + 2^{k-1}] = C_k[a] = C_{k-1}[a] + D_k[a], \tag{8}$$

where we define $C_0 := \langle 0 \rangle$ as the base case. Then, at the end of the k-th iteration of the outer for loop (after Line 7), the sequence A represented by $\mathcal{D}$ is exactly C_k. The running time depends on how $\mathcal{D}$ is implemented. In the following two subsections, we will present two different ways of implementing the data structure $\mathcal{D}$. These allow us to find an optimal shift in $O(u + N \log u)$ with simple arrays and $O(N \log^2 u)$ time with a dynamic DAG structure.

3.2.1 $O(u + N \log u)$-time algorithm

Our first solution is simple: we represent A implicitly by storing an array $\Delta[0..|A| - 1]$ of length $|A|$, encoding the differences between adjacent elements of A. At the beginning, Δ is initialized as an array of length 1 containing the integer 0. Operation $\mathcal{D}$.`add`(ℓ, r) is



$\mathcal{D}$ → $\mathcal{D}$.`extend`() → $\mathcal{D}$.`add`$(3, 6)$

Figure 6 DAG-compressed representation of A. (Left) Each element $A[i]$ is represented as the sum of the values stored in its corresponding root-to-leaf path. The two children of v_a and the left child of v_b are represented by a single node v_c. (Middle) Duplicating the whole content can be performed by creating a new root (v_d) with the old root (v_e) being referred as both of its left and right children. (Right) Incrementing $A[i]$ by 1 for each $i \in [3, 6) = [3, 5]$ is performed by incrementing the values in v_g and v_h by 1 each, which covers $[3, 3]$ and $[4, 5]$, respectively. We duplicate every node that has more than one incoming edges when it is visited so that the visited nodes (indicated with red nodes) should have unique paths from the root.

implementing by incrementing $\Delta[\ell]$ by 1 unit and (if $r < |\Delta|$) decrementing $\Delta[r]$ by 1 unit, in $O(1)$ time (Lines 4–5 of Algorithm 3 in Appendix A.2). As far as operations $\mathcal{D}$.`extend`() and $\mathcal{D}$.`argmin`() are concerned, they can be supported naively in linear $O(|\Delta|) = O(|A|)$ time with a constant number of scans of array Δ. See Algorithm 3 for the details.

Next, we analyze the running time of Algorithm 1 when using this simple implementation of $\mathcal{D}$. Operation $\mathcal{D}$.`add`() is called $O(N \log u)$ times, each of which costs $O(1)$ time. Operation $\mathcal{D}$.`extend`() is called $\log u$ times, and each call costs $O(|A|)$ time. Each time this operation is called, the length of A doubles (starting from $|A| = 1$). The overall cost of these calls is therefore $O(1 + 2 + 4 + \cdots + u) = O(u)$. Finally, $\mathcal{D}$.`argmin`() is called only once when $|A| = u$, and therefore it costs $O(u)$ time. We conclude that, using this implementation of $\mathcal{D}$, Algorithm 1 runs in $O(u + N \log u)$ time. Within this running time we actually obtain a much more general result, since we can evaluate any entry of A:

▶ **Theorem 8.** *Given a sequence $\mathcal{S} = \langle S_1, \cdots, S_n \rangle$ of n subsets of U, we can compute* $\mathrm{trie}(\mathcal{S} + a)$ *for all $a \in U$ in total $O(u + N \log u)$ time.*

3.2.2 $O(N \log^2 u)$-time algorithm

If u is too large compared to N, then the simple solution presented in the previous subsection is not efficient. In this case, we instead represent A with a DAG-compressed variant of the segment tree [4, 11]; see Algorithm 4 in Appendix A.3 for the details. Intuitively, consider the complete binary tree of height $\log |A|$ where the root stores a counter associated to the whole array A, its left/right children store a counter associated to $A[0, |A|/2 - 1]$ and $A[|A|/2, |A| - 1]$, respectively, and so on recursively (up to the leaves, which cover individual values of the array). Assume that these counters are initialized to 0. Observe that for every $0 \le \ell < r \le |A|$, there exists the coarsest partition of $A[\ell, r - 1]$ into $O(\log |A|)$ disjoint intervals covered by nodes of the tree. The idea is to support $\mathcal{D}$.`add`(ℓ, r) by incrementing by

1 unit the counter associated to those nodes. To avoid spending time $O(|A|)$ for operation $\mathcal{D}.\texttt{extend}()$, however, we cannot afford duplicating the whole tree when this operation is called. In this case, our idea is simple: since $\mathcal{D}.\texttt{extend}()$ duplicates the whole content of A, in $O(1)$ time we simply create a new node covering AA and make both its two outgoing edges (left/right child) lazily point to the node covering A, i.e., the old root of the tree. In other words, we represented the tree as a DAG, collapsing nodes covering identical sub-arrays. This modification makes it necessary to (possibly) duplicate at most $O(\log|A|)$ nodes at each call of $\mathcal{D}.\texttt{add}(\ell, r)$: a duplication happens when, starting from the root, we reach a node x having more than one incoming edges. Without loss of generality, assume that we are moving from y that covers sub-array $A[i, i+2^{k+1}-1]$ to its left child. Since x has more than one incoming edges, $A[i, i+2^k-1]$ is not the unique interval that x covers. We create a new node x' by duplicating x, then make x' the left child of y so that $A[i, i+2^k-1]$ is the unique interval that x' covers. Then proceed to x'. Since $A[\ell, r-1]$ is covered by $O(\log|A|)$ nodes, starting this procedure at the root duplicates at most $O(\log|A|)$ nodes of the DAG and operation $\mathcal{D}.\texttt{add}(\ell, r)$ therefore costs $O(\log|A|) \subseteq O(\log u)$ time. This slow-down (with respect to the $O(1)$ cost of the same operation in the previous subsection) is paid off by the fact that we do not need to create $O(|A|)$ new nodes (i.e. duplicate the tree) at each call of $\mathcal{D}.\texttt{extend}()$. With this representation, the operation $\mathcal{D}.\texttt{argmin}()$ can be implemented in $O(1)$ time as well by simply associating to each node the smallest value in the sub-array of A covered by that node, as well as its index (these values are updated inside $\mathcal{D}.\texttt{add}(\ell, r)$). Operations $\mathcal{D}.\texttt{initialize}()$ and $\mathcal{D}.\texttt{extend}()$ trivially take $O(1)$ time.

