Università di Pisa

Dipartimento di Informatica
Corso di Laurea Magistrale in Informatica

Compressed Representations of Set Collections

Candidato
Luca Lombardo

Relatore
Roberto Grossi

Anno Accademico 2025/2026

Abstract

This thesis studies compressed representations of collections of sets drawn from a common ordered universe. The representation must support standard ordered-set queries on each set: membership, access by position, rank (which counts elements up to a value) predecessor, and successor. A natural baseline is to compress each set in isolation, treating it only as a subset of the universe. Such an encoding may exploit the internal structure of each set, but it cannot use the redundancy that appears only when the sets are viewed together. Containment, overlap, and small differences between sets are precisely the regularities lost by this isolated view. Existing relation-based representations exploit these regularities by storing one set through another and answering queries along labelled trees. Their references, however, are usually restricted to the input sets or to a graph on them, so they do not measure how much information the whole collection contains.

To measure the redundancy that is visible only at collection level, we change how we look at the collection. Instead of describing one set at a time, we look at each universe element and record which input sets contain it. Elements that appear in exactly the same input sets can be grouped together. If the groups that occur and their sizes are known, describing the collection reduces to deciding which labelled universe elements belong to each group. Counting these assignments gives a reference for the bits needed to describe how elements are distributed across the collection. Taken alone, however, this reference is only a count. It says how many bits are needed in principle, but not how to organize those bits so that membership, access, rank, predecessor, and successor on a single set can be answered without decoding the global assignment or adding separate indexes.

We introduce Set-Union Matching (SUM) to turn this counting view into a queryable hierarchy. Existing encodings can exploit a relation between two sets in two natural ways: if one contains the other, the smaller set can be recovered by deleting elements from the larger one, and if two sets are close, one can be described by recording the elements on which they differ. SUM keeps the first style by creating, for each chosen pair of current sets, their union as the smallest common parent, so that both children are recovered from the parent only by deleting the elements that do not belong to the corresponding child and every edge in the hierarchy has the same containment meaning. Repeating this step in rounds builds a hierarchy of such parents, using the saving exposed by each union to decide which pairs to merge and keeping the least costly hierarchy reached by the construction. The resulting count never exceeds the isolated-set baseline and improves on it when the hierarchy captures shared structure, while the global assignment count remains the lower benchmark for the hierarchy model considered in the thesis. Because the hierarchy uses only containment edges, each connected part becomes a deletion tree, where a set is recovered from its root by removing the labels found along one root-to-leaf path. A dictionary for each root and path-query indexes on the tree then support membership, access, rank, predecessor, and successor, with logarithmic terms governed by the root of that tree.

Preface

The interest in the topics behind this thesis stems from a talk at SEA 2025, held during the Festschrift for Roberto Grossi's sixtieth birthday [1], where Alanko, Bille, Gørtz, Navarro, and Puglisi presented [2]. The paper presents a compact representation of set collections that encodes each set as a subset of a larger one in the collection and supports membership, rank, predecessor, and successor. The idea of encoding each set inside a larger one and recovering it by traversing the containments was close to what we had studied in my bachelor thesis on succinct data structures over directed acyclic graphs [3]. This resemblance made the problem a natural continuation of that work, now in the setting of set collections.

Their framework uses redundancy only when one set is contained in another set already present in the collection, because only then can the smaller set be reached by following a containment path. If two sets share many elements but neither contains the other, that overlap cannot be reached by such a path. The question we pursued was how to make this overlap usable by a containment-based representation while keeping the queries efficient.

The algorithm we present in Chapter 5 keeps the containment style by adding synthetic union nodes as common parents for pairs of overlapping sets. A child is then recovered from its union parent by deleting the elements outside it, so the hierarchy has the same containment meaning on every edge. Independently and in parallel, Gagie, He, and Navarro pursued the same problem through symmetric differences: their representation describes one set from a nearby one by the elements on which they differ, at the price of a more involved access procedure. These two routes make the contrast between containment paths and difference-based descriptions explicit, and the comparison closes Chapter 5.

This thesis is one of several projects I have worked on during my bachelor's and master's years, all sitting along the line between combinatorial algorithm design, succinct data structures, algorithm engineering, and high-performance computing. The most rewarding ones have always been those that demanded all four at once.

Contents

List of Figures

1 Introduction

Compression starts from the idea that a representation should spend bits only on information needed to identify the object. If two descriptions lead to the same object, the shorter one has removed some redundancy. This viewpoint abstracts away from file formats and coding schemes, because it treats the object as the thing to be identified and the representation as the information needed to identify it.

Kolmogorov complexity gives this ideal a precise form. After fixing a universal machine, descriptions become programs for that machine, and the Kolmogorov complexity of a binary string is the length of the shortest program that prints it [4]. In this sense, a highly regular string is compressible because a short program can describe the rule that generates it, while an irregular string may have no description substantially shorter than itself.

[4]: Li et al. (2008), *An introduction to Kolmogorov complexity and its applications*

Kolmogorov complexity marks the limit obtained by removing every redundant bit from an individual object, but this limit is not computable and does not prescribe a data structure. A shortest program that prints a set may describe it succinctly while giving no efficient way to answer queries about its elements.

We therefore work with computable counting bounds. Once an object is known to belong to a finite family $\mathcal{F}$, any lossless representation that distinguishes all objects in $\mathcal{F}$ needs at least $\lg |\mathcal{F}|$ bits in the worst case. For a single set, the family can be counted directly. A subset of size n drawn from a universe of size u can be chosen in $\binom{u}{n}$ ways, so the counting cost of that set is $\lg \binom{u}{n}$ bits.

This thesis considers instances in which several objects are described over the same universe U. Each object is represented by the subset of universe elements associated with it, so the instance is a finite family of subsets. We call such a family a set collection and write it as $\mathcal{S} = \{S_1, \ldots, S_m\}$, with each $S_i \subseteq U$. If the representation ignores relations inside $\mathcal{S}$, it describes each S_i independently and adds the corresponding counting costs. This baseline tells us how many bits are spent before any relation between sets is used. The compression question starts when the sets are related: one set may contain another, and two sets may overlap enough that independent descriptions repeat the same choices. We then need a smaller description that also leaves enough structure for later queries to reach the selected set.

1.1 Why compress set collections?

Set collections appear in applications that look unrelated until the universe is fixed. A web page contributes its outgoing neighbours, a term contributes the documents that contain it, and a sequence fragment contributes the genomes in which it occurs. In each case the sets are not arbitrary: nearby pages, similar terms, or adjacent fragments often select

almost the same elements. Treating the input as a collection matters because these repeated choices can be shared, while queries still ask for information about one selected set.

Compression becomes interesting when the subsets are dependent. Related objects often select overlapping parts of the universe, or differ only by a small boundary around a common core. If we encode each set as a fresh subset of the whole universe, these relations disappear from the count. The compression problem is therefore to exploit dependence between sets without losing direct access to the set named by a query.

In web graphs, this dependence appears in the adjacency lists. A graph representation must store the successor list of each page, and a query may later ask for the neighbours of one selected page. Boldi and Vigna show that URL order exposes two forms of dependence. Successors of one page often have close identifiers, and pages close in URL order often share successors. Their WebGraph format stores gaps between consecutive successors, then represents a list by referring to a nearby list, marking copied portions, and storing the remaining extra nodes [5]. The compression gain comes from the relation between adjacency sets, while the data structure still has to answer graph queries.

[5]: Boldi et al. (2004), *The webgraph framework I: compression techniques*

Inverted indexes have the same shape with documents as the universe. For each distinct term, the index stores the sorted list of document identifiers containing that term. Query processing then scans or intersects the lists selected by the query, so compression cannot be separated from access to individual lists. Pibiri and Venturini survey this problem as inverted index compression, where the lists must occupy little space and still support fast query processing [6]. This gives one subset of the document universe for each term. Similar terms can induce similar lists, and the collection may contain information that a per-list encoding never sees.

[6]: Pibiri et al. (2020), *Techniques for inverted index compression*

In coloured de Bruijn graphs, the universe is the set of genomes. The graph represents sequence fragments, and the colour set of a vertex records the genomes in which that fragment occurs [7]. Since neighbouring fragments often appear in nearly the same genomes, adjacent vertices often have identical or similar colour sets. Almodaresi et al. use this dependence by encoding colour classes through small differences from related classes [8]. Query access remains local. A compressed pangenomic index must answer which genomes contain a fragment without expanding all colour sets.

[7]: Alanko et al. (2023), *Themisto: a scalable colored k-mer index for sensitive pseudoalignment against hundreds of thousands of bacterial genomes*

[8]: Almodaresi et al. (2020), *An efficient, scalable, and exact representation of high-dimensional color information enabled using de Bruijn graph search*

In each case, the data contain many subsets of one universe, the subsets depend on one another, and the operations still ask about selected sets. A useful compression measure must say how much is saved by exploiting relations such as containment, overlap, or small difference. A useful representation must then realise that saving without turning the collection into an inaccessible archive. We now fix the common model and the independent baseline against which such savings will be measured.

1.2 Problem setting

We work with many finite sets over one ordered universe. The order lets the operations ask whether an element is present and where it lies among the elements of a set. The following definition fixes the notation used by the whole thesis.

Definition 1.2.1 (Set collection) *A set collection over a universe U is a family $\mathcal{S} = \{S_1, \ldots, S_m\}$ of m finite subsets of a finite, totally ordered set U. We call U the universe of the collection. We write $u = |U|$ for the size of the universe and $n = \sum_{i=1}^{m} |S_i|$ for the total size of the collection. We assume $U = [1..u] = \{1, 2, \ldots, u\}$ without loss of generality, since any totally ordered set of size u can be mapped to this interval by a rank function after preprocessing in $O(n \lg u)$ time.*

If the universe is not $[1..u]$, one collects the n elements, sorts them, and assigns integers in $[1..u]$ to the u distinct values. Queries and answers translate in $O(1)$ time via a lookup table.

Each set S_i inherits the total order from U, so its elements can be listed as $S_i[1] < S_i[2] < \cdots < S_i[|S_i|]$. The integer n counts elements with multiplicity across sets. If one universe element belongs to k sets, it contributes k to n. The parameter n can be much smaller than $m \cdot u$, but the representation must still distinguish the sets and support queries on each of them.

A representation of $\mathcal{S}$ must recover information about any chosen set S_i without decompressing the whole collection.

Gagie, He, and Navarro list membership, access, rank, predecessor, and successor [9]. Alanko et al. use the equivalent bitvector formulation obtained by identifying each set with its characteristic vector [2].

Definition 1.2.2 (Operations on set collections) *Given a set collection $\mathcal{S}$ over $U = [1..u]$, we define five operations on each $S \in \mathcal{S}$:*

- member(S, x)*: returns true if $x \in S$ and false otherwise.*
- access(S, i)*: given $i \in [1..|S|]$, returns the element of rank i in S, that is, $S[i]$.*
- rank(S, x)*: returns $|\{y \in S : y \leq x\}|$, the number of elements of S at most x.*
- predecessor(S, x)*: returns $\max\{y \in S : y \leq x\}$, or $-\infty$ if no such element exists.*
- successor(S, x)*: returns $\min\{y \in S : y \geq x\}$, or $+\infty$ if no such element exists.*

These five operations are the primitives on which higher level tasks depend. Set intersection, for instance, can be computed by interleaving predecessor and successor queries on both operands, advancing through the sorted elements in tandem.

Predecessor and successor reduce to rank followed by access. A predecessor query performs one rank computation and one access computation, and a successor query does the same. The three operations member, rank, and access are therefore the independent primitives. Any representation that supports these three automatically supports all five, at the cost of one additional access per predecessor or successor query.

To compute predecessor(S, x), first compute $r = \text{rank}(S, x)$. If $r = 0$, no predecessor exists. Otherwise, return access(S, r). Successor works symmetrically.

We know from information theory that for a finite class of objects $\mathcal{F}$, the worst-case entropy is $\lg |\mathcal{F}|$, the number of bits needed to distinguish one object of the class when no probability model is assumed. To apply this count to a set collection, we first ignore all relations between different sets and fix only the cardinalities. A set $S_i \subseteq U$ of size $|S_i|$ is one of $\binom{u}{|S_i|}$ possible subsets of U of that size. Any encoding that distinguishes

all such subsets uses at least $\lg \binom{u}{|S_i|}$ bits for S_i. If the sets are described independently, the admissible class has size $\prod_{i=1}^{m} \binom{u}{|S_i|}$, and its logarithm is the sum below.

Definition 1.2.3 (Independent encoding cost) *The independent encoding cost of a set collection $\mathcal{S}$ over U is*

$$H_{wc}(\mathcal{S}) \;=\; \sum_{i=1}^{m} \lg \binom{u}{|S_i|}.$$

The notation $H_{wc}(\mathcal{S})$ records that this is the worst-case entropy of the independent class determined by the set sizes. Alanko, Bille, Gørtz, Navarro, and Puglisi use the same quantity for set collections, written $H(\mathcal{S})$ [2]. We keep the subscript to distinguish this reference point from the finer measures introduced later. The bound is achievable up to lower order terms. Encoding each set independently costs $H_{wc}(\mathcal{S})+O(n+m \lg n)$ bits, where the $O(m \lg n)$ term gives a directory for the m sets and the $O(n)$ term accounts for the overheads of the sparse bitvectors. The representation supports rank and constant time access to the elements of any set.

The independent cost is the reference point for representations that ignore relations between sets. If the representation may use a relation, the class to be counted can become smaller. If $S_i \subset S_j$, then describing S_i relative to S_j means choosing $|S_i|$ elements from a universe of size $|S_j|$, for a cost of $\lg \binom{|S_j|}{|S_i|}$ bits. Containment-based representations use exactly this smaller universe when a contained set is encoded through a containing set [2]. If two sets differ in few elements, the description can record the small change between them. We can recover one set from the other by storing the insertions and deletions that separate them, which is the set-difference viewpoint of Gagie, He, and Navarro [9]. Thus $H_{wc}(\mathcal{S})$ measures the cost before inter-set relations are used, and later measures drop below it only when such relations become part of the description.

After fixing the independent count, two questions remain. We first need a measure that tells which relations between sets shrink the admissible class below the one counted by $H_{wc}(\mathcal{S})$. We then need a representation that stores such a description while queries still run on the compressed data. Section 1.3 explains how the chapters follow these questions.

1.3 Thesis structure

The independent count of Definition 1.2.3 is meaningful only if a single ordered set can be stored near its own counting bound and still queried. We therefore start from the one-set problem. Chapter 2 develops bitvectors, sparse dictionaries, and wavelet trees as concrete representations for membership, access, and rank on ordered sets. These structures give the independent representation that reaches $H_{wc}(\mathcal{S})$ up to lower order terms when no relation between sets is used.

Relations between sets change the shape of the query problem. If a set is stored through another set, a query cannot stop at one dictionary. It must follow a reference path and account for the changes stored along

that path. Chapter 3 develops the tree representations and path-query primitives needed for this step. Succinct ordinal trees represent the shape of a hierarchy, and labelled tree extraction turns path labels into the counting and selection operations needed by set queries.

Single set dictionaries and the path-query framework of Chapter 3 make relation-based compression queryable. Chapter 4 develops the recent state of the art in this direction, namely containment, insertion, and symmetric difference compressibility. Each choice changes the local description used on an edge. A containment edge records deletions from a larger set, an insertion edge records additions from a smaller set, and a symmetric difference edge records both additions and deletions. These regimes show the constraint that every later construction must satisfy. A smaller count matters only when it also produces paths on which the operations of Definition 1.2.2 can run.

Those constructions still choose their references among the sets already present in the collection. This restriction misses pairs of sets that overlap heavily without either set containing the other, since no existing set can serve as a containment parent for both. Chapter 5 addresses this case by separating the counting question from the query question. If two sets share many elements, counting them together can charge the shared part once instead of treating the two sets as unrelated choices. This explains where a smaller description can come from, but it does not tell a query how to recover either set. The chapter then turns the same overlap into a query path by adding the union of the two sets as a new parent. The two sets become children of a common containment node. Repeating this step keeps the query structure of deletion hierarchies while allowing the hierarchy to use references that were absent from the input.

Appendix A examines the same principle in a larger reference space. Once the representation may add union nodes, it is natural to ask what changes if it may also add larger unions, intersections, or mixed references. Closing the collection under union and intersection exposes the lattice of possible references, while bounded arity measures how much is lost when the hierarchy insists on binary splits. The appendix places the binary-union construction inside this larger optimisation problem and separates the tractable subproblem restricted to unions from the broader lattice search.

2 Succinct Dictionaries

Chapter 1 fixed a collection of ordered subsets and the five operations that a representation must answer. These operations are evaluated on one set at a time, so any representation of a collection contains a single-set problem as a necessary component. For a set $S \subseteq [1..u]$, the characteristic bitvector $B[1..u]$ records the ordered universe without changing the object, with $B[x] = 1$ exactly when $x \in S$. This binary representation preserves the order needed by the operations. Membership inspects the bit at x. Access asks for the position of the i-th one. Rank asks for the number of ones in the prefix $B[1..x]$, and predecessor and successor combine this count with the corresponding selected position.

The symbol n is local in this chapter. For the characteristic bitvector of a single set over $[1..u]$, the bitvector length is $n = u$.

The cost of this representation has two parts. The bitvector itself gives the lossless encoding of the set, while auxiliary information makes count and position queries fast. Succinct dictionaries keep the auxiliary part asymptotically smaller than the encoded object, and sparse variants replace the full universe cost by a term depending on the number of stored elements. The same binary primitives also support sequences over larger alphabets through a hierarchy of bitvectors. These structures provide the single-set baseline used later when a collection is encoded set by set before inter-set relations are exploited.

2.1 Rank and select

Throughout this chapter we work in the word RAM model with word size $\omega = \Omega(\lg n)$, so that a machine word holds a bitvector index and the bit-level primitives below admit constant-time arithmetic, bitwise, and shift operations. All space bounds are measured in bits, and lg denotes $\log_2$. Let $B[1..n]$ be a bitvector of length n. The two primitives we study are defined as follows.

Definition 2.1.1 (Rank) *For $b \in \{0,1\}$ and $1 \le i \le n$, the operation* $\mathrm{rank}_b(B, i)$ *returns the number of occurrences of b in the prefix $B[1..i]$,*

$$\mathrm{rank}_b(B, i) \;=\; |\{j : 1 \le j \le i,\ B[j] = b\}|.$$

By convention $\mathrm{rank}_b(B, 0) = 0$.

Definition 2.1.2 (Select) *For $b \in \{0,1\}$ and $j \ge 0$, the operation* $\mathrm{select}_b(B, j)$ *returns the position in B of the j-th occurrence of b, that is, the unique index i such that $B[i] = b$ and* $\mathrm{rank}_b(B, i) = j$. *By convention* $\mathrm{select}_b(B, 0) = 0$, *and* $\mathrm{select}_b(B, j) = n + 1$ *whenever* $j > \mathrm{rank}_b(B, n)$.

The two operations are mutual inverses on their domains. For any i with $B[i] = b$ we have $\mathrm{select}_b(B, \mathrm{rank}_b(B, i)) = i$, and for every valid j we have $\mathrm{rank}_b(B, \mathrm{select}_b(B, j)) = j$. The identity $\mathrm{rank}_0(B, i) = i - \mathrm{rank}_1(B, i)$ reduces rank for the two values of b to a single subtraction, so only rank_1

needs direct support. Select admits no analogous identity, and select_0 requires an auxiliary structure separate from select_1.

Figure 2.1 fixes a concrete bitvector of length $n = 20$ with population count 11. The red prefix gives $\text{rank}_1(B, 15) = 9$, and the blue-circled bit gives $\text{select}_1(B, 7) = 12$.

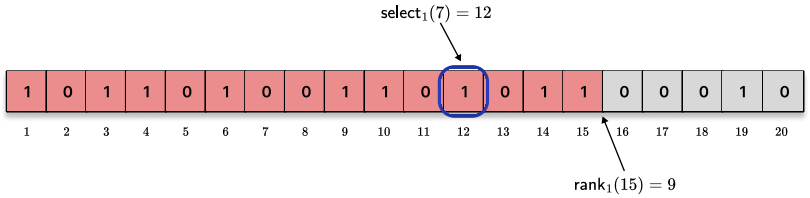

Figure 2.1: A bitvector $B[1..20]$. The prefix $B[1..15]$ (shaded red) contains nine ones, so $\text{rank}_1(B, 15) = 9$. The seventh one (circled blue) lies at position 12, so $\text{select}_1(B, 7) = 12$.

Without auxiliary structures, both queries reduce to a linear scan. Rank scans i bits and counts the ones. Select scans from the left and returns the position once it has counted j ones. In the worst case, both queries take $\Theta(n)$ time, with no extra storage beyond the n bits of B itself and $O(\lg n)$ bits for a loop counter.

The linear-time scan is acceptable for short bitvectors, but the set collections of later chapters need fast repeated queries on much longer inputs. We therefore adopt $O(1)$ query time as the goal, and measure the price in extra space above B. Distinguishing the 2^n bitvectors of length n takes at least n bits, and B already attains this lower bound, so the question is how many bits of auxiliary storage in addition to B are needed to support $O(1)$-time rank and select. An auxiliary index of size $o(n)$, growing strictly slower than n, gives a total of $n + o(n)$ bits, the target Jacobson called *succinct* [10].

A succinct representation adds only $o(n)$ bits to the verbatim n-bit input and answers each query in $O(1)$ time, matching the information-theoretic minimum up to a sublinear term.

[10]: Jacobson (1988), *Succinct static data structures*

The primitives of this section recur throughout the rest of the thesis as building blocks, and every bound built on them inherits their query time. A linear-time rank would propagate linearly, and a logarithmic rank would propagate logarithmically. The $O(1)$-time $o(n)$-bit target is therefore the operating assumption of every structure layered above bitvectors. Subsection 2.1.1 through Subsection 2.1.4 show how this target is met, why it is optimal in a precise sense, and what changes when the bitvector itself is encoded below n bits.

2.1.1 Constant-time rank

The first question is whether the linear-time baseline can be improved to $O(1)$ without paying more than $o(n)$ extra bits. A rank query at position i only needs the number of ones before a nearby boundary plus the number of ones inside the short suffix from that boundary to i. Jacobson's construction makes this decomposition explicit. It stores ranks at sparse boundaries, stores smaller ranks at denser boundaries, and answers the remaining within-block query by table lookup.

Jacobson [11] introduced the rank directory below. Clark later used the word-RAM view to state the same access pattern as constant-time rank [12].

Theorem 2.1.1 (Constant-time rank [11, 12]) *For any bitvector $B[1..n]$, there is a data structure of $n + O(n \lg\lg n / \lg n)$ bits that supports $\text{rank}_1(B, i)$ in $O(1)$ time on the word RAM with word size $\omega = \Omega(\lg n)$. The bitvector B is stored verbatim and accessed read-only.*

The construction partitions B into two nested levels, illustrated in Figure 2.2. Choose $z = \lfloor (\lg n)/2 \rfloor$ and let $Z = \Theta(\lg^2 n)$ be a multiple of z. For readability assume Z divides n, so each full superblock contains Z/z blocks. The remaining boundary cases are handled by padding or by storing one final partial superblock separately, changing only lower-order terms.

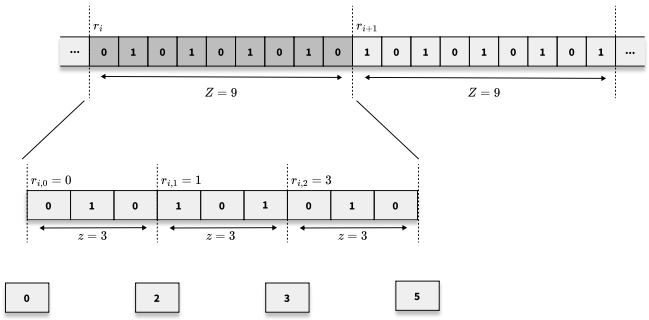

Figure 2.2: The two-level rank directory of Jacobson's construction. Superblocks of size Z store the absolute directory entries R_S. Blocks of size z store the relative directory entries R_B inside the enclosing superblock. The bottom row represents the lookup table indexed by a z-bit block pattern and an in-block offset.

Two auxiliary arrays store precomputed rank values. The *superblock directory* R_S holds absolute ranks at superblock boundaries. For $k = 0, 1, \ldots, n/Z - 1$ we set

$$R_S[k] \;=\; \mathrm{rank}_1(B, kZ).$$

The *block directory* R_B holds ranks relative to the enclosing superblock. For each block index $\ell = 0, 1, \ldots, n/z - 1$, belonging to superblock $k = \lfloor \ell z / Z \rfloor$, we set

$$R_B[\ell] \;=\; \mathrm{rank}_1(B, \ell z) - R_S[k].$$

A rank query at position $i \geq 1$ decomposes as the sum of three contributions. Let $k = \lfloor (i-1)/Z \rfloor$ be the enclosing superblock, let $\ell = \lfloor (i-1)/z \rfloor$ be the enclosing block, and let $j = i - \ell z$ be the offset inside that block. Then $\mathrm{rank}_1(B, i)$ is $R_S[k]$, plus the block-relative value $R_B[\ell]$, plus the rank inside block ℓ up to offset j.

The third contribution, rank within a z-bit block, is handled by a single universal lookup table. Since $z \leq \omega$, the block pattern $B[\ell z + 1..(\ell+1)z]$ fits in one machine word and is read directly from B. The table is indexed by the z-bit pattern p and the offset $j \in [1..z]$, returning the precomputed count $\mathrm{rank}_1(p, j)$. With $2^z \cdot z$ entries of $\lceil \lg(z+1) \rceil$ bits each, the total size is $O(2^z \cdot z \cdot \lg z)$ bits, which becomes $O(\sqrt{n}\, \lg n\, \lg\lg n) = o(n)$ because $z = \lfloor (\lg n)/2 \rfloor$ gives $2^z = \Theta(\sqrt{n})$.

Sketch. The superblock directory R_S stores n/Z absolute ranks of $\lceil \lg n \rceil$ bits each, totalling $O(n/\lg n)$ bits. The block directory R_B stores n/z values in $[0, Z]$ at $O(\lg\lg n)$ bits each, totalling $O(n \lg\lg n / \lg n)$ bits. The lookup table has $2^z \cdot z = O(\sqrt{n} \lg n)$ entries of $O(\lg\lg n)$ bits, which is $o(n)$. The auxiliary components therefore use $O(n \lg\lg n / \lg n)$ bits, dominated by R_B. Adding the verbatim copy of B gives total space $n + O(n \lg\lg n / \lg n)$. A query computes $k = \lfloor (i-1)/Z \rfloor$, $\ell = \lfloor (i-1)/z \rfloor$, and $j = i - \ell z$, reads $R_S[k]$, $R_B[\ell]$, the block pattern from B, and one table entry, performing $O(1)$ arithmetic. □

The lookup table is the precomputed-table step in the construction. It is small enough to keep the representation succinct, but large enough to

answer every within-block rank query in one word-RAM lookup. The construction exposes where the $o(n)$ overhead comes from. The block directory R_B contributes the dominant term $\Theta(n \lg\lg n/\lg n)$, because each of $\Theta(n/\lg n)$ blocks carries a directory entry of $\Theta(\lg\lg n)$ bits. The superblock directory contributes only $O(n/\lg n)$ bits, smaller than R_B by a $\lg\lg n$ factor since superblock boundaries are sparser and each entry pays $\lg n$ bits while a block entry pays $\lg\lg n$. The lookup table is smaller still, contributing only $O(\sqrt{n}\operatorname{polylog} n)$ bits. The block size $z = \Theta(\lg n)$ is the balance point. A larger z would shrink the block directory and blow up the lookup table, and a smaller z would do the opposite. The next subsection shows why this position-keyed directory is not enough for select, where the queried position is the unknown value.

2.1.2 Constant-time select

Rank and select are mutual inverses, but Jacobson's directory is indexed by positions. For rank, the query position i determines the enclosing superblock and block by division. For select, the input j names an occurrence, while the position containing that occurrence is the answer. A directory sampled by position therefore leaves a window that still has to be searched. The earlier select structure discussed by Clark stores explicit position markers at every $\lg^2 n$-th one and binary-searches the $\lg^2 n$-bit window between consecutive markers, giving $\Theta(\lg\lg n)$ query time [12]. To make the top-level navigation constant time, the first partition must be sampled by occurrence number.

Clark resolved this by partitioning the bitvector according to the number of ones instead of the number of positions. A *chunk* contains a fixed number K of ones, so its boundaries lie at the positions $\operatorname{select}_1(B, 1)$, $\operatorname{select}_1(B, K+1)$, $\operatorname{select}_1(B, 2K+1)$, and so on. Locating the j-th one at the top level reduces to computing $\lfloor (j-1)/K \rfloor$. The cost is that chunks now have variable length, and a second idea is needed to handle queries within a chunk. Clark's solution is a three-level hierarchy with a sparse/dense dichotomy at each level.

Clark [12] developed the three-level select hierarchy on top of Jacobson's rank directory. The construction keeps the bitvector verbatim and stores only position markers, offsets, and short-block tables.

Theorem 2.1.2 (Constant-time select [12]) *For any bitvector $B[1..n]$, there is a data structure of $n + o(n)$ bits that supports* $\operatorname{select}_1(B, j)$ *in $O(1)$ time on the word RAM with word size $\omega = \Omega(\lg n)$. The bitvector B is stored verbatim and accessed read-only. Applying the same auxiliary structure to the zero positions, equivalently to the complemented bitvector $\overline{B}$, supports* select_0 *with the same asymptotic overhead.*

The three-level partition is described here for select_1. The zero case uses the same construction on the positions where B has value 0.Let $m = \operatorname{rank}_1(B, n)$ be the number of ones in B, and set the top-level chunk population to $K = \lfloor \lg^2 n \rfloor$.

At the top level, we divide the positions of ones into groups of K. Let $q = \lceil m/K \rceil$. The chunk starts are recorded in an array P_1 with $q + 1$ entries, with

$$P_1[c] = \operatorname{select}_1(B, cK + 1) \qquad \text{for } c = 0, 1, \ldots, q-1,$$

With $K = \lfloor \lg^2 n \rfloor$, a sparse chunk has length at least $K^2 = \Theta(\lg^4 n)$, while a dense chunk fits in a polylogarithmic window. The threshold balances explicit storage against the recursive layer applied within dense chunks.

and sentinel value $P_1[q] = n + 1$. Each entry of P_1 fits in $\lceil \lg n \rceil$ bits, so the total space for P_1 is $O((m/K + 1) \lg n) = O(n/\lg n) + O(\lg n) = o(n)$ bits, using $m \leq n$. Given a query $\text{select}_1(B, j)$, the chunk index is $c = \lfloor (j-1)/K \rfloor$ and the local rank within the chunk is $j_c = ((j-1) \bmod K) + 1$, both computed in $O(1)$ time.

A chunk covers positions $P_1[c]$ through $P_1[c+1] - 1$ of B, so its length $Z_c = P_1[c+1] - P_1[c]$ varies with c. The construction splits into two cases by density. A chunk is *sparse* when $Z_c \geq K^2 = \Theta(\lg^4 n)$, and *dense* otherwise. Sparse chunks have disjoint lengths at least K^2, so there are at most n/K^2 of them. For a sparse chunk, the positions of its ones are stored explicitly as offsets relative to $P_1[c]$, each fitting in $\lceil \lg Z_c \rceil = O(\lg n)$ bits. The final partial chunk contributes only $O(K \lg n)$ bits if stored explicitly, so the total sparse-offset space is

$$\frac{n}{K^2} \cdot K \cdot O(\lg n) \;=\; \frac{n \lg n}{K} \;=\; O\left(\frac{n}{\lg n}\right) \text{ bits,}$$

which is $o(n)$. Explicit storage is not used for dense chunks. There may be n/K dense chunks, and storing K offsets of $O(\lg \lg n)$ bits in each of them would cost $O(n \lg \lg n)$ bits. Dense chunks are therefore sampled again by occurrence number.

Each dense chunk is subdivided by number of ones, with sub-chunk size $k = \lceil (\lg \lg n)^2 \rceil$. Inside dense chunk c, we record the start of each sub-chunk as an offset from $P_1[c]$, together with a local sentinel for the end of the dense chunk. Each offset fits in $\lceil \lg Z_c \rceil = O(\lg \lg n)$ bits, since $Z_c < K^2 = \Theta(\lg^4 n)$ implies $\lg Z_c = O(\lg \lg n)$. With at most n/K dense chunks contributing $(K/k) \cdot O(\lg \lg n)$ bits each, the total dense-chunk recursive space is

$$\sum_{\text{dense } c} \frac{K}{k} \cdot O(\lg \lg n) \;=\; O\left(\frac{n}{K} \cdot \frac{K \lg \lg n}{k}\right) \;=\; O\left(\frac{n}{\lg \lg n}\right),$$

which is $o(n)$. A sub-chunk spans k consecutive ones, so its length z_c is variable. The same sparse/dense dichotomy applies at this level. A sub-chunk is sparse when $z_c \geq k^2 = \Theta((\lg \lg n)^4)$ and dense otherwise. Sparse sub-chunks store the positions of their ones explicitly. Since these sub-chunks are disjoint and each stored offset fits in $O(\lg \lg n)$ bits, their total space is at most

$$\frac{n}{k^2} \cdot k \cdot O(\lg \lg n) \;=\; O\left(\frac{n}{\lg \lg n}\right),$$

which is $o(n)$.

A dense sub-chunk has length $O((\lg \lg n)^4) = o(\lg n)$. A precomputed table on all $(\lg n)/2$-bit patterns, of size $O(\sqrt{n} \lg n \lg \lg n) = o(n)$, resolves it after the appropriate local window has been extracted.

Dense sub-chunks satisfy $z_c < k^2 = \Theta((\lg \lg n)^4) = o(\lg n)$. A word read and a shift extract a pattern of length at most $\lfloor (\lg n)/2 \rfloor$ containing the sub-chunk, padded if necessary. A precomputed table indexed by this pattern and by the local rank j_s returns the offset of the requested one inside the pattern. With $2^{(\lg n)/2} = \Theta(\sqrt{n})$ patterns of $(\lg n)/2$ entries at $O(\lg \lg n)$ bits each, the table occupies $O(\sqrt{n} \lg n \lg \lg n) = o(n)$ bits. This is the four-Russians technique of Jacobson [11] for rank, extended to select by Clark [12].

The three levels assemble into a constant-time query. First the chunk

index c and local rank j_c are computed, and $P_1[c]$ gives the chunk start. For a sparse chunk, the answer is $P_1[c]$ plus the j_c-th stored offset. For a dense chunk, the sub-chunk index $\ell = \lfloor (j_c - 1)/k \rfloor$ and the local rank $j_s = ((j_c - 1) \bmod k) + 1$ give the sub-chunk start relative to $P_1[c]$ through the sub-chunk offset table. Sparse cases are addressed by flag bitvectors with rank support. The flag tells whether the current chunk or sub-chunk stores explicit offsets, and its rank gives the number of earlier explicit offset groups. A sparse sub-chunk then reads the local offset of its j_s-th one. A dense sub-chunk gets the same local offset from the dense-pattern table. Each level's branch choice is a single comparison, and the query reads $O(1)$ words.

Sketch. Each storage component is $o(n)$ bits. The top-level array P_1 costs $O(n/\lg n)$, the sparse-chunk offsets cost $O(n/\lg n)$, the dense-chunk recursive offsets cost $O(n/\lg\lg n)$, the sparse-sub-chunk offsets cost $O(n/\lg\lg n)$, and the bottom-level lookup table costs $o(n)$. The flag bitvectors and their rank indexes have one entry per chunk or sub-chunk, so they are absorbed by these bounds. A query computes the chunk and sub-chunk indices arithmetically, reads the relevant stored offsets, and either returns an explicit offset or uses one table lookup at the last level. Thus it performs $O(1)$ word probes. □

The same construction, applied separately to $\overline{B}$, handles select_0. Storing both structures doubles the auxiliary space while keeping it $o(n)$. Combined with the rank structure of Subsection 2.1.1, we obtain a bitvector representation that supports rank_b and select_b for $b \in \{0, 1\}$ in $O(1)$ time, with auxiliary space $o(n)$ bits on top of the verbatim bitvector.

2.1.3 Lower bounds

The structures above store B verbatim and add an index for fast queries. The next question is therefore model-specific. Among representations that leave the input bits untouched, how much auxiliary space is forced by constant-time rank and select?

The *indexing* model used by Miltersen [13] isolates this question.

Miltersen's indexing model stores the input verbatim and allows only an auxiliary index. The restriction is natural for arbitrary bitvectors, because the n bits of B already meet the information-theoretic minimum.

Definition 2.1.3 (Indexing data structure) *An indexing data structure for a problem on inputs $x \in \{0,1\}^n$ consists of the input x stored verbatim in $\lceil n/\omega \rceil$ words, together with an r-bit index $\phi(x)$ stored in $\lceil r/\omega \rceil$ words.*

The query algorithm against such a storage scheme is an adaptive cell-probe procedure. It may probe words of the verbatim input and words of the index, and all computation between probes is free. It has query time t if it makes at most t word probes in the worst case.The indexing model constrains the representation to store the bitvector B untouched while allowing an extra index. This matches Theorem 2.1.1 and Theorem 2.1.2, where B is accessed read-only and the auxiliary structures R_S, R_B, P_1, and the lookup tables form the index. The redundancy r in this model is precisely the size of the index. Miltersen's paper gives the following lower bound.