Plugging this structure into Algorithm 1, observe that the running time is dominated by $\mathcal{D}.\texttt{add}(\cdot)$, which is called $O(N\log u)$ times. Since each call to this function takes $O(\log u)$ time as discussed above, we finally obtain:

▶ **Theorem 9.** *Given a sequence $\mathcal{S} = \langle S_1, \cdots, S_n\rangle$ of n subsets of U, we can compute one value a minimizing* $\text{trie}(\mathcal{S}+a)$ *in $O(N\log^2 u)$ time.*

Finally, we remark that using more elaborate arguments one can obtain an algorithm running in $O(N\log u)$ time. Due to space limitations, we refrain from detailing this approach here, but will include it in the extended version of the article.

4 Optimal Ordered Encoding

To compute the optimal *ordered encoding*, we employ the equivalent problem formulation of Section 2.1. For any $x \in U$, let $A_x = \{i \in [n] \ : \ x \in S_i\}$. Recall that we can assume, w.l.o.g., that $A_x \neq \emptyset$ for all $x \in U$ and $S_i \neq \emptyset$ for all $i \in [n]$. Our goal is to find the binary tree T with leaves $0, \dots, u-1$ (in this order) minimizing $c(T) := \sum_{v \in T\setminus\{r(T)\}} |\bigcup_{x\in L(T_v)} A_x|$. For a binary tree T, let T_ℓ and T_r be the left and right sub-tree of the root of T, respectively. We show that the dynamic programming solution of Knuth [12] can be applied to our scenario. Consider the following alternative cost function: $d(T) := \left|\bigcup_{x\in L(T)} A_x\right| + d(T_\ell) + d(T_r)$. The (recursive) function $d(T)$ is related to $c(T)$ by the following equality: $c(T) = d(T) - |\bigcup_{x\in L(T)} A_x|$. It is not hard to turn the definition of $d(T)$ into a set of dynamic programming formulas computing both the best tree T and its cost $d(T)$ by creating, for every $0 \le x \le y < u$, a variable $d_{x,y}$ whose final value will be $\min_{T'} d(T')$ (where T' runs over all trees such that $L(T') = \{x, x+1, \dots, y\}$), and a pre-computed constant $a_{x,y} = \left|\bigcup_{t=x}^{y} A_t\right|$. Given the constants $a_{x,y}$, the following set of dynamic programming formulas finds (bottom up, i.e., by increasing $y-x$) the value $c(T) = d(T) - |\bigcup_{x\in L(T)} A_x| = d_{0,u-1} - a_{0,u-1}$ for the optimal tree T and, by standard backtracking, the tree T itself:

$$\begin{array}{rlll} \text{initialization:} & a_{x,y} := & \left|\bigcup_{t=x}^{y} A_t\right| & \text{for all } 0 \leq x \leq y \leq u-1 \\ & d_{x,x} := & a_{x,x} & \text{for all } x \in U \\ \text{recursion:} & d_{x,y} := & a_{x,y} + \min_{x<z\leq y} d_{x,z-1} + d_{z,y} & \text{for all } 0 \leq x < y \leq u-1 \end{array}$$

The optimal tree's topology can be retrieved by backtracking: if $L(T') = \{x, x+1, \dots, y\}$, then the number of leaves in T'_ℓ is equal to $(\text{argmin}_{x<z\leq y}\, d_{x,z-1} + d_{z,y}) - i$ (we start from the root of T with $x = 0$ and $y = u-1$). Below, we show how to compute constants $a_{x,y}$ in $O(N + u^2)$ time. Since the above formulas can be evaluated bottom-up (i.e. by increasing $y - x$) in $O(u^3)$ time[2], the overall running time of the algorithm is $O(N + u^3)$.

We now show how to compute efficiently the constants $a_{x,y}$. We describe an overview of the algorithm (see Algorithm 2 for the pseudocode). We add to the sequence two sets $A_{-1} = A_u = [n]$, respectively at the beginning and end: the new sequence becomes $A_{-1}, A_0, \dots, A_{u-1}, A_u$. The main idea behind the algorithm is to initially compute inside $a_{x,y}$ the number of elements from $[n]$ that are *missing* from $\left|\bigcup_{t=i}^{j} A_t\right|$. Then, the substitution $a_{x,y} \leftarrow n - a_{x,y}$ will yield the final result.