Theorem 2.1.3 (Index size lower bound [13]) *For any indexing data structure supporting* select_1 *on a bitvector of length* n *in the cell-probe model with word size* ω*, with* r *bits of index and query time* t*, it holds that*

$$3(r+2)(t\omega+1) \;\geq\; n.$$

For rank_1*, it holds that*

$$2(2r+\lg(\omega+1))t\omega \;\geq\; n\lg(\omega+1).$$

In the regime $\omega = \Theta(\lg n)$ and $t = O(1)$, the select bound becomes $r = \Omega(n/\lg n)$ and the rank bound becomes $r = \Omega(n \lg\lg n/\lg n)$. The rank bound matches Theorem 2.1.1 up to a constant factor. The select bound is weaker: against the $O(n \lg\lg n/\lg n)$ select indexes cited by Miltersen, it misses a factor of $\lg\lg n$. Miltersen proves the rank statement through a separate counting reduction to vector addition. We sketch the select statement, because it already explains how an indexing lower bound reconstructs the input from too small an auxiliary index.

Sketch of the select bound. First set $\omega = 1$, so every cell probe reads one bit. Assume $t \geq 1$ and restrict attention to inputs $x \in \{0,1\}^n$ of Hamming weight $m = \lfloor (n/3-1)/t \rfloor$. Let $\tau'(x)$ be the length-mt trace obtained by running the m select queries on x and concatenating the probed bits. From this trace form $\tau(x)$ by zeroing every position that records an index bit and every later position, in trace order, that records an input bit already recorded before. Given $\phi(x)$ and $\tau(x)$, we reconstruct the original trace by simulating the same queries in increasing order of j. Index probes are answered from $\phi(x)$, repeated input probes reuse the earlier reconstructed value for that input position, and first-time input probes read the corresponding position of $\tau(x)$. The reconstructed trace gives all select answers for $j = 1, \ldots, m$, hence all one-positions of x and therefore x itself.

The string $\tau(x)$ has length mt and Hamming weight at most m, because only first probes to one-bits of the input can contribute ones. Since the pair $(\phi(x), \tau(x))$ identifies every length-n bitvector of Hamming weight m, counting possible pairs gives

$$2^r \sum_{i=0}^{m} \binom{mt}{i} \;\geq\; \binom{n}{m}.$$

Miltersen bounds this binomial tail against the type class of weight-m inputs, obtaining

$$r+1 \;\geq\; \lg\binom{n}{m} - \lg\binom{mt}{m}.$$

Since $m \leq n/3$ and $mt \leq n/3$, each factor in $\binom{n}{m}/\binom{mt}{m}$ is at least 2, so $r+1 \geq m$. This gives $3(r+2)(t+1) \geq n$ for $\omega = 1$. A word-probe algorithm with word size ω can be simulated by a bit-probe algorithm using at most $t\omega$ probes, so substituting $t\omega$ for t gives $3(r+2)(t\omega+1) \geq n$. □

[14]: Golynski (2007), *Optimal lower bounds for rank and select indexes*

Golynski strengthens the model before proving the matching lower bound [14]. The algorithm may inspect the index freely, and only the probes to B are charged. Since a constant number of word probes to B

reveal only $O(\lg n)$ bits when $\omega = \Theta(\lg n)$, a lower bound in this stronger bit-probe model also applies to the indexing structures above.

Theorem 2.1.4 (Tight rank and select index bound [14]) *Any indexing data structure for* rank_1 *or* select_1 *on a bitvector of length* n*, with an* r*-bit index, unlimited probes to the index, unlimited computation, and* $O(\lg n)$ *bit-probes per query to the bitvector, has index size* $r = \Omega(n \lg\lg n / \lg n)$*.*

The proof fixes an index value and turns the remaining probes to B into a decision process over many queries. For rank, write the bit-probe budget as $t = f \lg n$, split the bitvector into blocks of length $k = t + \lg n$, so that $p = \Theta(n/\lg n)$, and place the queries at block endpoints. At each leaf, the probed bits reveal only part of the input, while the correct endpoint answers force the population count of every block. A counting argument shows that one fixed index value can cover only a $2^{-\Omega(p \lg\lg n)}$ fraction of the 2^n possible bitvectors. Covering all inputs therefore requires $2^r \geq 2^{\Omega(p \lg\lg n)}$ index values, so $r = \Omega(n \lg\lg n / \lg n)$. For select, the same choices-tree argument is applied to bitvectors with $n/2$ ones and to queries whose selected positions delimit the corresponding blocks.

Golynski's paper also gives a density-sensitive lower bound. If the bitvector has m ones and the query algorithm uses t bit-probes to the bitvector, the index must have size $\Omega((m/t)\lg t)$ bits. For $m = \Theta(n)$ and $t = \Theta(\lg n)$, this recovers $\Omega(n \lg\lg n / \lg n)$. For sparser bitvectors, the dependence on m points toward the compressed representations studied in Subsection 2.1.4.

The $\Omega((m/t)\lg t)$ bound is density-sensitive. For very sparse bitvectors, where $m \ll n$, the lower bound allows less redundancy if the query algorithm is allowed more probes.

For a set represented by its characteristic bitvector, rank and select also give predecessor. Count how many stored elements are at most the query value, then select that counted element. Beame and Fich [15] established cell-probe lower bounds for predecessor, so re-encoded bitvectors with rank and select also sit in the predecessor-search setting.

[15]: Beame et al. (2002), *Optimal bounds for the predecessor problem and related problems*

The lower bound of Theorem 2.1.4 applies to indexing data structures, where B itself is stored untouched. Pătraşcu [16] showed that this restriction is essential. If the bitvector is re-encoded, the redundancy can be pushed far below $\Theta(n \lg\lg n / \lg n)$, at the cost of an exponential trade-off with query time. A related line of work by Grossi, Orlandi, Raman, and Srinivasa Rao [17] lowers the redundancy of fully indexable dictionaries through a parametric family of representations refining Theorem 2.1.5.

[16]: Patrascu (2008), *Succincter*

[17]: Grossi et al. (2009), *More haste, less waste: Lowering the redundancy in fully indexable dictionaries*

Theorem 2.1.5 (Succincter [16]) *On a word RAM with cells of* $\Omega(\lg n)$ *bits, a bitvector of length* n *with* m *ones can be stored in*

$$\lg \binom{n}{m} + \frac{n}{((\lg n)/t)^t} + \widetilde{O}(n^{3/4}) \quad \textit{bits},$$

supporting rank_1 *and* select_1 *queries in* $O(t)$ *time, for any integer* $t \geq 1$*.*

Setting $t = 2$ in Theorem 2.1.5 gives redundancy $O(n/(\lg n)^2)$, since the additive $\widetilde{O}(n^{3/4})$ term is lower order, and this is strictly smaller than the Jacobson-Clark bound. Setting $t = \lceil \lg\lg n \rceil$ gives redundancy

$$O\left(\frac{n}{((\lg n)/(\lg\lg n))^{\lg\lg n}}\right) = \frac{n}{2^{\Omega((\lg\lg n)^2)}}.$$

The ceiling in t only changes the constant hidden in Ω, and the additive $\widetilde{O}(n^{3/4})$ term is dominated by the displayed quantity. The lower bound of Theorem 2.1.4 is compatible with this upper bound, because Pătraşcu's construction departs from the indexing model and re-encodes the data through spill-over representations, which carry fractional bits of entropy between levels.

The re-encoding step has to solve a small arithmetic problem. A local state set X may not have size exactly equal to a power of two, so storing an element of X in $\lceil \lg |X| \rceil$ bits wastes part of a bit at each level. A *spill-over representation* replaces this plain code by M memory bits and a spill value in $\{0, 1, \ldots, K-1\}$, with $K \cdot 2^M \geq |X|$. For a target redundancy parameter r, Pătraşcu [16] chooses K with $r \leq K < 2r$, which makes the excess at one level at most $2/r$ bits per represented state. Composing this local representation through the augmented search tree used by the construction yields redundancy $n/((\lg n)/t)^t$. The query time is $O(t)$ because a query touches t levels. Thus Theorem 2.1.5 shows that the Jacobson-Clark bound is optimal only within the indexing model, and that dropping the verbatim-storage restriction permits exponential savings in the time parameter.

A spill-over representation stores M memory bits plus a spill value in $\{0, \ldots, K-1\}$. The spill carries the fractional part that would otherwise be rounded up at the same level.

The same space term now has two different meanings. In an indexing structure, redundancy is the extra space needed after the bitvector has already been stored verbatim. In a re-encoding, the whole representation is measured against $\lg \binom{n}{m}$. The next section develops this second viewpoint for one bitvector, because it becomes the set-by-set baseline for collections.

2.1.4 Compressed bitvectors

The lower bounds above explain the cost of adding an index to a verbatim bitvector. For the single-set baseline used later, we also need to encode the bitvector itself. A bitvector of length n with m ones is one of $\binom{n}{m}$ possibilities, so the information-theoretic cost of encoding it is

$$\mathcal{B}(n, m) = \left\lceil \lg \binom{n}{m} \right\rceil = m \lg(n/m) + O(m)$$

bits for $m \leq n/2$,which is much smaller than n when $m \ll n$. To use this encoding as a query structure, the compressed representation must still support rank and select. Raman and Rao [18] give an intermediate dictionary structure with membership and rank for stored elements, and Raman, Raman, and Rao [19] give the fully indexable bitvector representation.

[18]: Raman et al. (1999), *Static dictionaries supporting rank*

[19]: Raman et al. (2007), *Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets*

Raman and Rao combine the classical FKS perfect-hashing dictionary [20] with a quotient representation to achieve the following.

The quotient representation stores only enough information to distinguish an element from universe values sent to the same hash location. This saves a linear term compared with storing every universe element outright.

[20]: Fredman et al. (1984), *Storing a sparse table with 0 (1) worst case access time*

Theorem 2.1.6 (Quotient dictionary with $O(1)$ member rank [18]) *A subset $S \subseteq [1..n]$ of size m can be stored in $m \lg n - \Theta(m) + O(\lg \lg n)$ bits so that membership for universe elements and rank for elements of S are answered in $O(1)$ time.*

The bound in Theorem 2.1.6 combines the main dictionary construction with a block-wise refinement of the stored ranks. The dictionary first

places the elements of S by perfect hashing. Instead of storing each universe element in full, it stores a quotient value that identifies the element among the universe values sent to the same hash-table location. Membership queries recompute the hash location and quotient and compare them with the stored entry. Rank is available only after membership has found an element of S. A direct version stores the full rank with each entry, while the refined version divides the sorted set into blocks, stores the local rank inside the block with each dictionary entry, and stores block starts separately. A constant-size search over those starts identifies the block in the corollary used here, and the global rank is the block's starting rank plus the local rank. This is not yet a fully indexable bitvector at the stated space bound. It does not also give select, and its rank guarantee is for stored elements rather than arbitrary positions. It does show how hashing and local rank information can beat the explicit $m \lg n$ representation while keeping constant-time queries.

Raman, Raman, and Rao [19] then answer the bitvector query problem. Their construction, known as the RRR structure, replaces fixed-width blocks by class/offset codes whose total length follows $\mathcal{B}(n, m)$. It then adds prefix-sum support and lookup tables so that the compressed code can still answer rank and select.

[19]: Raman et al. (2007), *Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets*

The bitvector $B[1..n]$ is partitioned into $\lceil n/b \rceil$ blocks of length $b = \lfloor (\lg n)/2 \rfloor$, so each full block fits in a constant number of words and the universal tables remain sublinear. The final block may be shorter, and this only changes lower-order terms. Each block is replaced by a (class, offset) pair in place of its verbatim content.For a full block $B_j = B[(j-1)b+1..jb]$, define its *class* c_j as the population count,

$$c_j \;=\; \sum_{i=(j-1)b+1}^{jb} B[i] \;\in\; \{0, 1, \ldots, b\},$$

and its *offset* o_j as the index of the block's pattern within the lexicographic enumeration of the $\binom{b}{c_j}$ b-bit patterns of population count c_j. The pair (c_j, o_j) uniquely determines B_j, and inversely B_j determines (c_j, o_j). The class fits in $\lceil \lg(b+1) \rceil$ bits. The offset fits in $\lceil \lg \binom{b}{c_j} \rceil$ bits, which depends on the class.

The variable-length offsets save space only if a query can find the correct block and the correct offset field in constant time. RRR therefore stores searchable prefix sums, prefix sums equipped with constant-time search by cumulative value, for the classes and for the offset lengths. The first structure locates the block containing a requested occurrence, the second locates the encoded block in the offset stream, and lookup tables on all b-bit patterns answer the remaining in-block query.

Theorem 2.1.7 (Fully indexable dictionary (RRR) [19]) *A bitvector $B[1..n]$ with m ones can be stored in*

$$\mathcal{B}(n, m) + O(n \lg\lg n / \lg n) \;=\; \mathcal{B}(n, m) + o(n) \quad \textit{bits},$$

where $\mathcal{B}(n, m) = \lceil \lg \binom{n}{m} \rceil$, so that $\mathrm{rank}_b(B, i)$ and $\mathrm{select}_b(B, i)$ for $b \in \{0, 1\}$ are answered in $O(1)$ time on the word RAM with word size $\omega = \Omega(\lg n)$.

Sketch. Partition B into $p = \lceil n/b \rceil$ blocks of length at most $b = \lfloor (\lg n)/2 \rfloor$, encode each block by its (c_j, o_j) pair, and store the class array together with the variable-length offset stream. The offset stream has length at most

$$\sum_{j=1}^{p} \left\lceil \lg \binom{b_j}{c_j} \right\rceil \;\le\; p + \lg \prod_{j=1}^{p} \binom{b_j}{c_j} \;\le\; \mathcal{B}(n, m) + O(n/\lg n),$$

where b_j is the length of block j and the last inequality is Vandermonde's identity over the block partition. The class array, the searchable prefix sums for class values, and the searchable prefix sums for offset lengths each use $O(n \lg\lg n/\lg n)$ bits. Lookup tables over all b-bit block patterns use $2^b \operatorname{polylog}(n) = o(n)$ bits and return rank or select inside a decoded block. A rank query adds the prefix sum of the classes before the block to the table answer inside the block. A select query uses the searchable prefix sums of the class array to find the block containing the requested one, then uses the block table for the final position. The stated space and time bounds follow. □

The term $\mathcal{B}(n, m)$ is the single-set component of Definition 1.2.3. For a collection over a universe of size u, encoding each set separately gives the counting term $H_{wc}(\mathcal{S}) = \sum_i \lg \binom{u}{|S_i|}$, and Theorem 2.1.7 shows that this term is compatible with rank and select up to lower-order space. Later compressibility measures start from this set-by-set baseline and ask how much additional space can be saved by using structure between sets.

2.2 Sparse bitvectors

Section 2.1 shows that a bitvector with m ones can be stored near the population-count term $\mathcal{B}(n, m)$ while still supporting rank and select. When $m \ll n$, the positions of the ones form a monotone sequence that can be encoded directly, without paying for every block of B. A set of m elements drawn from a universe of size n is one of $\binom{n}{m}$ possibilities, so the information-theoretic minimum is

$$\mathcal{B}(n, m) \;=\; \lceil \lg \tbinom{n}{m} \rceil \;=\; m \lg(n/m) + O(m) \quad \text{bits},$$

a quantity much smaller than n in the sparse regime. Theorem 2.1.7 reaches $\mathcal{B}(n, m) + o(n)$ bits with constant-time rank and select through class/offset encoding over blocks of length $(\lg n)/2$. Elias-Fano takes the other route. It splits each one position into a compact high part and an explicit low part, obtaining a simpler layout that trades slower rank and access for fast select. The two approaches stay near the same counting term, but they distribute the cost differently across operations.

The construction is named after the sparse integer-set representations of Elias [21] and Fano [22]. The word-RAM presentation follows Grossi and Vitter [23].

2.2.1 Elias-Fano

Let $B[1..n]$ be a bitvector with exactly m ones, and let $x_h = \operatorname{select}_1(B, h) - 1$ be the zero-based position of the h-th one. Then $0 \le x_1 < x_2 < \cdots < x_m < n$. The sequence $(x_1, \ldots, x_m)$ is a monotone sequence of integers in the half-open universe $[0, n)$, and the bitvector and the sequence

are equivalent because $\text{select}_1(B, h) = x_h + 1$. RRR compresses B at the block level, without acknowledging monotonicity. Elias-Fano exploits monotonicity directly. Split each x_h into a low part and a high part,

$$x_h \;=\; q_h \cdot 2^{w-z} + r_h,$$

where $z = \lfloor \lg m \rfloor$, $w = \lceil \lg n \rceil$, the high part q_h occupies the top z bits of x_h, and the low part r_h occupies the remaining $w - z$ bits. The low parts $r_1, r_2, \ldots, r_m$ are stored verbatim as a fixed-width array L, one $(w-z)$-bit field per element. The high parts are stored in a bitvector H obtained by unary-coding the differences of consecutive quotients,

$$H \;=\; 0^{q_1}\,1\,0^{q_2-q_1}\,1\,\cdots\,0^{q_m-q_{m-1}}\,1.$$

The ones in H mark the m quotients, and the zeros interpolate the gaps between successive quotient values. Since $q_h \in [0, 2^z)$ and $2^z \le m$, there are at most m zeros and exactly m ones, so $|H| \le 2m$.

Figure 2.3 illustrates the encoding on a concrete example. The positions $5, 7, 8, 15, 32$ form a strictly increasing sequence in $[0, 36)$, so $n = 36$, $m = 5$, $w = 6$, and $z = \lfloor \lg 5 \rfloor = 2$. The low parts are the bottom $w - z = 4$ bits of each position and form the array L. The top $z = 2$ bits of each position form the quotient sequence $(0, 0, 0, 0, 2)$, unary-coded into H.

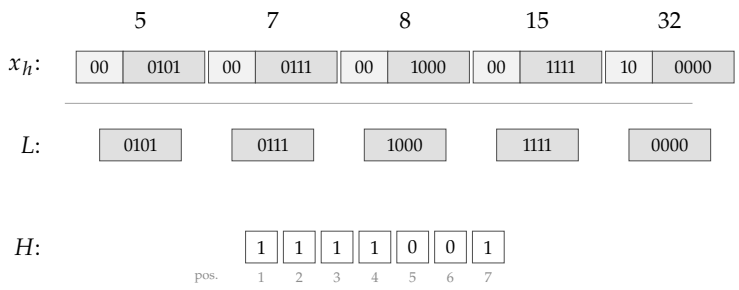

Figure 2.3: Elias-Fano encoding of the set $\{5, 7, 8, 15, 32\}$ with universe $[0, 36)$ and $m = 5$, so $w = 6$ and $z = \lfloor \lg 5 \rfloor = 2$. Each element is split into a high part of z bits and a low part of $w - z = 4$ bits. The array L stores the low parts, while $H = 1111001$ unary-codes the successive increments of the high parts $(0, 0, 0, 0, 2)$. With one-based positions in H, the j-th one lies at $p = \text{select}_1(H, j)$, so $q_j = p - j$ recovers the high part and $L[j]$ gives the low part.

Adding a constant-time select index to H turns this split into a constant-time retrieval scheme with only a linear additive term [23].

[23]: Grossi et al. (2000), *Compressed suffix arrays and suffix trees with applications to text indexing and string matching*

Theorem 2.2.1 (Elias-Fano encoding [23]) *Given m integers in sorted order, each containing w bits, with $m < 2^w$, there is a representation of*

$$m\,\big(2 + w - \lfloor \lg m \rfloor\big) \;+\; O\big(m / \lg \lg m\big) \quad \textit{bits}$$

that retrieves the h-th integer in $O(1)$ time on the word RAM with word size $\omega = \Omega(w)$. For the characteristic bitvector of a subset $S \subseteq [1..n]$ of size $m < n$, we encode the zero-based one positions with $w = \lceil \lg n \rceil$. The bound becomes $m(\lceil \lg n \rceil - \lfloor \lg m \rfloor) + 2m + O(m/\lg \lg m) \le m\lceil \lg(n/m) \rceil + 3m + O(m/\lg \lg m)$ bits, and the h-th retrieval gives $\text{select}_1(B, h) - 1$.

Proof. We follow the construction of Grossi and Vitter [23]. Take $z = \lfloor \lg m \rfloor$ and split each x_h into the quotient $q_h \in [0, 2^z)$ and the remainder $r_h \in [0, 2^{w-z})$, as above. Encode L as the concatenation $r_1 r_2 \cdots r_m$ of fixed-width $(w-z)$-bit fields, totalling $|L| = m(w-z)$ bits. Encode H as the unary concatenation of the differences $q_1, q_2 - q_1, \ldots, q_m - q_{m-1}$, which has m ones (one per integer) and at most $2^z - 1 \le m$ zeros (one per occupied cell of the quotient universe), so $|H| \le 2m$ bits.

[23]: Grossi et al. (2000), *Compressed suffix arrays and suffix trees with applications to text indexing and string matching*

The two arrays together occupy $|L| + |H| \le m(w - z) + 2m = m(2 + w - \lfloor \lg m \rfloor)$ bits. To support select_1 on H in constant time, we augment H with a select structure of $O(|H|/\lg\lg|H|) = O(m/\lg\lg m)$ bits. The total space is $m(2 + w - \lfloor \lg m \rfloor) + O(m/\lg\lg m)$ bits.

To recover x_h we must compute its quotient q_h and its remainder r_h in constant time. The quotient is stored implicitly in H through the position of its h-th one. Indeed, the h-th one of H lies at position $q_h + h$, because the unary code for the differences places the h-th one after exactly q_h zeros cumulatively. Therefore

$$q_h \;=\; \text{select}_1(H, h) \;-\; h,$$

computable in $O(1)$ time by the auxiliary select structure. The remainder is read directly from L through the fixed-width access $r_h = L[h]$. The answer is $x_h = q_h \cdot 2^{w-z} + r_h$, a constant-time composition of two word-level arithmetic operations. □

The space calculation in Theorem 2.2.1 has two parts. The remainder array L costs $m(w - z)$ bits, within an additive m-bit slack of $m\lceil \lg(n/m) \rceil$. The high-part bitvector H costs at most $2m$ bits, because it has one 1 per stored integer and its total number of zeros is the last quotient, which is smaller than $2^z \le m$. Thus the leading term is $m \lg(n/m) + O(m)$ bits, within a linear additive term of $\mathcal{B}(n, m) = \lg \binom{n}{m}$ by the standard Stirling bounds.

The retrieval algorithm in the proof of Theorem 2.2.1 returns the zero-based position $x_h = \text{select}_1(B, h) - 1$, so the usual one-based select answer is obtained by adding one. A select query on H recovers the high part, and a fixed-width lookup in L recovers the low part. Rank has the opposite input shape. Given a position $i \in [1..n]$, computing $\text{rank}_1(B, i)$ requires identifying the largest h such that $x_h \le i - 1$, so the standard implementation binary-searches $\text{select}_1(B, \cdot)$, trading $O(\lg m)$ time for $O(1)$ space overhead. Access reduces to rank by $\text{access}(B, i) = [\text{rank}_1(B, i) > \text{rank}_1(B, i - 1)]$, inheriting the same $O(\lg m)$ cost. This is the operational asymmetry that distinguishes Elias-Fano from Theorem 2.1.7. The latter buys $O(1)$ rank and select simultaneously, at the cost of a two-level segment structure and variable-length offsets. Elias-Fano buys $O(1)$ select at the cost of logarithmic rank and access.

2.2.2 Rank and access

Theorem 2.2.1 supports select_1 in $O(1)$ time but leaves rank_1 and access at $\Theta(\lg m)$, the cost of a binary search over $\text{select}_1(B, \cdot)$. Okanohara and Sadakane [24] give a different access path at the same leading space term by using the sparse component of their SDarray construction, called sarray in the paper. The construction keeps the Elias-Fano split but fixes the low-part width to the sparse scale and pads the high-part bitvector so that it has m ones and m zeros. This dense bit profile makes select_0 on the high-part bitvector efficient.

[24]: Okanohara et al. (2007), *Practical entropy-compressed rank/select dictionary*

Theorem 2.2.2 (sarray, sparse component of sdarray [24]) *A bitvector* $B[1..n]$ *with* m *ones can be represented in*

$$m\lceil \lg(n/m) \rceil \; + \; 2m \; + \; o(m) \quad \textit{bits},$$

supporting $\mathrm{select}_1(B, j)$ *in* $O(\lg^4 m/\lg n)$ *time and* $\mathrm{rank}_1(B, i)$ *and* $\mathrm{access}(B, i)$ *in* $O(\lg(n/m) + \lg^4 m/\lg n)$ *time, on the word RAM with word size* $\omega = \Omega(\lg n)$.

Vigna [25] adapts Elias-Fano to inverted indexes. Forward pointers speed sequential access, and skip pointers on the negated unary code support finding the smallest stored value above a query bound.

The sarray representation is a tuned Elias-Fano layout. Let $x_j = \mathrm{select}_1(B, j) - 1$ be the zero-based position of the j-th one of B, and set $\ell = \lceil \lg(n/m) \rceil$. Write $x_j = q_j 2^\ell + r_j$, where $0 \le r_j < 2^\ell$. The low parts form the fixed-width array $L[1..m]$, using $m\ell$ bits. The high-part bitvector $H[1..2m]$ has its j-th one at position $q_j + j$, with trailing zeros added if needed, so H contributes exactly $2m$ bits. Then

$$\mathrm{select}_1(B, j) \; = \; \big(\mathrm{select}_1(H, j) - j\big)2^\ell + L[j] + 1.$$

Thus select_1 on B reduces to one select on H and one fixed-width read on L. The select on H is answered by darray, which contributes the $O(\lg^4 m/\lg n)$ term in the theorem. The extra operation needed for rank and access is select_0 on H.

The darray subroutine is applied to the dense bitvector H of length $2m$. For a bitvector of length N, Okanohara and Sadakane use parameters $L = O(\lg^2 N)$, $L_2 = O(\lg^4 N)$, and $L_3 = O(\lg N)$, so on H these become $O(\lg^2 m)$, $O(\lg^4 m)$, and $O(\lg m)$. To support select_0, apply the same structure to the zeros of H. Blocks spanning more than L_2 positions store zero positions explicitly, while shorter blocks sample every L_3-th zero and scan within one sampled interval. This gives $o(m)$ auxiliary bits and worst-case time $O(\lg^4 m/\lg n)$ for select_0 on a word RAM with $\Theta(\lg n)$-bit words.

Rank on B reduces to select_0 on H together with a local search on L. Given a query position $i \in [1..n]$, put $y = i - 1$, $q = \lfloor y/2^\ell \rfloor$, and $r = y \bmod 2^\ell$. If $q = 0$, no stored position has smaller quotient. If $q > 0$, the number of ones of B with quotient strictly less than q is the number of ones of H before the q-th zero, namely $\mathrm{select}_0(H, q) - q$. The next zero of H gives the number of ones with quotient at most q, so the candidates with quotient exactly q form one contiguous range in L. There are at most $2^\ell < 2n/m$ such candidates, so a binary search over their remainders costs $O(\lg(n/m))$ time. The total cost of rank is $O(\lg(n/m) + \lg^4 m/\lg n)$, and access follows the same route.

Theorem 2.2.2 and Theorem 2.1.7 both start from the population-count scale $\mathcal{B}(n, m) = \lg \binom{n}{m}$, but they reach it with different additive terms and different query costs. Theorem 2.1.7 achieves $\mathcal{B}(n, m) + o(n) + O(\lg \lg n)$ bits with $O(1)$ rank and select for both values, a uniform guarantee across all densities. Theorem 2.2.2 uses $m\lceil \lg(n/m) \rceil + 2m + o(m)$ bits, within $O(m)$ bits of $\mathcal{B}(n, m)$, with select_1 in $O(\lg^4 m/\lg n)$ and rank_1 and access in $O(\lg(n/m) + \lg^4 m/\lg n)$. The $\lg^4 m/\lg n$ term is the worst-case overhead from select on the dense high-part bitvector, and the original analysis treats it as constant in the practical word-RAM regime.

RRR carries the apparatus of class/offset encoding over every block of the bitvector regardless of density, with variable-length offsets and

a searchable prefix-sum structure. sdarray avoids this apparatus by committing to a single global split of each one position into high and low parts, and recovers the operations from select structures on H. When $m \ll n$, the high-part bitvector has length only $2m$ and the lower-order space is tied to m, but rank still pays the $O(\lg(n/m))$ search inside one high-part bucket. Neither representation dominates the other. The density m/n and the operation mix determine which is preferable.

Each gap $g = s_i - s_{i-1}$ needs $1+\lfloor \lg g \rfloor$ bits of binary representation. When the gap distribution concentrates on small values, $\text{gap}(\hat{S})$ falls well below the $2m$ bits spent by Elias-Fano on the unary H, and the Sadakane-Grossi bound beats Elias-Fano by more than a constant factor.

[26]: Sadakane et al. (2006), *Squeezing succinct data structures into entropy bounds*

The Elias-Fano $2m$-bit overhead on H is tight against the gap distribution only up to a constant. When the gaps between consecutive ones are highly clustered, H contains large runs of zeros that encode redundant information, and the gap distribution itself admits coding well below $2m$ bits. Sadakane and Grossi [26] closed this gap with *entropy-bound indexable dictionaries*, a construction that achieves the following bound.

$$\min\{nH_k,\ \text{gap}(\hat{S})\} \ +\ O(n \lg\lg n/\lg n) \quad \text{bits},$$

Here H_k is the k-th order empirical entropy of B, and $\text{gap}(\hat{S}) = \sum_{i=1}^{m}(1 + \lfloor \lg(s_i - s_{i-1}) \rfloor)$ sums the minimum binary lengths of the gaps between consecutive ones, with $s_0 = 0$.

Like Theorem 2.1.5, the Sadakane-Grossi construction steps outside the indexing model of Subsection 2.1.3 and avoids the lower bounds of Theorem 2.1.3 and Theorem 2.1.4.

The construction is structurally different from sdarray. Sadakane and Grossi replace the verbatim bitvector B with a compressed representation from which any $O(\lg n)$-bit substring decodes in constant time, and build rank/select on top of the compressed form. The nH_k argument of the min is tight on dense inputs with symbol-level regularity. The $\text{gap}(\hat{S})$ argument is tight on sparse inputs with clustered gaps. The min adapts to whichever term is smaller on the given input, without committing at encoding time.

The bound is most useful when entropy or gap structure is much smaller than the population-count term. Rank and select remain $O(1)$ under the word-RAM assumption, but the redundancy term is tied to the universe length n, so very sparse inputs may still favor sdarray's $o(m)$ overhead. The result confirms that neither $m\lceil \lg(n/m) \rceil$ nor $\mathcal{B}(n, m)$ is the last word on the sparse-bitvector problem.

The quantity $\mathcal{B}(n, m) = \lg \binom{n}{m}$, in the single-set notation of this chapter, is the independent-encoding cost H_{wc} of Definition 1.2.3 evaluated on a single set. Returning to the collection notation of Definition 1.2.1, with u for the universe size, n for the total size $\sum_i |S_i|$, and m for the number of sets, applying Theorem 2.2.2 (or Theorem 2.1.7) to each characteristic bitvector of $\mathcal{S} = \{S_1, \ldots, S_m\}$ yields a representation of total space $H_{wc}(\mathcal{S}) + O(n + m \lg u)$ bits, the set-by-set baseline. The single-set baseline still leaves one use of binary dictionaries unexplained. In many later structures the stored object is not a subset but a sequence, where each position carries one of several symbols. Wavelet trees answer sequence queries by reducing the larger alphabet to a hierarchy of bitvectors, linking the dictionaries of this chapter to the labeled-tree structures used later.

2.3 Wavelet trees

Rank and select on a bitvector solve a problem with only two possible symbols. A sequence $S[1..n]$ over an alphabet $\Sigma = [1..\sigma]$ requires the same kind of positional navigation, but each position now carries one of σ possible values. Storing every symbol in a fixed $\lceil \lg \sigma \rceil$-bit field costs $n\lceil \lg \sigma \rceil$ bits, yet the fields do not by themselves count how many copies of a symbol occur in a prefix. The other direct reduction, one characteristic bitvector B_c for each symbol $c \in \Sigma$, gives rank and select immediately but repeats the length-n universe σ times, for $n\sigma$ bits. The binary viewpoint can still do the work without materializing one bitvector for every symbol. A single bit asks whether the current symbol lies in the left half or in the right half of the alphabet. Rank on that bitvector tells how many symbols of a prefix survive inside the chosen half, so it gives the prefix length for the next restricted sequence. Repeating the same question on the chosen half identifies the symbol after $\lceil \lg \sigma \rceil$ binary decisions, while preserving the position remapping needed by rank and select. Grossi, Gupta, and Vitter's wavelet tree [28] stores this hierarchy of binary decisions.

Chazelle's functional approach to multidimensional searching [27] is an older range-search ancestor. Grossi, Gupta, and Vitter [28] made the wavelet tree a sequence data structure over finite alphabets, with rank and select on the induced bitmaps.

[28]: Grossi et al. (2003), *High-order entropy-compressed text indexes*

Definition 2.3.1 (Sequence operations) *Let $S[1..n]$ be a sequence over the alphabet $[1..\sigma]$. For $c \in [1..\sigma]$, $1 \le i \le n$, and $j \ge 1$, we define three operations on S:*

- $\mathrm{access}(S, i)$*: returns the symbol $S[i]$.*
- $\mathrm{rank}_c(S, i)$*: returns $|\{k : 1 \le k \le i,\ S[k] = c\}|$, the number of occurrences of c in the prefix $S[1..i]$, with $\mathrm{rank}_c(S, 0) = 0$ by convention.*
- $\mathrm{select}_c(S, j)$*: returns the unique position k such that $S[k] = c$ and $\mathrm{rank}_c(S, k) = j$, or $\perp$ if fewer than j occurrences of c appear in S.*

For $\sigma = 2$, these operations reduce to access, rank_b, and select_b on the characteristic bitvector, recovering Definition 2.1.1 and Definition 2.1.2.

2.3.1 Structure and navigation

Begin with the whole alphabet interval $[1..\sigma]$ and the whole sequence S. At any stage, suppose the current interval is $[a_v, b_v]$ and the symbols outside it have been filtered out on this branch. The remaining symbols form an implicit subsequence S_v, kept in their original order. If $[a_v, b_v]$ contains more than one symbol, the balanced version asks the next binary question by splitting the interval at its midpoint. Let $m_v = \lfloor (a_v + b_v)/2 \rfloor$. For each position of S_v, store whether the symbol lies in the left half $[a_v, m_v]$ or in the right half $[m_v + 1, b_v]$. This gives a bitmap $B_v[1..|S_v|]$ with

$$B_v[i] = \begin{cases} 0 & \text{if the } i\text{-th symbol of } S_v \text{ lies in } [a_v, m_v], \\ 1 & \text{if it lies in } [m_v + 1, b_v]. \end{cases}$$

The zeros of B_v form the subsequence $S_0 = S_v\langle B_v = 0\rangle$ on which the same construction continues over $[a_v, m_v]$. The ones form $S_1 = S_v\langle B_v = 1\rangle$ over $[m_v + 1, b_v]$. The process stops when the interval is a singleton $[c, c]$. At that point no bitmap is needed, because every remaining position contains the same symbol c, and the label is already determined by the previous binary choices. The recursive construction is a balanced binary tree of height $\lceil \lg \sigma \rceil$: the root has $S_{\text{root}} = S$ and covers $[1..\sigma]$, while each leaf corresponds to one symbol, as in Figure 2.4.

At every level ℓ of the tree, the active node subsequences $\{S_v\}_{v \text{ at level } \ell}$ are disjoint sublists of S, because each symbol that has not yet reached

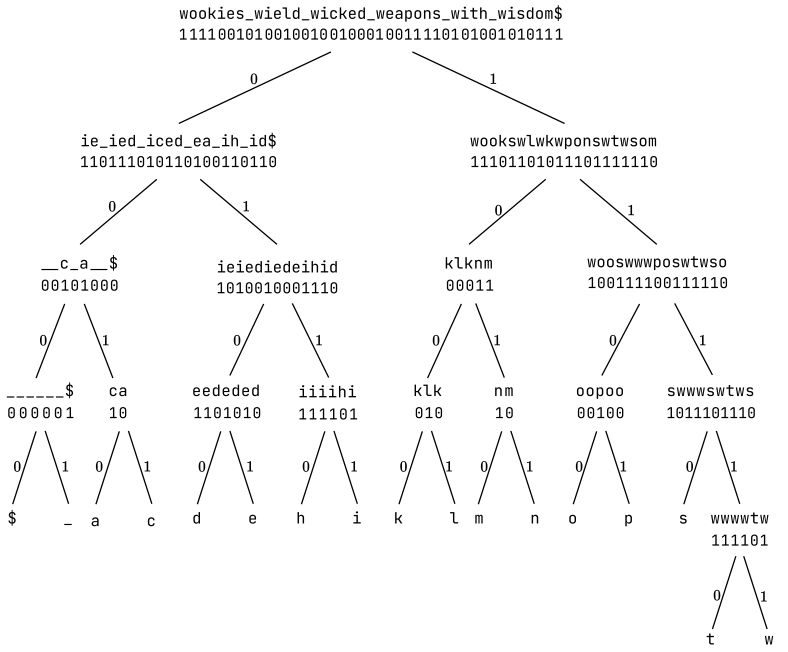

Figure 2.4: Wavelet tree for the string $S =$ `wookies_wield_wicked_weapons_with_wisdom$` over an alphabet of 17 symbols. Each internal node carries a subalphabet interval. Its bitmap encodes, for every symbol of the implicit subsequence, which side of the midpoint split it falls on. The subsequences are drawn beside each node for illustration. The structure stores only the topology and the bitmaps.

a leaf belongs to exactly one alphabet interval at that level. Therefore $\sum_{v \text{ at level } \ell} |B_v| \le n$, with equality as long as no node at level ℓ is a leaf. Deeper levels may store strictly fewer than n bits once some subtrees have already collapsed onto single-symbol leaves. Summing over the $\lceil \lg \sigma \rceil$ levels, the total length of the bitmaps is at most $n \lceil \lg \sigma \rceil$ bits. The constant-time rank and select auxiliaries of Theorem 2.1.1 and Theorem 2.1.2 can be charged level by level. Since the bitmaps on one level have total length at most n, the auxiliaries add $o(n)$ bits per level and $o(n) \lg \sigma$ bits in total. The tree topology, encoded with explicit child pointers, takes $O(\sigma \lg n)$ additional bits, since the tree has σ leaves and $\sigma - 1$ internal nodes each storing pointers and bitmap offsets of $O(\lg n)$ bits. The total space of the pointered wavelet tree is therefore $n \lceil \lg \sigma \rceil + o(n) \lg \sigma + O(\sigma \lg n)$ bits. The tree is built in $O(n \lg \sigma)$ time by linear-time processing at each internal node, where the node reads its subsequence S_v, writes its bitmap B_v, and partitions S_v into the two children's subsequences. Since the active subsequences at any level are disjoint and have total length at most n, the work at each level sums to $O(n)$, and summing over the $\lceil \lg \sigma \rceil$ levels yields the claimed bound.

Once every bitmap supports constant-time binary rank and select, the hierarchy turns the three sequence operations into root-to-leaf or leaf-to-root position remappings. Each level contributes one binary query and one child choice, so the cost is $O(\lg \sigma)$. Access, rank, and select differ only in where the traversal starts and in how the next edge is chosen.

Access reads the i-th symbol of S by a top-down traversal starting at the root. At the current node v, consult $B_v[i]$. If $B_v[i] = 0$, the i-th symbol of S_v lies in $[a_v, m_v]$ and appears in the left child's subsequence S_0. Its position within S_0 is the number of zeros of B_v up to position i, namely $\mathrm{rank}_0(B_v, i)$. Descend to the left child with index updated to this value. If $B_v[i] = 1$, descend symmetrically to the right child with index $\mathrm{rank}_1(B_v, i)$. The traversal terminates at a leaf $[c, c]$, which identifies the symbol $S[i] = c$. It performs at most $\lceil \lg \sigma \rceil$ binary rank queries and constant-time navigation at each step, for $O(\lg \sigma)$ time.

Symbol rank computes $\text{rank}_c(S, i)$, the number of occurrences of c in $S[1..i]$, by a top-down traversal along the fixed root-to-leaf path of c. At node v the direction of descent depends on c, not on the bitmap. When $c \le m_v$, the occurrences of c among $S[1..i]$ contribute to the zeros of $B_v[1..i]$. Update i to $\text{rank}_0(B_v, i)$ and descend to the left child. When $c > m_v$, update i to $\text{rank}_1(B_v, i)$ and descend to the right child. At the leaf $[c, c]$, the final index is $\text{rank}_c(S, i)$ for the original query. The path has length at most $\lceil \lg \sigma \rceil$ and each step performs one binary rank, yielding $O(\lg \sigma)$ time. Figure 2.5 traces a concrete instance of this descent.

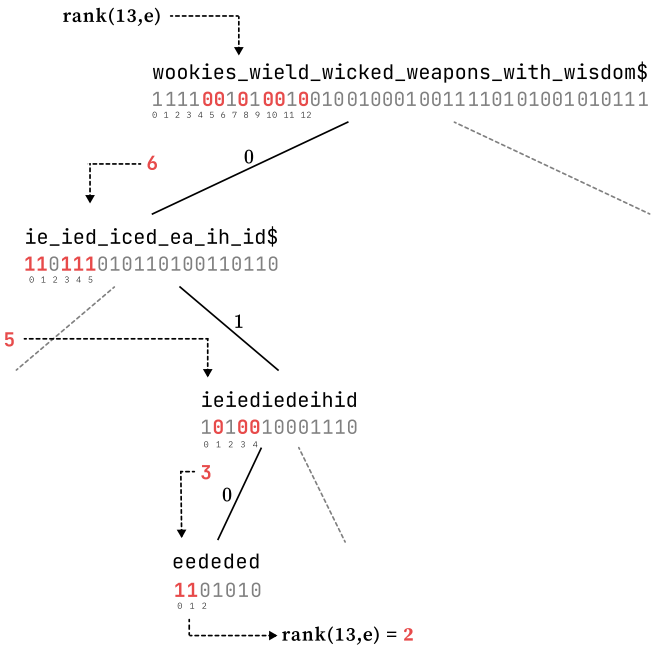

Figure 2.5: Trace of $\text{rank}_{\texttt{e}}(S, 13) = 2$ on the wavelet tree of Figure 2.4. The descent follows the root-to-leaf path of the target symbol `e`, remapping the query prefix length through one binary rank at each of four levels. The intermediate values $13 \to 6 \to 5 \to 3 \to 2$ trace the query index from the root to the leaf $[\texttt{e}, \texttt{e}]$. Each remapping is a rank_0 or rank_1 on the local bitmap, selected by whether `e` falls in the left or right half of the current alphabet interval.

Symbol select computes $\text{select}_c(S, j)$, the position of the j-th occurrence of c in S, by a bottom-up traversal starting at the leaf $[c, c]$ with $j_0 = j$. If fewer than j copies of c reach this leaf, the query returns $\perp$. Otherwise, moving from the current node v' to its parent v, the index is mapped through a binary select on B_v. If v' is the left child of v, the j-th occurrence of c inside $S_{v'}$ corresponds to the j-th zero of B_v, so update j to $\text{select}_0(B_v, j)$. If v' is the right child, update j to $\text{select}_1(B_v, j)$. When the traversal reaches the root, the final index is the answer $\text{select}_c(S, j)$ in S. It performs at most $\lceil \lg \sigma \rceil$ binary select queries and constant-time navigation at each step, for $O(\lg \sigma)$ time.

Combining the level-by-level bitmap bound, the explicit topology term, and the traversal costs gives the plain wavelet-tree representation.

Theorem 2.3.1 (Wavelet tree [28]) *A sequence $S[1..n]$ over its effective alphabet $[1..\sigma]$, with $\sigma \le n$, can be represented in*

$$n\lceil \lg \sigma \rceil \; + \; o(n) \lg \sigma \; + \; O(\sigma \lg n) \quad \textit{bits}$$

so that $\text{access}(S, i)$, $\text{rank}_c(S, i)$, *and* $\text{select}_c(S, j)$ *are supported in $O(\lg \sigma)$ time on the word RAM with word size $\omega = \Omega(\lg n)$.*

Sketch. The level-by-level accounting above gives at most $n\lceil \lg \sigma \rceil$ bitmap bits, $o(n) \lg \sigma$ auxiliary bits, and $O(\sigma \lg n)$ topology bits. The access, rank, and select procedures described above visit at most one root-to-leaf

path and perform one binary operation per visited level, so their time is $O(\lg \sigma)$. □

The $O(\sigma \lg n)$ overhead comes entirely from the explicit tree topology and the per-node offsets into the global bit storage. For alphabets of size $\sigma = \Theta(n)$ this term dominates and can overwhelm the $n\lceil \lg \sigma \rceil$ leading term, motivating the topology-free layouts surveyed by Navarro [29].

The topology term $O(\sigma \lg n)$ is asymptotically smaller than $n\lceil \lg \sigma \rceil$ only when $\sigma = o(n)$. For alphabets growing with the input, the pointer overhead can dominate the bitmaps themselves, and the representation is no longer succinct. Later pointerless layouts remove this explicit topology term at no asymptotic penalty in query time.

To remove the explicit topology, store the node bitmaps by level. Number the root level as 1. At level ℓ, concatenate the bitmaps of the nodes at that level into a single bitmap of length n, and concatenate the $\lceil \lg \sigma \rceil$ level bitmaps into one global string $B[1..n\lceil \lg \sigma \rceil]$. Level ℓ occupies positions $n(\ell-1)+1$ through $n\ell$ of B. The shape of the tree is altered so that all leaves are aligned to the left, which lets every stored level be filled to length exactly n without gaps. A node v at level ℓ is identified by its bitmap's range $[l, r] \subseteq [n(\ell-1)+1, n\ell]$ within the level. The range of its left child at level $\ell+1$ starts at $n+l$ and ends at $n+l+\text{rank}_0(B, l, r)-1$, where $\text{rank}_0(B, l, r) = \text{rank}_0(B, r) - \text{rank}_0(B, l-1)$ counts the zeros of B_v. The right child starts immediately after, extending for $\text{rank}_1(B, l, r)$ positions. Navigation from a node to its children requires one rank query per descent, so the downward traversal costs $O(\lg \sigma)$ rank queries, matching the pointered version. Upward traversal is handled by descending from the root to the target leaf to record every node's range, then unwinding the recursion. The asymptotic time is preserved. The topology is entirely encoded by B itself together with the level offsets, and the space drops to $n\lceil \lg \sigma \rceil + o(n) \lg \sigma$ bits, with no explicit $O(\sigma \lg n)$ term.

2.3.2 Entropy compression

The plain space bound counts one stored bit for every original position at every alphabet level. This is the correct worst-case accounting, but it treats a balanced bitmap and an almost constant bitmap as equally expensive. If the symbols of S are skewed, many splits produce skewed bitmaps, and those bitmaps are precisely the inputs on which the compressed dictionaries of Theorem 2.1.7 save space. Write $n_c = |\{k : S[k] = c\}|$ for the number of occurrences of symbol c in S. The zero-order empirical entropy of S is

$$H_0(S) = \sum_{\substack{c \in [1..\sigma] \\ n_c > 0}} \frac{n_c}{n} \lg \frac{n}{n_c} \leq \lg \sigma,$$

with equality only when all alphabet symbols occur with the same frequency. Thus the compression should be applied locally, to the bitmaps that record the binary choices. If each B_v is stored with an entropy-compressed indexable dictionary while preserving constant-time binary rank and select, the traversals above are unchanged. This local replacement gives a compressed sequence representation only if the bitmap costs, summed over all internal nodes, match the entropy of S. Grossi, Gupta, and Vitter [28] prove this entropy telescoping for the wavelet trees used in their index. In the zero-order sequence setting it gives the equality below.

[28]: Grossi et al. (2003), *High-order entropy-compressed text indexes*

For a binary wavelet tree T, write $n_v = |S_v|$ for the length of the subsequence at node v. If an internal node v has children v_0 and v_1, then B_v has n_{v_0} zeros and n_{v_1} ones. Omitting zero-frequency bitmap symbols, its entropy cost expands to

$$|B_v| H_0(B_v) = \sum_{\substack{b \in \{0,1\} \\ n_{v_b} > 0}} n_{v_b} \lg \frac{n_v}{n_{v_b}} = n_v \lg n_v - n_{v_0} \lg n_{v_0} - n_{v_1} \lg n_{v_1}.$$

The identity depends only on the bottom-up composition of conditional probabilities through the binary partition. The shape of the binary wavelet tree is immaterial. A skewed or Huffman-shaped layout yields the same $nH_0(S)$ as the balanced one.

The final expression uses the usual convention that $0 \lg 0$ contributes zero. Summing it over all internal nodes cancels every non-root internal term. It appears once with positive sign in its own bitmap and once with negative sign in its parent's bitmap. After cancellation, the remaining root and leaf terms give

$$\sum_{v \text{ internal}} |B_v| \cdot H_0(B_v) \;=\; nH_0(S).$$

The per-node zero-order costs therefore reproduce the zero-order cost of the whole sequence.

Theorem 2.3.2 (Entropy-compressed wavelet tree [28]) *A sequence $S[1..n]$ over an alphabet $[1..\sigma]$ can be represented in*

$$nH_0(S) \;+\; o(n \lg \sigma) \quad \textit{bits}$$

so that access(S, i), rank$_c(S, i)$, *and* select$_c(S, j)$ *are supported in $O(\lg \sigma)$ time on the word RAM with word size $\omega = \Omega(\lg n)$.*

Sketch. Use the topology-free level layout described above for the binary wavelet tree. For each nonempty internal node v, store B_v with the entropy-compressed fully indexable dictionary of Theorem 2.1.7. It stores the bitmap within its zero-order entropy plus lower-order redundancy and supports rank$_b$ and select$_b$ in $O(1)$ time.The entropy terms of these dictionaries sum to $nH_0(S)$ by the identity above. The cited compressed binary wavelet-tree analysis controls the lower-order terms after summing over all node bitmaps, giving $o(n \lg \sigma)$ bits. The total space is therefore $nH_0(S) + o(n \lg \sigma)$. The traversals of Theorem 2.3.1 do not change. Each step now performs the constant-time binary rank or select operation supplied by Theorem 2.1.7 on the local dictionary for the current bitmap. A traversal visits $\lceil \lg \sigma \rceil$ levels, so access, rank, and select take $O(\lg \sigma)$ time. □

The binary wavelet tree spends one level for each binary decision on the alphabet. Ferragina, Mäkinen, Manzini, and Navarro [30] reduce this height by replacing the binary split with an r-ary alphabet partition, while treating the binary wavelet tree of Grossi, Gupta, and Vitter as the baseline that already gives $nH_0(S) + o(n \lg \sigma)$ bits and $O(\lg \sigma)$ time. Each internal node then stores a sequence over child indices $[1..r]$ instead of a bitmap, so the height drops once access, rank, and select over the smaller node alphabets are supported.

[30]: Ferragina et al. (2007), *Compressed representations of sequences and full-text indexes*

The resulting bound depends on the alphabet size. For polylogarithmic alphabets, the generalized structure uses $nH_0(S) + o(n)$ bits and answers access, rank, and select in constant time. For larger alphabets, choosing $r = O(\lg n / (\lg \lg n)^2)$ gives $nH_0(S) + O(\sigma \lg n) + o(n \lg \sigma)$ bits and time

$O(\lg \sigma / \lg \lg n)$. When $\sigma = o(n)$, the space simplifies to $nH_0(S) + o(n \lg \sigma)$ bits.

The leaf order also gives a stable sorting of S. Traversing leaves from left to right groups equal symbols, and the j-th position inside the leaf $[c, c]$ corresponds to the j-th occurrence of c in S. The bottom-up select traversal of Theorem 2.3.1 converts this sorted position into the original position in S via one select_0 or select_1 per level. The sequence S and its stable sorting are both directly readable from the same bitmaps, related to each other by a $\lceil \lg \sigma \rceil$-level chain of rank/select operations.

The sequence queries and the stable sorting both rely on a hierarchical binary partition of the alphabet, instantiated level by level as a bitmap on which rank and select answer the primitive queries. The same template, a hierarchical binary partition supported by rank on the induced bitmaps, generalizes from sequences to labeled ordinal trees in a later chapter of this thesis. There the partition acts on the alphabet of node labels, and rank on bitmaps becomes the primitive for queries on labeled trees. The wavelet tree is the prototypical instance of this template on strings.

Succinct Representations of Trees | 3

A single set can be stored as a bitvector: once the universe positions are marked, the rank and select structures of Chapter 2 make those marks searchable within a low-order term of the relevant information-theoretic bound. A collection adds structure that no single bitvector contains. Sets can be organized so that one is decoded relative to another, and a query then follows the dependencies chosen by the representation. When those dependencies form a tree, the query structure has two compressed layers to manage, the data stored at the nodes and the tree that tells the query where to move. This chapter isolates the second layer. It first represents the shape of an ordinal tree in succinct space while preserving the navigation expected from pointers. It then adds node labels, so later constructions can attach set information to a node rather than treating the node as a position only.

3.1 Encoding problem

A pointer representation makes navigation direct: a node stores references to the neighbors a query may ask for. On an n-node tree this costs $O(n \lg n)$ bits, because each reference has width $\Theta(\lg n)$. A succinct representation has to replace most of those references by a bit string that still determines the shape. This section separates the problem into three steps. We first fix the class of trees, then list the operations that a replacement for pointers must support, and finally count how many shapes an encoding has to distinguish before adding the first concrete representation.

3.1.1 Catalan lower bound

Before comparing encodings, we need a common model. We encode rooted trees whose children have a left-to-right order. A replacement for pointers must still answer questions about parents, children, siblings, ancestors, and traversal positions. With the object and operations fixed, we can count how many shapes an encoding has to distinguish.

An *ordinal tree* is a rooted tree whose children at every node are linearly ordered, so the children of a node v form a sequence $c_1, c_2, \ldots, c_{\deg(v)}$ rather than an unordered set. Munro and Raman call the same object a *rooted ordered tree* [31]. We use ordinal tree throughout. The unique node with no parent is the root. A node with no children is a leaf, and a node with at least one child is internal. The degree of v is the number of children of v. The depth of v is its distance from the root, with the root at depth 0. The subtree rooted at v, written T_v, is the induced subtree on v and all its descendants, and its size is $|T_v|$. A preorder traversal visits a node before its children, recursively from left to right. A postorder traversal visits the children before the node. Both traversals order all

[31]: Munro et al. (2001), *Succinct representation of balanced parentheses and static trees*

nodes, and preorder will be the default numbering when nodes are addressed by integers in $[1..n]$.

Since the representation replaces pointers, it must still answer the elementary moves that pointers expose. Jacobson's binary-tree model used the three operations $\{\texttt{left-child}, \texttt{right-child}, \texttt{null}\}$, while later ordinal-tree representations support more operations [11, 32]. The definition below records the operations used in this chapter.

[11]: Jacobson (1989), *Space-efficient static trees and graphs*
[32]: Navarro et al. (2014), *Fully functional static and dynamic succinct trees*

Definition 3.1.1 (Operations on ordinal trees) *Given an ordinal tree T on n nodes addressed by integers in $[1..n]$, a representation should support, for every node $x \in T$, the following operations.*

- $\texttt{parent}(x)$, *the parent of x (undefined at the root).*
- $\texttt{first\_child}(x)$, *the leftmost child of x (undefined at a leaf).*
- $\texttt{next\_sibling}(x)$, *the sibling of x immediately to its right (undefined at the rightmost child).*
- $\texttt{child}(x, i)$, *the i-th child of x in left-to-right order.*
- $\texttt{degree}(x)$, *the number of children of x.*
- $\texttt{subtree\_size}(x) = |T_x|$, *the size of the subtree rooted at x.*
- $\texttt{depth}(x)$, *the depth of x measured from the root at 0.*
- $\texttt{level\_ancestor}(x, d)$, *the ancestor of x at d levels above x.*
- $\texttt{ancestor}(x, d)$, *the ancestor of x at depth d.*
- $\texttt{LCA}(x, y)$, *the lowest common ancestor of x and y.*
- $\texttt{preorder\_rank}(x)$ *and* $\texttt{preorder\_select}(i)$, *mapping nodes to their preorder positions and back.*

The shape count is easiest through parentheses. During a preorder traversal, write an open parenthesis when entering a node and a close parenthesis when leaving its subtree. An n-node tree gives a balanced string with n pairs whose first parenthesis matches the last one [31]. Removing the outer pair gives the inner balanced string, and adding the outer pair back gives the traversal string of one ordinal tree. This is the count used by Navarro and Sadakane for n-node ordinal trees [32]:

[31]: Munro et al. (2001), *Succinct representation of balanced parentheses and static trees*
[32]: Navarro et al. (2014), *Fully functional static and dynamic succinct trees*

$$N_n = \frac{1}{2n-1}\binom{2n-1}{n-1} = \frac{2^{2n}}{\Theta(n^{3/2})}.$$

Distinguishing all these shapes requires one memory state per shape, so any encoding needs at least $\lg N_n$ bits in the worst case. In the lower-bound form used for succinct trees, this is

$$\lg N_n = 2n - \Theta(\lg n),$$

so the information-theoretic minimum is $2n$ bits up to a lower-order correction. A pointer representation uses $O(n \lg n)$ bits, since it stores $O(n)$ references of $\Theta(\lg n)$ bits each. Compared with the lower bound, this is a factor $\Theta(\lg n)$ away from optimal.

Theorem 3.1.1 (Catalan lower bound [32]) *Every representation that distinguishes all ordinal trees on $n \geq 2$ nodes has worst-case space at least $\lg N_n = 2n - \Theta(\lg n)$ bits, where $N_n = \frac{1}{2n-1}\binom{2n-1}{n-1}$ is the number of such trees.*

Sketch. The count above gives N_n possible inputs. If the worst-case space is b bits, the representation has at most 2^b memory states. Distinguishing

all inputs requires $2^b \geq N_n$, hence $b \geq \lg N_n$. The displayed bound gives $\lg N_n = 2n - \Theta(\lg n)$. □

Since Theorem 3.1.1 gives $\lg N_n = 2n - \Theta(\lg n)$, using $2n + o(n)$ bits is the same as using $(1 + o(1)) \lg N_n$ bits. This is the succinct-space criterion used by Navarro and Sadakane [32] and matches Jacobson's ratio-to-lower-bound formulation [11]. A tree representation must also answer the operations of Definition 3.1.1 in constant time on the word RAM.

The preorder parenthesis string above has the right order of space and determines the tree, yet finding the matching close, the enclosing pair, or a subtree boundary may still require a scan. The encodings that follow keep a near-minimum description of the shape and add sublinear indexing so that tree operations reduce to a constant number of queries on bitvectors or balanced-parenthesis sequences.

3.1.2 Level-order degrees as a first encoding

To obtain a first concrete encoding, ignore depth-first structure and record only how many children each node has. If the nodes are read in level order, each degree tells how many children must appear later in the same order. Jacobson [11] writes these degrees in unary and calls the result the *level-order unary degree sequence*. Later work abbreviates this name as LOUDS [32, 33].

[11]: Jacobson (1989), *Space-efficient static trees and graphs*

[32]: Navarro et al. (2014), *Fully functional static and dynamic succinct trees*

[33]: Benoit et al. (2005), *Representing trees of higher degree*

We build B from the augmented tree. First, we add a synthetic *super-root*, a new node of degree one whose only child is the original root. The super-root is not part of T, and its only role is to make the original root, like every other original node, appear as a child of some node and therefore have one associated 1-bit. Second, we list the degrees of all nodes of the augmented tree in level order, starting from the super-root. Third, we encode each degree d in unary as 1^d0, namely d ones followed by one zero, and concatenate the codewords in the same order. The resulting bitvector B is the unary-encoded BFS list of child counts. Within each codeword, the 1-bits announce the children of the current node, and the final 0 terminates that node's block.

Figure 3.1 shows the construction on a tree of seven nodes, augmented to eight by the super-root.

Counting bits is straightforward. Each non-root node contributes exactly one 1-bit, since it is the child of some other node, so the 1-count equals the number of children summed over all parents, which equals the number of non-root nodes. Each node contributes exactly one 0-bit, the terminator of its own degree codeword. With the super-root added, the augmented tree has $n + 1$ nodes, of which n are non-root, so B contains n ones and $n + 1$ zeros, for a total length of $2n + 1$.

Theorem 3.1.2 (LOUDS [11, 33]) *An ordinal tree of n nodes can be represented in $2n + 1 + o(n)$ bits so that, when a node is addressed by the position of its associated 1-bit in B, the operations* `parent`, `first_child`, `next_sibling`, `child`, *and* `degree` *run in $O(1)$ time on the word RAM with word size $\omega = \Omega(\lg n)$. The bitvector B is stored verbatim with the rank/select structures of Theorem 2.1.1 and Theorem 2.1.2 on top.*

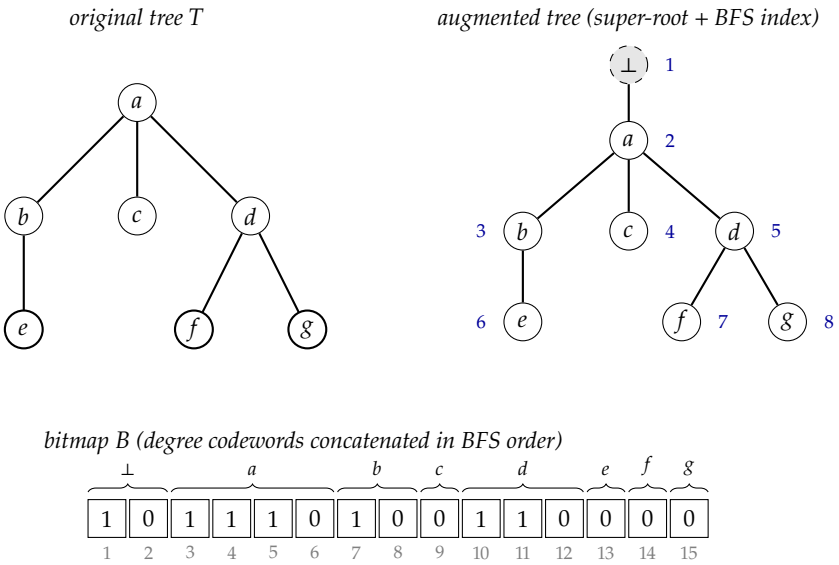

Figure 3.1: LOUDS encoding of an ordinal tree on seven nodes. The super-root $\perp$ is added so that every node, including the original root, has a unique parent and contributes a unary degree codeword. The augmented tree is traversed in BFS order, the BFS indices $1, \ldots, 8$ are shown next to each node, and the degree of each node in turn is written in unary as 1^d0, concatenated to form the bitmap B. Grouping braces above B mark the codewords contributed by the super-root and by nodes a, b, c, d, e, f, g in BFS order.

Proof. We have already counted the bits. The $o(n)$ overhead is the rank-and-select index of Theorem 2.1.1 and Theorem 2.1.2 stored with B, which gives $O(1)$ time for rank_b and select_b for $b \in \{0, 1\}$. Jacobson states the formulas for `parent`, `first_child`, and `next_sibling` in a bit-access model, where each rank or select query costs $O(\lg n)$ bit inspections. We use the same formulas with the word-RAM rank/select structures already introduced in this thesis, so each formula costs $O(1)$ time.

We use the local node address of this encoding, not the preorder address fixed in Definition 3.1.1. A node x of the original tree is represented by the position p_x of the 1-bit that announces x as a child in the augmented tree. Let $\beta_x = \text{rank}_1(B, p_x)$ be the BFS index of x among original nodes. The degree codeword of x is the block between the β_x-th and the $(\beta_x + 1)$-st 0-bit, namely the range $\text{select}_0(B, \beta_x) + 1, \ldots, \text{select}_0(B, \beta_x + 1)$.

For the parent query, we count how many degree codewords have ended before p_x is reached. The value $\text{rank}_0(B, p_x)$ is exactly the BFS index, among original nodes, of the parent that emitted this 1-bit. Hence, for every nonroot node x, the address of the parent is $p_{\texttt{parent}(x)} = \text{select}_1(B, \text{rank}_0(B, p_x))$. The original root is the special case whose parent is undefined in T.

For the first-child query, the child block of x begins at $s_x = \text{select}_0(B, \beta_x) + 1$. If $B[s_x] = 1$, then $p_{\texttt{first\_child}(x)} = s_x$. If $B[s_x] = 0$, the degree block is empty and x is a leaf. For the next-sibling query, we use the fact that siblings are emitted as consecutive 1-bits inside the same degree block. Therefore $p_{\texttt{next\_sibling}(x)} = p_x + 1$ exactly when $B[p_x + 1] = 1$.

We derive `child` and `degree` from the same block. If i is at most the number of 1-bits before the terminator of the block, then $p_{\texttt{child}(x,i)} = s_x + i - 1$. The number of such children is the length of the run of 1-bits before the next 0, so $\texttt{degree}(x) = \text{select}_0(B, \beta_x + 1) - \text{select}_0(B, \beta_x) - 1$. Every expression uses a constant number of bit accesses, rank/select calls, and arithmetic operations, so the claimed operations take $O(1)$ word-RAM time. □

LOUDS gives a succinct baseline: it stores the tree in $2n + 1 + o(n)$ bits

and answers the operations of Theorem 3.1.2 with a constant number of rank/select calls. This is succinct because Theorem 3.1.1 gives $2n - \Theta(\lg n)$ bits as the information-theoretic lower bound. The explicit bitmap is only $\Theta(\lg n)$ bits above that bound, apart from the $o(n)$ auxiliary index. What LOUDS does not give directly is the depth-first locality needed by operations such as `subtree_size`, `depth`, `ancestor`, `LCA`, `level_ancestor`, and `preorder_rank`. In BFS order, the descendants of a node may span many later levels, and their codewords are interleaved with codewords of nodes outside the subtree.

This limitation explains the move to depth-first encodings. Instead of adding more local formulas to the same level-order bitmap, the next representations write the tree in an order where every subtree occupies a contiguous range. Balanced parentheses obtains this locality by writing one pair of parentheses per node in DFS order, while DFUDS keeps the unary degree blocks but writes them in DFS order.

3.2 Depth-first encodings

The limitation of LOUDS is not its space but its order. A level-order sequence keeps the children of a node close to the node's degree block, which is why the formulas of Theorem 3.1.2 are simple, but it spreads a subtree across several BFS levels and interleaves it with nodes outside the subtree. The operations still missing from the LOUDS baseline, including `subtree_size`, `depth`, `ancestor`, `LCA`, `level_ancestor`, and `preorder_rank`, are easier to express when descendants form a contiguous range or when ancestry becomes parenthesis nesting. We now replace the level-order layout with depth-first layouts. First, we use balanced parentheses, which writes one open and one close parenthesis per node in preorder. Then we keep the unary degree blocks of LOUDS but write them in preorder instead of BFS order, obtaining DFUDS.

3.2.1 Balanced parentheses

We begin with the simplest depth-first layout. A preorder traversal of T writes an open parenthesis when a node is first visited and the matching close parenthesis when the traversal leaves its subtree. The result is a string $B[1..2n]$ over $\{$ `(` , `)` $\}$ in which each node contributes exactly one matched pair. We fix this preorder convention and identify each node v with the position i_v of its open parenthesis in B. The preorder rank of v is then $\mathrm{rank}_{(}(B, i_v)$. This address makes subtree intervals explicit. If $j_v = \texttt{findclose}(i_v)$, then the substring $B[i_v..j_v]$ is exactly the balanced-parenthesis sequence of the subtree rooted at v, and its length is $2 \cdot |T_v|$. Figure 3.2 shows the matched pairs and subtree intervals on the running tree.

The basic numerical quantity on B is the *excess* at a position. For $i \in [1..2n]$ define

$$e(B, i) \;=\; \mathrm{rank}_{(}(B, i) \;-\; \mathrm{rank}_{)}(B, i),$$

the number of open parentheses minus the number of close parentheses among the first i symbols. With this inclusive convention, the excess at an open position i_v is $\texttt{depth}(v) + 1$, because the unmatched opens are exactly

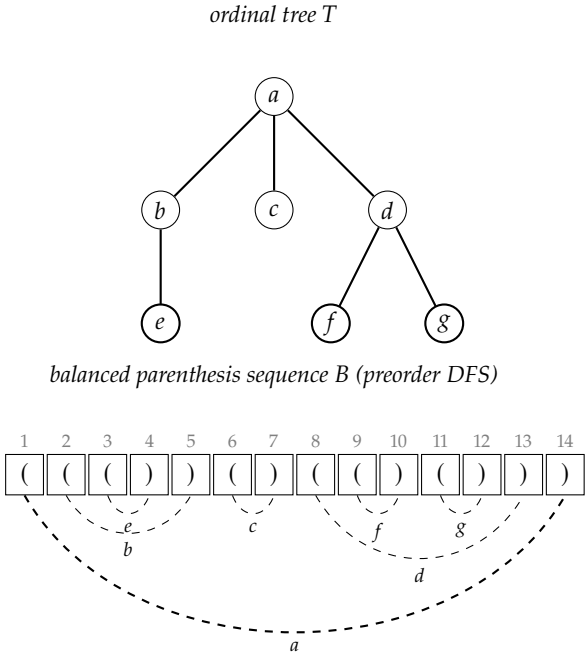

Figure 3.2: Balanced parenthesis encoding of the same seven-node ordinal tree used in Figure 3.1. A preorder DFS visit writes an open parenthesis on entry to a node and a close on exit. Each dashed arc joins the matching pair that represents one node, labelled by that node. The substring from a node's open parenthesis through its matching close is exactly the encoding of its subtree, of length $2 \cdot |T_v|$.

the ancestors of v together with v itself. The excess is non-negative for every prefix of B and equals zero only at the end of B, since the sequence is balanced. If i is an open parenthesis, then its matching close is the smallest $j > i$ such that $e(B, j) = e(B, i) - 1$. If v is not the root, then the open parenthesis of its parent is the largest open position $i' < i_v$ such that $e(B, i') = e(B, i_v) - 1$.

[31]: Munro et al. (2001), *Succinct representation of balanced parentheses and static trees*

Munro and Raman [31] index the balanced sequence through five primitives. The operation `findclose`(i) returns the close parenthesis matching the open at i. The operation `findopen`(i) returns the open parenthesis matching the close at i. The operation `excess`(i) returns the excess value at i. The operation `enclose`(i) returns the open parenthesis of the closest matched pair that strictly contains the pair beginning at i. The operation `double_enclose`(i, j) returns the closest matched pair that contains two non-overlapping matched pairs beginning at i and j, when such a pair exists. The rank identity above lets us compute `excess`(i) from binary rank, so the explicit tree-navigation primitives we keep are `findclose`, `findopen`, `enclose`, and `double_enclose`. Together with binary rank, these primitives are enough for the parent, subtree-size, depth, sibling, preorder-rank, and LCA operations below, while the i-th child still costs $O(i)$ in this encoding.

Theorem 3.2.1 (Balanced parentheses [31]) *Given a balanced parenthesis sequence B of length $2n$, there is a data structure of $o(n)$ bits attached to B such that on the word RAM with word size $\omega = \Omega(\lg n)$ the operations `findclose`, `findopen`, `excess`, `enclose`, and `double_enclose` on B all run in $O(1)$ time. Consequently, an ordinal tree on n nodes admits a representation in $2n + o(n)$ bits, with each node represented by its opening parenthesis, supporting `parent`(v), `subtree_size`(v), `depth`(v), `first_child`(v), `next_sibling`(v), `preorder_rank`(v), and `LCA`(u, v) in $O(1)$ time, with `child`(v, i) supported in $O(i)$ time.*

Sketch. The auxiliary structure is a three-level decomposition of B that mirrors the hierarchical layout used by Jacobson [10, 11] for rank and select, but indexes the excess sequence rather than raw bit counts. Writing $b = \lg^2 n$, the sequence B is partitioned into blocks of length b. Each such block is partitioned into subblocks of length $\lg^2 b = 4(\lg \lg n)^2$, and each subblock is partitioned into bottom-level chunks of length $2 \lg \lg n$. For cross-block matches, the structure stores pointers only for selected representatives: whenever consecutive cross-block parentheses stop matching into the same block, the first parenthesis of the new run is selected. At each top-level boundary it stores the current excess and the immediately preceding selected representative. Subblock headers carry the analogous information one level finer. If a string is divided into k blocks, then it has at most $2k - 3$ such representatives [11]. Since the number of top-level blocks is $O(n/\lg^2 n)$, the cumulative space for these pointers is $O(n/\lg n)$ bits. A precomputed lookup table [34], indexed by a low-level block pattern and an in-block position, resolves local `findclose` and `findopen` queries in $O(1)$ word-RAM operations. Its size is polynomial in $\lg n$ and therefore $o(n)$. A query $\texttt{findclose}(i)$ first uses the local table. If the match is not local, the procedure checks the subblock and top-level block summaries and uses the stored representative links to jump to the block that contains the answer. The same fixed hierarchy handles `findopen` by the symmetric search on right excess. Analogous bookkeeping of enclosing pairs supports `enclose` and `double_enclose` in the same asymptotic budget. The reductions of the operations above are direct. For nonroot nodes, $\texttt{parent}(v) = \texttt{enclose}(i_v)$. We also have $\texttt{subtree\_size}(v) = (\texttt{findclose}(i_v) - i_v + 1)/2$, $\texttt{depth}(v) = e(B, i_v) - 1$, $\texttt{first\_child}(v) = i_v + 1$ when $B[i_v + 1] =$ (and undefined otherwise, $\texttt{next\_sibling}(v) = \texttt{findclose}(i_v) + 1$ when that position exists and holds an open parenthesis, and $\texttt{preorder\_rank}(v) = \text{rank}_(B, i_v)$. For $\texttt{LCA}(u, v)$, assume $i_u \le i_v$. If $i_v \le \texttt{findclose}(i_u)$, then u is an ancestor of v and the answer is u. Otherwise the two matched pairs are nonoverlapping, and $\texttt{double\_enclose}(i_u, i_v)$ returns their closest enclosing pair, namely the lowest common ancestor. □

[10]: Jacobson (1988), *Succinct static data structures*
[11]: Jacobson (1989), *Space-efficient static trees and graphs*

[34]: Munro (1996), *Tables*

The lookup-table device is the four-Russians technique in the abstract sense [35], but Munro-Raman attribute the load-bearing tabulation construction to [34].

The reductions are immediate consequences of the bijection node $\leftrightarrow$ matched pair. A nonroot node's parent is the closest enclosing pair, which is what $\texttt{enclose}(i_v)$ returns. The subtree of v occupies the substring $B[i_v..\texttt{findclose}(i_v)]$, of length $2|T_v|$, so the subtree size is half the difference. The depth of v is the number of unmatched ancestor opens before i_v, hence $e(B, i_v) - 1$ under our inclusive convention. The first child of v exists exactly when the symbol at $i_v + 1$ is an open parenthesis, since otherwise $B[i_v + 1] =$) matches i_v and v is a leaf. The next sibling lives at $\texttt{findclose}(i_v) + 1$ when that position exists and holds an open parenthesis, since $\texttt{findclose}(i_v)$ is the close of v's subtree, the next position belongs to the same parent, and an open there starts the next sibling. The preorder rank of v is the count of open parentheses up to i_v, which is the binary $\text{rank}_(B, i_v)$ of Theorem 2.1.1. The i-th child is reachable by $i - 1$ successive applications of `next_sibling` starting from $\texttt{first\_child}(v)$, each costing $O(1)$ via `findclose`, for total cost $O(i)$.

The lowest common ancestor also admits a range-minimum view. During a DFS traversal of T, the lowest common ancestor of u and v is the shallowest node visited between the first occurrences of u and v in the traversal. Bender and Farach-Colton [36] use this observation to

[36]: Bender et al. (2000), *The LCA problem revisited*

reduce LCA to RMQ on the level array of an Euler tour, and then give an $\langle O(n), O(1)\rangle$ preprocessing and query algorithm for the special case of ± 1 range minimum queries. The excess sequence of a balanced-parenthesis traversal has the same ± 1 property. Under our inclusive convention, the excess at an open parenthesis is the node depth plus one, and at a close parenthesis it is the depth after leaving the corresponding subtree. Thus minima in the excess encode the same ancestor-boundary information used by the Euler-tour level minimum. We will use this RMQ viewpoint when discussing LCA, while the BP theorem above can already support LCA through `double_enclose`.