We initialize $a_{x,y} \leftarrow 0$ for all $0 \leq x, y \leq u$. Let $A_{x+1}, A_{x+2}, \dots, A_{y-1}$ be a maximal contiguous subsequence of sets not containing a given $i \in [n]$, that is, (i) $i \notin A_{x'}$ for all $x' = x+1, x+2, \dots, y-1$, (ii) $i \in A_x$, and (iii) $i \in A_y$. Then, $i \notin \left|\bigcup_{t=x'}^{y'} A_t\right|$ for every $i \leq x' \leq y' \leq y$. Imagine then adding one unit to $a_{x',y'}$ for all $(x', y') \in [x+1, y-1] \times [x+1, y-1]$. Clearly, if we can achieve this for every $i \in [n]$ and every such maximal contiguous subsequence of sets not containing i, at the end each $a_{x,y}$ will contain precisely the value $n - \left|\bigcup_{t=x}^{y} A_t\right|$. The issue is, of course, that adding one unit to $a_{x',y'}$ for all $x < x' \leq y' < y$, costs time $O(xy)$ if done naively. The crucial observation is that this task can actually be performed in constant time using (bidimensional) *partial sums*: see Figure 7 for an example. Ultimately, this trick allows us computing all $a_{x,y}$ in the claimed $O(N + u^2)$ running time.

x \ y	0	1	2	3
0	1	0	−1	0
1	0	1	0	−1
2	−1	0	1	0
3	0	−1	0	1

x \ y	0	1	2	3
0	0	−1	−1	0
1	0	0	−1	−1
2	0	1	1	0
3	0	0	1	1

x \ y	0	1	2	3
0	0	0	0	0
1	0	1	1	0
2	0	1	2	1
3	0	0	1	1

Figure 7 Matrix $a_{x,y}$. Suppose our goal is to add 1 unit to the two sub-matrices with indices $[1,2] \times [1,2]$ and $[2,3] \times [2,3]$. We show how to achieve this by performing 4 updates for each of these two submatrices, and then performing two scans (partial sums) of total cost $O(u^2)$ over the full matrix. Letting t be the number of sub-matrices to update (in this example, $t = 2$), this means that we can perform the t updates in total $O(t + u^2)$ time. Assume we start by a matrix $a_{x,y}$ containing only zeros. To update the sub-matrix with indices $[x', y'] \times [x', y']$, we perform these 4 operations: (1) $a_{y',y'} \leftarrow a_{y',y'} + 1$, (2) $a_{x'-1,x'-1} \leftarrow a_{x'-1,y'-1} + 1$, (3) $a_{x'-1,y'} \leftarrow a_{x'-1,y'} - 1$, and (4) $a_{y',x'-1} \leftarrow a_{y',x'-1} - 1$. **Left:** applying these four operations for each of the two sub-matrices $[1,2] \times [1,2]$ and $[2,3] \times [2,3]$. **Center**: we compute partial sums row-wise, cumulating from right to left. **Right**: we compute partial sums column-wise, cumulating bottom-up. At the end, we correctly added 1 unit to the target sub-matrices.

---

[2] In his work [12], Knuth shows how to evalutate these recursive formulas to $O(u^2)$ time. In the simpler case considered in his article (corresponding to our problem with pairwise-disjoint $A_0, \dots, A_{u-1}$), one can bound $\text{argmin}_{x<z\leq y}\, d_{x,z-1} + d_{z,y}$ so that the overall cost of looking for all those optimal values of z amortizes to u^2. Unfortunately, in our scenario we haven't been able to prove that the same technique can still be applied.

Algorithm 2 Compute $a_{x,y}$.

```
input  : Sets $A_0, \dots, A_{u-1} \subseteq [n]$ of total cardinality $N = \sum_{x=0}^{u-1} |A_x|$, s.t. for every
         $i \in [n]$ there exists $x \in U$ with $i \in A_x$.
output : $a_{x,y} = \left|\bigcup_{t=x}^{y} A_t\right|$, for each $0 \leq x \leq y < u$

Add dummy sets $A_{-1} = A_u = [n]$
for each $0 \leq x, y \leq u$ do $a_{x,y} \leftarrow 0$
$P[1, \dots, n] \leftarrow -1$  // Previous occurrence of $i \in [n]$, all initialized to $-1$

for $y = 0, \dots, u$ do
    for $i \in A_y$ do
        $x \leftarrow P[i]$        // $A_{x+1}, \dots, A_{y-1}$: maximal sequence not containing $i$
        $P[i] \leftarrow y$
        if $x < y - 1$ then
            $a_{y-1,y-1} \leftarrow a_{y-1,y-1} + 1$
            $a_{x,x} \leftarrow a_{x,x} + 1$
            $a_{x,y-1} \leftarrow a_{x,y-1} - 1$
            $a_{y-1,x} \leftarrow a_{y-1,x} - 1$

for $x = 0, \dots, u - 1$ do
    for $y = u - 2, \dots, 0$ do
        $a_{x,y} \leftarrow a_{x,y} + a_{x,y+1}$                // Partial sums, row-wise

for $y = 0, \dots, u - 1$ do
    for $x = u - 2, \dots, 0$ do
        $a_{x,y} \leftarrow a_{x,y} + a_{x+1,y}$             // Partial sums, column-wise

for each $0 \leq x, y < u$ do $a_{x,y} \leftarrow n - a_{x,y}$
return $a_{x,y}$ for all $0 \leq x \leq y < u$
```

4.1 Optimal Shifted Ordered Encoding

Observe that the optimal ordered encoding is not necessarily better than the optimal shifted encoding, because shifted encodings are not ordered (except the case $a = 0$). It is actually very easy to get the best of both worlds and compute the ordered encoding $\text{enc} : \{0,1\}^* \to \{0,1\}^*$ minimizing $\min_{a \in U} \text{trie}(\text{enc}(\mathcal{S} + a))$. This encoding is guaranteed to be no worse than *both* the best shifted encoding and the best ordered encoding.