Example 3.2.1 For the tree of Figure 3.2, the BP sequence is $B =$ `((())()(())())` and has length 14. Node b sits at $i_b = 2$, and its matching close is at position 5. Therefore

$$\texttt{subtree\_size}(b) \;=\; \frac{5 - 2 + 1}{2} \;=\; 2,$$

in agreement with the subtree of b, which contains b and its only child e. The closest enclosing parenthesis of position $i_b = 2$ is at 1, so $\texttt{parent}(b) = a$. Finally, $e(B, 2) = 2$, and therefore $\texttt{depth}(b) = e(B, 2) - 1 = 1$.

The bound $O(i)$ for $\texttt{child}(v, i)$ is the remaining limitation of the encoding. For trees of bounded degree this is constant, but for high-degree trees a query for the i-th child of a node of degree $d \gg 1$ still pays $\Theta(i)$ findclose calls in the worst case. Munro and Raman raise this gap as an open question in their original paper. We next recover $\texttt{child}(v, i)$ in $O(1)$ while keeping constant-time parent and subtree-size access.

3.2.2 DFUDS

[33]: Benoit et al. (2005), *Representing trees of higher degree*

We follow the construction of Benoit, Demaine, Munro, Raman, Raman, and Rao [33], which keeps the unary degree blocks of LOUDS but changes their order. LOUDS writes the unary degree of each node in level order. BP writes one open and one close parenthesis per node in depth-first order. DFUDS combines these choices by writing unary degree blocks in depth-first preorder. At node v of degree d_v, the traversal emits d_v open parentheses followed by one close. Concatenating these $d_v + 1$ symbols across all nodes yields a string of length $\sum_v (d_v + 1) = (n - 1) + n = 2n - 1$. A leading open parenthesis is prepended to balance the string, bringing the total length to $2n$. The result is the *depth-first unary degree sequence* encoding, abbreviated DFUDS, shown in Figure 3.3.

The string is balanced by induction on the tree. A single-node tree is encoded as `()`. Now suppose that the root has d children and that the DFUDS strings of the child subtrees are balanced. The root contributes the artificial open, then d opens and one close for its own block. For each child subtree, we append its balanced DFUDS string after removing its artificial leading open. Removing that open leaves one extra close, and the d extra closes from the d child strings match the d opens in the root block. The artificial open at the beginning matches the last close of the whole string.

We identify a node v with the position p_v of the first parenthesis of its block. The artificial open at position 1 has no node, so the root starts at

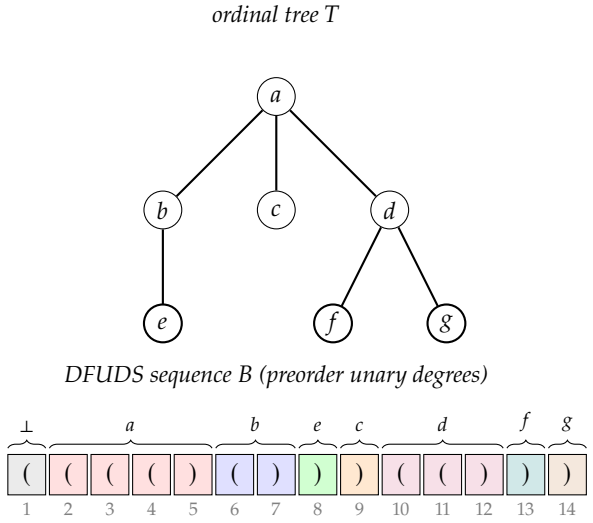

Figure 3.3: DFUDS encoding of the same seven-node ordinal tree of Figure 3.2. Each node v contributes a block of d_v open parentheses followed by one close, written in DFS preorder. A leading open parenthesis (block $\perp$) is prepended so that the resulting string of length $2n = 14$ is balanced. The colored blocks above the sequence highlight the per-node contributions of a, b, e, c, d, f, g in preorder. The position p_v of the first parenthesis of each block is the address by which the node is referenced.

position 2. The address $p_v \in [1..2n]$ is the local node label used by the formulas below. When an integer preorder label in $[1..n]$ is needed, it is enough to count the blocks completed before p_v. Under our inclusive rank convention, we write $r_v = \text{rank}_)(B, p_v - 1)$ for the number of close parentheses strictly before the block of v, so the preorder label of v is $r_v + 1$.

The DFUDS encoding supports degree, i-th child, parent, and subtree size in constant time on top of Theorem 3.2.1. The degree of the node at position p_v is the length of its run of opens, which is the gap from p_v to the next close in B. A single $\text{rank}_)$ followed by a $\text{select}_)$ identifies that close, giving $\texttt{degree}(v) = \text{select}_)(B, r_v + 1) - p_v$. For $1 \le i \le \texttt{degree}(v)$, the i-th child is found by counting the opens of v's block from right to left. The open at position $\text{select}_)(B, r_v+1)-i$ has its matching close immediately before the block of the i-th child, so $\texttt{child}(v, i) = \texttt{findclose}(\text{select}_)(B, r_v+1)-i)+1$. If $i > \texttt{degree}(v)$, the child is undefined. Thus the query uses one rank call, one select call, one `findclose` call, and constant arithmetic.

The parent of v is recovered from the close immediately preceding v's block. If $p_v = 2$, then v is the root. Otherwise let $q_v = \texttt{findopen}(p_v - 1)$ and $s_v = \text{rank}_)(B, q_v - 1)$. The position q_v lies inside the block of the parent. If $s_v = 0$, then the parent is the root, whose address is 2. Otherwise the parent block starts immediately after the s_v-th close, so $\texttt{parent}(v) = \text{select}_)(B, s_v) + 1$.

Subtree size is not read from the matching close of p_v. If v is a leaf, then $\texttt{subtree\_size}(v) = 1$. Otherwise let $c_v = \texttt{findclose}(\texttt{enclose}(p_v))$. The substring $B[p_v..c_v]$ is the DFUDS description of the subtree rooted at v with its artificial leading open removed. Its length is $2|T_v| - 1$, and therefore

$$\texttt{subtree\_size}(v) = \frac{c_v - p_v}{2} + 1.$$

Theorem 3.2.2 (DFUDS [33]) *An ordinal tree on n nodes admits a representation in $2n + o(n)$ bits that supports $\texttt{degree}(v)$, $\texttt{child}(v, i)$, $\texttt{parent}(v)$, and $\texttt{subtree\_size}(v)$ in $O(1)$ time on the word RAM with word size $\omega = \Omega(\lg n)$. The representation is the DFUDS string of length $2n$, the auxiliary structure of Theorem 3.2.1, and binary* rank *and* select *indexes on the two parenthesis symbols.*

Sketch. The DFUDS string is balanced and has length $2n$, so Theorem 3.2.1 attaches an $o(n)$-bit auxiliary structure that supports `findclose`, `findopen`, `excess`, `enclose`, and `double_enclose` in $O(1)$ time, together with rank and select on the open and close alphabets via Theorem 2.1.1 and Theorem 2.1.2. The four operations claimed are obtained by the formulas above, each evaluating to $O(1)$ word-RAM steps. □

DFUDS changes syntax, not the tree traversal. BP and DFUDS read nodes in preorder and use $2n$ parentheses, but a parenthesis carries different information in the two encodings. In BP, an open marks node entry and a close marks subtree exit. In DFUDS, the opens of a node's block give one handle per child, and the trailing close ends the block. This shift turns the cost of $\texttt{child}(v, i)$ from a chain of i `next_sibling` jumps into a single `findclose` on a position computed by a rank/select pair, while preserving constant-time parent and subtree-size queries. We have now seen a level-order encoding that makes children local, a parenthesis encoding that makes subtrees local, and a unary depth-first encoding that combines the two properties for the four operations of Theorem 3.2.2. The auxiliary structures used so far are still built around separate primitives. In the next section we replace them with a uniform tree representation supporting the full operation set of Definition 3.1.1.

3.3 Fully functional encodings

The encodings above reach the succinct space target, but they do not yet give the full operation set of Definition 3.1.1. BP gives constant-time parent, subtree size, depth, siblings, preorder rank, and LCA, but pays $O(i)$ for $\texttt{child}(v, i)$. DFUDS restores constant-time child, parent, degree, and subtree size, yet its support still rests on separate indexes for parenthesis primitives. As operations such as `leftmost_leaf`, `leaf_select`, `degree`, `child_rank`, and `level_ancestor` are added, the classical BP line accumulates auxiliary structures tied to individual queries. We now move from encoding-specific support to uniform representations. The first, due to He, Munro, and Rao [37], augments a tree covering with an $o(n)$-bit catalog of microtree patterns and supports every operation of Definition 3.1.1 in $O(1)$ time within $2n + o(n)$ bits. The second, due to Sadakane and Navarro [32], keeps the BP sequence but answers the needed excess primitives through one range min-max tree, reaching the same static bound and extending naturally to the dynamic setting.

[37]: He et al. (2007), *Succinct ordinal trees based on tree covering*

[32]: Navarro et al. (2014), *Fully functional static and dynamic succinct trees*

The starting point is a recursive cover of T at two carefully chosen sizes. With parameter M, the covering procedure covers the nodes of T by *minitrees*, connected subgraphs that may share only a common root. Every minitree has size between M and $3M - 4$, except for a possible smaller minitree containing the root of T. The same procedure is then applied inside each minitree with parameter M', producing a cover by *microtrees*. Geary, Raman, and Raman [38] introduced the covering algorithm and proved the size bound. He, Munro, and Rao [37] use this two-level cover in their fully functional representation. Choosing $M = \max\{\lceil \lg^4 n \rceil, 2\}$ and $M' = \max\{\lceil \lg n/24 \rceil, 2\}$ gives $O(n/\lg^4 n)$ minitrees and $O(n/\lg n)$ microtrees in total. Figure 3.4 sketches the nesting pattern of this two-level cover on a small example.

[38]: Geary et al. (2006), *Succinct ordinal trees with level-ancestor queries*

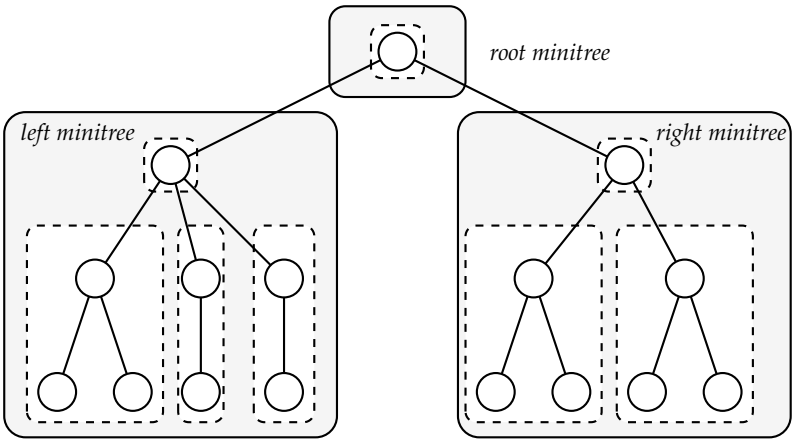

Figure 3.4: A schematic two-level tree covering of an ordinal tree. Solid rectangles show minitree regions produced with parameter M. Dashed rectangles show microtree regions produced inside them with parameter $M' < M$, including the possible smaller piece that contains the root of a recursive cover. Different pieces may intersect only at a common root. The exact size bounds are those of Lemma 3.3.1. The drawing shows nesting and boundary sharing on a tree too small for the asymptotic parameters used in the representation.

Lemma 3.3.1 (Tree covering size bound [38]) *For any ordinal tree T on n nodes and any integer $M \geq 2$, the covering procedure covers the nodes of T with connected subgraphs of T. Any two pieces are either disjoint or intersect only at their common root. Every piece A has $|A| \leq 3M - 4$, and $|A| \geq M$ unless A contains the root of T. Consequently the cover has $\Theta(\max\{1, n/M\})$ pieces.*

A boundary node may belong to more than one minitree or microtree, so He, Munro, and Rao address it by recording the containing minitree, the containing microtree, and the local preorder position. If a node has several such addresses, we use the lexicographically smallest one as its canonical address. Extended microtrees keep promoted boundary copies so that local navigation remains possible, and the conversion between canonical addresses and preorder indices is supported in $O(1)$ time. We use this conversion as an available constant-time primitive.

The size cap $3M - 4$ is what makes microtrees small enough for a catalog while keeping the number of pieces $\Theta(n/M)$ in the regime used below. The lower bound M rules out pathological splits and is essential for the count of pieces to be sublinear.

The second-level parameter $M' = \max\{\lceil \lg n/24 \rceil, 2\}$ is calibrated so that the catalog of microtree shapes is sublinear. For the asymptotic range in which $\lceil \lg n/24 \rceil \geq 2$, the size bound of Lemma 3.3.1 gives at most $3M' - 4 \leq \lceil \lg n/8 \rceil$ nodes per microtree, which is the upper bound used by He, Munro, and Rao. The elementary Catalan bound on ordinal trees of size at most h gives fewer than 4^h shapes. Substituting $h = \lceil \lg n/8 \rceil$ yields $O(n^{1/4})$ possible microtree shapes, which is the shape universe used in their accounting. Since each shape has only $O(\lg n)$ local positions, lookup tables indexed by a shape and a constant number of local positions occupy $o(n)$ bits.

The local component is a catalog over microtree-sized objects. Once a query has been reduced to data that lie inside a single microtree, the shape of that microtree and the local positions determine the answer. He, Munro, and Rao therefore precompute one set of universal tables over all possible microtree shapes. One table, indexed by a shape and two local preorder positions, returns the local node of maximum depth between those positions, which is the local subproblem needed by the height query. Another table, indexed by a shape and two local nodes, returns their lowest common ancestor when both nodes lie in the same microtree. These tables are stored once for the whole representation, not once per occurrence of a microtree, and occupy $o(n)$ bits because the shape universe has size $O(n^{1/4})$ and each shape contributes only a polylogarithmic number of local entries. This is the microtree analogue of the block lookup tables used in Theorem 2.1.1.

The remaining structures reconcile queries that cross microtree boundaries. A bitvector of length n marks the tier-1 *preorder changers*, namely the $O(n/\lg^4 n)$ nodes whose preorder predecessor lies in a different minitree. A second bitvector marks the tier-2 preorder changers, namely the $O(n/\lg n)$ nodes whose predecessor lies in a different microtree. Both bitvectors are sparse and are stored with the rank/select dictionaries of Theorem 2.1.7 in $o(n)$ bits. The associated arrays store the local-address components of boundary nodes and the representatives needed for range maxima on the conceptual depth array. These summaries do not give a single formula for every operation. Instead, they move a query between global preorder, the relevant minitree or microtree, and the local catalog entry required by the operation. For LCA, a macro tree on the $O(n/\lg^4 n)$ minitree roots is preprocessed with the Bender-Farach-Colton LCA structure [36], taking $O(n/\lg^4 n \cdot \lg n) = o(n)$ bits. Distance then follows from LCA and depth. Analogous tier-2 macro trees handle microtree roots inside each minitree, and the same sparse-boundary accounting keeps their total space $o(n)$.

[36]: Bender et al. (2000), *The LCA problem revisited*

Theorem 3.3.2 (Tree covering succinct [37]) *An ordinal tree on n nodes admits a representation in $2n + o(n)$ bits that supports every operation of Definition 3.1.1 in $O(1)$ time on the word RAM with word size $\omega = \Theta(\lg n)$. The same representation supports, in $O(1)$ time within the same space bound, the additional operations*

- `child_rank`(x), *the position of x among its siblings.*
- `height`(x), *the height of the subtree rooted at x.*
- `distance`(x, y), *the length of the tree path between x and y.*
- `leftmost_leaf`(x) *and* `rightmost_leaf`(x), *the leftmost and rightmost leaves of the subtree rooted at x.*
- `leaf_rank`(x), `leaf_select`(i), *and* `leaf_size`(x), *mapping leaves to leaf-only positions, mapping those positions back, and counting leaves inside a subtree.*
- `level_leftmost`(d) *and* `level_rightmost`(d), *the leftmost and rightmost nodes at depth d.*
- `level_succ`(x) *and* `level_pred`(x), *the next and previous nodes at the same depth.*
- *the postorder rank and select operations, and the DFUDS rank and select counterparts of* `preorder_rank` *and* `preorder_select`.

Sketch. We start from the extended-microtree representation of Geary, Raman, and Raman [38]. Its main structures store the tree in $2n + o(n)$ bits. He, Munro, and Rao add only $o(n)$ further bits: sparse rank/select dictionaries for the tier-1 and tier-2 preorder changers, arrays storing the corresponding local-address components, catalog tables for microtree-local subproblems, range-extremum summaries on the conceptual depth array, and macro trees on the minitree and microtree roots. The counts established above give $O(n/\lg^4 n)$ minitrees, $O(n/\lg n)$ microtrees, and $O(n^{1/4})$ possible microtree shapes, so each family of added structures stays sublinear.

The query algorithms use these structures by operation family. Operations inherited from the Geary, Raman, and Raman representation keep their constant-time support on extended microtrees. Height reduces to a range maximum over the conceptual preorder depth array and is answered by

the tiered range-extremum summaries. LCA first uses an in-microtree catalog lookup when both nodes lie in one microtree, and otherwise moves to the appropriate tier-2 or tier-1 macro tree. Distance follows from LCA and depth. Leaf operations use marked pseudo-leaves at the mini and micro levels. DFUDS rank and select use the extended-mini-tree sums and the conversion between DFUDS numbers and local addresses. Level operations use the per-level summaries described for `level_leftmost`, `level_rightmost`, `level_succ`, and `level_pred`. In every case the algorithm performs a constant number of rank/select calls, address conversions, macro-tree queries, or catalog lookups, each of which takes $O(1)$ time.

The operation list of Definition 3.1.1 follows from the operations stated by He, Munro, and Rao. Their child, degree, depth, level-ancestor, LCA, and preorder rank/select operations are already in our list. We recover $\texttt{parent}(x)$ as $\texttt{level\_ancestor}(x, 1)$, $\texttt{first\_child}(x)$ as $\texttt{child}(x, 1)$ when $\texttt{degree}(x) > 0$, $\texttt{next\_sibling}(x)$ from $\texttt{child\_rank}(x)$ and the parent of x, $\texttt{subtree\_size}(x)$ by adding one to the number of descendants of x, and $\texttt{ancestor}(x, d)$ as $\texttt{level\_ancestor}(x, \texttt{depth}(x) - d)$. The detailed definitions of local addresses, extended microtrees, and the auxiliary arrays are those of He, Munro, and Rao [37]. □

The two-level cover predates Theorem 3.3.2 and is a result in its own right. Geary, Raman, and Raman used it to obtain a $2n + o(n)$-bit representation that supports `level_ancestor` in $O(1)$ time [38]. As presented above, LOUDS, BP, and DFUDS do not provide that operation in constant time. Farzan and Munro later developed a related recursive-decomposition framework for ordinal, cardinal, and free trees [39]. Theorem 3.3.2 extends the Geary, Raman, and Raman construction by augmenting it with the auxiliaries needed for every operation of Definition 3.1.1, while keeping the asymptotic space bound.

[38]: Geary et al. (2006), *Succinct ordinal trees with level-ancestor queries*

[39]: Farzan et al. (2008), *A uniform approach towards succinct representation of trees*

A second route reaches the same bound without covering the tree. Sadakane and Navarro keep the BP sequence and attach one range min-max tree to it [32]. Their reduction expresses the operations through a small set of bitvector and excess primitives, including forward search, backward search, prefix sum, and range minimum or maximum queries. The same auxiliary structure supports these primitives, replacing the per-primitive hierarchies used by earlier BP representations.

[32]: Navarro et al. (2014), *Fully functional static and dynamic succinct trees*

The auxiliary is the *range min-max tree*. Let $B[1..2n]$ be the BP sequence of T, and let $e(B, i)$ be the excess at position i of Section 3.2. For intuition, divide B into blocks of length $\Theta(\lg n)$ and build a complete tree whose leaves are the blocks of B, with each internal node storing the minimum and maximum excess below it. Sadakane and Navarro first implement the polylogarithmic-size building block as a complete k-ary tree with arity $k = \Theta(w/(c_0 \lg w))$ and chunks of size $w/2$, where w is the machine word and c_0 is the constant controlling the depth of that local structure. Three per-chunk arrays store the last excess, minimum excess, and maximum excess, and each occupies $O(N c_0 \lg w/w)$ bits for a segment of length N inside that building block. The in-chunk step is answered by a universal lookup table of $O(\sqrt{2^w} \lg w)$ bits. The full static construction combines these blocks with higher-level summaries and then reparameterizes the constants, giving the $O(n/\lg^c n)$ redundancy stated in Theorem 3.3.3 for any final constant $c > 0$.

The parenthesis operations reduce to these primitives in two different ways. The operations `findclose`, `findopen`, `enclose`, and `level_ancestor` become forward or backward searches for a target excess value. Such a search starts at the leaf containing the query position, walks upward until it finds a sibling subtree whose stored range contains the target, and then walks downward to localize the answer. By contrast, `rmq` and `RMQ` ask directly for a minimum or maximum excess in an interval and use the same min-max summaries. The lookup table answers the final in-chunk step in constant time. After the large-input reduction and the constant reparameterization, each primitive runs in $O(c)$ time on the word RAM, where $c > 0$ is the final constant fixing the redundancy bound.

Theorem 3.3.3 (Range min-max tree [32]) *Let $B[1..2n]$ be the balanced parenthesis sequence of an ordinal tree T on n nodes. For any constant $c > 0$, the BP sequence augmented with the range min-max tree occupies $2n + O(n/\lg^c n)$ bits and supports every operation of Definition 3.1.1 in $O(c)$ time on the word RAM with word size $\Theta(\lg n)$. The same approach extends to a dynamic ordinal tree on n nodes that supports every operation of Definition 3.1.1, together with insertions and deletions, in $O(\lg n)$ time per operation, within $2n + O(n/\lg n)$ bits of space.*

The proof of Theorem 3.3.3 is constructive. Each primitive resolves to a traversal of the range min-max tree, with the leaf step answered by the lookup table on $\Theta(\lg n)$-bit blocks. The dynamic extension is one advantage over Theorem 3.3.2. The two-level catalog used by the tree-covering scheme is hard to maintain under updates, while the range min-max tree stores the sequence in updatable blocks and refreshes the affected min-max summaries. Sadakane and Navarro [32] also present a finer dynamic tradeoff with $2n + O(n \lg\lg n/\lg n)$ bits. In that variant many operations take $O(\lg n/\lg\lg n)$ time, while updates and some degree or child-related queries have larger nonuniform bounds. We state the coarser dynamic variant in Theorem 3.3.3 because it gives a single $O(\lg n)$ bound for all supported operations.

Theorem 3.3.2 and Theorem 3.3.3 both attain $2n + o(n)$ bits with constant query time for every operation of Definition 3.1.1, the first with $O(1)$ time and the second with $O(c)$ time for any fixed constant $c > 0$. They reach this bound by two different routes. One builds a catalog of microtree patterns on a recursive cover. The other keeps the BP sequence and stores range min-max summaries of its excess sequence. Together they give the fully functional unlabeled tree representations needed in the rest of the thesis. The next section turns to the question of what changes when each node carries a label drawn from an alphabet Σ.

3.4 Labeled trees

The encodings developed so far address ordinal trees in which a node is identified by its position in some traversal order and nothing else. Every operation of Definition 3.1.1 returns or consumes a position. No operation carries a node label. Theorem 3.3.2 and Theorem 3.3.3 both close the unlabeled question with $2n + o(n)$ bits and constant query time. The natural next step is the case in which each node carries a label drawn from an alphabet Σ of size σ. Once labels enter, the basic navigation

queries have label-aware counterparts: instead of asking for the parent of x, one asks for the closest ancestor of x whose label is a specified $\alpha \in \Sigma$, and instead of counting all descendants of x, one counts the descendants whose label is α. Setting $\sigma = 1$ recovers the same navigation, up to the depth convention in which $\texttt{depth}_\alpha(v)$ counts v itself.

We follow the terminology of He, Munro, and Zhou [40], who write *α-node* for a node whose label is α and refer to the label-aware variants as *α-operations*. The α-rank of a node x in a list is the number of α-nodes to the left of x in that list. The five α-operations below are tree analogues of the rank_c and select_c primitives that drive the wavelet tree of Section 2.3 on a sequence over an alphabet of size σ.

[40]: He et al. (2014), *A framework for succinct labeled ordinal trees over large alphabets*

Definition 3.4.1 (α-operations on labeled ordinal trees) *Let T be an ordinal tree on n nodes whose nodes are labeled with symbols from an alphabet Σ of size σ. For every $\alpha \in \Sigma$ and every node $v \in T$, a representation of the labeled tree should support the following operations.*

- $\texttt{parent}_\alpha(v)$, *the closest α-labeled ancestor of v (undefined when no α-ancestor exists).*
- $\texttt{depth}_\alpha(v)$, *the number of α-labeled nodes on the path from v to the root, counting v itself when its label is α.*
- $\texttt{child\_select}_\alpha(v, i)$, *the i-th α-labeled child of v in left-to-right order.*
- $\texttt{nbdesc}_\alpha(v)$, *the number of α-labeled nodes in the subtree rooted at v.*
- $\texttt{level\_anc}_\alpha(v, d)$, *the α-labeled ancestor w of v with $\texttt{depth}_\alpha(v) - \texttt{depth}_\alpha(w) = d$.*

When $\sigma = 1$, every node is an α-node for the unique label, so these operations recover the corresponding unlabeled navigation queries, with $\texttt{depth}_\alpha(v) = \texttt{depth}(v) + 1$ under the inclusive convention above.

Two routes from Section 3.3 to a labeled representation are immediate but inadequate. The first builds, for every $\alpha \in \Sigma$, a separate 0/1-labeled index of T in which the α-nodes are marked with 1 and all other nodes with 0. This gives constant-time access to the α-filtered queries on each copy, but it stores σ indexed trees on the same n nodes, for $\Theta(\sigma n)$ bits even before lower-order terms. The second route stores one unlabeled structure for T and keeps the labels in an explicit array of $\lceil \lg \sigma \rceil$ bits per node. This uses $n \lg \sigma + 2n + o(n)$ bits, but without an index over the labels a query such as $\texttt{level\_anc}_\alpha$ or $\texttt{nbdesc}_\alpha$ may scan a path or subtree of size $\Theta(n)$. Neither route gives $n \lg \sigma + O(n)$ bits together with sublogarithmic query time.

The information-theoretic lower bound for an n-node ordinal tree whose nodes are labeled with symbols from an alphabet of size σ is $n(\lg \sigma + 2) - O(\lg n)$ bits, by combining the unlabeled bound of Theorem 3.1.1 with the $\lg(\sigma^n) = n \lg \sigma$ bits required to store one label per node. A representation that meets this lower bound to within a low-order term, while supporting every α-operation of Definition 3.4.1 in $O(1)$ time, would close the labeled question in the same sense as Theorem 3.3.2 closed the unlabeled one. Geary, Raman, and Raman achieve $O(1)$ time within a controlled additive overhead, and their bound is the technical centerpiece of the section.

Theorem 3.4.1 (Labeled ordinal trees [38, 40]) *An ordinal tree on n nodes whose nodes carry labels from an alphabet Σ of size σ admits a representation in*

$$n(\lg \sigma + 2) \; + \; O\left(\frac{\sigma \, n \, \lg\lg\lg n}{\lg\lg n}\right)$$

bits that supports the α-operations of Definition 3.4.1, together with the labeled counterparts of the preorder and postorder rank *and* select, *in $O(1)$ time on the word RAM under the logarithmic word-size convention.*

Sketch. The construction modifies Theorem 3.3.2 to integrate the labels into the catalog of microtree shapes. Reduce the second-level parameter to $M' = \max(\lceil \lg n/(24 \lg \sigma) \rceil, 2)$ so that, whenever a microtree is answered by local table lookup, its labels fit alongside its tree structure within $\lceil |\mu| \lg \sigma \rceil + 2|\mu|$ bits while keeping the lookup table sublinear. For microtree-local queries that are handled by explicit label filters, store σ extra bitvectors per microtree, the α-th of which marks the nodes labeled α. At the canonical root of each microtree, store further bitvectors encoding the distribution of children by label and the per-label skip pointers needed to handle $\texttt{parent}_\alpha$ and $\texttt{level\_anc}_\alpha$ across microtree boundaries. The per-microtree label-aware bookkeeping sums to $o(n)$ bits over all microtree roots once M' is set as above. The macro tree on minitree roots is replicated once per $\alpha \in \Sigma$, with each macro edge weighted by the number of α-labeled nodes in the skipped portion of the original tree. The weights are polylogarithmic in n, so the polylog-weight level-ancestor algorithm answers $\texttt{level\_anc}_\alpha$ on each copy in $O(1)$ time. The σ copies of the macro tree contribute the $O(\sigma n \lg\lg\lg n/\lg\lg n)$ additive overhead of the bound. The remaining α-operations reduce to a constant number of in-microtree lookups, rank/select calls on the per-label microtree bitvectors, and a single macro-level query. □

Theorem 3.4.1 settles the labeled question for very small alphabets. When $\sigma = o(\lg\lg n)$, the displayed overhead is lower order than $n(\lg \sigma + 2)$, so the representation matches the lower bound to within a low-order correction. Outside that regime, the same upper bound no longer shows how to stay near the information-theoretic cost. If σ is polynomial in n, the displayed additive term can be much larger than the label entropy term $n \lg \sigma$, so the bound no longer explains how to store large-alphabet labels near their information-theoretic cost.

The threshold $\sigma = o(\lg\lg n)$ is what makes the additive overhead lower order in Theorem 3.4.1. For polynomial alphabets, the displayed overhead grows far beyond $n \lg \sigma$, so the stated bound ceases to be informative.

The set-collection applications that motivate the later chapters use labels drawn from the universe, so the alphabet can be much larger than the small range covered by Theorem 3.4.1. Geary, Raman, and Raman leave open whether labeled ordinal trees can remain space-efficient for larger alphabets. He, Munro, and Zhou [40] answer this question with a representation that occupies $nH_0(PLS_T) + O(n)$ bits, where $H_0(PLS_T)$ is the zeroth-order empirical entropy of the preorder label sequence of T and is bounded above by $\lg \sigma$, while supporting the α-operations of Definition 3.4.1 in $O(1)$ time when $\sigma = O(\text{polylog}(n))$ and in $O(\lg\lg \sigma)$ time for general Σ. The remaining sections of this chapter develop the answer.

[40]: He et al. (2014), *A framework for succinct labeled ordinal trees over large alphabets*

3.5 Tree extraction

The representation we seek must map every α-operation on a labeled ordinal tree T to a small number of operations on auxiliary structures whose size depends on n and on the entropy of the labels, not on σ and n jointly. A primitive that isolates, for each label α, the part of T relevant to that label, and does so without storing a separate tree per label, is the technical device that realises the reduction. Tree extraction is that primitive. Given a subset X of the nodes of T containing the root, the X-extraction of T is a new ordinal tree T_X on exactly the nodes of X, whose shape encodes the ancestor, order, and depth relations among those nodes inherited from T.

Definition 3.5.1 (X-extraction) *Let T be an ordinal tree on n nodes and let $X \subseteq V(T)$ be a subset containing the root. The X-extraction of T is the ordinal tree T_X obtained from T by deleting, one by one in level order, every node $v \notin X$. When v is deleted, the children of v are inserted into the list of children of the parent of v, in place of v, preserving their original left-to-right order among themselves. The deletion leaves a tree on exactly the nodes of X, and the assignment $v \mapsto v$ is a natural bijection between $V(T) \cap X$ and $V(T_X)$.*

The deletion rule above is the delete operation of tree edit distance used by Tai [41]. It deletes a non-root node together with its edge to the parent, and the subtree below the deleted node slides into the vacated slot. Bille's survey records the equivalent rule used here, where the children of the deleted node enter the parent's child list as a contiguous left-to-right subsequence [42]. He, Munro, and Zhou use the same deletion rule as a data-structure primitive on ordinal trees, state the preservation results for X-extraction, reduce path queries to depth queries on extracted forests, and give the forest form and proofs used below [43, 44]. Figure 3.5 shows the operation on a concrete sixteen-node tree.

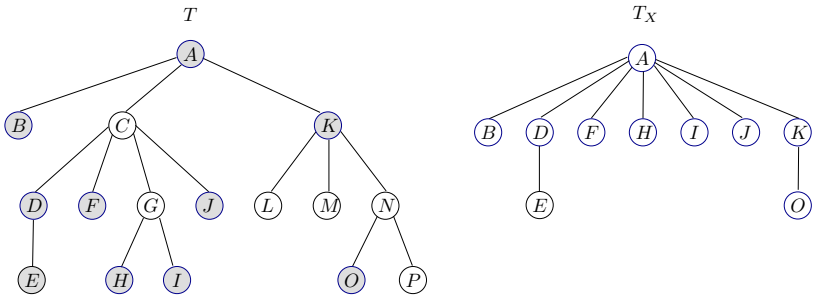

Figure 3.5: X-extraction on a sixteen-node ordinal tree, reproduced from Gagie, He, and Navarro [9]. The shaded nodes form X, and the extracted tree T_X retains exactly them after each unshaded node promotes its children into the sibling list of its parent.

We now record the three preservation properties of T_X that justify calling it a structural projection of T onto the nodes of X. The deletion order fixes a concrete construction, but the extracted forest is independent of that order, and the tree construction used for labeled operations follows the same preservation rules [40, 44]. The first property says that ancestry transfers between T and T_X. The second says that preorder and postorder transfer. The third relates the depth of the nearest retained ancestor in the extracted tree to the number of retained nodes on the original root path.

Lemma 3.5.1 (Ancestry preservation [44]) *Let T be an ordinal tree and let $X \subseteq V(T)$ contain the root of T. For every pair $u, v \in X$, the node u is an ancestor of v in T if and only if u is an ancestor of v in T_X.*

Proof. Fix a level-order enumeration $z_1, z_2, \ldots, z_k$ of the nodes of T not in X, and write $T^{(0)} = T$ and $T^{(i)} = T^{(i-1)}$ with z_i deleted, so that $T^{(k)} = T_X$. We show, by induction on i, that ancestry among the nodes of X is the same in $T^{(i)}$ as in T. The claim is immediate for $i = 0$.

Assume the claim for $i - 1$ and consider the single deletion of $z = z_i$ in $T^{(i-1)}$. Let p be the parent of z in $T^{(i-1)}$ and let $c_1, \ldots, c_d$ be the children of z in $T^{(i-1)}$, in left-to-right order. After the deletion, p acquires $c_1, \ldots, c_d$ as consecutive children in place of z, and every other parent-child edge is untouched. Take $u, v \in X$ with u an ancestor of v in $T^{(i-1)}$. The unique path from v up to u in $T^{(i-1)}$ either avoids z, in which case the path is intact in $T^{(i)}$ and u remains an ancestor of v, or passes through z, in which case the sub-path of the form $v \to \cdots \to c_j \to z \to p \to \cdots \to u$ becomes $v \to \cdots \to c_j \to p \to \cdots \to u$ in $T^{(i)}$, a path in $T^{(i)}$ that still makes u an ancestor of v. Conversely, if u is an ancestor of v in $T^{(i)}$ with $u, v \in X$, the promotion rule is the only way a new edge is created, and it merely splices p between z's old position and z's children. Walking backwards along the ancestry path in $T^{(i)}$ and reinserting z whenever an edge of the form $p \to c_j$ was produced by the deletion reconstructs a path in $T^{(i-1)}$ witnessing the same ancestry. Both directions extend the claim from $i - 1$ to i, and the induction closes at $i = k$. □

Lemma 3.5.2 (Traversal-order preservation [40, 44]) *Let T be an ordinal tree and let $X \subseteq V(T)$ contain the root. For every pair $u, v \in X$, u precedes v in the preorder traversal of T if and only if u precedes v in the preorder traversal of T_X. The same equivalence holds for postorder.*

Proof. It is enough to inspect one deletion. Let z be a non-root node with parent p and children $c_1, \ldots, c_d$ in left-to-right order. In the preorder traversal before the deletion, the block contributed by the subtree of z is z followed by the preorder blocks of the subtrees rooted at $c_1, \ldots, c_d$. After the deletion, the same child blocks occupy the slot formerly occupied by z, in the same left-to-right order, and the only removed entry is z. All traversal entries outside the subtree of z keep their relative order. Thus the preorder sequence of the surviving nodes is the old preorder sequence with z removed.

In postorder, the same local block consists of the postorder blocks of the subtrees rooted at $c_1, \ldots, c_d$, followed by z. After the deletion, those child blocks again occupy the same slot and the same order, and the only removed entry is z. Therefore the postorder sequence of the surviving nodes is the old postorder sequence with z removed. Iterating over the deletion sequence gives both traversal equivalences for the final tree T_X. □

Lemma 3.5.3 (Depth identity [44]) *Let T be an ordinal tree and let $X \subseteq V(T)$ contain the root, under the convention that a node is its own 0-th ancestor. For every $v \in V(T)$, let $w = \mathrm{anc}_X(T, v)$ be the lowest node of X on the root-to-v path, and let w_X be the image of w under the natural bijection between X and $V(T_X)$. Write $\mathrm{dep}^+_X(T, v)$ for the number of nodes of X on the root-to-w path, including w. Then*

$$\mathrm{dep}_{T_X}(w_X) + 1 \;=\; \mathrm{dep}^+_X(T, v).$$

Equivalently, if a dummy root is added above T_X, the depth of w_X in the dummy-rooted tree is exactly $\mathrm{dep}^+_X(T, v)$.

Proof. Since w is the lowest node of X on the root-to-v path, the nodes counted by $\mathrm{dep}^+_X(T, v)$ are exactly w together with the nodes of X that are strict ancestors of w in T. By Lemma 3.5.1, a node $x \in X$ is a strict ancestor of w in T if and only if its image x_X is a strict ancestor of w_X in T_X. Thus the strict ancestors of w_X in T_X are precisely the images of the strict X-ancestors of w in T. The number of such strict ancestors is $\mathrm{dep}_{T_X}(w_X)$ under the zero-based depth convention of Section 3.1. Adding the node w itself gives $\mathrm{dep}_{T_X}(w_X) + 1 = \mathrm{dep}^+_X(T, v)$. □

Lemma 3.5.1, Lemma 3.5.2, and Lemma 3.5.3 together make T_X a structural projection of T onto X. Ancestry is preserved, so any navigational query on T that only consults ancestor or descendant relations among X-nodes can be answered on T_X. Preorder and postorder are preserved, so subtree-rank and subtree-select primitives on X-nodes transfer. The depth identity says that, once the extraction is viewed below a dummy root, the depth of the image of the nearest retained ancestor counts the retained nodes on the original root path up to that ancestor. This is the form the binary partition of Section 3.7 exploits to reduce path counting to depth queries on extracted forests. These three properties recur as building blocks in the two representations developed in the remainder of this chapter.

3.6 α-operations

The three preservation lemmas of Section 3.5 make the extracted tree T_X a structural projection of the ordinal tree T onto the subset $X \subseteq V(T)$. Ancestry transfers in both directions, preorder and postorder transfer, and the depth identity counts retained ancestors after the extraction is placed below a dummy root. We now put this projection to work on the α-operations problem that Section 3.4 left open, where the bound of Theorem 3.4.1 does not explain how to keep large alphabets near their information-theoretic cost.

Definition 3.6.1 (α-operations on the ancestor axis) *Let T be an ordinal tree on n nodes, each node carrying a label from $[1..u]$. For a node v of T, a label $\alpha \in [1..u]$, and an integer $i \geq 1$, we define three α-operations on the root-to-v path:*

- $\texttt{parent}_\alpha(v)$*: the lowest proper ancestor of v whose label is α, or $\perp$ if no proper ancestor of v carries label α.*
- $\mathrm{rank}_\alpha(v)$*: the number of nodes with label α on the root-to-v path, including v when its label is α.*
- $\mathrm{select}_\alpha(v, i)$*: the i-th node with label α on the root-to-v path, counted from the root downward, or $\perp$ if fewer than i such nodes exist.*

Let $\Sigma_T = \{\alpha_1 < \alpha_2 < \cdots < \alpha_\sigma\}$ be the labels that occur in T. For every active label $\alpha \in \Sigma_T$, we build an extracted tree T_α that retains only the nodes of T whose label information is relevant to α. Let X_α consist of a fresh root r_α, every node of T whose label is α, and every parent of such a node, where r_α is attached as the sole parent of the original root of T in

He, Munro, and Zhou [40] list a larger family of α-operations, including $\texttt{depth}_\alpha$, $\texttt{level\_anc}_\alpha$, and descendant-axis queries. We restrict the exposition to the three ancestor-axis operations required by the set-collection constructions of later chapters.

the augmented tree. The X_α-extraction of the augmented tree, as given by Definition 3.5.1, is the desired T_α. Each node of T_α carries a single bit, namely the marker 1 if its corresponding node in T is an α-node, and the marker 0 otherwise. In particular, the root r_α of T_α always carries marker 0.

Each α-node of T contributes one 1-marked node and at most one further 0-marked parent to T_α, while the fresh root r_α adds a single 0-marked node. Thus $|T_\alpha| \le 2n_\alpha + 1$ and $\sum_{\alpha\in\Sigma_T} |T_\alpha| \le 2n + \sigma$, where n_α counts the α-nodes of T. Since $\sigma \le n$, the active family has $O(n)$ nodes. By Lemma 3.5.1, the α-labeled nodes on the root-to-y path of any α-node y correspond bijectively to the 1-marked nodes on the root-to-y' path of its image $y' \in V(T_\alpha)$. The three α-operations on T reduce to marker-1 ancestor operations after the query is routed to the relevant α-node on the root-to-query path.

The active trees already have linear total size, but the framework stores them as one object. We concatenate the family into a single ordinal tree $\mathcal{T}$ with a fresh root $\mathcal{R}$, whose children are the roots of $T_{\alpha_1}, T_{\alpha_2}, \ldots, T_{\alpha_\sigma}$ in that order. The tree for the label of the original root saves one parent marker, so adding $\mathcal{R}$ keeps the merged tree $\mathcal{T}$ within $2n + \sigma$ nodes. The preorder traversal of each T_α occurs as a contiguous substring of the preorder traversal of $\mathcal{T}$. The same holds for the DFUDS traversal once the new root r_α of each T_α is removed. In the representation we build below, $\mathcal{T}$ plays the role of a single 0/1-labeled ordinal tree on which we support marker operations, as Figure 3.6 makes concrete on a five-node example.

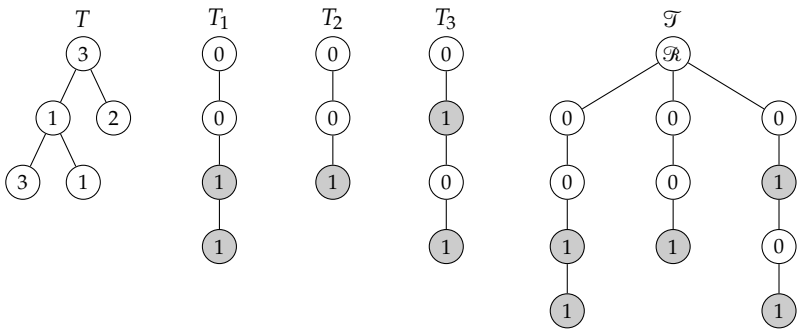

Figure 3.6: From left to right, an ordinal tree T on five nodes with labels in $\{1, 2, 3\}$, the extracted trees T_1, T_2, and T_3 obtained from the corresponding X_α-extractions, and the merged tree $\mathcal{T}$ obtained by placing those roots below a fresh root $\mathcal{R}$. Empty circles carry marker 0, and shaded circles carry marker 1. After the mapping steps handled by the other framework blocks, the ancestor query inside each T_α becomes a marker-1 ancestor query inside the corresponding subtree of $\mathcal{T}$, which is the part represented by $\mathcal{D}_3$ in Lemma 3.6.1.

The construction leaves three kinds of information to index: the shape of T, the preorder label sequence of T, and the marker tree $\mathcal{T}$. The next lemma is the part of the framework of He, Munro, and Zhou needed for these ancestor-axis operations.

Lemma 3.6.1 (Ancestor-axis framework decomposition [40]) *Let T be an ordinal tree on n nodes with active labels $\Sigma_T = \{\alpha_1 < \alpha_2 < \cdots < \alpha_\sigma\} \subseteq [1..u]$, let $PLS_T[1..n]$ be the preorder label sequence of T remapped to $[1..\sigma]$, and let $\mathcal{T}$ be the merged tree built above. Suppose that three data structures are available:*

- $\mathcal{D}_1$**:** *an unlabeled ordinal tree on the shape of T, occupying $\mathcal{S}_1(n)$ bits and supporting parent, depth, preorder rank, subtree size, level-ancestor, and lowest-common-ancestor in constant time.*
- $\mathcal{D}_2$**:** *a representation of PLS_T as a string over $[1..\sigma]$, occupying $\mathcal{S}_2(PLS_T)$ bits and supporting* rank_j *and* select_j *for every active-rank label $j \in [1..\sigma]$.*

$\mathcal{D}_3$: *a* 0/1*-labeled ordinal tree on* $\mathcal{T}$*, occupying* $\mathcal{S}_3(2n+\sigma)$ *bits and supporting marker-*1 *parent, marker-*1 *depth, marker-*1 *ancestor selection by marker rank, marker preorder rank and select, and the unlabeled navigational operations in constant time.*

These structures represent T *in* $\mathcal{S}_1(n)+\mathcal{S}_2(PLS_T)+\mathcal{S}_3(2n+\sigma)$ *bits and support every* α*-operation of Definition 3.6.1 with a constant number of calls to the operations of* $\mathcal{D}_1$, $\mathcal{D}_2$, *and* $\mathcal{D}_3$ *after an active-label lookup.*

Proof. The cited framework proves this decomposition for a larger set of labeled-tree operations. We keep only the operations that appear in Definition 3.6.1. Those reductions use the original tree shape, the preorder label sequence, and the merged marker tree. They do not use the leaf bitvector needed by the broader statement. Remapping the occurring labels to $[1..\sigma]$ preserves preorder ranks and selections after the active-label lookup, so the same reductions apply to the active alphabet. The space bound is the sum of the three retained blocks, using $|\mathcal{T}| \le 2n+\sigma$ from the construction above. □

The following theorem records only the consequence needed later: the ancestor-axis operations of Definition 3.6.1 can be answered within the same word-space budget used by the set-collection constructions. It is a specialization of the labeled-tree framework, with the construction-time bound taken from the set-difference use of tree extraction.

Theorem 3.6.2 (Ancestor-axis α-operations [9, 40]) *Let* T *be an ordinal tree on* n *nodes, each carrying a label from* $[1..u]$*, and assume the word RAM model with word size* $\omega = \Omega(\lg n)$*. There exists a representation of* T *in* $O(n)$ *words of space that supports* $\texttt{parent}_\alpha(v)$, $\mathrm{rank}_\alpha(v)$, *and* $\mathrm{select}_\alpha(v,i)$ *of Definition 3.6.1 in* $O(\lg \lg_\omega u)$ *time, and that can be built in* $O(n \lg u)$ *time.*

Proof. Store Σ_T with the quotient dictionary of Theorem 2.1.6. It maps an occurring label α_j to its active rank j in $O(1)$ time and detects labels outside Σ_T, for which $\texttt{parent}_\alpha$ and select_α return $\perp$ and rank_α returns 0. Its $O(\sigma \lg u)$-bit cost is within the final $O(n \lg u)$ budget because $\sigma \le n$.

Instantiate $\mathcal{D}_1$ with the tree-covering representation of Theorem 3.3.2. Instantiate $\mathcal{D}_2$ with the sequence representation of Belazzougui and Navarro [45] on the active-rank sequence PLS_T, which gives constant-time select_j and $O(\lg \lg_\omega \sigma) \subseteq O(\lg \lg_\omega u)$-time rank_j in $O(n \lg u)$ bits. Instantiate $\mathcal{D}_3$ with the binary-alphabet case of the labeled-tree framework on $\mathcal{T}$ [40], which occupies $O(n)$ bits and answers marker operations in constant time because $|\mathcal{T}| = O(n)$.

[45]: Belazzougui et al. (2015), *Optimal lower and upper bounds for representing sequences*

These blocks satisfy Lemma 3.6.1. The total space is $O(n \lg u)$ bits, equivalently $O(n)$ words under the word-space convention used in the set-collection chapters. Each query makes only a constant number of calls to the three blocks, and the only nonconstant call is rank_j on $\mathcal{D}_2$, so the time is $O(\lg \lg_\omega u)$. The active-label dictionary, extracted trees, merged marker tree, and component indexes can be built within the $O(n \lg u)$ word-RAM budget used for the tree-extraction overlay of Gagie, He, and Navarro [9]. □

The representation of Theorem 3.6.2 answers the ancestor-axis α-operations in time sublogarithmic in the alphabet size and in space proportional to n words, which is the bound used by the membership reduction in Section 5.3. The next section extends this single-label primitive to path queries over ranges of labels.

3.7 Path queries

He, Munro, and Zhou [44] also handle node-to-node paths and path-median queries. The set-collection constructions of later chapters only need the root-to-node case, which already admits the sharper time bounds of the present section.

The α-operations of Definition 3.6.1 aggregate information along the root-to-v path when the query is pinned to a single label. Many downstream uses of labeled trees ask a broader question. How many ancestors of v carry a label in a query range $[p..q]$, and what is the i-th smallest label on the path. We call these *path queries*, and we restrict attention to the root-to-node version, which is what the set-collection constructions of later chapters need. The construction develops in two steps, each with its own query time bound, and this section covers the first of them.

3.7.1 Binary partition

[28]: Grossi et al. (2003), *High-order entropy-compressed text indexes*

The closing of Section 2.3 promised that the hierarchical binary partition of the alphabet, which organises rank and select on a sequence through a chain of bitmaps, would transfer from sequences to labeled ordinal trees. We use that connection here. The wavelet tree of Grossi, Gupta, and Vitter [28] splits the alphabet $[1..u]$ at each recursive level by a midpoint, stores at every range a bitmap that records, for each symbol of the underlying subsequence, whether it falls in the left or right half of the range, and answers rank_c and select_c through a root-to-leaf traversal whose per-level cost is a single binary rank or select. We replicate this plan on a labeled ordinal tree T on n nodes with labels drawn from $[1..u]$. The alphabet is partitioned by the same range tree, and at every range $[a..b]$ we store a 0/1-labeled extracted tree $T_{a,b}$ whose 0 markers flag the nodes with labels in the left half and whose 1 markers flag those with labels in the right half. Level transitions move between extracted trees through a constant number of marker operations, and the three preservation lemmas of Section 3.5 guarantee that depths and ancestries transfer across the transitions in the form the queries need.

Definition 3.7.1 (Path queries) *Let T be an ordinal tree on n nodes with labels in $[1..u]$, and let $v \in V(T)$. For a range $[p..q] \subseteq [1..u]$ and an integer $i \geq 1$, we define two operations on the root-to-v path of T:*

- `path_count`$(v, [p..q])$*: the number of nodes on the root-to-v path whose label belongs to $[p..q]$, with equal labels on different nodes counted separately.*
- `path_select`(v, i)*: the i-th smallest label in the multiset of labels appearing on the root-to-v path. Return $\perp$ if i exceeds the length of the path.*

Under the convention that a node is its own 0-th ancestor, the root-to-v path includes v itself.

The alphabet is organised by a *conceptual range tree* on $[1..u]$. The root range is $[1..u]$. A non-leaf range $[a..b]$ with $a < b$ has two child ranges $[a..m]$ and $[m+1..b]$, where $m = \lfloor (a+b)/2 \rfloor$. The recursion stops at the u leaf ranges of the form $[\alpha..\alpha]$, one for every value $\alpha \in [1..u]$. The conceptual range tree has height $\lceil \lg u \rceil$ and is balanced up to a unit imbalance in sibling lengths. No material storage is devoted to the range tree itself. Each range is an abstract index under which we store the labeled tree associated with it.

For each range $[a..b]$ in the conceptual range tree we store an extracted ordinal tree $T_{a,b}$. Let $R_{a,b}$ denote the set of nodes of T whose label lies in $[a..b]$, and let $F_{a,b}$ be the forest analogue of Definition 3.5.1, obtained by deleting from T every node outside $R_{a,b}$. The forest $F_{a,b}$ has exactly $|R_{a,b}|$ nodes. The object $T_{a,b}$ is $F_{a,b}$ with a dummy root $\rho_{a,b}$ attached above, following He, Munro, and Zhou [44], so that $T_{a,b}$ is a genuine ordinal tree and its nondummy nodes correspond bijectively to the nodes of $R_{a,b}$. At the top of the recursion, $T_{1,u}$ contains every node of T under the dummy root, and its preservation properties reduce to those of T itself. At every nonleaf range $[a..b]$ with children $[a..m]$ and $[m+1..b]$, each nondummy node of $T_{a,b}$ carries a single marker bit. The marker is 0 if the label of the corresponding node of T lies in $[a..m]$ and 1 if it lies in $[m+1..b]$. The child trees $T_{a,m}$ and $T_{m+1,b}$ are, up to a relabelling of the dummy root, the extractions of $T_{a,b}$ onto its marker-0 and marker-1 nondummy nodes, respectively, under the convention that the dummy root is preserved. At a leaf range $[\alpha..\alpha]$, $T_{\alpha,\alpha}$ collects the nodes of T whose label equals α.

[44]: He et al. (2016), *Data structures for path queries*

Each $T_{a,b}$ is accessed through three navigation primitives on every node, namely the depth of the node within $T_{a,b}$, a link back to the corresponding node of T, and child-image transitions leading to the two children $T_{a,m}$ and $T_{m+1,b}$. In the original pointer-machine construction these values are stored explicitly. In the succinct encoding the weak marker-ancestor step behind a child-image transition is realised by the binary instance of the α-operations of Definition 3.6.1, which has constant cost because the marker alphabet has size two. The preservation lemmas of Section 3.5 justify the navigation. For a marker $\beta \in \{0,1\}$, the transition first finds the lowest node on the root-to-v path of $T_{a,b}$ whose marker is β, using the dummy root when no such node exists. Its image in the corresponding child tree is then the natural retained-node image given by the extraction.

Theorem 3.7.1 (Binary-partition path queries [9, 44]) *Let T be an ordinal tree on n nodes with labels in $[1..u]$, and assume the word-space convention that a universe value fits in one machine word. The binary partition represents T in $O(n)$ words of space, supports the operations* `path_count`$(v, [p..q])$ *and* `path_select`(v, i) *of Definition 3.7.1 in $O(\lg u)$ time, and can be built in $O(n \lg u)$ time.*

Proof. The construction above is the binary range-tree construction of He, Munro, and Zhou [44] restricted to root-to-node paths. Their pointer-machine presentation stores explicit depths and child-image links. We use the same hierarchy with the bit-level storage recorded by Gagie, He, and Navarro [9]. At a fixed level ℓ of the conceptual range tree, the

[44]: He et al. (2016), *Data structures for path queries*

materialised ranges partition the labels occurring in T, so

$$\sum_{[a..b] \text{ at level } \ell} |R_{a,b}| = n,$$

and the number of dummy roots at that level is at most the number of nonempty ranges, hence at most n. Empty ranges are not materialised. Since every stored tree has a binary marker alphabet, one level costs $O(n)$ bits, and all $\lceil \lg u \rceil$ levels cost $O(n \lg u)$ bits. Under the assumption that a universe value fits in one word, this is $O(n)$ words. The same hierarchy can be built in $O(n \lg u)$ time.

The query algorithms are the corresponding root-to-node specializations of the path-counting and path-selection algorithms of He, Munro, and Zhou [44]. For counting, a fully covered range $[a..b]$ contributes the depth of the current node in $T_{a,b}$, which equals the number of retained ancestors by Lemma 3.5.3 with the dummy root placed above the extracted forest. The remaining work follows the canonical decomposition of $[p..q]$ in the binary range tree, so it visits $O(\lg u)$ ranges. For selection, the depth of the left child extraction tells whether the requested label lies in the left or right half, and the search descends one root-to-leaf branch. Each child-image transition has constant cost on the binary marker alphabet, so both operations take $O(\lg u)$ time. □

3.7.2 Sublogarithmic refinement

The bound of Theorem 3.7.1 is the tree analogue of the $O(\lg u)$ cost that a binary wavelet tree pays on sequences, one alphabet level at a time. On sequences, Ferragina, Manzini, Mäkinen, and Navarro [30] replace the binary partition by an r-ary partition and store each child-index sequence so that access, rank, and select inside a node take constant time. With a suitable polylogarithmic branching factor this gives $O(\lg u / \lg \lg n)$ levels for large alphabets. He, Munro, and Zhou [44] adapt the same multiary idea to labeled ordinal trees, where a level transition is reduced to constant-time primitives on small-alphabet sequences.

[30]: Ferragina et al. (2007), *Compressed representations of sequences and full-text indexes*

We fix a constant $\epsilon \in (0, 1)$ and set the fanout to $f = \lceil \lg^{\epsilon} n \rceil$. The conceptual range tree on $[1..u]$ is built by splitting each nonleaf range into f subranges of as nearly equal length as possible, following He, Munro, and Zhou [44]. We write $h = \lceil \log_f u \rceil$ for the number of nonbottom transition levels, so the bottom level in their level numbering is $h + 1$. Since $\lg f = \Theta(\lg \lg n)$,

$$h = O\left(\frac{\lg u}{\lg \lg n} + 1\right).$$

Path counting calibrates the fanout to $f = \lceil \lg^{\epsilon} n \rceil$, giving $O(\lg u / \lg \lg n + 1)$ transition levels. Path selection will instead use $f' = \lceil \lg^{\epsilon'} u \rceil$, giving $O(\lg u / \lg \lg u)$ transition levels. The two denominators agree only when u is polynomial in n. Outside that regime the two queries genuinely separate.

[44]: He et al. (2016), *Data structures for path queries*

Under the logarithmic word-size convention $\omega = \Theta(\lg n)$, this is $O(\lg_{\omega} u)$. The saving is paid for at every nonbottom level, where the alphabet of markers is now $[1..f]$. The binary transition of Theorem 3.7.1 has to be replaced by constant-time operations on an f-labeled sequence.

The first primitive counts, in a sequence over a small integer alphabet, how many entries up to a given position are not larger than a threshold.

Lemma 3.7.2 (Counting up to a threshold [44, 46]) *In the word RAM model, let $S[1..n]$ be a sequence over the alphabet $[1..f]$ with $f = O(\lg^{\epsilon} n)$ for some constant $\epsilon \in (0, 1)$. Assume that S is stored in a representation from which any $\lceil \lg^{\lambda} n \rceil$ consecutive entries are accessible in $O(1)$ time, for some constant $\lambda \in (\epsilon, 1)$. There exists an auxiliary structure of $o(n)$ bits that supports, in $O(1)$ time, the query*

$$\texttt{count}_{\beta}(S, i) \;=\; |\{\, j \le i : S[j] \le \beta \,\}|$$

for every $\beta \in [1..f]$ and every $i \in [1..n]$.

The second primitive counts how many entries in a prefix carry the same value as the last entry of that prefix.

Lemma 3.7.3 (Constant-time partial rank [47]) *In the word RAM model, let $S[1..n]$ be a sequence over the alphabet $[1..f]$. There is a representation of S in $O(n \lg f)$ bits supporting, in $O(1)$ time, the query*

$$\texttt{partial\_rank}(S, i) \;=\; |\{\, j \le i : S[j] = S[i] \,\}|.$$

At each nonbottom level $\ell \in \{1, 2, \ldots, h\}$ of the f-ary range tree we merge the family $\{T_{a,b}\}$ at that level into a single labeled ordinal tree T_{ℓ}, following the construction of He, Munro, and Zhou [44]. The ranges at level ℓ partition the n weighted nodes of T into disjoint groups, so the forests $\{F_{a,b}\}$ at level ℓ are node-disjoint. Collecting them under a common dummy root in the order imposed by the ranges produces T_{ℓ}, whose n nondummy nodes correspond bijectively to the nodes of T. Each nondummy node of T_{ℓ} carries a label from $[1..f]$, namely the index of the child range at level $\ell + 1$ to which the node descends. We store T_{ℓ} with the small-alphabet labeled-tree representation used by He, Munro, and Zhou, charged here by the weaker bound $O(|T_{\ell}| \lg f) = O(n \lg f)$ bits. We also store the preorder label sequence of T_{ℓ} in packed fixed-width form, which gives the block access required by Lemma 3.7.2, and index it by Lemma 3.7.2 and Lemma 3.7.3. These additions fit within the same $O(n \lg f)$-bit asymptotic budget.

[44]: He et al. (2016), *Data structures for path queries*

Theorem 3.7.4 (Word-RAM path-query corollary [44]) *Let T be an ordinal tree on n nodes with labels in $[1..u]$, under the word RAM model with word size $\omega = \Theta(\lg n)$, and assume that a universe value fits in one word. There exists a representation of T in $O(n)$ words of space that supports the operation $\texttt{path\_count}(v, [p..q])$ of Definition 3.7.1 in $O(\lg_{\omega} u)$ time and the operation $\texttt{path\_select}(v, i)$ of Definition 3.7.1 in $O(\lg u / \lg \lg u)$ time.*

Proof. He, Munro, and Zhou [44] give two word-RAM structures for weighted ordinal trees. Their path-counting structure uses the f-ary hierarchy described above and answers node-to-node counting in $O(\lg u / \lg \lg n + 1)$ time. Their path-selection structure uses a separate hierarchy with fanout $\lceil \lg^{\epsilon'} u \rceil$ and answers selection in $O(\lg u / \lg \lg u)$ time. A root-to-node query is a special case of their node-to-node query, and Definition 3.7.1 asks selection to return the selected label, which is the value returned by their selection query.

[44]: He et al. (2016), *Data structures for path queries*

It remains only to match the word-space convention used in the set-collection chapters. The two structures have entropy-compressed bit bounds. Since the labels lie in $[1..u]$, the zeroth-order entropy term is

at most $\lg u$, so each hierarchy uses $O(n \lg u)$ bits in the worst case. Storing the counting and selection structures in parallel therefore still costs $O(n \lg u)$ bits, which is $O(n)$ words under the assumption that a universe value fits in one word. Under $\omega = \Theta(\lg n)$, the counting time $O(\lg u / \lg \lg n + 1)$ is $O(\lg_\omega u)$, while the selection time remains $O(\lg u / \lg \lg u)$. □

The word-space statement of Theorem 3.7.4 records only the coarser consequence of sharper bit-level bounds. He, Munro, and Zhou also bound their structures by the zeroth-order entropy $H(W_T)$ of the multiset of node labels of T. Since the labels lie in $[1..u]$, $H(W_T) \leq \lg u$, with equality only when the empirical label distribution is uniform over all values of the universe. Their path-counting representation uses $n(H(W_T) + 2) + o(n)$ bits when $u = O(\lg^\epsilon n)$ for some constant $\epsilon \in (0, 1)$, and $nH(W_T) + O(n \lg u / \lg \lg n)$ bits otherwise, with counting time $O(\lg u / \lg \lg n + 1)$. Their path-selection representation uses $n(H(W_T)+2)+o(n)$ bits when $u = O(1)$, and $nH(W_T)+O(n \lg u / \lg \lg u)$ bits otherwise, with selection time $O(\lg u / \lg \lg u)$. Here we keep the word-space form, because the next chapters use these structures as query primitives.

Hierarchical Encodings 4

We now have data structures that answer queries on labels along a root-to-node path. A set collection does not come with such paths. Storing each set alone pays again for elements repeated across related sets. Given a reference set, we can instead store only what must be deleted or inserted to obtain the target set. Choosing references for the whole collection arranges these differences as labels on a tree. The path to one node records the changes that define one set, so membership, access, rank, predecessor, and successor reduce to queries on path labels. The literature gives two ways to choose the references [2, 9].

[2]: Alanko et al. (2025), *Compact data structures for collections of sets*
[9]: Gagie et al. (2026), *Compressed Set Representations based on Set Difference*

4.1 Independent encoding

We return to the setting of Section 1.2. A collection $(\mathcal{S}, U)$ consists of a totally ordered universe $U = [1..u]$ of size u and a family $\mathcal{S} = \{S_1, \ldots, S_m\}$ of m finite subsets of U, with $n = \sum_{i=1}^{m} |S_i|$ counting elements across sets with multiplicity (Definition 1.2.1). In this chapter we follow the convention $U = \bigcup_i S_i$ used by Alanko et al. [2] and by Gagie, He, and Navarro [9]. If the input names a larger ordered universe, we apply the results to the restriction $U' = \bigcup_i S_i$ and translate query keys through the induced order, since every element outside U' is absent from every set. A representation of $\mathcal{S}$ must support the five operations of Definition 1.2.2, namely member, access, rank, predecessor, and successor. The independent-encoding baseline of Definition 1.2.3,

Section 1.2 fixed the model and operations. Chapter 2 showed how sparse bitvectors meet the set-by-set baseline. This chapter asks what falls below that baseline when $\mathcal{S}$ has structure.

$$H_{wc}(\mathcal{S}) = \sum_{i=1}^{m} \lg \binom{u}{|S_i|},$$

is the per-set information content of the collection when each S_i is viewed as an isolated subset of U. No inter-set relationship contributes to this cost.

The baseline is achievable up to lower-order terms. Applying Theorem 2.2.2 to the characteristic bitvector of each S_i separately stores the whole collection in

$$H_{wc}(\mathcal{S}) + O(n + m \lg n) \quad \text{bits},$$

with the select_1, rank_1, and bit-access times of Theorem 2.2.2 [24]. The set query $\text{access}(S, j)$ is select_1 on the characteristic bitvector of S. Membership is bit access at position x, or equivalently the check that $r = \text{rank}(S, x)$ is nonzero and $\text{access}(S, r) = x$. Predecessor and successor use the rank-then-access reduction of Section 1.2. The $O(n)$ slack collects the high-part bitvectors, auxiliary select structures, and rounding terms in the m per-set Elias-Fano layouts (Subsection 2.2.2). The $O(m \lg n)$ slack pays for a directory of pointers into the m separate bitvectors, so that each set can be addressed in $O(1)$ time.

[24]: Okanohara et al. (2007), *Practical entropy-compressed rank/select dictionary*

The set-by-set representation is the right starting point when the collection really is a bag of unrelated sets. For every S_i, there are $\binom{u}{|S_i|}$ subsets of U of

In the applications of Section 1.1, $\mathcal{S}$ is not a bag of unrelated sets. Adjacency lists of neighbouring web nodes often share many links, and colour sets of neighbouring k-mers are often similar. The baseline ignores this overlap.

that size, and distinguishing them costs $\lg \binom{u}{|S_i|}$ bits. For the independent class that fixes only the set sizes, the worst-case counting lower bound is exactly the sum of the per-set costs. The difficulty is that collections arising in practice are almost never independent in this sense. Two sets that differ in only one universe element pay full price twice under H_{wc}, and nothing in the baseline records that one is a one-edit variant of the other. If $S_i \subset S_j$, describing S_i as a subset of S_j costs $\lg \binom{|S_j|}{|S_i|}$ bits, at most $\lg \binom{u}{|S_i|}$ and strictly less whenever $0 < |S_i|$ and $|S_j| < u$. More generally, if the set of elements that belong to exactly one of S_i and S_j, written $S_i \triangle S_j$, is small, a description of S_j as an edit of S_i costs $|S_i \triangle S_j| \lg u$ bits by listing the changed universe elements. Every such relationship is invisible to H_{wc}.

The rest of this chapter answers both the combinatorial and the structural side of the question, taking each compressibility measure end to end. Section 4.2 starts from directional references. Containment entropy encodes a set against a larger reference and pays a binomial cost, while insertion compressibility encodes a set against a smaller reference and pays for the inserted elements. The same section shows how the insertion view becomes a labelled ordinal tree on which the tree-extraction primitives of Section 3.5 answer the five operations of Definition 1.2.2. Section 4.3 then drops the containment requirement and pays in symmetric-difference labels of two signs, with a parallel structural realisation. Throughout, cost is measured in bits of information content for the combinatorial side and in words of space and query time for the structural side.

4.2 Insertion compressibility

The question is posed on a single pair (S, R). The next two subsections apply each primitive to the whole collection, obtaining two compressibility measures whose definitions disagree on which direction of the containment is assumed but whose information-theoretic flavour is the same.

Given a set S and a reference set R, how many bits are needed to describe S once R is known? Two containment cases give useful bounds. When $S \subseteq R$, describing S reduces to choosing $|S|$ positions out of the $|R|$ elements of R, at the exact cost $\lg \binom{|R|}{|S|}$ bits. When $R \subseteq S$, a label-wise description lists the $|S \setminus R|$ elements that must be added to R to obtain S. These elements lie in $U \setminus R$, so writing them over that alphabet costs $|S \setminus R| \lg(u - |R|)$ bits, and writing them over U costs $|S \setminus R| \lg u$ bits. The tighter unordered count is $\lg \binom{u-|R|}{|S \setminus R|}$, but the insertion measure below follows the label-count view used by the tree representation. The two cases are duals of one another. One specifies which elements of a larger reference are retained, the other specifies which elements outside a smaller reference are inserted.

[2]: Alanko et al. (2025), *Compact data structures for collections of sets*

[9]: Gagie et al. (2026), *Compressed Set Representations based on Set Difference*

At the level of a whole collection $\mathcal{S}$, each primitive yields a compressibility measure. Choosing the reference $R = p(S)$ to be a smallest proper superset of S in $\mathcal{S}$, or U if no such superset exists, instantiates the first primitive and gives the containment entropy $L(\mathcal{S})$ of Alanko, Bille, Gørtz, Navarro, and Puglisi [2]. Choosing $R = p'(S)$ to be a strict subset of S in $\mathcal{S}$ of maximum cardinality, or $\emptyset$ if none exists, instantiates the second and gives the insertion compressibility $I(\mathcal{S})$ of Gagie, He, and Navarro [9]. The rest of this section develops the two primitives as counting bounds, lifts each one to a measure on $\mathcal{S}$, recalls the containment-bounded representation, and then turns the insertion measure into a labelled tree that supports the

five operations of Definition 1.2.2 through the tree-extraction primitives of Section 3.5.

4.2.1 Reference encoding

We isolate the two primitives as purely combinatorial counting bounds, independently of any data-structure representation. Throughout, U is the universe of size u and $S, R \subseteq U$ are finite.

The two primitives do not exhaust the space of reference encodings. When R and S are incomparable, a reference description can record the elements that belong to exactly one of them, namely $S \triangle R$. The next section turns this idea into a measure.

Proposition 4.2.1 (Subset primitive) *If $S \subseteq R \subseteq U$, then S can be described in $\lg \binom{|R|}{|S|}$ bits given R. The bound is tight over the class of subsets of R of size $|S|$.*

Proof. There are exactly $\binom{|R|}{|S|}$ subsets of R of size $|S|$, and S is one of them. An encoding restricted to this class uses $\lg \binom{|R|}{|S|}$ bits by indexing. Since every subset of R of size $|S|$ is a valid value of S, no shorter prefix-free code exists. □

For $0 < |S|$, the subset primitive strictly improves on the independent cost $\lg \binom{u}{|S|}$ of Definition 1.2.3 whenever $|R| < u$. Indeed $\binom{|R|}{|S|} \leq \binom{u}{|S|}$, with equality only when $|R| = u$. Thus $R = U$ recovers the baseline, and every proper reference set yields a strict saving.

Proposition 4.2.2 (Superset primitive) *If $R \subseteq S \subseteq U$, then S can be described by specifying the $|S \setminus R|$ elements of $S \setminus R$. A label-wise encoding with alphabet $U \setminus R$ uses $|S \setminus R| \lg(u - |R|)$ bits, and one with alphabet U uses $|S \setminus R| \lg u$ bits.*

Proof. The elements of $S \setminus R$ are distinct members of $U \setminus R$. Encoding each as an integer in $[1..u - |R|]$ costs $\lg(u - |R|)$ bits. Encoding each as an integer in $[1..u]$ costs $\lg u$ bits. The second encoding discards the knowledge of R and is looser. □

The two primitives describe S given R in opposite directions. The subset primitive measures S as a fraction of a larger reference. The superset primitive measures how many elements must be added to a smaller reference. Neither exhausts the possibilities when R and S are incomparable. The next section develops the corresponding primitive for a close but unordered neighbour.

4.2.2 Containment entropy

Alanko et al. [2] apply Proposition 4.2.1 to every set in $\mathcal{S}$ at once. For each S_i, the reference is a smallest proper superset of S_i among the sets of $\mathcal{S}$. If none exists, the reference is U itself. After arbitrary tie-breaking among equal-size choices, this parent map induces a tree on $\mathcal{S} \cup \{U\}$.

Alanko et al. write n_i for $|S_i|$, p_i for $|p(S_i)|$, n for the total size, and s for the number of sets. We keep the thesis notation from Definition 1.2.1.

[2]: Alanko et al. (2025), *Compact data structures for collections of sets*

Definition 4.2.1 (Containment hierarchy [2]) *Let $\mathcal{S} = \{S_1, \ldots, S_m\}$ be a collection over universe U. The containment hierarchy of $\mathcal{S}$ has U at the root. For each $S_i \in \mathcal{S}$, the parent $p(S_i)$ is any smallest set in $\mathcal{S}$ strictly containing S_i, with ties broken arbitrarily. If no such set exists, then $p(S_i) = U$.*

Definition 4.2.2 (Containment entropy [2]) *The containment entropy of $\mathcal{S}$ is*

$$L(\mathcal{S}) = \sum_{i=1}^{m} \lg \binom{|p(S_i)|}{|S_i|},$$

where $p(S_i)$ is the parent of S_i in the containment hierarchy.

The definition is an instance-by-instance application of Proposition 4.2.1. Every S_i is a subset of its parent $p(S_i)$, so the subset primitive gives a description of length $\lg \binom{|p(S_i)|}{|S_i|}$ bits. Summing over i yields the containment entropy. When the hierarchy is trivial and every $p(S_i)$ equals U, the containment entropy collapses to the baseline.

Proposition 4.2.3 (Containment bound [2]) *For every collection $\mathcal{S}$,*

$$L(\mathcal{S}) \le H_{wc}(\mathcal{S}).$$

Proof. Since $p(S_i) \subseteq U$, we have $|p(S_i)| \le u$, and hence $\binom{|p(S_i)|}{|S_i|} \le \binom{u}{|S_i|}$. Taking logarithms and summing over i gives $L(\mathcal{S}) \le \sum_i \lg \binom{u}{|S_i|} = H_{wc}(\mathcal{S})$. □

To translate the bound into space, the representation must shorten the root-to-leaf paths in the hierarchy. A direct representation stores each S_i as a sparse bitvector of length $|p(S_i)|$ indicating which elements of the parent are retained. Queries on S_i then walk the chain of ancestors up to U, which can be as long as the hierarchy itself. Alanko et al. introduce a path-compressed variant in which the parent of S_i skips all intermediate ancestors whose size is at most $2|S_i|$.