The solution is a straightforward extension of the technique used for the best ordered encoding. Let $A_0, \dots, A_{u-1}$ be the sets defined previously. Build the sequence of sets $A_0, \dots, A_{u-1}, A_u, \dots, A_{2u-1}$, where $A_{u+x} = A_i$ for all $0 \leq x < u$ (that is, we simply create an extra copy of the sets). Compute $a_{x,y}$ and $d_{x,y}$ as discussed previously, with the only difference that now the domain of x, y is doubled: $0 \leq x, y < 2u$. It is not hard to see that the optimal shifted ordered encoding has then cost $c'(T) = \min_{0 \leq a < u}(d_{a,a+u-1} + a_{a,a+u-1}) = (\min_{0 \leq a < u} d_{a,a+u-1}) + a_{0,u-1}$. Again, the tree topology is obtained by backtracking starting from the optimal interval $[a', a'+u)$ given by $a' = \text{argmin}_{0 \leq a < u} d_{a,a+u-1}$ (the optimal ordered encoding discussed previously is simply the particular case $a' = 0$). The asymptotic cost of computing this optimal shifted ordered encoding is still $O(N + u^3)$.

5 Experimental Results

We implemented our algorithms for computing the optimal shifted encoding and the optimal shifted ordered encoding in `C++` and made them available at `https://github.com/regindex/trie-measure`. We computed the sizes of these encodings on thirteen datasets, the details of which can be found in Table 1 in Appendix 5. The datasets can be naturally split in four groups according to their origin: 1) eight sequences of sets from [3], 2) a dataset of paper tags from dblp.org xml dump [14], 3) a dataset containing amino acid sets from two protein sequence collections [16, 13], 4) and two datasets containing ratings and tags of 10000 popular books [15]. For the last three groups, we generated sequences of sets (i.e. the inputs of our algorithms) by extracting sets of features from the original datasets: in 2) we extracted sets of tags for all dblp entries, in 3) we extracted the set of amino acids contained in each protein sequence, and in the two datasets of group 4) we extracted a set of tags and ratings for each book, respectively. The repository above contains all the generated set sequences.

As far as the shifted trie measure $\text{trie}(\mathcal{S} + a)$ of Section 3 is concerned, we evaluated it on the above datasets for all shifts $a \in U$ and reported the following statistics in Table 2: the trie cost for the optimal and worst shifts, the average cost over all shifts, and the percentage differences between the average/worst shifts and the optimal shift. Our results indicate that the optimal, average, and worst shifts lead in practice to similar costs. In particular, only five datasets showed a difference larger than 5% between the optimal and worst shift (opt-shift/worst-shift(%) < 95). Among these, the differences range between 15.85% for `DBLP.xml` and 6.01% for `tags-math-sx-seqs`. The differences are even smaller when comparing the optimum with the average shift. In this case, only `DBLP.xml` shows an average difference (of 6.95%) being larger than 5% (opt-shift/avg-shift(%) < 95), meaning that the trie measure computed with a random shift on this dataset is, on average, 6.95% larger than with the optimal shift. These results suggest that data structures which encode integer sets as tries using the standard binary integer encoding (such as the subset wavelet tree of Alanko et al. [1]) are often efficient in practice since even arbitrary shifts allow to obtain an encoding being not far from the optimal shifted encoding.

As far as the optimal shifted ordered encoding of Section 4.1 is concerned, the last two columns of Table 2 show the size of this encoding for the ten datasets on which we could run our cubic dynamic programming algorithm. The results indicate that the size of this encoding tends to be significantly smaller than the optimal shifted encoding discussed in the previous paragraph: seven datasets showed a percentage difference between the optimal shifted encoding and the optimal shifted ordered encoding being larger than 10%, with a peak of 27.31% on `tags-math-sx-seqs`.

We leave it as an open question whether it is possible to compute the optimal shifted ordered encoding in sub-cubic time as a function of u. Another interesting research direction is to design fast heuristic (e.g. ILP formulations) for computing the globally-optimal prefix free encoding (NP-hard to compute).

References

1 Jarno N. Alanko, Elena Biagi, Simon J. Puglisi, and Jaakko Vuohtoniemi. Subset wavelet trees. In Loukas Georgiadis, editor, *21st International Symposium on Experimental Algorithms, SEA 2023, July 24-26, 2023, Barcelona, Spain*, volume 265 of *LIPIcs*, pages 4:1–4:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. `doi:10.4230/LIPIcs.SEA.2023.4`.

2 Diego Arroyuelo and Juan Pablo Castillo. Trie-Compressed Adaptive Set Intersection. In Laurent Bulteau and Zsuzsanna Lipták, editors, *34th Annual Symposium on Combinatorial*

Pattern Matching (CPM 2023), volume 259 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 1:1–1:19, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. `doi:10.4230/LIPIcs.CPM.2023.1`.

3 Austin R. Benson, Ravi Kumar, and Andrew Tomkins. Sequences of sets. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1148–1157, New York, NY, USA, 2018. Association for Computing Machinery. `doi:10.1145/3219819.3220100`.

4 Jon Louis Bentley and Derick Wood. An Optimal Worst Case Algorithm for Reporting Intersections of Rectangles. *IEEE Transactions on Computers*, C29(7):571–577, 1980. `doi:10.1109/TC.1980.1675628`.

5 Alexander Bowe, Taku Onodera, Kunihiko Sadakane, and Tetsuo Shibuya. Succinct de Bruijn Graphs. In Ben Raphael and Jijun Tang, editors, *Algorithms in Bioinformatics*, pages 225–235, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. `doi:10.1007/978-3-642-33122-0_18`.

6 Nicola Cotumaccio, Giovanna D'Agostino, Alberto Policriti, and Nicola Prezza. Co-lexicographically ordering automata and regular languages - part i. *J. ACM*, 70(4), August 2023. `doi:10.1145/3607471`.

7 Paolo Ferragina, Fabrizio Luccio, Giovanni Manzini, and S. Muthukrishnan. Compressing and indexing labeled trees, with applications. *J. ACM*, 57(1), November 2009. `doi:10.1145/1613676.1613680`.

8 Travis Gagie, Giovanni Manzini, and Jouni Sirén. Wheeler graphs: A framework for bwt-based data structures. *Theoretical Computer Science*, 698:67–78, 2017. Algorithms, Strings and Theoretical Approaches in the Big Data Era (In Honor of the 60th Birthday of Professor Raffaele Giancarlo). `doi:10.1016/j.tcs.2017.06.016`.

9 Mainak Ghosh, Indranil Gupta, Shalmoli Gupta, and Nirman Kumar. Fast compaction algorithms for NoSQL databases. In *2015 IEEE 35th International Conference on Distributed Computing Systems*, pages 452–461, 2015. `doi:10.1109/ICDCS.2015.53`.

10 Ankur Gupta, Wing-Kai Hon, Rahul Shah, and Jeffrey Scott Vitter. Compressed data structures: Dictionaries and data-aware measures. *Theoretical Computer Science*, 387(3):313–331, 2007. `doi:10.1016/j.tcs.2007.07.042`.

11 Nick Kline and Richard T. Snodgrass. Computing temporal aggregates. In *Proceedings of the 11th International Conferencec on Data Engineering*, pages 222–231, 1995. `doi:10.1109/ICDE.1995.380389`.

12 Donald E. Knuth. Optimum binary search trees. *Acta informatica*, 1:14–25, 1971. `doi:10.1007/BF00264289`.

13 AFproject high-id proteins dataset. Accessed: 2024-11-15. URL: `https://afproject.org/app/benchmark/protein/high-ident/dataset/`.

14 DBLP XML dump. Accessed: 2024-11-15. URL: `https://dblp.org/xml/`.

15 goodbooks-10k dataset. Accessed: 2024-11-15. URL: `https://github.com/zygmuntz/goodbooks-10k`.

16 UniProt database. Accessed: 2024-11-15. URL: `https://www.uniprot.org/uniprotkb`.

A Additional Material for Section 3

A.1 Proof of Lemma 4

▶ **Lemma 4.** *Let $S = \{x_1, \cdots, x_m\}$ be a set of m integers with $0 \le x_1 < \cdots < x_m < u$. Let us define $x_{m+1} := x_1 + u$. For every $a \in U$, it holds that*

$$\mathrm{trie}(S+a) = \sum_{k=1}^{\log u} \sum_{i=1}^{m} \max_{j \in [x_i+a, x_{i+1}+a)} c_j^{(k)}$$

Proof. Let $a \in U$ be arbitrary. We divide S into $S_< = \{x \in S : x + a < u\}$ and $S_\ge = \{x - u : x \in S \text{ and } x + a \ge u\}$ to consider the effect of computing modulo u. We distinguish three cases.

First, consider the case where $S_\ge = \emptyset$. Observe that $c_{u-1}^{(k)} = 1$ for every $k \in [\log u]$ by definition and the assumption that u is a power of 2. Therefore we have

$$\log u = \sum_{k=1}^{\log u} 1 = \sum_{k=1}^{\log u} c_{u-1}^{(k)}. \tag{9}$$

Furthermore, we have $c_{u-1}^{(k)} = \max_{j \in [x_m, x_{m+1})} c_j^{(k)} = \max_{[x_m+a, x_{m+1}+a)} c_j^{(k)}$, using the assumption that $x_m + a < u \le x_1 + u = x_{m+1}$ as $S_\ge = \emptyset$. From Definition 1, Equations (2) and (9), we then obtain

$$\begin{aligned}\mathrm{trie}(S+a) &= \log u + \sum_{k=1}^{\log u} \sum_{i=1}^{m-1} \max_{j \in [x_i+a, x_{i+1}+a)} c_j^{(k)} \\ &= \sum_{k=1}^{\log u} \left(\max_{j \in [x_m+a, x_{m+1}+a)} c_j^{(k)} + \sum_{i=1}^{m-1} \max_{j \in [x_i+a, x_{i+1}+a)} c_j^{(k)} \right) \\ &= \sum_{k=1}^{\log u} \sum_{i=1}^{m} \max_{j \in [x_i+a, x_{i+1}+a)} c_j^{(k)}. \end{aligned} \tag{10}$$

The claim follows analogously when $S_< = \emptyset$, since then $c_j^{(k)} = c_{j+u}^{(k)}$ for every $k \in [\log u]$ and every $j \ge 0$.