Definition 4.2.3 (Contracted hierarchy [2]) *The contracted hierarchy of $\mathcal{S}$ has U at the root. For each S_i, its parent $c(S_i)$ is the highest ancestor S_t of $p(S_i)$ in the containment hierarchy satisfying $|S_t| \le 2|S_i|$. If no such ancestor exists, $c(S_i) = p(S_i)$.*

The threshold $2|S_i|$ balances space and traversal time. Replacing $p(S_i)$ by $c(S_i)$ may move the reference upward, but the new reference is either still at most twice as large as S_i or it is the original parent itself. Thus the sparse bitvector for S_i pays only the additive slack accounted for below. In exchange, the contracted edge skips every ancestor whose size is too close to $|S_i|$ to help the depth. After two contracted edges, the set size has more than doubled, which is why the next proposition gives logarithmic depth.

Proposition 4.2.4 (Depth of the contracted hierarchy [2]) *In the contracted hierarchy of $\mathcal{S}$, every node S_i has depth $O(\log(u/|S_i|))$.*

Proof. Let $A_0, A_1, \ldots$ be the contracted path from $A_0 = S_i$ toward U. Whenever A_{t+2} exists, the set A_{t+2} contains the original-hierarchy parent of $A_{t+1} = c(A_t)$. By the definition of $c(A_t)$, that original parent has size greater than $2|A_t|$, since otherwise $c(A_t)$ would not be the highest eligible ancestor. Hence $|A_{t+2}| > 2|A_t|$. It follows that $|A_{2j}| > 2^j|S_i|$, and the inequality can hold for only $O(\log(u/|S_i|))$ values of j because all sets have size at most u. □

Encoding each S_i as a sparse bitvector relative to $c(S_i)$ adds only $O(n)$ bits over the containment-entropy sum, so the total remains $L(\mathcal{S}) + O(n + m \lg n)$ bits. The construction uses the sparse-bitvector representation of Section 2.2.

Theorem 4.2.5 (Containment-bounded representation [2]) *Let $\mathcal{S}$ be a collection of m sets over a universe of size u, with total size $n = \sum_i |S_i|$ as in Definition 1.2.1. The contracted hierarchy of $\mathcal{S}$ can be stored in*

$$L(\mathcal{S}) + O(n + m \lg n) \quad \textit{bits}$$

using one sparse bitvector per edge and the contracted-hierarchy traversal, supporting on every $S \in \mathcal{S}$ the five operations of Definition 1.2.2 in time $O(\log(u/|S|))$.

The additive term matches the slack already present in the baseline of Section 4.1. Each of the m sparse bitvectors spends $O(|S_i|)$ additive bits in its Elias-Fano layout, summing to $O(n)$, and $O(\lg n)$ bits per set are spent on a directory that addresses the m separate structures.

Alanko et al. observe that the containment hierarchy need not be the cheapest possible. When many sets are subsets of both S_i and S_j, introducing a new set $S_i \cap S_j$ as an internal node could shorten the descriptions even though $S_i \cap S_j$ is not in $\mathcal{S}$. They also point out that $L(\mathcal{S})$ is computable in polynomial time, while more general measures that allow new sets may not be. The later construction uses the same tradeoff: synthetic sets can reduce description length, but the augmented structure must still support queries.

4.2.3 Insertion compressibility

Gagie, He, and Navarro [9] apply Proposition 4.2.2 to every set in $\mathcal{S}$. For each $S \in \mathcal{S}$, the reference is a strict subset of S in $\mathcal{S}$ of maximum cardinality, with ties broken arbitrarily. If none exists, the reference is the empty set.

Gagie, He, and Navarro [9] call the measure insertion compressibility because every set is described by the elements *inserted* into a smaller reference. The dual of containment entropy describes the elements *deleted* from a larger reference to obtain the set. The two are the same primitive applied in opposite directions.

[9]: Gagie et al. (2026), *Compressed Set Representations based on Set Difference*

Definition 4.2.4 (Insertion compressibility [9]) *Let $\mathcal{S}$ be a collection. For each $S \in \mathcal{S}$, let $p'(S)$ be a strict subset of S in $\mathcal{S}$ of maximum cardinality, with ties broken arbitrarily, or $\emptyset$ if no such subset exists. The insertion compressibility of $\mathcal{S}$ is*

$$I(\mathcal{S}) = \sum_{S \in \mathcal{S}} |S \setminus p'(S)| = \sum_{S \in \mathcal{S}} \big(|S| - |p'(S)|\big).$$

The description length implied by the superset primitive is $|S \setminus p'(S)| \lg u$ bits per set. Dropping the $\lg u$ factor, $I(\mathcal{S})$ counts the total number of inserted elements across all sets, which is bounded by the total size of $\mathcal{S}$.

Proposition 4.2.6 (Insertion bound [9]) *For every collection $\mathcal{S}$ with total size $n = \sum_i |S_i|$,*

$$I(\mathcal{S}) \le n.$$

Proof. By Definition 4.2.4, $I(\mathcal{S}) = \sum_{S \in \mathcal{S}} |S \setminus p'(S)|$. For each $S \in \mathcal{S}$, $|S \setminus p'(S)| \le |S|$, so $I(\mathcal{S}) \le \sum_{S \in \mathcal{S}} |S| = n$. □

Equality holds when $p'(S) = \emptyset$ for every S, which happens whenever no two sets of $\mathcal{S}$ are in a containment relation. At the opposite extreme, a deep chain $S_1 \subset S_2 \subset \cdots \subset S_m$ gives $I(\mathcal{S}) = |S_m|$, a quantity that can be far below n when the chain is long. A direct comparison with $H_{wc}(\mathcal{S})$ is unit-dependent: the insertion representation pays $I(\mathcal{S}) \lg u$ bits against $H_{wc}(\mathcal{S}) = \sum_i \lg \binom{u}{|S_i|}$ bits, and neither dominates the other across all collections. Thus $I(\mathcal{S})$ records containment chains that the per-set baseline ignores.

[9]: Gagie et al. (2026), *Compressed Set Representations based on Set Difference*

The dual construction that realises $I(\mathcal{S})$ structurally is the insertion graph of [9]. Its nodes are the sets of $\mathcal{S}$ together with $\emptyset$. For every nonempty $S \in \mathcal{S}$ there is a single outgoing edge to $p'(S)$ of weight $|S \setminus p'(S)|$. If $\emptyset \in \mathcal{S}$, the root also represents that set and has no outgoing edge. The edge weights sum to $I(\mathcal{S})$ by Definition 4.2.4. Since every edge points from a set to a strict subset, sizes decrease along directed edges. Hence no directed cycle exists, and every path eventually reaches $\emptyset$.

Chain expansion turns an insertion graph into a node-labelled ordinal tree. Weights are replaced by element labels, and the universe labels on the root-to-$v(S)$ path become exactly the set S.

The graph is not directly a node-labelled ordinal tree, because its edges carry numeric weights rather than element labels. Gagie et al. expand each weighted edge into a chain.

Definition 4.2.5 (Insertion tree [9]) *Let $\mathcal{S}$ be a collection, with $p'(S)$ as in Definition 4.2.4. The insertion tree of $\mathcal{S}$ is a rooted node-labelled ordinal tree. The root is $v(\emptyset)$ and carries a dummy label outside the universe alphabet. If $\emptyset \in \mathcal{S}$, this root also represents that set. For every nonempty $S \in \mathcal{S}$ there is a node $v(S)$, and the tree contains a downward chain of $|S \setminus p'(S)|$ new nodes whose first node is a child of $v(p'(S))$ and whose last node is $v(S)$. These chain nodes carry the distinct labels of $S \setminus p'(S)$ in arbitrary order. The tree may contain additional internal chain nodes not associated with any set of $\mathcal{S}$.*

Proposition 4.2.7 ([9]) *The insertion tree of $\mathcal{S}$ has $1 + I(\mathcal{S})$ nodes. Every nonroot label lies in $[1..u]$, and for every $S \in \mathcal{S}$ the set of universe labels on the path from $v(\emptyset)$ to $v(S)$ is exactly S.*

Proof. The root contributes one node and no universe label. Each nonempty $S \in \mathcal{S}$ contributes a chain of $|S \setminus p'(S)|$ new nodes. Summing over $\mathcal{S}$ adds $\sum_S |S \setminus p'(S)| = I(\mathcal{S})$ nodes, since the empty set contributes zero when it belongs to the collection. Every nonroot label lies in $U = [1..u]$ by construction. An induction on the insertion-graph path shows that the universe labels on the root-to-$v(S)$ path accumulate exactly the elements of $p'(S) \cup (S \setminus p'(S)) = S$. The induction base is the root, which represents $\emptyset$, and each step adds the labels of $S \setminus p'(S)$ to the path for $p'(S)$. □

Figure 4.1 shows the insertion graph for a nine-set collection and the insertion tree obtained from it by chain expansion. The edge weights on the left add up to $I(\mathcal{S}) = 20$, and the tree on the right has 21 nodes in total. The labels on the root-to-$v(S_i)$ path reproduce the content of S_i.

Apart from the dummy root label, the insertion tree carries labels from $[1..u]$ and no deletion marks. Compared to an encoding that lists for each S_i the elements of $S_i \setminus p'(S_i)$ as an unordered collection, the tree does not increase the number of element labels, and the total universe-label count is $I(\mathcal{S})$.

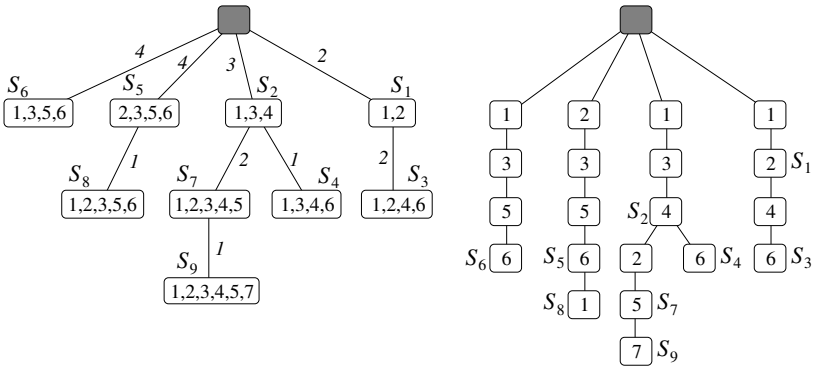

Figure 4.1: Reproduced from Gagie, He, and Navarro [9]. Left: an insertion graph for a collection $\mathcal{S} = \{S_1, \ldots, S_9\}$, with edge weights written as slanted values. The edge weights sum to $I(\mathcal{S}) = 20$. Right: the corresponding insertion tree obtained by expanding each edge $(v(p'(S)), v(S))$ of weight $w = |S \setminus p'(S)|$ into a chain of w labelled nodes. The tree has $1 + I(\mathcal{S}) = 21$ nodes. The annotation S_i marks the chain endpoint $v(S_i)$.

Proposition 4.2.8 (Insertion-graph construction [9]) *Given a collection $\mathcal{S}$ of m sets with total size n over a universe of size u, the insertion graph and its chain expansion into the insertion tree can be built in $O(n \lg u + mn)$ time on a word RAM with $\omega = \Omega(\lg n)$.*

Sketch. Sort the sets by increasing size in $O(m \lg m)$ time and the elements of every set in increasing order at total cost $O(n \lg u)$. Since $\mathcal{S}$ is a collection of distinct sets, either $m = 1$ and this term is zero, or $n \geq m - 1$, so $O(m \lg m)$ is absorbed by $O(mn)$. Insert the sets into a growing graph that starts from the single node $\emptyset$. For each nonempty set S, a DFS from $\emptyset$ identifies a maximum-cardinality strict subset $p'(S)$ among the already-inserted sets through merge-based containment checks, each costing time linear in the two sets involved. Summed over all insertions the traversal cost is $O(mn)$, and chain expansion of the graph into the insertion tree adds $O(I(\mathcal{S})) \subseteq O(n)$. □

4.2.4 Insertion-tree queries

The insertion tree of Definition 4.2.5 is a node-labelled ordinal tree of $1+I(\mathcal{S})$ nodes. Its root is $v(\emptyset)$ and carries a dummy label $u+1$, while every other label lies in $[1..u]$. Its defining property, recorded in Proposition 4.2.7 and illustrated in Figure 4.1, is that the universe labels on the path from $v(\emptyset)$ to $v(S)$ are exactly the elements of S. Every operation of Definition 1.2.2 on S thus becomes a question about the ancestors of $v(S)$.

Among this chapter's representations, the insertion tree is the one whose access operation reaches the sublogarithmic bound of Theorem 3.7.4. The indel construction of the next section pays $O(\lg u)$ for access for a different structural reason.

We build both indexes over the alphabet $[1..u+1]$. The α-operations index of Theorem 3.6.2 supports $\texttt{parent}_\alpha$ and rank_α for every label $\alpha \in [1..u+1]$. The path-queries index of Theorem 3.7.4 supports `path_count` and `path_select` for arbitrary label ranges. Each index stores a labelled ordinal tree of $|T|$ nodes in $O(|T|)$ words. Since $|T| = 1 + I(\mathcal{S})$ by Proposition 4.2.7, the overlay occupies $O(I(\mathcal{S}) + 1)$ words in total. The dummy label $u + 1$ is outside every query range $[1..x]$ and is larger than every universe label, so it does not change ranks or the first $|S|$ path selections.

Membership asks whether $x \in S$. Since the elements of S are exactly the universe labels on the root-to-$v(S)$ path, the test reduces to $\text{rank}_x(v(S)) > 0$, which the α-operations index answers in $O(\lg \lg_\omega u)$ time.

Rank asks for $|\{y \in S : y \leq x\}|$. Because the universe labels on the root-to-$v(S)$ path form S, this count equals the number of path nodes whose

label lies in $[1..x]$, which $\texttt{path\_count}(v(S), [1..x])$ returns in $O(\lg_\omega u)$ time.

Access asks for the i-th smallest element of S. This is the i-th smallest universe label on the root-to-$v(S)$ path, returned by $\texttt{path\_select}(v(S), i)$ in $O(\lg u / \lg\lg u)$ time.

Predecessor and successor follow the rank-then-access reduction recalled in Section 1.2, with the convention $\text{rank}(S, 0) = 0$. The predecessor of x in S is $\text{access}(S, \text{rank}(S, x))$ when $\text{rank}(S, x) \geq 1$ and undefined otherwise. The successor is $\text{access}(S, \text{rank}(S, x-1)+1)$ when that index does not exceed $|S|$. Each query performs one rank and one access, and the access term $O(\lg u / \lg\lg u)$ dominates.

Theorem 4.2.9 ([9]) *Let $\mathcal{S}$ be a collection of subsets of $[1..u]$ with total size n, and assume the word RAM model with word size $\omega = \Omega(\lg n)$. The insertion tree of $\mathcal{S}$ admits a representation in $O(I(\mathcal{S}) + 1)$ words of space supporting, for every $S \in \mathcal{S}$, the operations of Definition 1.2.2 within the following time bounds.*

- $\text{member}(S, x)$ *in time* $O(\lg\lg_\omega u)$.
- $\text{rank}(S, x)$ *in time* $O(\lg_\omega u)$.
- $\text{access}(S, i)$ *in time* $O(\lg u / \lg\lg u)$.
- $\text{predecessor}(S, x)$ *in time* $O(\lg u / \lg\lg u)$.
- $\text{successor}(S, x)$ *in time* $O(\lg u / \lg\lg u)$.

Chain-like containment is what the insertion tree compresses. When $S \subsetneq S'$ and $p'(S') = S$, the chain from $v(S)$ to $v(S')$ has length $|S' \setminus S|$, so it contributes little to $I(\mathcal{S})$ when the difference is small. A close incomparable neighbour cannot be chosen as the parent in Definition 4.2.4. Any saving for such a set must come from some other strict subset in the collection. If no strict subset exists for either set, both chains start at $v(\emptyset)$ and the two sets contribute their full sizes to $I(\mathcal{S})$. The next section develops a measure that pays for the elements on which two sets differ, regardless of inclusion.

4.3 Symmetric-difference compressibility

The subset and superset primitives of Subsection 4.2.1 both require a directional relationship between S and its reference. Neither applies when S and R are incomparable. The next primitive removes this direction requirement by recording insertions and deletions relative to the same neighbour.

Containment and insertion, developed in Section 4.2, both pick a directional reference. The containment hierarchy assigns each $S \in \mathcal{S}$ a parent $p(S) \supseteq S$ and pays $\lg \binom{|p(S)|}{|S|}$ bits. The insertion hierarchy assigns a parent $p'(S) \subseteq S$ and pays $|S \setminus p'(S)| \lg u$ bits. When two sets $A, B \in \mathcal{S}$ are incomparable, neither can serve as the other's reference under these primitives. Any saving for either set must come from a third comparable reference. If no such reference exists, the directional schemes describe the two sets separately and miss their pairwise closeness.

The elements that belong to exactly one of A and B give a non-directional cost. We write this set as $A \triangle B$, so $|A \triangle B| = |A \setminus B| + |B \setminus A|$. A small value means that one set can be obtained from the other by a short list of insertions and deletions, without requiring containment. Gagie, He, and Navarro [9] turn this cost into a compressibility measure $\Delta(\mathcal{S})$ and give a representation of size $O(\Delta(\mathcal{S}))$ words that supports the five operations of Definition 1.2.2 within logarithmic time. The task is to choose the edit

[9]: Gagie et al. (2026), *Compressed Set Representations based on Set Difference*

neighbours globally, turn the chosen edges into paths, and answer the operations from those paths without expanding the sets themselves.

4.3.1 Historical context

The idea of encoding a set as the symmetric difference with a close neighbour predates the compressibility measure by more than thirty years. The same pattern reappears in information retrieval, table compression, matrix multiplication, web-graph compression, and bioinformatics, usually under application-specific terminology and with only partial connection to earlier occurrences.

One early treatment in this lineage is by Bookstein and Klein [48], motivated by inverted indexes in information retrieval. For each term of a text collection, a posting bitvector records the segments in which the term occurs. Semantically correlated terms, such as the pair *Security* and *Council*, can produce posting bitvectors with heavy overlap. Bookstein and Klein form the complete weighted graph on the posting bitvectors, with edge weight equal to the Hamming distance, introduce an artificial zero vector as an extra node, and take a minimum spanning tree of the augmented graph. Every posting bitvector is then encoded as the XOR with its parent in the tree for non-root bitvectors, or directly for root bitvectors. They report that this preprocessing nearly doubles the compression saving on the Hebrew Bible corpus when combined with sparse-bitvector coders.

Bookstein and Klein [48] treat each bitvector as a set and identify the Hamming distance between two bitvectors with the cardinality of the symmetric difference between the corresponding sets.

[48]: Bookstein et al. (1991), *Compression of correlated bit-vectors*

The same broader idea of encoding each object via its difference from a neighbour reappears a decade later in two different communities. Buchsbaum, Caldwell, Church, Fowler, and Muthukrishnan [49] study compression of massive tables through differential dependencies, storing each derived column relative to a selected source column. In a parallel development, Björklund and Lingas [50] study fast Boolean matrix multiplication for highly clustered data. They express the running time in terms of the minimum spanning-tree weight of the Hamming graph on the rows or columns. Once the product for one row is known, the product for another row can be obtained by correcting only the entries on which the two rows differ. Exact MST construction was costly in their setting, and they used approximate MSTs instead.

The Hamming distance between two binary row vectors equals $|A \triangle B|$ when the rows are read as characteristic vectors over the column universe. The MST argument is Bookstein and Klein's. The application to matrix multiplication is Björklund and Lingas's.

[49]: Buchsbaum et al. (2000), *Engineering the compression of massive tables: an experimental approach*

[50]: Björklund et al. (2001), *Fast Boolean matrix multiplication for highly clustered data*

The web-graph compression literature gives a major application. Boldi and Vigna [5] observe that when the rows of the adjacency matrix of a web graph are sorted by URL, nearby rows often share many outgoing edges. The WebGraph framework stores each adjacency list against a chosen reference list drawn from a bounded backreference window of size W, recording by a copy list which successors of the reference to retain and by a list of extra nodes which successors to add. This backreference rule restricts candidate references to nearby URL-sorted rows and bounds reference-chain depth, so it is an engineering heuristic rather than the full minimum-spanning-tree construction. The construction still describes one list by differences from another, but it optimizes for extraction time as well as space instead of searching for a minimum spanning tree in the full Hamming graph.

Boldi and Vigna's [5] reference compression restricts candidate references to URL-sorted neighbours within a window W and bounds the reference count. This keeps decoding depth under control while exploiting similarity between nearby adjacency lists.

[5]: Boldi et al. (2004), *The webgraph framework I: compression techniques*

The technique reaches bioinformatics with Almodaresi, Pandey, Ferdman, Johnson, and Patro [8], who compress the colour sets of coloured de

[8]: Almodaresi et al. (2020), *An efficient, scalable, and exact representation of high-dimensional color information enabled using de Bruijn graph search*

Bruijn graphs in the Mantis tool. In that setting, each k-mer in a de Bruijn graph carries a colour set recording the genomes in which the k-mer occurs. Colour sets of neighbouring k-mers are often identical or similar. Almodaresi et al. use this locality by restricting the similarity graph to edges between colour sets of neighbouring vertices in the de Bruijn graph, taking a minimum spanning tree of the resulting graph, and encoding every colour set on that MST, so the representation stores small deltas between related colour classes. A more recent instance is due to Alves, Moustafa, Benkner, Francisco, Gansterer, and Russo [51], who apply the same idea to compress binary adjacency matrices for graph neural network acceleration. Gagie, He, and Navarro [9] take Alves et al. as their direct predecessor, and the two-rooted symdiff graph of Definition 4.3.1 is a variant of the one-rooted model used there.

[51]: Alves et al. (2025), *Accelerating graph neural networks using a novel computation-friendly matrix compression format*

The parameter $\Delta(\mathcal{S})$ is the minimum total number of insertions and deletions needed by a two-rooted symdiff graph.

[9]: Gagie et al. (2026), *Compressed Set Representations based on Set Difference*

For set collections, the same MST-on-Hamming graph has a precise cost. Gagie, He, and Navarro [9] define $\Delta(\mathcal{S})$ as the minimum weight of a two-rooted symdiff graph, prove $\Delta(\mathcal{S}) \leq I(\mathcal{S})$ for every collection, give a construction algorithm whose running time depends on the heaviest edge in the minimum-weight graph and on pairwise differences truncated at that threshold, and represent the collection in $O(\Delta(\mathcal{S}))$ words while supporting member, access, rank, predecessor, and successor. This gives the non-directional primitive a space measure comparable to the two directional measures used earlier in this chapter. A queryable representation also needs a decoding structure: every chosen neighbour relation must lead back to known anchors, and every edge must carry the edits that reconstruct the child set.

4.3.2 Symdiff graph

The root nodes labelled $\emptyset$ and U are canonical anchors. If either set belongs to $\mathcal{S}$, the corresponding root also represents that collection set.

The symmetric-difference primitive uses a directed graph whose edges are reconstruction steps. Its vertices are the sets of $\mathcal{S}$ together with two anchors, $\emptyset$ and U. An edge from S to a neighbour $p(S)$ says that S is stored by inserting and deleting elements relative to $p(S)$, and its weight is the number of edits.

Definition 4.3.1 (Symdiff graph [9]) *A symdiff graph on a collection $\mathcal{S}$ is a weighted directed graph on the node set $\mathcal{S} \cup \{\emptyset, U\}$, where $U = \bigcup_{S \in \mathcal{S}} S$ is the universe spanned by $\mathcal{S}$. The nodes $\emptyset$ and U are canonical roots, and if either set belongs to $\mathcal{S}$ the same root also represents that set. Every $S \in \mathcal{S} \setminus \{\emptyset, U\}$ has exactly one outgoing edge. Its target $p(S) \in \mathcal{S} \cup \{\emptyset, U\}$ is the neighbour of S, and the weight of the edge $(S, p(S))$ is $|S \triangle p(S)|$. The graph is required to satisfy the reachability condition that from every $S \in \mathcal{S}$ one of $\emptyset$ or U is reachable by following outgoing edges. The weight of the graph is the sum of the edge weights.*

Given a symdiff graph, each non-anchor $S \in \mathcal{S}$ is reconstructed from its neighbour $p(S)$ by inserting the elements of $S \setminus p(S)$ and deleting those of $p(S) \setminus S$. The total number of labels stored across the whole collection equals the weight of the graph. The reachability condition ensures that the decoding terminates at one of the two canonical anchors, $\emptyset$ or U, whose content is known a priori.

Definition 4.3.2 (Symdiff compressibility [9]) *The symmetric-difference compressibility of $\mathcal{S}$ is*

$$\Delta(\mathcal{S}) \;=\; \min\{\, W(G) : G \text{ is a symdiff graph on } \mathcal{S} \,\},$$

where $W(G)$ denotes the weight of G.

Because each non-anchor has one outgoing edge and every path reaches an anchor, a symdiff graph is a pair of directed trees oriented toward $\emptyset$ and U. Every set of $\mathcal{S}$ lies in one of these trees, possibly as an anchor when it equals $\emptyset$ or U. The choice is therefore a two-rooted spanning structure, and its minimum weight can be found by adding the zero-weight edge between $\emptyset$ and U and solving one spanning-tree problem.

Lemma 4.3.1 (Minimum-weight symdiff graph [9]) *Let K be the complete undirected graph on $\mathcal{S} \cup \{\emptyset, U\}$ with edge weights $w(A, B) = |A \triangle B|$ for $A, B \in \mathcal{S} \cup \{\emptyset, U\}$, except for the edge between $\emptyset$ and U, which is given weight 0. The weight of a minimum spanning tree of K equals $\Delta(\mathcal{S})$. The two trees achieving $\Delta(\mathcal{S})$ are obtained by removing the zero-weight edge between $\emptyset$ and U from an MST chosen to contain that edge.*

Proof. Choose an MST that contains the edge between $\emptyset$ and U. Such a tree exists. If an MST omits this edge, adding it creates a cycle, and replacing any edge on the resulting $\emptyset$-to-U path by the zero-weight edge preserves minimum weight. Removing this zero-weight edge disconnects the chosen MST into two trees, one containing $\emptyset$ and the other containing U. The two trees span $\mathcal{S} \cup \{\emptyset, U\}$ and their total weight equals that of the MST. Conversely, any pair of trees rooted at $\emptyset$ and U together spanning $\mathcal{S} \cup \{\emptyset, U\}$, plus the zero-weight edge between $\emptyset$ and U, gives a spanning tree of K of the same weight. The minimum over pairs of rooted trees therefore coincides with the MST weight, and the minimum-weight two-tree configuration is recovered by deleting the zero-weight edge. □

The insertion measure recalled in Definition 4.2.4 is $I(\mathcal{S}) = \sum_{S \in \mathcal{S}} |S \setminus p'(S)|$, where $p'(S)$ is the chosen maximum-cardinality strict subset of S, or $\emptyset$ if no such set exists. Keeping the same parent map, adding the unused anchor U, and reading each insertion edge as a symmetric-difference edge gives a symdiff graph that uses only the tree rooted at $\emptyset$. Since $p'(S) \subseteq S$, each retained edge has weight $|S \triangle p'(S)| = |S \setminus p'(S)|$.

Proposition 4.3.2 (Symdiff-insertion inequality [9]) *For every collection $\mathcal{S}$,*

$$\Delta(\mathcal{S}) \;\le\; I(\mathcal{S}).$$

Proof. The insertion graph of Subsection 4.2.3 has one outgoing edge from each nonempty $S \in \mathcal{S}$ to its largest strict subset $p'(S) \in \mathcal{S} \cup \{\emptyset\}$. Every such S reaches $\emptyset$ along this chain. Add the canonical U root as an isolated root, and if $U \in \mathcal{S}$ let that root represent U instead of using its insertion edge. The resulting graph satisfies Definition 4.3.1 and has weight at most $I(\mathcal{S})$. Because $p'(S) \subseteq S$ on every retained insertion edge, no deletions contribute and $|S \triangle p'(S)| = |S \setminus p'(S)|$. Therefore $\Delta(\mathcal{S}) \le I(\mathcal{S})$. □

For two large incomparable sets that differ in only a few elements, the two measures can differ sharply. An insertion graph has no containment

edge between them, so each set must be paid from a smaller contained set or from $\emptyset$. A symdiff graph can store one of the two through a small symmetric-difference edge, and the characterisation of Lemma 4.3.1 selects such edges globally rather than inside a containment chain.

4.3.3 Adapted Prim construction

The same MST characterisation makes $\Delta(\mathcal{S})$ computable. Once the edge weights of the augmented complete graph are known, an ordinary minimum-spanning-tree algorithm gives the two rooted components. A direct exact construction sorts all sets and computes all pairwise weights. The adapted construction of Gagie, He, and Navarro [9] instead discovers each weight only as far as Prim's algorithm needs it. After an $O(n \lg u)$ preprocessing of the sorted sets, their suffix-tree iterator exposes the elements of $S \triangle S'$ one at a time. Prim is run with unknown edge weights, revealing all edges below the current threshold before it accepts a heavier edge.

[9]: Gagie et al. (2026), *Compressed Set Representations based on Set Difference*

Theorem 4.3.3 (Construction of $\Delta(\mathcal{S})$ [9]) *Let $\mathcal{S}$ be a collection of m sets over a universe of size u, with total size n. Let ℓ be the maximum edge weight in a minimum-weight symdiff graph of $\mathcal{S}$. A minimum-weight symdiff graph, and hence $\Delta(\mathcal{S})$, can be computed in time*

$$O\left(n \lg u + \sum_{S,S' \in \mathcal{S}} \min(\ell, |S \triangle S'|)\right) \subseteq O\left(n \lg u + \min(m^2 \ell, mn)\right).$$

Once the minimum-weight graph is fixed, the query structure no longer uses the state of Prim's algorithm. It uses only the two rooted components and the edit labels on their edges.

4.3.4 Indel trees

Where the insertion tree of Definition 4.2.5 carries only positive labels drawn from $U = [1..u]$, the indel tree carries labels from $[-u..-1] \cup [1..u]$. Doubling the alphabet is the price of allowing deletions.

Inside each rooted component, the directed edge from S to its neighbour $p(S)$ is read downward from $v(p(S))$ to $v(S)$. Its weight decomposes as $|S \triangle p(S)| = |S \setminus p(S)| + |p(S) \setminus S|$ insertions and deletions. Unlike the insertion tree of Definition 4.2.5, whose nonroot nodes carry only insertions, the indel tree of [9] carries both kinds of edit. Translating a weighted edge into a chain of labelled nodes therefore requires two label classes, one for insertions and one for deletions.

Definition 4.3.3 (Indel trees [9]) *Let $\mathcal{S}$ be a collection, with $p(S)$ the neighbour of S in a minimum-weight symdiff graph of $\mathcal{S}$. The indel trees of $\mathcal{S}$ are two rooted labelled ordinal trees. The root of one tree is $v(\emptyset)$, and the root of the other is $v(U)$. If $\emptyset$ or U belongs to $\mathcal{S}$, the corresponding root also represents that collection set. For every $S \in \mathcal{S} \setminus \{\emptyset, U\}$ there is a node $v(S)$, lying in the same tree as $v(p(S))$. The connection from $v(p(S))$ down to $v(S)$ is replaced by a chain of $|S \triangle p(S)|$ new nonroot nodes, with the last new node equal to $v(S)$. Each new node carries a distinct signed element of the insertion set $S \setminus p(S)$ or the deletion set $p(S) \setminus S$, in arbitrary order. Labels from the insertion set are written $+x$ for $x \in S \setminus p(S)$, and labels from the deletion set are written $-y$ for $y \in p(S) \setminus S$. The label alphabet is $[-u..-1] \cup [1..u]$, of size $2u$. Gagie et al. write this as $[-u..u]$, but the label 0 is unused since $0 \notin U$.*

Proposition 4.3.4 (Indel-tree structure [9]) *The two indel trees of $\mathcal{S}$ together contain $\Delta(\mathcal{S}) + 2$ nodes. For every $S \in \mathcal{S}$, the content of S is recovered from the root of its tree by walking the path from the root to $v(S)$ and applying each label in order. An insertion label $+x$ adds x to the running set, and a deletion label $-y$ removes y.*

Proof. The two roots contribute two nodes. Each non-anchor $S \in \mathcal{S}$ contributes $|S \triangle p(S)|$ new nonroot nodes, with the last one equal to $v(S)$. Summing over the symdiff-graph edges adds $\sum_{S \in \mathcal{S} \setminus \{\emptyset, U\}} |S \triangle p(S)| = \Delta(\mathcal{S})$ nodes. The decoding semantics follows by induction on depth. At the root the running set is $\emptyset$ or U as given. An insertion label $+x$ increments the running set by x, and a deletion label $-y$ decrements it by y. By Definition 4.3.1, the labels on the chain from $v(p(S))$ to $v(S)$ realise exactly the symmetric difference between $p(S)$ and S, so the running set at $v(S)$ equals $p(S) \cup (S \setminus p(S)) \setminus (p(S) \setminus S) = S$. □

Figure 4.2 shows the minimum-weight symdiff graph of a nine-set collection, together with the two indel trees obtained by chain expansion. The MST edges have total weight $\Delta(\mathcal{S}) = 13$, and the two trees together have 15 nodes.

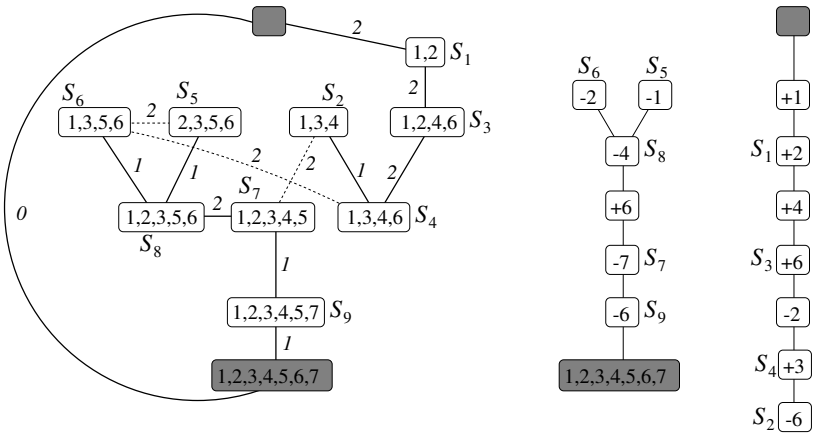

Figure 4.2: Reproduced from Gagie, He, and Navarro [9]. Left, a minimum-weight symdiff graph on $\mathcal{S} = \{S_1, \ldots, S_9\}$ with $\Delta(\mathcal{S}) = 13$. Dashed edges are full-graph edges of weight at most $\ell = 2$ that are considered by the construction but not selected by the MST. Right, the two indel trees rooted at U (drawn upside down) and at $\emptyset$. Chain labels $+x$ are insertions, and chain labels $-x$ are deletions relative to the parent.

In the figure, each set has become a node reached by a signed edit history. A membership or rank query must read that history without reconstructing the set, and access must select among the elements that survive all earlier insertions and deletions.

4.3.5 Indel-tree queries

A query on S is now a query on the signed root-to-$v(S)$ history. The insertion tree of Definition 4.2.5 had only positive labels, so the labels on a path were exactly the elements of the set. Here a later $-x$ can cancel an earlier $+x$, and in the tree rooted at U every element starts present until a deletion removes it. The indexes must therefore support tests for the last edit of one element and signed range counts on the same paths.

The label alphabet doubles from $[1..u]$ to $[-u..-1] \cup [1..u]$. Since $\lg(2u) = \lg u + 1$, the asymptotic path-query bounds keep the same logarithmic parameters.

We overlay the two indexes of Section 3.5 on each of the two trees. The α-operations index of Theorem 3.6.2, instantiated at alphabet $[-u..-1] \cup [1..u]$, supports $\texttt{parent}_\alpha$, rank_α, and select_α for every signed label α. The path-queries index of Theorem 3.7.4, instantiated at the same alphabet, supports `path_count` for arbitrary label ranges and `path_select` on the

root-to-node path. Each index stores a labeled ordinal tree of $|T|$ nodes in $O(|T|)$ words, and $|T^{\emptyset}| + |T^{U}| = \Delta(\mathcal{S}) + 2$ by Proposition 4.3.4, so the overlay occupies $O(\Delta(\mathcal{S}))$ words in total. The alphabet size $2u$ enters the query bounds only through $\lg(2u) = \lg u + 1$, which leaves every asymptotic in $\lg u$ and $\lg \lg_{\omega} u$ unchanged.

Membership on (S, x) asks which of the two signed labels $+x$ and $-x$ appears last on the root-to-$v(S)$ path. Let $r^{+} = \text{rank}_{+x}(v(S))$ and $r^{-} = \text{rank}_{-x}(v(S))$. If $r^{+} > 0$, let $u^{+} = \text{select}_{+x}(v(S), r^{+})$, and otherwise set $u^{+} = \perp$. Define u^{-} from r^{-} in the same way. If both nodes exist, the last edit that affected x is the deeper of the two, so $x \in S$ iff $\text{depth}(u^{+}) > \text{depth}(u^{-})$. If only u^{+} exists, x was inserted and never removed, so $x \in S$. If only u^{-} exists, x was removed and never reinstated, so $x \notin S$. If neither exists, no label for x appears on the path, so $x \in S$ iff $v(S)$ descends from $v(U)$, since the elements of U are implicitly present at the root of that tree. The constant number of α-operations and the depth comparison run in $O(\lg \lg_{\omega} u)$ time.

Rank on (S, x) is a difference of two path counts. The ancestors of $v(S)$ with label in $[1..x]$ record the elements of $[1..x]$ inserted along the path, and the ancestors with label in $[-x.. -1]$ record those deleted. The rank is the excess of insertions over deletions. When $v(S)$ descends from $v(U)$, the root holds all of $[1..x]$ implicitly, so the excess must be added to x. Both counts are single calls to `path_count` of Theorem 3.7.4, and the total cost is $O(\lg_{\omega} u)$.