Now we assume that both $S_<$ and $S_\ge$ are not empty. Observe that $x_1 \in S_<$ because it is the smallest element in S and $S_<$ is not empty. Recalling that $x_{m+1} := x_1 + u$, we can observe that the trie for $(S_\ge \cup \{x_{m+1} - u\}) + a$ and the trie for $S_< + a$ share exactly one path from the root to $x_1 + a$. Therefore,

$$\mathrm{trie}(S+a) = \mathrm{trie}((S_\ge \cup \{x_{m+1} - u\}) + a) + \mathrm{trie}(S_< + a) - \log u. \tag{11}$$

Let $i^* \in [2, m]$ be the integer such that $x_{i^*} = \min S_\ge$. Then $S_< = \{x_1, \cdots, x_{i^*-1}\}$ and $S_\ge \cup \{x_{m+1}\} = \{x_{i^*} - u, x_{i^*+1} - u, \cdots, x_{m+1} - u\}$. Since $0 \le x_1 + a < u \le x_{i*} + a$, for every $k \in [\log u]$ it holds that $\max_{j \in [x_1+a, x_{i*}+a)} c_j^{(k)} = c_{u-1}^{(k)} = 1$. Therefore we have

$$\mathrm{trie}(S_< + a) = \log u + \sum_{k=1}^{\log u} \sum_{i=1}^{i^*-2} \max_{j \in [x_i+a, x_{i+1}+a)} c_j^{(k)} = \sum_{k=1}^{\log u} \sum_{i=1}^{i^*-1} \max_{j \in [x_i+a, x_{i+1}+a)} c_j^{(k)}$$

Algorithm 3 Implementation of $\mathcal{D}$ for the $O(u + N \log u)$-time algorithm.

```
function $\mathcal{D}$.initialize():
    $\Delta \leftarrow \langle 0 \rangle$
function $\mathcal{D}$.add($\ell$, $r$):
    $\Delta[\ell] \leftarrow \Delta[\ell] + 1$        // equivalent to $A[a] \leftarrow A[a] + 1, \forall a \in [\ell, r)$
    if $r < |\Delta|$ then $\Delta[r] \leftarrow \Delta[r] - 1$
function $\mathcal{D}$.extend():
    $i \leftarrow |\Delta|$; $s \leftarrow \sum_{a \in [0,i)} \Delta[a]$        // $s$ becomes $A[i-1]$
    $\Delta \leftarrow \Delta\Delta$; $\Delta[i] \leftarrow \Delta[i] - s$        // $\Delta[i]$ becomes $A[0] - A[i-1]$
function $\mathcal{D}$.argmin():
    $a^* \leftarrow 0$; $v^* \leftarrow \Delta[0]$; $v \leftarrow \Delta[0]$        // finding the minimum in the prefix sum
    for $a = 1..|\Delta| - 1$ do
        $v \leftarrow v + \Delta[a]$
        if $v^* > v$ then  $a^* \leftarrow a$; $v^* \leftarrow v$;
    return $a^*$
```

Applying this with Equation (10) to Equation (11), we obtain

$$\begin{aligned}\mathrm{trie}(S+a) &= \sum_{k=1}^{\log u}\left(\sum_{i=i^*}^{m+1} \max_{j \in [x_i + a, x_{i+1} + a)} c_j^{(k)} + \sum_{i=1}^{i^*-1} \max_{j \in [x_i + a, x_{i+1} + a)} c_j^{(k)}\right) - \log u \\ &= \sum_{k=1}^{\log u}\sum_{i=1}^{m} \max_{j \in [x_i + a, x_{i+1} + a)} c_j^{(k)} + \sum_{k=1}^{\log u} \max_{j \in [x_{m+1} + a, x_{m+2} + a)} c_j^{(k)} - \log u.\end{aligned}$$

where we define $x_{m+2} := x_{i^*} + u$.

Now note that $x_{m+1} + a = x_1 + a + u < 2u = u + u \leq x_{i^*} + a + u = x_{m+2} + a$ and thus $x_{m+1} + a \leq 2u - 1 < x_{m+2} + a$. Since $c_{2u-1}^{(k)} = 1$ by definition, it follows that the second maximum is always 1. Hence the second term equals $\log u$, which is canceled with the last term, and thus the claim follows. ◀

A.2 Implementation of $\mathcal{D}$ for the $O(u + N \log u)$-time algorithm

All functions implementing $\mathcal{D}$ besides the `extend` function are self-explanatory. Thus, we proceed with a detailed explanation of $\mathcal{D}$.`extend`(), which corresponds to duplicating the sequence A to $A' = AA$. Assume that A is of length i, i.e., contains elements $A[0], \ldots, A[i-1]$ and recall that Δ at every point has to encode A via differences, i.e., $\Delta[0] = A[0]$ and $\Delta[a] = A[a] - A[a-1]$ for any $a \in [1, i)$. Hence, when duplicating the sequence A, we can simply duplicate Δ to $\Delta' = \Delta\Delta$ but have to change the value at the "boundary", i.e., $\Delta'[i]$. This value has to become $A'[i] - A'[i-1] = A[0] - A[i-1]$. We can simply obtain $A[0]$ from $\Delta[0]$ and we can reconstruct $A[i-1]$ as $\sum_{a=0}^{i-1} \Delta[a]$.

A.3 Implementation of $\mathcal{D}$ for the $O(N \log^2 u)$-time algorithm

The pseudocode implementing the interface of the DAG-compressed variant of dynamic segment trees that we described in Section 3.2.2 is given in Algorithm 4. Algorithm 5 describes instead the private functions called by Algorithm 4. In what follows, we give all implementation details.

Algorithm 4 Implementation of $\mathcal{D}$ for the $O(N \log^2 u)$-time algorithm.

```
1  function $\mathcal{D}$.initialize():
2      $\mathcal{D}$.root ←reallocate_node(null, null, null)
3      $\mathcal{D}$.height ← 1
4  function $\mathcal{D}$.add($\ell$, $r$):
5      $\mathcal{D}$.root ← increment($\mathcal{D}$.root, $\ell, r$, $\mathcal{D}$.height)
6  function $\mathcal{D}$.extend():
7      $u$ ← $\mathcal{D}$.root; $\mathcal{D}$.root ←reallocate_node($u, u, u$)
8      $\mathcal{D}$.root.val ← 0 ; $\mathcal{D}$.height ← $\mathcal{D}$.height + 1
9  function $\mathcal{D}$.argmin():
10     return $\mathcal{D}$.root.argmin
```

The structure of the DAG-compressed tree is as follows. It is built in $\log u$ iterations. At each iteration $k \in [\log u]$, the height of the DAG (i.e. the height of the tree resulting from the expansion of the DAG) is k and it is equal to the number of nodes on the path from the root to the leaves. At any point, the expansion of the DAG we are building is a complete binary tree of height k. We store this height k in a variable $\mathcal{D}$.`height`. The root node is stored in $\mathcal{D}$.`root`, and it covers the range $[0, 2^{k-1})$. We say that the height of the leaves in the tree is 1, parents of leaves are at height 2, and so on. A node v at height $h \geq 1$ covers a range $[x, x + 2^{h-1})$ for some integer x. Such a node v may have zero or two children. If it has two children, the left child covers $[x, x + 2^{h-2})$ and the right child covers $[x + 2^{h-2}, x + 2^{h-1})$, respectively. Note that in the pseudocodes, neither the value of x nor the value of h are stored in the node v explicitly, but they can be reconstructed while navigating the tree.

Node v, corresponding to the subsequence $A[x, x + 2^{h-1} - 1]$, has four variables v.`val`, v.`argmin`, v.`min`, and v.`ref`, in addition to the pointers to its left and right child v.`left` and v.`right`. Recall that the purpose of $\mathcal{D}$ is to maintain an integer array A, increment by one unit all entries belonging to range $A[\ell, r - 1]$ via operation $\mathcal{D}$.`add`(ℓ, r), and duplicate the array via operation $\mathcal{D}$.`extend`(). For a node v that corresponds to a range $A[x, x + 2^{h-1} - 1]$, we store those increments that apply to the whole range in a variable v.`val`. The variable v.`argmin` stores $\mathrm{argmin}_{a \in [0, 2^{h-1})} A[x + a]$, while v.`min` stores $\min_{a \in [0, 2^{h-1})} A[x + a]$.

Finally, v.`ref` stores the number of pointers of other nodes to the node v. The role of this reference counter is to duplicate nodes only when necessary (more details are given below). This reference counter v.`ref` is managed in function `reallocate_node`$(\cdot)$ in Lines 1-11. This function takes three arguments u, w_L and w_R to create a new node. If $u \neq$ null, it makes a copy of u, and decrements u.`ref` because it means one pointer will replace u with its copy. Then it sets the left and right child of the new node to w_L and w_R, and (if they are not null pointers) increment w_L.`ref` and w_R.`ref` by 1 each because the new node will point to them.

All functions implementing $\mathcal{D}$ besides $\mathcal{D}$.`extend`() and $\mathcal{D}$.`add`(ℓ, r) are self-explanatory so we do not discuss them. Function $\mathcal{D}$.`extend`() is also simple as it just allocates a new root, sets its left and right children pointers to the old root, and sets its height to the height of the old root incremented by one unit. Note that the counter `ref` associated with the old root is correctly set at 2. The counter `ref` associated with the new root is set at 1 (even though the new root is not referenced by any node, this value prevents the code from re-allocating the new root each time function $\mathcal{D}$.`add`(ℓ, r) is called).

We proceed by discussing the details of $\mathcal{D}$.`add`(ℓ, r). When this function is invoked, we call a subroutine `increment`$(\cdot)$ with arguments representing the root node, the interval $[\ell, r)$, and the height of the tree (Line 5). We start from the root node, and perform `increment`$(\cdot)$ recursively. Suppose we arrive at a node v of height h covering a range $[x, x + 2^{h-1})$. In

Algorithm 5 Private functions of the data structure from Algorithm 4.

```
1  function reallocate_node($u$,$w_L$,$w_R$):
2      Create a new node $v$
3      if $u \neq$ null then
4          $u$.ref $\leftarrow$ $u$.ref $- 1$
5          ($v$.min, $v$.argmin, $v$.val) $\leftarrow$ ($u$.min, $u$.argmin, $u$.val)
6      else  ($v$.val, $v$.min, $v$.argmin) $\leftarrow (0, 0, 0)$
7      ($v$.left, $v$.right) $\leftarrow (w_L, w_R)$
8      if $w_L \neq$ null then  $w_L$.ref $\leftarrow$ $w_L$.ref $+ 1$
9      if $w_R \neq$ null then  $w_R$.ref $\leftarrow$ $w_R$.ref $+ 1$
10     $v$.ref $\leftarrow 1$
11     return $v$
12 function increment($v$,$\ell$,$r$,$h$):
13     if $\ell \geq r$ then return $v$
14     if $v$.ref $> 1$ then $v$ $\leftarrow$reallocate_node($v$, $v$.left, $v$.right)
15     if $r - \ell = 2^{h-1}$ then
16         ($v$.val,   $v$.min) $\leftarrow$ ($v$.val $+ 1$,   $v$.min $+ 1$)
17         return $v$
18     $v$.left $\leftarrow$ increment($v$.left, $\ell$, $\min\{r, 2^{h-2}\}$, $h - 1$)
19     $v$.right $\leftarrow$ increment($v$.right, $\max\{\ell - 2^{h-2}, 0\}$, $r - 2^{h-2}$, $h - 1$)
20     if $v$.left.min $\leq$ $v$.right.min then
21         ($v$.min,  $v$.argmin) $\leftarrow$ ($v$.left.min $+$ $v$.val,  $v$.left.argmin)
22     else  ($v$.min,  $v$.argmin) $\leftarrow$ ($v$.right.min $+$ $v$.val,  $v$.right.argmin $+ 2^{h-2}$)
23     return $v$
```