Access is the one operation where the reductions used for the insertion tree break down. Path selection on the signed alphabet returns the i-th smallest signed label on the path, which may be a deletion mark and need not be a surviving element of S. A separate algorithm, developed in the coordinated-descent paragraph below, achieves $\text{access}(S, i)$ in $O(\lg u)$ time by descending two parallel hierarchies in lockstep.

Predecessor and successor follow the rank-then-access reduction recalled in Section 1.2. Each performs one rank and one access, and the access term $O(\lg u)$ dominates.

Theorem 4.3.5 (Indel tree representation [9]) *Let $\mathcal{S}$ be a collection of subsets of $[1..u]$ with total size n, and assume the word RAM model with word size $\omega = \Theta(\lg n)$ and with universe values fitting in one word. The two indel trees of $\mathcal{S}$ admit a joint representation in $O(\Delta(\mathcal{S}))$ words of space supporting, for every $S \in \mathcal{S}$, the operations of Definition 1.2.2 within the following time bounds.*

- $\text{member}(S, x)$ *in time* $O(\lg \lg_{\omega} u)$.
- $\text{rank}(S, x)$ *in time* $O(\lg_{\omega} u)$.
- $\text{access}(S, i)$ *in time* $O(\lg u)$.
- $\text{predecessor}(S, x)$ *in time* $O(\lg u)$.
- $\text{successor}(S, x)$ *in time* $O(\lg u)$.

Coordinated descent

Access on (S, i) asks for the i-th smallest element x of S. A direct binary search on $[1..u]$ driven by rank queries on $v(S)$ would probe $\lceil \lg u \rceil$ candidate values and spend $O(\lg_{\omega} u)$ per probe, for a total of $O(\lg u \cdot \lg_{\omega} u)$. The algorithm of this subsection removes the extra $\lg_{\omega} u$

factor by replacing external rank probes with an internal descent on hierarchical partitions built by tree extraction, so that every level takes $O(1)$ time.

For each indel tree $\mathcal{T}$ rooted at $v(\emptyset)$ or $v(U)$, form two sign-specific extracted trees by the binary labels $\{+, -\}$. The counted nodes of $\mathcal{T}^+$ are the insertion labels, and the counted nodes of $\mathcal{T}^-$ are the deletion labels. Both trees are obtained through the tree-extraction operation of Definition 3.5.1 with the associated image map $f_{\mathcal{T}}$. Within each of $\mathcal{T}^+$ and $\mathcal{T}^-$, apply the hierarchical binary universe partition of Subsection 3.7.1 on $[1..u]$, halving the label range at every level, producing a family of binary-labeled trees $\mathcal{T}^+_{a,b}$ and $\mathcal{T}^-_{a,b}$ indexed by the current range. Figure 4.3 diagrams the extraction for one of the two hierarchies, and $\mathcal{T}^+$ is the parallel instance built on insertion marks. Each extracted tree has a binary marker alphabet, so one level costs linear bits in the number of retained nodes at that level. The retained sets at a fixed level partition the labels of $\mathcal{T}^{\pm}$, and the hierarchy therefore costs $O(\Delta(\mathcal{S}) \lg u)$ bits of additional state, which is $O(\Delta(\mathcal{S}))$ words.

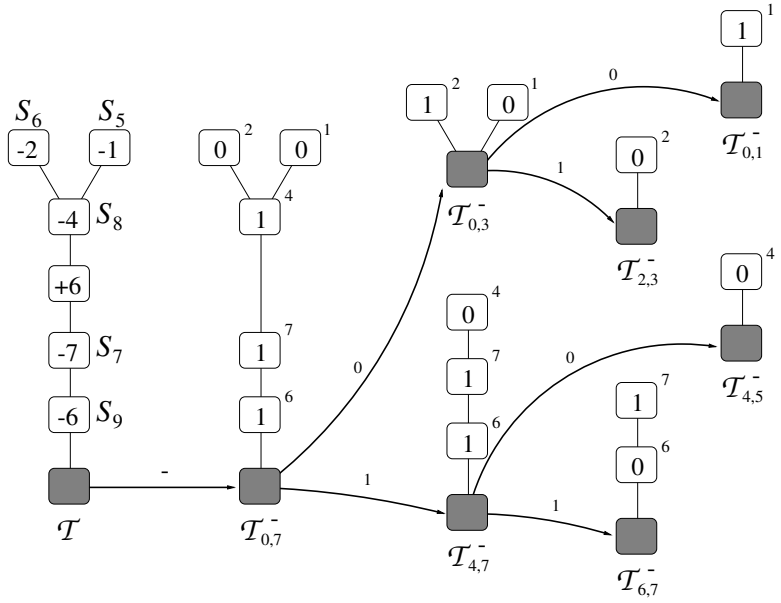

Figure 4.3: Hierarchical extraction of one indel-tree hierarchy, reproduced from Gagie, He, and Navarro [9]. The diagram shows the source example $\mathcal{T}^- = \mathcal{T}^-_{0,7}$ obtained from the indel tree rooted at $v(U)$ of Figure 4.2 (drawn upside down). The source figure pads the example to the range $[0..7]$, while the construction in this thesis uses $[1..u]$. At each intermediate range $[a..b]$, the 0/1-labelled tree $\mathcal{T}^-_{a,b}$ extracts to $\mathcal{T}^-_{a,m}$ on label 0 and to $\mathcal{T}^-_{m+1,b}$ on label 1, following the bold rightward arrows. The same construction yields $\mathcal{T}^+$ on insertion marks. The coordinated descent of this subsection tracks images v^+ and v^- in two parallel instances of this structure.

At a range $[a..b]$, the descent needs the number of elements of S that lie in the left half $[a..m]$. The two rank_0 calls in Algorithm 1 count left-half insertion labels and left-half deletion labels in $O(1)$ time by Theorem 3.6.2, since the alphabet of $\mathcal{T}^{\pm}_{a,b}$ is $\{0, 1\}$. Their difference $s^+ - s^-$ is the left-half count when $\mathcal{T}$ is rooted at $v(\emptyset)$. The additive correction $m - a + 1$ when $\mathcal{T}$ is rooted at $v(U)$ accounts for the elements of $[a..m]$ that start as implicitly present at the root and are removed only by deletions along the path.

Correctness follows from the loop invariant that, at every level, i is the rank of x among the elements of S in $[a..b]$. At initialisation the invariant holds because x is the i-th smallest element of S and $[a..b] = [1..u]$. The branch rule preserves it by a per-value invariant. If $\mathcal{T}$ is rooted at $v(\emptyset)$, then for every value y, the quantity $\#(+y) - \#(-y)$ on the root-to-v path is 1 exactly when y is present at v and 0 otherwise. If $\mathcal{T}$ is rooted at $v(U)$, then $1 + \#(+y) - \#(-y)$ is the same indicator. Summing over $y \in [a..m]$ gives the left-half count used in the branch test. The values s^+ and s^-

Algorithm 1: Coordinated descent for access(S, i) on indel trees [9]

Input: Indel tree $\mathcal{T}$ rooted at $v(\emptyset)$ or $v(U)$, node $v = v(S)$, rank $i \in [1..|S|]$.
Output: The i-th smallest element of S.

$a \leftarrow 1$;
$b \leftarrow u$;
$v^+ \leftarrow f_{\mathcal{T}}(v, +) \in \mathcal{T}^+_{1,u}$;
$v^- \leftarrow f_{\mathcal{T}}(v, -) \in \mathcal{T}^-_{1,u}$;
while $a < b$ **do**
  $m \leftarrow \lfloor (a + b)/2 \rfloor$;
  $s^+ \leftarrow \mathrm{rank}_0(v^+)$ in $\mathcal{T}^+_{a,b}$;
  $s^- \leftarrow \mathrm{rank}_0(v^-)$ in $\mathcal{T}^-_{a,b}$;
  $c \leftarrow s^+ - s^-$;
  if $\mathcal{T}$ *is rooted at* $v(U)$ **then**
    $c \leftarrow c + (m - a + 1)$;
  $a_0 \leftarrow a$;
  $b_0 \leftarrow b$;
  if $i \le c$ **then**
    $v^+ \leftarrow f^+_{a_0,b_0}(v^+, 0)$;
    $v^- \leftarrow f^-_{a_0,b_0}(v^-, 0)$;
    $b \leftarrow m$;
  else
    $v^+ \leftarrow f^+_{a_0,b_0}(v^+, 1)$;
    $v^- \leftarrow f^-_{a_0,b_0}(v^-, 1)$;
    $a \leftarrow m + 1$;
    $i \leftarrow i - c$;
return a;

realise these sums as instances of the depth identity of Lemma 3.5.3 applied to $\mathcal{T}^{\pm}_{a,b}$, where $\mathrm{rank}_0(v^{\pm})$ counts retained left-half labels on the path.

Each level performs two $O(1)$ rank queries, two $O(1)$ tree-extraction remappings, and one arithmetic comparison, for $O(1)$ time overall. The number of levels is $\lceil \lg u \rceil$, so the algorithm runs in $O(\lg u)$ time.

Gagie, He, and Navarro [9] obtain $O(\lg u)$ access time for the signed extension and leave the $O(\lg_\omega u)$ analogue of unsigned path selection as future work.

The $O(\lg u)$ bound comes from the signed access problem itself. Membership and rank ask for last edits or signed counts on one path, but access must choose the i-th surviving value while insertions and deletions are kept in separate hierarchies. The descent therefore maintains two images, $v^+ \in \mathcal{T}^+_{a,b}$ and $v^- \in \mathcal{T}^-_{a,b}$, and each level branches using the difference of the two left-half counts.

The insertion and indel structures show that chosen inter-set relations can be made queryable within their own space measures. They do not identify the information-theoretic optimum for a collection. The next chapter starts from membership patterns, derives a lower bound, and then builds a containment hierarchy by adding synthetic union nodes.

5 Set-Union Matching

The representations of Chapter 4 show how to use an inter-set relation once that relation has been fixed. A containment edge records the elements lost from a parent, and a relation based on small changes records the elements that differ between two close sets. Query support then comes from reading these labels along paths. The remaining problem is not how to encode a fixed relation, but how to decide which relations should be introduced. To make that decision independent of a particular hierarchy, we first examine the information contained in the collection itself.

This information is determined at the level of single universe elements. For each $x \in U$, the collection determines the set of indices i such that $x \in S_i$. Elements with the same index set are indistinguishable with respect to all sets in the collection, so the universe decomposes into Venn regions. Once the occurring regions and their sizes are fixed, a collection is determined by assigning one region label to each labelled element of U. Counting those assignments gives a lower bound on any representation that receives the region data as side information.

This counting view abstracts from the eventual representation, and consequently it has no query mechanism. It treats the collection as one global string over Venn-region labels. Membership, rank, and access should not require reconstructing the entire sequence of region labels before the query can proceed. To obtain the locality of the previous hierarchies, the same information must be reorganised as parent-child changes that can be read along paths.

Containment already has the local form needed by the path-query structures. If $Q \subseteq P$, the edge from P to Q stores the deleted elements $P \setminus Q$, and those structures already know how to count and select the edge labels. When two sets overlap but neither contains the other, no containment edge can place their shared elements above both sets. Adding a set that contains both would create two containment edges, and the smallest such set is their union. The two edge labels then store only the elements that must be deleted to recover each child.

Restricting the added reference to the union keeps the choice local. Before such local choices can be compared, the cost they reduce must be expressed without reference to the hierarchy that will later realise it. A newly created parent may itself overlap with another current set, so the same cost comparison can be repeated at the next level. In this way, the elementwise count guides the selection of local parents, while the final object remains a hierarchy of containment edges. Allowing arbitrary added references would broaden the search beyond pairwise unions. Appendix A studies that larger model, while this chapter keeps to binary unions because their cost can be evaluated locally and the resulting hierarchy can be stored with the deletion-tree query structure.

5.1 Information theoretic bound

For $\mathcal{S} = \{S_1 = \{a, b\}, S_2 = \{b, c\}\}$ over $U = \{a, b, c\}$, the partition has three Venn regions, $\{a\}$ in S_1 only, $\{b\}$ in both S_1 and S_2, and $\{c\}$ in S_2 only.

Comparing possible parents requires a reference cost that is defined before any parent is chosen. The collection determines such a cost through the way elements are distributed among the input sets. We first group elements that belong to exactly the same input sets, then count how many assignments of labelled universe elements to those groups remain possible. This count is conditional: it ignores the cost of naming the groups and their sizes, while a complete data structure must also store that side information and the auxiliary indices used by queries.

5.1.1 Membership patterns

The count starts from the label carried by one universe element: the subset of input-set indices whose sets contain it. Equal labels place elements in the same Venn region, and different labels place them in different regions. Thus the label records membership, while the region is the class of elements carrying that label.

Definition 5.1.1 (Membership pattern) *The membership pattern of an element $x \in U$ relative to $\mathcal{S} = \{S_1, \ldots, S_m\}$ is the set*

$$\sigma(x) \;=\; \{\, i \in [m] : x \in S_i \,\}.$$

The convention $U = \bigcup_i S_i$ fixed in Section 4.1 means that no element has the empty membership pattern. We therefore only need nonempty patterns $\tau \subseteq [m]$. For such a pattern, the associated Venn region is the set of all elements with membership pattern τ. Since every set generated from $S_1, \ldots, S_m$ either contains all of that region or none of it, the region is an indivisible part of the generated Boolean algebra. The standard algebraic term for such a nonzero element is atom.

Definition 5.1.2 (Atom [52]) *Let $\mathcal{B}$ be a Boolean algebra with least element 0. An atom of $\mathcal{B}$ is an element $a \neq 0$ such that the only elements $b \in \mathcal{B}$ satisfying $b \leq a$ are $b = 0$ and $b = a$.*

For $\mathcal{S} = \{S_1, \ldots, S_m\} \subseteq 2^U$, let $\mathcal{B}(\mathcal{S})$ be the Boolean algebra of subsets of U generated by $S_1, \ldots, S_m$. For every nonempty $\tau \subseteq [m]$, set

$$A_\tau \;=\; \{\, x \in U : \sigma(x) = \tau \,\}.$$

If $A_\tau \neq \emptyset$, we call A_τ the atom of pattern τ.

The specialised sets A_τ are atoms of $\mathcal{B}(\mathcal{S})$ because the sets $S_1, \ldots, S_m$ act on each nonempty A_τ all at once. For each index i, either $A_\tau \subseteq S_i$ or $A_\tau \cap S_i = \emptyset$, according to whether $i \in \tau$. The same all or nothing behaviour holds for every set obtained from $S_1, \ldots, S_m$ by unions, intersections, and complements. Thus no set in $\mathcal{B}(\mathcal{S})$ can split a nonempty A_τ into two nonempty parts.

In the running example, $\sigma(a) = \{1\}$, $\sigma(b) = \{1, 2\}$, and $\sigma(c) = \{2\}$. The atoms are $A_{\{1\}} = \{a\}$, $A_{\{1,2\}} = \{b\}$, and $A_{\{2\}} = \{c\}$, all of size one.

The nonempty sets A_τ form a partition of U, and their cardinalities sum to u. There are at most $2^m - 1$ of them, one for each nonempty subset of $[m]$. If $u = 0$, all multinomial coefficients are one and every bit bound is zero. We assume $u > 0$ from now on.

Let $R(\mathcal{S}) = \{\tau \subseteq [m] : \tau \neq \emptyset,\ A_\tau \neq \emptyset\}$ denote the set of realised patterns and write $k = |R(\mathcal{S})|$. This number ranges from 1, when all sets coincide, to $2^m - 1$, when every nonempty Venn region is realised. Enumerate $R(\mathcal{S}) = \{\tau_1, \dots, \tau_k\}$ and set $c_{\tau_j} = |A_{\tau_j}|$. Fixing $R(\mathcal{S})$ and the vector $(c_{\tau_j})_{j=1}^k$ still leaves the labelled elements of U to place into the pattern classes. Every labelling of the u elements by patterns in $R(\mathcal{S})$ with prescribed counts $(c_\tau)_{\tau \in R(\mathcal{S})}$ produces one collection, and every collection with these parameters gives the labelling $x \mapsto \sigma(x)$. If the decoder receives these parameters outside the bit count, the remaining class to distinguish is exactly the fixed-count class counted next.

5.1.2 Atom bound

Once the realised patterns and their counts are fixed, only the positions of the patterns on the labelled universe remain. A collection with these parameters is a labelling of the u elements of U by patterns in $R(\mathcal{S})$, using each pattern τ exactly c_τ times. The number of such labellings is the multinomial coefficient

$$\binom{u}{(c_\tau)_{\tau \in R(\mathcal{S})}} = \frac{u!}{\prod_{\tau \in R(\mathcal{S})} c_\tau!},$$

obtained by ordering the u elements in $u!$ ways and dividing by the $\prod_\tau c_\tau!$ permutations within each pattern class that produce the same labelling.

Proposition 5.1.1 (Atom bound) *Let $R(\mathcal{S}) = \{\tau_1, \dots, \tau_k\}$ be the realised patterns of $\mathcal{S}$ and $c_{\tau_j} = |A_{\tau_j}|$ the atom counts. Set*

$$L^*(\mathcal{S}) = \lg \binom{u}{c_{\tau_1}, \dots, c_{\tau_k}}.$$

Given the side information $(R(\mathcal{S}), (c_{\tau_j})_{j=1}^k)$ outside the bit count, every fixed length representation that distinguishes all indexed collections sharing these parameters uses at least $\lceil L^*(\mathcal{S}) \rceil$ bits. The longest codeword of every prefix code for the same class has length at least $\lceil L^*(\mathcal{S}) \rceil$ bits.

Proof. Let $\mathcal{C}(R, (c_\tau))$ denote the class of indexed collections

$$\mathcal{S}' = \{S'_1, \dots, S'_m\}$$

over U whose realised pattern set is R and whose atom counts match (c_τ). The map sending $\mathcal{S}' \mapsto \ell_{\mathcal{S}'}$ with $\ell_{\mathcal{S}'}(x) = \{i \in [m] : x \in S'_i\}$ takes $\mathcal{C}(R, (c_\tau))$ into the set of pattern labellings $\ell : U \to R$ in which τ occurs exactly c_τ times. Its inverse sends a labelling ℓ back to the collection with $S'_i = \{x \in U : i \in \ell(x)\}$, whose realised pattern set is the image of ℓ (which equals R since every $\tau \in R$ has $c_\tau \geq 1$) and whose atom counts are (c_τ). The two maps are inverses, hence $|\mathcal{C}(R, (c_\tau))| = \binom{u}{(c_\tau)}$.

If a fixed length representation uses b bits, it has at most 2^b distinct codewords. Since it must distinguish every member of $\mathcal{C}(R, (c_\tau))$, we have $2^b \geq |\mathcal{C}(R, (c_\tau))|$, hence $b \geq \lceil \lg |\mathcal{C}(R, (c_\tau))| \rceil = \lceil L^*(\mathcal{S}) \rceil$. If a prefix code has longest codeword length L, its code tree has at most 2^L leaves, so the same counting argument gives $L \geq \lceil L^*(\mathcal{S}) \rceil$. □

Enumerative coding of fixed count classes is treated in Cover and Thomas [53, Ch. 11].

Proposition 5.1.1 gives a conditional lower bound. It charges the assignment of labelled universe elements to fixed pattern classes, and it leaves the cost of describing those classes to the complete representation.

5.1.3 Coarser bounds

The bound $L^*(\mathcal{S})$ assumes the most detailed side information used in this section: the decoder receives the realised patterns and their counts outside the bit count. If this side information is reduced, the representation must distinguish a larger class and the lower bound becomes coarser.

The atom counts (c_τ) are strictly finer side information than the row sizes $(|S_i|)$. The row sizes follow from the atom counts via $|S_i| = \sum_{\tau \ni i} c_\tau$, but multiple atom count configurations can yield the same row sizes.

The first weakening keeps row sizes and forgets the rest of the pattern data. With only row sizes available, the independent row-size count is $H_{wc}(\mathcal{S}) = \sum_i \lg \binom{u}{|S_i|}$, as in Definition 1.2.3. Once the sets A_τ have prescribed sizes, every row size is forced by $|S_i| = \sum_{\tau \ni i} c_\tau$. A code that knows only the row sizes sees a larger class, because it also has to distinguish row tuples whose pattern counts differ from those of $\mathcal{S}$.

Proposition 5.1.2 *For every collection $\mathcal{S}$ with row sizes $(|S_i|)_{i=1}^m$,*

$$L^*(\mathcal{S}) \;\le\; H_{wc}(\mathcal{S}).$$

Proof. Let R be the realised pattern set and (c_τ) the atom counts of $\mathcal{S}$. Consider one of the collections counted by the multinomial defining $L^*(\mathcal{S})$, and write $(A'_\tau)_{\tau \in R}$ for its sets of elements with pattern τ. It determines an m-tuple of rows $(S'_1, \dots, S'_m)$ by

$$S'_i \;=\; \bigcup_{\tau \ni i} A'_\tau.$$

The row sizes satisfy $|S'_i| = \sum_{\tau \ni i} c_\tau = |S_i|$, so the tuple belongs to the class counted by $H_{wc}(\mathcal{S})$.

The map is injective. Given the m-tuple $(S'_1, \dots, S'_m)$, the set A'_τ is recovered as $\bigcap_{i \in \tau} S'_i \cap \bigcap_{i \notin \tau} (U \setminus S'_i)$. Thus the partition $(A'_\tau)_{\tau \in R}$ is determined by the tuple, and distinct partitions map to distinct tuples.

Comparing the cardinality of the domain with the cardinality of the target gives

$$\binom{u}{(c_\tau)} \;\le\; \prod_{i=1}^{m} \binom{u}{|S_i|},$$

and taking logarithms to base 2 yields $L^*(\mathcal{S}) \le H_{wc}(\mathcal{S})$. □

The second weakening forgets the row sizes as well. If k pattern labels occur, then the sequences counted by $L^*(\mathcal{S})$ are exactly the strings over this alphabet with prescribed frequencies. They form a subset of all k^u strings over the same alphabet. Replacing k by its largest possible value $2^m - 1$ gives a bound that depends only on u and m.

Proposition 5.1.3 *If $\mathcal{S}$ realises k patterns, with $1 \le k \le 2^m - 1$, then*

$$L^*(\mathcal{S}) \;\le\; u \lg k \;\le\; u \lg(2^m - 1).$$

Proof. The multinomial coefficient satisfies

$$\binom{u}{(c_\tau)} = \frac{u!}{\prod_\tau c_\tau!} \leq k^u$$

since the left side counts pattern sequences of length u over an alphabet of size k with prescribed frequencies, and k^u counts all pattern sequences of length u over the same alphabet. Taking logarithms gives $L^*(\mathcal{S}) \leq u \lg k$. Under the convention of Section 4.1, the empty pattern is excluded, so $k \leq 2^m - 1$ and the second inequality follows. □

These inequalities place $L^*(\mathcal{S})$ below the coarser counts obtained from weaker side information. They do not yet say whether the exact fixed-pattern count can be reached as an encoding length.

5.1.4 Achievability and its limits

$L^*(\mathcal{S})$ is the logarithm of a finite class of pattern sequences, and enumerating that class gives a fixed length code. Once the decoder knows the realised patterns and their counts, the encoder lists the membership patterns in the order of the universe and stores the enumerative index of that sequence. This reaches the lower bound up to the ceiling. The statement concerns only the codeword length, and it excludes the cost of storing the side information and the indices used by queries.

Proposition 5.1.4 (Achievability of L^*) *For every collection $\mathcal{S}$ under the convention $U = \bigcup_i S_i$, there is an encoding of $\mathcal{S}$ of length exactly $\lceil L^*(\mathcal{S}) \rceil$ bits given the side information $(R(\mathcal{S}), (c_\tau)_{\tau \in R(\mathcal{S})})$.*

Proof. Order the universe as $U = \{x_1, \ldots, x_u\}$ and form the pattern sequence $\pi(\mathcal{S}) = (\sigma(x_1), \ldots, \sigma(x_u))$ over the alphabet $R(\mathcal{S})$. By the bijection in the proof of Proposition 5.1.1, the map $\mathcal{S} \mapsto \pi(\mathcal{S})$ is a bijection between indexed collections with parameters $(R, (c_\tau))$ and sequences of length u over $R(\mathcal{S})$ with prescribed frequencies (c_τ). Enumerative coding assigns a distinct codeword of length $\lceil \lg \binom{u}{(c_\tau)} \rceil = \lceil L^*(\mathcal{S}) \rceil$ bits to each sequence.

The decoder, given the side information and the codeword, reconstructs $\pi(\mathcal{S})$ and then sets $S_i = \{x_j : i \in \sigma(x_j)\}$ for each $i \in [m]$, recovering $\mathcal{S}$. □

The code of Proposition 5.1.4 stores an index of the whole sequence $(\sigma(x_1), \ldots, \sigma(x_u))$ among all sequences with the prescribed pattern counts. A membership query for S_i and x needs the pattern of x and tests whether it contains i. A rank query for S_i up to position x needs the number of earlier patterns that contain i. Since even these two values are not available from the index alone, bit optimality does not by itself give a representation for the five operations of Definition 1.2.2.

To support these operations in the construction developed next, we replace the global sequence by local descriptions of how one set is recovered from another. The basic local description is containment: from a parent P to a child $Q \subseteq P$, the edge stores the deleted elements $P \setminus Q$, and path queries count or select those deletions. Two incomparable sets

lack such a parent inside the pair itself. Their union contains both, and using it as a synthetic parent turns the pair into two containment edges.

The construction in the next section keeps only this binary union step, which is local, creates containment edges, and can be iterated into a forest whose root-to-leaf paths recover the input sets. Broader closure models are left to Appendix A.

5.2 Union matching

The previous section leaves a gap between counting and queries. The atom code counts the membership-pattern sequence directly, but a query on one set should not decode that whole sequence. To keep the locality of the hierarchical representations, each input set should instead be recovered from a nearby reference by a sequence of small changes. Containment already has that form: if $Q \subseteq P$, then the edge from P to Q stores the deleted elements $P \setminus Q$.

This containment view breaks when two useful labels are not contained in one another. Neither label can be the parent of the other, but both can be placed below any set that contains their union. Choosing $M = A \cup B$ adds no element beyond what is needed to contain both labels, and it creates two containment edges, from M to A and from M to B. Once M has been introduced, it can be treated like any other current label and paired again with another label.

Iterating this operation produces a rooted forest whose leaves carry the input sets and whose internal nodes carry synthetic unions. Every parent-child edge is a containment edge, so each input set can be recovered from its component root by deleting the elements stored along one root-to-leaf path. Set-Union Matching, abbreviated SUM, is the construction that chooses these binary union parents. We first charge one merge, then add the charges over a whole forest, and only then turn the resulting cost into a matching rule.

5.2.1 Merge cost

Fix two sets $A, B \subseteq U$ and write $M = A \cup B$, $k = |A \cap B|$, $l = |A \setminus B|$, and $r = |B \setminus A|$. Then $|M| = k + l + r$. Once M is known, recovering A and B is the same as assigning every element of M to one of the three regions $A \cap B$, $A \setminus B$, and $B \setminus A$. Two fixed-count bitvectors encode this ternary assignment, because once two of the three regions are identified the third is determined by elimination. First, a bitvector of length $|M|$ marks the k elements in the intersection, selecting one of $\binom{|M|}{k}$ possibilities. After those elements are removed, a second bitvector of length $l + r$ marks the l elements in $A \setminus B$, selecting one of $\binom{l+r}{l}$ possibilities. The remaining r elements belong to $B \setminus A$ and require no third bitvector. The product of the two counts is the multinomial coefficient $\binom{|M|}{k,l,r}$. A short header records k and l, after which $r = |M| - k - l$ follows.

Write the elements of M in the order inherited from U as $m_1 < m_2 < \cdots < m_{|M|}$. The example in Figure 5.1 uses the seven positions $m_1, \ldots, m_7$ to show how the two bitvectors are indexed. The first bitvector isolates

the intersection positions, and the second bitvector is indexed by the compacted sequence of elements that remain.

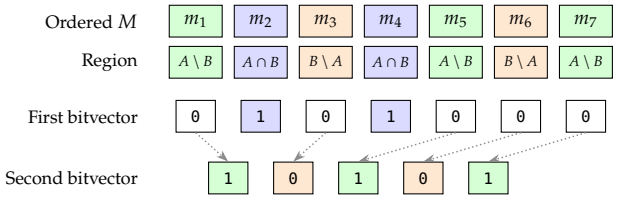

Figure 5.1: The two bitvectors used to encode the three regions of M. The top row fixes the order of M. The first bitvector marks the k positions of $A \cap B$. The unmarked positions are compacted, and the second bitvector marks the l positions of $A \setminus B$. The remaining positions belong to $B \setminus A$. The number of choices is $\binom{|M|}{k}\binom{l+r}{l} = \binom{|M|}{k,l,r}$.

Definition 5.2.1 (Merge cost) *For $A, B \subseteq U$ with $M = A \cup B$, $k = |A \cap B|$, $l = |A \setminus B|$, $r = |B \setminus A|$, the merge cost of the pair (A, B) is*

$$w(A, B) \;=\; \lg \binom{|M|}{k, l, r} \;+\; c(|M|, k), \tag{5.1}$$

where $c(|M|, k) = \lceil \lg(|M| + 1) \rceil + \lceil \lg(|M| - k + 1) \rceil$ encodes k and l.

When $k = 0$, the first bitvector is fixed and the multinomial part is $\binom{l+r}{l}$. We still store the two-field header so that one decoder handles all pairs.

The multinomial in (5.1) is bounded above by $\binom{|M|}{|A|}\binom{|M|}{|B|}$, the cost of encoding A and B as two separate bitvectors of length $|M|$, one marking the elements of A and one marking the elements of B. This is the local analogue of the $H_{wc}(\mathcal{S})$ baseline, where each S_i is encoded independently as a bitvector of length u. Every joint labelling of M by the three regions $A \cap B$, $A \setminus B$, $B \setminus A$ determines both bitvectors uniquely, so the count $\binom{|M|}{k,l,r}$ injects into their product.

The merge cost is local, but the representation must describe the whole collection. We distinguish node occurrences from their set labels, because two nodes may carry the same subset of U while playing different roles in the forest. Each time we merge two current occurrences a and b with set labels P_a and P_b, we create a new parent occurrence c with set label $P_c = P_a \cup P_b$ and keep a and b as its two children. Repeating this operation turns the indexed input occurrences into the leaves of rooted binary components. The internal occurrences are precisely the synthetic unions introduced by the construction. A component root has no parent that can describe it, while an internal occurrence is described through its parent-child decomposition. The global cost must therefore charge roots and internal occurrences in different ways.

Definition 5.2.2 (SUM forest) *A SUM forest on $\mathcal{S}$ is a rooted forest F in which every internal node has exactly two children. We call the nodes of F occurrences, because two distinct nodes may carry the same set label. Each occurrence a carries a set label $P_a \subseteq U$. The leaves of F are exactly m distinguished occurrences $a_1, \ldots, a_m$ with $P_{a_i} = S_i$, kept distinct even when two input sets coincide. For every internal occurrence c with children a and b, $P_c = P_a \cup P_b$.*

A SUM forest records which union occurrences have been introduced. To decode it, we start from each root and move downward. At an internal occurrence c, once the decoder knows the set label P_c, the merge description charged by $w(P_{\text{left}(c)}, P_{\text{right}(c)})$ recovers the set labels of its two child occurrences. A root occurrence has no parent from which it can be recovered. The representation therefore stores each root label P_a directly as a subset of U, at cost $\lg \binom{u}{|P_a|}$. This gives two charges, one for starting a component and one for splitting a known occurrence.

Different choices of merges produce different SUM forests, so the construction needs a single value with which to compare them. We assign to a forest the sum of all root charges and all merge charges. This value is denoted by $\Phi(F)$.

Definition 5.2.3 (SUM cost) *The cost of a SUM forest F on $\mathcal{S}$ is*

$$\Phi(F) \;=\; \sum_{a \in \mathrm{roots}(F)} \lg \binom{u}{|P_a|} \;+\; \sum_{c \in \mathrm{internal}(F)} w(P_{\mathrm{left}(c)}, P_{\mathrm{right}(c)}). \tag{5.2}$$

In (5.2)*, the first sum is the support cost, with each root occurrence encoded against U. The second is the structural cost, with each internal occurrence contributing the merge cost of its decomposition. The terms* $\mathrm{left}(c)$ *and* $\mathrm{right}(c)$ *denote the two child occurrences of c.*

$\Phi(F)$ is a conditional cost. The forest skeleton and the map from distinguished leaf occurrences a_i to input indices i are outside this count. Given that side information, the root terms count supports of known sizes, and the multinomial terms count local splits after their compositions are fixed. The header term in w instead pays for the local fields k and l used by the uniform decoder. The bits needed to encode the skeleton, the leaf map, and the auxiliary indices that support queries are accounted for separately in Section 5.3.

5.2.2 Greedy matching

$\Phi(F)$ ranks SUM forests by cost, but the construction still needs a rule for choosing which merges to perform. The rule assigns a score to every pair of current roots and selects compatible pairs through a weighted matching at each level.

We score a candidate merge by the change it induces in the forest cost Φ. Take two current root occurrences a and b with set labels $A = P_a$ and $B = P_b$. Replacing them with a new root occurrence labelled $M = A \cup B$ adds a root charge for M as a subset of U and removes the old root charges for a and b. The merge description then pays $w(A, B)$ to recover the two child labels from M. The resulting difference is the merge score.

Definition 5.2.4 (Merge score) *Let a and b be two root occurrences of a SUM forest, set $A = P_a$, $B = P_b$, and $M = A \cup B$. The merge score of the occurrence pair (a, b) is*

$$\Delta\Phi(a, b) \;=\; \lg \binom{u}{|M|} \;+\; w(A, B) \;-\; \lg \binom{u}{|A|} \;-\; \lg \binom{u}{|B|}. \tag{5.3}$$

In (5.3), the term $\lg \binom{u}{|M|}$ stores the new root label as a subset of U, and $w(A, B)$ stores the split of M into its two child labels. The two negative terms remove the former root charges of the two occurrences. Hence $\Delta\Phi(a, b) < 0$ means that this replacement lowers Φ, while $\Delta\Phi(a, b) > 0$ means that it raises Φ.

A level may contain several merges as long as no two pairs share a root. Each merge changes Φ only through the support charges of its two roots and the new support and merge charges introduced by their parent.

Disjoint pairs of merges therefore modify disjoint terms of Φ, and the total change at the level is the sum of the individual scores $\Delta\Phi(a, b)$. For a prescribed number q of pairs, the level cost is minimised by a minimum-weight matching of cardinality q in the complete graph on the current root occurrences, with edge weight $\Delta\Phi(a, b)$ on the edge $\{a, b\}$. SUM takes the maximum-cardinality choice, $q = \lfloor r/2 \rfloor$, when the level starts with r roots. The next level has $\lceil r/2 \rceil$ roots, so at most $\lceil \lg m \rceil$ levels are produced.

Since the matching has cardinality $\lfloor r/2 \rfloor$, it may contain pairs with positive score, and a level can raise Φ above the previous one. SUM therefore records the forest at every level and returns the cheapest recorded forest. The first recorded forest is the trivial F_0 on m isolated roots, with cost $\Phi(F_0) = \sum_i \lg \binom{u}{|S_i|} = H_{wc}(\mathcal{S})$. The independent encoding therefore remains available as a recorded option.

The score compares the cost of two encodings of the pair (A, B) over U. For the same candidate pair, write $k = |A \cap B|$, $l = |A \setminus B|$, and $r = |B \setminus A|$. As separate roots, A and B are encoded as two binary strings of length u with $|A|$ and $|B|$ ones, giving $\binom{u}{|A|} \cdot \binom{u}{|B|}$ choices. As a single union root, the pair is encoded by assigning each universe element to one of the four regions $U \setminus M$, $A \cap B$, $A \setminus B$, and $B \setminus A$, with counts $u - |M|$, k, l, and r. This gives $\binom{u}{u-|M|,k,l,r}$ choices.

For a candidate occurrence pair (a, b) with labels $A = P_a$ and $B = P_b$, the signed saving of the merge is $-\Delta\Phi(a, b)$. Substituting (5.1) into (5.3) gives

$$\begin{aligned} -\Delta\Phi(a,b) = \lg \binom{u}{|A|} \; + \; \lg \binom{u}{|B|} \\ - \; \lg \binom{u}{|M|} \; - \; \lg \binom{|M|}{k,l,r} \; - \; c(|M|,k). \end{aligned} \tag{5.4}$$

The two negative logarithms in (5.4) combine because the product $\binom{u}{|M|}\binom{|M|}{k,l,r}$ counts the same choices in two ways. We may choose the support M and then split it into the three regions inside M, or we may partition all u universe elements directly into $U \setminus M$, $A \cap B$, $A \setminus B$, and $B \setminus A$. Hence

$$\begin{aligned} -\Delta\Phi(a,b) = \lg \binom{u}{|A|} \; + \; \lg \binom{u}{|B|} \\ - \; \lg \binom{u}{u-|M|,k,l,r} \; - \; c(|M|,k). \end{aligned} \tag{5.5}$$

Algorithm 2 performs the level-by-level construction and returns the cheapest forest visited along the way. It maintains a working forest F, updated at each level by a minimum-weight matching of maximum cardinality on its current roots, alongside a recorded forest F^* initialised to the trivial F_0. The invariant $\Phi^* = \min_{0 \le s \le t} \Phi(F_s)$ holds at the end of level t, where F_s is the working forest after s levels.

Let r_t be the number of roots at the start of level t. A sorted-set implementation of Algorithm 2 first computes $\Delta\Phi(a, b)$ for every unordered pair of current roots. Computing one score amounts to finding $|P_a \cap P_b|$, $|P_a \setminus P_b|$, and $|P_b \setminus P_a|$, so a linear merge of the two sorted set labels costs $O(|P_a| + |P_b|)$ time. Assume that the logarithmic binomial and

Algorithm 2: Set-Union Matching.