Line 14, if v is referred by more than one pointer (i.e., if v.`ref` > 1), this means that the range $[x, x+2^{h-1})$ is not the unique range that v is covering; in other words, v is being reused at more than one place. Thus we make a copy of v, and proceed with the copied node. This new node will be returned by the function (Lines 17,23) so that it can replace the old node properly (Lines 5,18-19). Now we are to perform an update according to the given interval. When $[\ell, r)$ is passed as an argument of the function, it means that we are to increment $A[a]$ for $a \in [x + \ell, x + r)$. If the size of the interval is exactly 2^{h-1}, we need to increase $A[a]$ by 1 unit for all $a \in [x, x + 2^{h-1})$, which can be performed by incrementing v.`val` and v.`min` by 1 each (Lines 15-17). Otherwise, we split the interval at position $x + 2^{h-2}$ (i.e., split $[x + \ell, x + r)$ into $[x + \ell, x + 2^{h-2})$ and $[x + 2^{h-2}, x + r)$), then process each split segment with its left and right child, respectively (Lines 18-19). After processing the insertion at the child nodes, we update v.`min` and v.`argmin` according to the minimum computed in the children (Lines 20-22), which will propagate to the root node. If the minimum is from the right child, we add the offset 2^{h-2} for updating v.`argmin` properly.

The cost of procedure $\mathcal{D}$.`add`$(\cdot)$ is proportional to $O(\log |A|)$. This is because in Line 15 we do not further recurse on the children of v if the local interval $[\ell, r)$ spans the entire range of length 2^{h-1} associated with v; in turn, this means that the recursive calls to `increment` stop on the $O(\log |A|)$ nodes whose associated intervals partition the initial range $[\ell, r)$ on which function $\mathcal{D}$.`add`(ℓ, r) was called. Additionally, all ancestors of those nodes have at least two children (notice that, if `increment` is called recursively, the recursion always occurs on both children of the node: see Lines 18 and 19), therefore the total number of nodes recursively visited by $\mathcal{D}$.`add`(ℓ, r) is $O(\log |A|)$.

B Additional Material for Section 5

Table 1 Summary of the thirteen datasets. From left to right, we report the dataset id and name, the number of distinct universe elements belonging to the sets (σ), the size of the universe ($u = 2^{\lceil \log |U| \rceil}$), the total length ($N$), the number of sets ($n$), and the average size of a set (N/n).

dataset		σ	u	N	n	N/n
1	email-Enron-core-seqs	141	256	14148	10428	1.36
2	contact-prim-school-seqs	242	256	251546	174796	1.44
3	contact-high-school-seqs	327	512	377000	308990	1.22
4	email-Eu-core-seqs	937	1024	252872	202769	1.25
5	tags-mathoverflow-seqs	1399	2048	125056	44950	2.78
6	tags-math-sx-seqs	1650	2048	1177312	517810	2.27
7	coauth-Business-seqs	236226	2097152	849838	463070	1.84
8	coauth-Geology-seqs	525348	2097152	3905349	1438652	2.71
9	DBLP.xml	26	32	28815437	3939813	7.31
10	proteins-high-id	20	32	40664	2128	19.11
11	proteins-long	25	32	11051828	571282	19.35
12	book-ratings	5	8	48763	10000	4.88
13	tags-book	34252	65536	999904	10000	99.99

Table 2 Experimental result. From left to right, we report the dataset id, and the optimal shifted encoding size (opt-shift) followed by the average shifted encoding size (avg-shift), the ratio between opt-shift and avg-shift, the worst shifted encoding size (worst-shift), and the ratio between opt-shift and worst-shift. The last two columns show the optimal shifted ordered encoding size (opt-ord), and the ratio between opt-shift and opt-ord (only for datasets with small universe size u).

id	opt-shift	avg-shift	$\frac{\text{opt-shift}}{\text{avg-shift}}$(%)	worst-shift	$\frac{\text{opt-shift}}{\text{worst-shift}}$(%)	opt-ord	$\frac{\text{opt-shift}}{\text{opt-ord}}$(%)
1	105708	106787	98.99	108027	97.85	84843	124.59
2	1864240	1870395	99.67	1876731	99.33	1815243	102.70
3	3227940	3244486	99.49	3257990	99.07	2906848	111.05
4	2457044	2460375	99.86	2464644	99.69	2181173	112.65
5	1092079	1138803	95.90	1177890	92.71	880232	124.07
6	10727737	11069006	96.92	11413439	93.99	8426162	127.31
7	14940827	15073904	99.12	15171942	98.47	-	-
8	66864377	67972793	98.37	68812124	97.17	-	-
9	70403690	75662919	93.05	81715980	86.15	58285023	120.79
10	83178	86608	96.04	89503	92.93	80989	102.70
11	22454406	23371923	96.07	24144585	93.00	21868947	102.68
12	96467	97865	98.57	98714	97.80	87566	110.16
13	7923290	8022933	98.76	8146000	97.27	-	-