Input: Collection $\mathcal{S} = \{S_1, \ldots, S_m\}$ over universe U.
Output: A SUM forest F^* achieving the smallest value of Φ across all levels $0, 1, \ldots, \lceil \lg m \rceil$.

```
F ← trivial forest F_0 on S;
F* ← F; Φ* ← Φ(F);
for t = 1, 2, ..., ⌈lg m⌉ do
    compute ΔΦ(a, b) for every pair of current root occurrences a, b in F;
    M_t ← minimum weight matching of maximum cardinality on the current root occurrences, with edge weights ΔΦ;
    foreach pair (a, b) ∈ M_t do
        replace a, b in F by a new root occurrence c with P_c = P_a ∪ P_b and child occurrences a, b;
    if Φ(F) < Φ* then
        F* ← F; Φ* ← Φ(F);
return F*;
```

multinomial terms in $\Delta\Phi$ are available for comparison in constant time once these three cardinalities are known. Let $N = \sum_i |S_i|$. Every current root label is a union of input sets, hence has size at most N, and one score costs $O(N)$ time in the worst case. The level has $\binom{r_t}{2}$ candidate pairs, so scoring costs $O(r_t^2 N)$ time.

The same level then solves a weighted matching instance on the complete graph over the r_t roots. This graph has $\Theta(r_t^2)$ edges. Weighted matching on a general graph with arbitrary edge weights runs in $O(|E|r + r^2 \log r)$ time [54]. Substituting $|E| = \Theta(r_t^2)$ gives $O(r_t^3)$ matching time at level t. Since $r_t \le \lceil m/2^{t-1} \rceil$, the scoring costs over all levels sum to $O(m^2 N)$, and the matching costs sum to $O(\sum_t r_t^3) = O(m^3)$. This sorted-set implementation runs in $O(m^2 N + m^3)$ time.

[54]: Gabow (2018), *Data structures for weighted matching and extensions to b-matching and f-factors*

SUM is a greedy heuristic. The matching at each level minimises that level's cost at fixed cardinality but not the global cost over all binary union forests, so Φ may rise as well as fall over the levels. Even the first level can raise Φ when every candidate pair has positive score and the cardinality $\lfloor r/2 \rfloor$ forces a merge. A level may also pass over an individually profitable merge, because a negative-score edge enters the matching only if it belongs to the cheapest matching of that cardinality, and its endpoints may otherwise be matched to other roots when that lowers the total level cost.

The instance $\mathcal{S} = \{\{1\}, \{2\}, \{3\}\}$ over $U = \{1, 2, 3\}$ in Figure 5.2 exhibits this phenomenon. Every level-1 candidate merges two singletons under a two-element union, giving cost $2 \lg 3 + 5$. Since $m = 3$, the algorithm then has one two-element root and one singleton root, and the second forced merge gives cost $9 + \lg 3$. Both nontrivial recorded forests are more expensive than F_0, so SUM returns F_0.

Because F_0 is recorded before any forced merge, Algorithm 2 returns the forest with smallest Φ across all levels visited, including level zero.

Definition 5.2.5 (Minimum recorded forest) *Let F_t be the forest held by Algorithm 2 after level t, with F_0 the trivial forest. The minimum recorded forest*

is any forest F^* satisfying

$$\Phi(F^*) = \min_{0 \le t \le \lceil \lg m \rceil} \Phi(F_t).$$

We write $L_{\mathrm{SUM}}(\mathcal{S}) = \Phi(F^*)$.

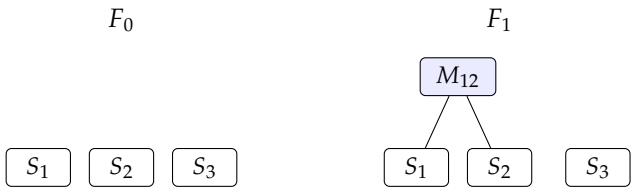

Figure 5.2: Trivial forest F_0 on $\mathcal{S} = \{\{1\}, \{2\}, \{3\}\}$ over $U = \{1, 2, 3\}$, and a level-1 candidate F_1 obtained by merging two singletons under the synthetic union $M_{12} = S_1 \cup S_2$. The costs are $\Phi(F_0) = 3 \lg 3$, $\Phi(F_1) = 2 \lg 3 + 5$, and $\Phi(F_2) = 9 + \lg 3$ for the unshown second-level forest. Both forests after F_0 are more expensive, so SUM returns F_0.

Recording F_0 at level zero gives a uniform upper bound against the baseline.

Theorem 5.2.1 *For every collection $\mathcal{S}$,*

$$L_{\mathrm{SUM}}(\mathcal{S}) \;\le\; H_{wc}(\mathcal{S}).$$

Proof. F_0 is a recorded forest and $\Phi(F_0) = H_{wc}(\mathcal{S})$, so the minimum across recorded forests is at most $H_{wc}(\mathcal{S})$. □

The atom bound L^* of Proposition 5.1.1 still controls any fixed SUM skeleton. Once the skeleton and the map from leaves to input indices are fixed, each occurrence represents the union of a fixed set of input indices. The atom counts therefore determine all root sizes and all local split compositions. This lets us compare the record count bounded by Φ with the atom type class counted by L^*.

Proposition 5.2.2 *For every SUM forest F on $\mathcal{S}$, with its skeleton and leaf-to-input map fixed as side information,*

$$L^*(\mathcal{S}) \;\le\; \Phi(F).$$

Consequently,

$$L^*(\mathcal{S}) \;\le\; L_{\mathrm{SUM}}(\mathcal{S}) \;\le\; H_{wc}(\mathcal{S}).$$

Proof. Fix the skeleton of F and the leaf-to-input map. Consider any collection $\mathcal{S}' = \{S'_1, \dots, S'_m\}$ with the same realised patterns and atom counts c_τ as $\mathcal{S}$. For an occurrence a, let I_a be the set of input indices whose distinguished leaves lie below a. The label represented by a in this fixed skeleton is $P'_a = \bigcup_{i \in I_a} S'_i$. Its size is $\sum_{\tau : \tau \cap I_a \neq \emptyset} c_\tau$, so each root size is determined by the atom counts and the fixed skeleton. If an internal occurrence c has children a and b, the three local sizes $|P'_a \cap P'_b|$, $|P'_a \setminus P'_b|$, and $|P'_b \setminus P'_a|$ are determined in the same way by whether an atom pattern intersects I_a, I_b, or both.

Encode $\mathcal{S}'$ using the fixed skeleton. Store each root support, and for each internal occurrence store the ternary split of its parent label into $P'_a \cap P'_b$, $P'_a \setminus P'_b$, and $P'_b \setminus P'_a$. These choices reconstruct every child label from its parent label, and therefore reconstruct all distinguished leaves. Distinct collections give distinct records. The number of possible records over the whole atom type class is the product of the root support counts and the

multinomial split counts. Its logarithm is the sum of the root terms and multinomial terms in $\Phi(F)$, while the header terms in w only increase $\Phi(F)$. Since the atom type class has size $2^{L^*(\mathcal{S})}$, we get $L^*(\mathcal{S}) \le \Phi(F)$. Specialising to the minimum recorded forest gives the left inequality, and Theorem 5.2.1 gives the right one. □

Remark 5.2.1 (Strict descent variant) Replacing $\mathcal{M}_t$ in Algorithm 2 by a minimum-total-score matching of arbitrary cardinality in the graph induced by root-occurrence pairs with $\Delta\Phi < 0$, and skipping the level when no such edge exists, gives a variant in which every nonempty level lowers Φ by the sum of the selected negative scores. The terminal forest $\widehat{F}$ inherits $\Phi(\widehat{F}) \le H_{wc}(\mathcal{S})$ by monotone descent rather than by the best-recorded mechanism. This variant changes the search behaviour, not the representation. The rest of the chapter uses the minimum recorded forest F^* returned by Algorithm 2.

5.2.3 Deletion trees

The cost Φ says which forest is cheaper, but the representation also needs an operational meaning for the forest. Once a component root is stored, a leaf set should be recoverable by reading the unique path from that root to the leaf. SUM makes this possible because every synthetic parent is a union of its children: the child contains no element outside the parent, so the edge only records elements deleted from the parent. Along a root-to-leaf path, a deleted element cannot reappear below the edge where it is removed. Thus the path carries exactly the difference between the root label and the leaf label, with no repetitions, which gives the deletion-tree view of a SUM forest.

Proposition 5.2.3 (Containment edges) *For every internal occurrence c with child occurrences a and b, both inclusions $P_a \subseteq P_c$ and $P_b \subseteq P_c$ hold, so every parent-child edge in a SUM forest is a containment.*

Proof. By Definition 5.2.2, $P_c = P_a \cup P_b$. Hence $P_a \subseteq P_c$ and $P_b \subseteq P_c$. □

Lemma 5.2.4 (Path deletions) *Let $a_0, a_1, \dots, a_t$ be a root-to-leaf path in a SUM forest, with $R = P_{a_0}$ and $S = P_{a_t}$. For each step $1 \le i \le t$, set $D_i = P_{a_{i-1}} \setminus P_{a_i}$. Then the sets $D_1, \dots, D_t$ are pairwise disjoint and*

$$S = R \setminus \bigcup_{i=1}^{t} D_i. \tag{5.6}$$

Equivalently, every element of $R \setminus S$ is removed exactly once along the path.

Proof. By Proposition 5.2.3, $P_{a_i} \subseteq P_{a_{i-1}}$ at every step. Induction gives (5.6). If $x \in D_i$, then $x \notin P_{a_i}$. Every later label P_{a_j} with $j \ge i$ is contained in P_{a_i}, so x cannot appear in any later deletion set. Thus the deletion sets are pairwise disjoint, and the equality says that every element removed from the root is removed exactly once. □

Definition 5.2.6 (SUM deletion tree) *Let F be a SUM forest. For each component root r, the SUM deletion tree of that component is obtained by creating one structural node v_a for every forest node occurrence a in the component. Distinct occurrences with the same set label create distinct structural nodes. For every forest edge $a \longrightarrow b$, replace it by a chain*

$$v_a \longrightarrow z_1 \longrightarrow z_2 \longrightarrow \cdots \longrightarrow z_h \longrightarrow v_b,$$

where $h = |P_a \setminus P_b|$ and the nodes $z_1, \ldots, z_h$ are labelled with the distinct elements of $P_a \setminus P_b$ in arbitrary order. If $P_a \setminus P_b = \emptyset$, no deletion node is inserted and the structural edge $v_a \longrightarrow v_b$ remains.

The deletion labels on the path from v_{a_0} to v_{a_t} are exactly the elements of $P_{a_0} \setminus P_{a_t}$, with no repetition, by Lemma 5.2.4. A membership query checks whether a root element has no deletion label on the path. A rank query counts deleted local ranks up to a prefix of the order inherited from P_{a_0}. An access query selects the q-th element of P_{a_0} whose local deletion label is absent from the path.

SUM deletion trees fit the tree-extraction framework of Chapter 4, with a path semantics specific to SUM.

Insertion paths (Definition 4.2.5) accumulate from $\emptyset$, indel paths (Definition 4.3.3) apply signed edits, SUM paths remove elements from a stored root R.

The construction contains ordinary containment encoding as a special case. If $A \subseteq B$ already holds in $\mathcal{S}$, then the merge of two occurrences carrying labels A and B gives $M = A \cup B = B$, $k = |A|$, $l = 0$, and $r = |B| - |A|$. Hence

$$w(A, B) \;=\; \lg \binom{|B|}{|A|, 0, |B| - |A|} + c(|B|, |A|) \;=\; \lg \binom{|B|}{|A|} + O(\lg |B|).$$

The binary merge creates a new internal node with set label B and two children carrying labels A and B. The edge to the child labelled B inserts no deletion node, while the edge to the child labelled A stores $B \setminus A$. The SUM forest still keeps the two occurrences labelled B distinct. For comparison with an ordinary containment hierarchy, if we ignore the zero-deletion structural edge, the remaining nonzero edge is the containment edge from B to A. Its cost is the containment-entropy contribution of A given B in Definition 4.2.2, up to the self-delimiting header.

For incomparable root labels A and B, a containment-only hierarchy without new synthetic references can place both sets under one nontrivial parent only when an already available set contains both. If no such label is present, the universal common parent is U. SUM instead inserts the smallest possible common parent, namely $A \cup B$. If this new parent remains a component root, the forest pays its support cost $\lg \binom{u}{|A \cup B|}$. If a later merge places it below a larger union node, the support charge for $A \cup B$ is no longer paid as a root term. The larger merge pays a local split cost that recovers $A \cup B$ as one child label. Every descendant is still obtained by deleting labels from its component root.

The binary union step is one choice among several possible ways to create deletion paths. Appendix A studies what changes if we admit additional reference sets generated from $\mathcal{S}$, or if one parent may split into more than two children. Such choices can reduce the counting cost because one parent can describe a larger Venn pattern at once. They also no longer lead to the pairwise matching subproblem used by Algorithm 2. The data structure of Section 5.3 keeps the binary deletion trees produced

here and adds the root dictionaries and tree-extraction indices needed for the five operations.

5.3 Queries

The deletion-tree expansion of Definition 5.2.6 turns each component of the minimum recorded forest F^* of Definition 5.2.5 into a labelled ordinal tree. For input index i, let a_i be the distinguished leaf, let r_i be its component root, set $R_i = P_{r_i}$, and let v_i be the structural node created from a_i. By Lemma 5.2.4, the deletion labels on the path from v_{r_i} to v_i are exactly the elements of $R_i \setminus S_i$. The five operations of Definition 1.2.2 on S_i will therefore be implemented by testing, counting, or selecting local ranks on this path.

Empty input sets and components with $|R_i| = 0$ are stored as sentinels outside the deletion trees; member returns false, rank returns zero, and access, predecessor, and successor return their respective failure values.

[40]: He et al. (2014), *A framework for succinct labeled ordinal trees over large alphabets*
[44]: He et al. (2016), *Data structures for path queries*

The representation performs these translations through a query directory $\mathcal{D}$. For a non-sentinel input, $\mathcal{D}[i] = (r_i, v_i)$ stores the component root occurrence and the structural node of the leaf occurrence for S_i, so v_i identifies the deletion tree in which the path query is evaluated. Each SUM deletion tree is converted into the node-labelled form expected by the tree-extraction primitives of Section 3.5 [40, 44], and each component root carries a dictionary between universe values and their ranks inside that root. In the algorithms below, an expression such as $\mathrm{rank}(i, x)$ is shorthand for the representation query that answers the public operation $\mathrm{rank}(S_i, x)$ of Definition 1.2.2.

5.3.1 Forest encoding

Fix a component root occurrence r with set label $R = P_r$. The chain expansion of Definition 5.2.6 produces a structural node v_a for every forest occurrence a in the component below r, and inserts one deletion node z_j for each element removed along the edge from a to a child. Since every set label below r is a subset of R, every deletion label is an element of R.

For every universe value x, $\mathrm{rank}_R(x)$ denotes the number of elements of R at most x, so it lies in $[0..|R|]$ on arbitrary query values. At a deletion node, the deleted value belongs to R, and we store its local rank $\mathrm{rank}_R(x) \in [1..|R|]$. At each structural node we store the dummy label $\delta_R = |R| + 1$. The labelled ordinal tree thus has technical alphabet $[1..|R| + 1]$, while every query uses only the deletion alphabet $[1..|R|]$.

By Lemma 5.2.4, the deletion labels on each root-to-leaf path are pairwise distinct. The path therefore represents a set of deleted local ranks, so path counting counts deleted elements in a prefix of R, and access becomes selection from the complementary ranks.

Public queries use universe values, while the tree index stores local ranks. Each root occurrence with label R is therefore stored as an RRR dictionary from Theorem 2.1.7, supporting membership, rank_R, and select_R in constant time. The dictionary occupies

$$\lg \binom{u}{|R|} + o(u) + O(\lg \lg u)$$

bits. The leading term is the support cost of R in Definition 5.2.3, while the lower-order terms belong to the query index outside L_{SUM}.

The α-operations index of Theorem 3.6.2 and the path-queries index of Theorem 3.7.4 overlay each labelled deletion tree at technical alphabet size $|R| + 1$. The α-operations index supports $\texttt{parent}_\alpha(v)$, the lowest proper ancestor of v with label α, for every $\alpha \in [1..|R|]$ in $O(\lg \lg_\omega |R|)$ time. The path-queries index supports $\texttt{path\_count}(v, [a..b])$, the number of nodes on the path from the root to v whose label lies in $[a..b]$, in $O(\lg_\omega |R|)$ time, and $\texttt{path\_select}(v, j)$, the j-th smallest label among the labels on that path, in $O(\lg |R| / \lg \lg |R|)$ time.

Each index stores a labelled ordinal tree of $|T|$ nodes in $O(|T|)$ words. Summed over all roots of F^*, the deletion forest has

$$O\left(m + \sum_{c \in \mathrm{internal}(F^*)} |P_{\mathrm{left}(c)} \triangle P_{\mathrm{right}(c)}|\right)$$

nodes, and the tree-extraction overlay occupies the same number of words.

5.3.2 Complement selection

Access asks for the q-th smallest element of the component root whose local rank does not occur as a deletion label on the path. The path-selection primitive of Theorem 3.7.4 selects labels that occur on the path, hence deleted root elements. We therefore need the complementary selection problem over the deletion alphabet. Membership and rank do not need this adaptation, since they use $\texttt{parent}_\alpha$ and $\texttt{path\_count}$ directly.

Definition 5.3.1 (Complement path selection) *Let T be a labelled ordinal tree on alphabet $[1..\sigma + 1]$. Labels in $[1..\sigma]$ are deletion labels, label $\sigma + 1$ is a dummy structural label, and deletion labels are unique on each root-to-leaf path. For a node v, let $D_T(v)$ be the set of labels in $[1..\sigma]$ that occur on the root-to-v path, including v itself. For every integer i with $1 \le i \le \sigma - |D_T(v)|$, the complement path selection $\texttt{path\_compselect}(v, i)$ is the i-th smallest element of $[1..\sigma] \setminus D_T(v)$.*

This definition isolates the operation that the standard path-query index does not already provide. The next lemma shows that selecting absent deletion labels keeps the same asymptotic cost as path selection, because the same alphabet descent can be driven by survivor counts instead of deletion counts.

Lemma 5.3.1 *The path-queries index of Theorem 3.7.4 on a labelled ordinal tree T satisfying Definition 5.3.1 supports $\texttt{path\_compselect}(v, i)$ in constant time for $\sigma \le 1$ and in $O(\lg \sigma / \lg \lg \sigma)$ time for $\sigma \ge 2$, under the word-RAM hypotheses of Theorem 3.7.4.*

Proof. If $\sigma \le 1$, the only possible answer is checked directly. Assume $\sigma \ge 2$. Use the alphabet descent of $\texttt{path\_select}$, restricted to the deletion alphabet $[1..\sigma]$, so the dummy label $\sigma + 1$ is outside every interval considered. At a state with current interval $I \subseteq [1..\sigma]$, maintain the invariant that the answer is the i-th smallest label of $I \setminus D_T(v)$. If $J \subseteq I$ is

a child interval, let $C_v(J) = \texttt{path\_count}(v, J)$. Since deletion labels are unique on the root-to-v path, the number of surviving labels in J is

$$|J| - C_v(J).$$

The values $C_v(J)$ are the child counts already used by the selection descent. Replacing each count by its complement within the child interval adds one subtraction per child. We choose the first child whose cumulative survivor count reaches i, subtract the survivor counts of previous children, and recurse into that child. The invariant is preserved, and a leaf with positive survivor count contains the desired deletion-alphabet label. The descent depth and per-level packed-comparison cost are unchanged, so the time bound follows. □

Thus access can use the same tree overlay as rank and membership. The reductions below differ only in which local-rank question they ask on the root-to-leaf path.

5.3.3 Query reductions

The previous subsections have built the objects that a query can inspect: the directory locates the path for S_i, the root dictionary translates universe values to ranks in R_i, and the tree overlay answers questions about deletion labels on that path. A public query is still phrased in the language of Definition 1.2.2. The reductions below bridge this mismatch by turning each operation into dictionary calls and a small number of path queries over local ranks.

Every reduction starts from the same translation. The query reads $\mathfrak{D}[i] = (r_i, v_i)$ and sets $R_i = P_{r_i}$. The deletion labels on the root-to-v_i path are exactly the elements of $R_i \setminus S_i$, so a label $j \in [1..|R_i|]$ appears on the path if and only if the descent from R_i to S_i removes $\text{select}_{R_i}(j)$. When no proper ancestor of v_i carries label j, we write $\texttt{parent}_j(v_i) = \texttt{null}$, distinct from the dummy label $\delta_{R_i} = |R_i| + 1$ that sits outside the deletion alphabet.

Membership asks whether x belongs to S_i. Since $S_i \subseteq R_i$, if x is not in the component root then x cannot belong to S_i either, and the root dictionary disposes of this case in constant time. When x does belong to R_i, the question is whether the descent from R_i down to v_i deletes x. We translate x to its local rank $j = \text{rank}_{R_i}(x)$, and the path query reduces to asking whether any proper ancestor of v_i carries label j. The α-operations index answers this in one call. The value $\texttt{parent}_j(v_i) = \texttt{null}$ tells us that no such ancestor exists, in which case x has survived to S_i. The query in Algorithm 3 returns exactly this absence test.

Rank asks how many elements of S_i are at most x. The elements of R_i at most x are easy to count, since the root dictionary returns $j = \text{rank}_{R_i}(x)$ in constant time, regardless of whether x itself lies in R_i. Among those j candidates, only the survivors of the deletions on the path to v_i also belong to S_i. Since deletion labels are unique on this path, $\texttt{path\_count}(v_i, [1..j])$ counts exactly the deleted candidates in the prefix. The query in Algorithm 4 returns the resulting difference.

Algorithm 3: SUM membership query.

Input: An input index i, a value x, the query directory $\mathfrak{D}$, and the root dictionaries and tree indices.
Output: Whether x belongs to S_i.

Query member(i, x)**:**
  $(r_i, v_i) \leftarrow \mathfrak{D}[i]$;
  $R_i \leftarrow P_{r_i}$;
  if member$_{R_i}(x)$ = false **then return** false;
  $j \leftarrow \text{rank}_{R_i}(x)$;
  return `parent`$_j(v_i)$ = `null`;

Algorithm 4: SUM rank query.

Input: An input index i, a value x, the query directory $\mathfrak{D}$, and the root dictionaries and tree indices.
Output: The number of elements of S_i at most x.

Query rank(i, x)**:**
  $(r_i, v_i) \leftarrow \mathfrak{D}[i]$;
  $R_i \leftarrow P_{r_i}$;
  $j \leftarrow \text{rank}_{R_i}(x)$;
  if $j = 0$ **then return** 0;
  return $j -$ `path_count`$(v_i, [1..j])$;

Access is the first reduction that must select a survivor rather than test or count deleted labels. For valid q, the complement path selection `path_compselect`(v_i, q) of Definition 5.3.1 returns the local rank j in R_i of the q-th label not deleted on the path to v_i. The query in Algorithm 5 converts this local rank back to the universe value $\text{select}_{R_i}(j)$.

Algorithm 5: SUM access query.

Input: An input index i, a position $q \in [1..|S_i|]$, the query directory $\mathfrak{D}$, and the root dictionaries and tree indices.
Output: The q-th smallest element of S_i.

Query access(i, q)**:**
  $(r_i, v_i) \leftarrow \mathfrak{D}[i]$;
  $R_i \leftarrow P_{r_i}$;
  $j \leftarrow$ `path_compselect`(v_i, q);
  return $\text{select}_{R_i}(j)$;

Predecessor returns the largest element of S_i at most x, or $-\infty$ if no such element exists. The rank $r = \text{rank}(i, x)$ counts the elements of S_i at most x, so the predecessor is $\text{access}(i, r)$ when $r > 0$ and $-\infty$ when $r = 0$. The query in Algorithm 6 uses only this rank test and access call.

Successor returns the smallest element of S_i at least x, or $+\infty$ if none exists. For $x = 1$, the candidate position is $r = 1$. For $x > 1$, the candidate position is $r = \text{rank}(i, x - 1) + 1$. The size $|S_i| = |R_i| -$ `path_count`$(v_i, [1..|R_i|])$ counts the survivors. The successor is $\text{access}(i, r)$ when $r \leq |S_i|$, and $+\infty$ otherwise. The query in Algorithm 7 computes this candidate position and uses the survivor count as the failure test.

Algorithm 6: SUM predecessor query.

Input: An input index i, a value x, the query directory $\mathfrak{D}$, and the root dictionaries and tree indices.
Output: The predecessor of x in S_i, or $-\infty$.

```
1 Query predecessor(i, x):
2     r ← rank(i, x);
3     if r = 0 then return −∞;
4     return access(i, r);
```

Algorithm 7: SUM successor query.

Input: An input index i, a value x, the query directory $\mathfrak{D}$, and the root dictionaries and tree indices.
Output: The successor of x in S_i, or $+\infty$.

```
1 Query successor(i, x):
2     (r_i, v_i) ← 𝔇[i];
3     R_i ← P_{r_i};
4     r ← 1;
5     if x > 1 then
6         r ← rank(i, x − 1) + 1;
7     s_i ← |R_i| − path_count(v_i, [1..|R_i|]);
8     if r > s_i then return +∞;
9     return access(i, r);
```

5.3.4 Representation bound

The SUM representation uses one RRR dictionary for each component root and overlays the α-operations and path-query indexes on the labelled deletion forest. By Lemma 5.3.1, complement selection reuses the path-query index.

The space bound separates root support costs from word-level query indexes. The query directory and tree overlays pay for the structural nodes and deletion labels used by the reductions above. Empty inputs, empty component roots, and roots of constant size are answered directly.

Theorem 5.3.2 *Let $\mathcal{S}$ be a collection of subsets of $[1..u]$ with total size n, let F^* be the minimum recorded forest returned by Algorithm 2, and assume the word RAM model with word size $\omega = \Theta(\lg n)$ and universe values fitting in one word. The SUM representation stores the root dictionaries in*

$$\sum_{r \in \text{roots}(F^*)} \left(\lg \binom{u}{|P_r|} + o(u) + O(\lg \lg u) \right) \text{ bits,}$$

and stores the query directory and tree-extraction overlay in

$$O\left(m + \sum_{c \in \text{internal}(F^*)} |P_{\text{left}(c)} \triangle P_{\text{right}(c)}| \right) \text{ words.}$$

It supports the operations of Definition 1.2.2 on every S_i. If $\mathfrak{D}[i] = (r_i, v_i)$ is a non-sentinel entry and $R_i = P_{r_i}$, then the bounds are as follows, with constant time for roots of constant size.

- member(i, x) *in time* $O(\lg \lg_\omega |R_i|)$.

- rank(i, x) *in time* $O(\lg_\omega |R_i|)$.
- access(i, q), predecessor(i, x), successor(i, x) *in time* $O(\lg |R_i| / \lg \lg |R_i|)$.

Proof. For each component root r, Theorem 2.1.7 stores P_r in the displayed number of bits and answers membership, rank_{P_r}, and select_{P_r} in constant time. The query directory costs $O(m)$ words. The deletion expansion creates one structural node per forest occurrence and, for each internal occurrence c, exactly $|P_{\text{left}(c)} \triangle P_{\text{right}(c)}|$ deletion nodes across the two outgoing edges. The deletion forest has the stated number of nodes, and the indexes of Theorem 3.6.2 and Theorem 3.7.4 store it in linear word space.

Sentinel entries and roots of constant size satisfy the claimed bounds directly. For the remaining inputs, Algorithm 3, Algorithm 4, Algorithm 5, Algorithm 6, and Algorithm 7 give the reductions after the directory lookup. Membership makes one $\texttt{parent}_j$ call, costing $O(\lg \lg_\omega |R_i|)$ by Theorem 3.6.2. Rank makes one `path_count` call, costing $O(\lg_\omega |R_i|)$ by Theorem 3.7.4. Access makes one `path_compselect` call, costing $O(\lg |R_i| / \lg \lg |R_i|)$ by Lemma 5.3.1. Predecessor and successor call rank and access a constant number of times, and the access term dominates under the word-size assumption. □

The redundancy in each root dictionary is $o(u)$ because Theorem 2.1.7 stores a bitvector of length u, not one measured against $|P_r|$. After summing over roots, the total redundancy need not be $o(L_{\text{SUM}}(\mathcal{S}))$, and it can dominate the leading support terms when many component roots are small.

The query times in Theorem 5.3.2 use the component-root size $|R_i|$ where Theorem 4.2.9 and Theorem 4.3.5 use u. This is a parameter refinement, not a different primitive. If $|R_i|$ is much smaller than u, the logarithmic factors can decrease. If $|R_i| = u$, membership and rank match the earlier bounds, while access, predecessor, and successor match the insertion bound and improve over the indel bound by a factor of $\lg \lg u$.

The cost $L_{\text{SUM}}(\mathcal{S})$ of Definition 5.2.3 should not be read as the size of the query index in Theorem 5.3.2. It is a counting objective for choosing a forest, where each merge contributes one ternary split term $\lg \binom{|P_c|}{k_c, l_c, r_c}$ plus the header and root support terms. The query index instead exposes the chosen forest to path queries. For an internal occurrence c, let $\rho(c)$ be the root of its component. Packed deletion labels still have payload

$$O\left(\sum_{c \in \text{internal}(F^*)} |P_{\text{left}(c)} \triangle P_{\text{right}(c)}| \, \lg |P_{\rho(c)}| \right)$$

bits, plus structural terms and the root dictionaries. Thus the theorem gives a word-level query index, not a succinct realization of the counting bound. Whether SUM admits a queryable representation of size $L_{\text{SUM}}(\mathcal{S})$ bits within the time bounds of Theorem 5.3.2 remains open. The set-difference representation of Gagie, He, and Navarro has the same gap [9]: it uses $O(\Delta(\mathcal{S}))$ words, while the bit-level target $\Delta(\mathcal{S}) \lg u + o(\Delta(\mathcal{S}) \lg u)$ is left open.

[9]: Gagie et al. (2026), *Compressed Set Representations based on Set Difference*

6 Conclusions

The five operations are those of Definition 1.2.2. They remain in view here because every compressibility measure in the thesis is judged by whether those operations stay queryable.

We have treated compression for set collections as a two-part requirement: a relation between sets must reduce the number of possible descriptions, and it must preserve enough structure to answer membership, rank, access, predecessor, and successor. The independent baseline of Definition 1.2.3 gives the cost before any relation is used. The regimes surveyed in Chapter 4 lower that cost by choosing a reference for each set and storing the difference from that reference. Containment gives deletion paths, insertion compressibility gives insertion paths, and symmetric-difference compressibility gives signed paths. In all three cases, tree extraction turns the saving into a queryable representation.

Chapter 5 starts from the case left open by these regimes: two sets can share enough structure to be worth encoding together even when neither contains the other and no useful reference set is already present. The atom bound $L^*(\mathcal{S})$ shows how far the count can drop if we describe the collection through membership patterns. This bound is tight for a flat encoding, but a flat encoding gives no path for a query to follow. SUM (Section 5.2) restores such paths by creating a reference set. For incomparable roots A and B, it adds the synthetic parent $A \cup B$, whose edges to both children are containment edges and therefore deletion edges. The local Venn count pays for the merge, and the resulting forest supports the query structure of Section 5.3.

SUM therefore sits between the atom bound and the earlier hierarchies. The atom bound counts all membership patterns at once, while the earlier hierarchies keep queries by following references that are already available. SUM adds a restricted form of reference, the binary union, and keeps every edge a containment edge. This restriction is what makes the construction queryable: each component becomes a deletion tree with local alphabet R_i, so the same tree-extraction primitives answer the five operations. It also leaves the bit-level gap exposed by Theorem 5.3.2. The queryable overlay stores explicit deletion labels and root dictionaries, so its bit cost need not match the counting objective $L_{\mathrm{SUM}}(\mathcal{S})$.

Appendix A asks what happens when the representation is no longer restricted to the binary unions chosen by SUM. The point is to view the closure of $\mathcal{S}$ under union and intersection as the ambient space of possible references. This space contains the input sets, the union parents created by SUM, and the additional synthetic sets that could shorten a hierarchy if they were allowed. It also separates two choices that SUM makes together: which reference sets may be used, and how many children a parent may describe at once. With unrestricted arity, one parent can describe the whole atom pattern and reach the atom bound in one level. With binary arity, the construction stays closer to SUM, but the full search can choose labels from the whole lattice rather than only unions of descendant leaves.

The larger reference space in Appendix A connects SUM to the open problems in the work it extends [2, 9]. Synthetic references can reduce

space, but once they are not prescribed by the input, choosing them becomes part of the problem. Queryable path representations have a related difficulty: the combinatorial description can be small while the indexes that navigate it still cost more bits. SUM stays on the controlled side of both issues. It admits only binary unions, so every added reference is a parent in a deletion hierarchy and tree extraction applies without signed labels. This gives a concrete queryable augmentation, but it leaves open both the succinct-space question and the search for better auxiliary references.

This leaves a bit-level question. The value $L_{\mathrm{SUM}}(\mathcal{S})$ counts the chosen forest under the conditional cost of Definition 5.2.3, while Theorem 5.3.2 stores that forest with root dictionaries, explicit deletion labels, and tree indexes. The missing step is a representation whose size follows $L_{\mathrm{SUM}}(\mathcal{S})$ up to lower-order terms while keeping the same query times. This is the SUM analogue of the succinctness gap in the signed symmetric-difference setting, where the combinatorial description is smaller than the direct query overlay.

A separate question concerns how far we can move inside the lattice without losing a polynomial construction. SUM makes a local greedy choice among pairs of current roots. The appendix gives an exact exponential dynamic programme for the union-only binary restriction, and the full lattice model allows more labels than that restriction. Thus the remaining algorithmic problem is not only to improve the matching heuristic, but to understand which larger families of synthetic references admit polynomial-time optimisation, approximation guarantees, or hardness bounds. Until then, SUM is the polynomial slice whose query structure is fully explicit.

Large collections raise a more practical version of the same selection problem. Section 5.2 defines the score to use once two current roots are compared, but the sorted-set implementation compares every pair at every level and spends $O(m^2 N)$ time just to decide which edges deserve attention. The needed object is a sparse candidate graph that keeps most pairs with low $\Delta\Phi$ and discards pairs that are unlikely to improve the forest. Resemblance and containment sketches can provide such a filter [55], and locality sensitive hashing can group roots whose sketches make them plausible candidates [56]. SUM would then compute the exact $\Delta\Phi$ only on retained edges and run matching on that graph. This changes the search procedure, not the representation: every accepted edge still creates the exact union parent, and the resulting forest is still a deletion hierarchy. The level structure also supports incremental sketches, because the MinHash signature of $A \cup B$ is the componentwise minimum of the signatures of A and B. The missing analysis is to bound the loss caused by omitted edges, especially pairs with high containment but low resemblance.

[55]: Broder (1997), *On the resemblance and containment of documents*

[56]: Gionis et al. (1999), *Similarity search in high dimensions via hashing*

Taken together, the results point to one constraint: a compressed description must expose paths for the five operations, not only reduce the count. SUM shows that such paths can be created when the input does not supply them. The remaining questions ask whether this can be done with fewer bits, better references, and faster candidate selection.

Appendix

Lattice Models A

The atom bound $L^*(\mathcal{S})$ of Proposition 5.1.1 counts one global object. Once the realised membership patterns of Definition 5.1.1 and their multiplicities are fixed, the remaining freedom is the placement of those patterns on the labelled elements of U. The flat encoding of Proposition 5.1.4 can meet this count, but it stores the collection as a single pattern sequence. A hierarchy for the operations of Definition 1.2.2 must expose the same information through local parent-child descriptions, so it also has to choose reference labels, paths, and local decoding rules.

Set-Union Matching of Section 5.2 chooses one local route. It adds only the union $A \cup B$ of two current roots, and it decodes each child by a containment edge from that union parent. This keeps the construction local and gives the deletion paths used by the query structure. It also fixes two modelling choices in advance. Reference labels come only from repeated binary unions, and each local split has exactly two children.

This appendix keeps the containment requirement and removes those two restrictions one at a time. First we enlarge the family of candidate references. Unions supply common parents for incomparable labels, while intersections supply common lower references once the candidates are ordered by inclusion. Section A.1 closes $\mathcal{S}$ under these two operations and treats the result as a bounded inclusion lattice. Since every candidate is a union of atoms, the realised membership patterns index the closure by upsets of their inclusion order.

Then we vary arity. A parent with several children stores one local Venn table for all of them, instead of several marginal descriptions. With all m input sets sitting directly under U, this table is the atom table and the cost is $L^*(\mathcal{S})$. At the binary endpoint we recover the arity used by SUM, where every local decision has two children and the hierarchy remains compatible with pairwise containment paths.

A.1 Lattice closure

A reference hierarchy assigns each nonroot label a parent label that contains it. The input family $\mathcal{S}$ alone may contain no label that contains two incomparable labels. Unions create the smallest common parents for such labels, and intersections create common lower references once the candidates are ordered by inclusion. We therefore start from the smallest family containing $\mathcal{S}$ and closed under these two operations.

Definition A.1.1 (Lattice closure) *Let $\mathcal{S} = \{S_1, \ldots, S_m\}$ be a family of subsets of the finite universe U. The lattice closure $\mathcal{C}(\mathcal{S})$ is the smallest family $\mathcal{F} \subseteq 2^U$ such that $\mathcal{S} \subseteq \mathcal{F}$ and, whenever $A, B \in \mathcal{F}$, both $A \cap B$ and $A \cup B$ belong to $\mathcal{F}$.*

Containment is the comparison needed by a reference hierarchy. If $Q \subseteq P$, then a node labelled P can be a parent of a node labelled Q, and the edge

from P to Q has the deletion meaning used by the query structure. If two labels are incomparable, neither can be placed directly above the other. The closure records the two references associated with such a pair: $A \cup B$ is the smallest generated set that contains both, and $A \cap B$ is the largest generated set contained in both.

If S_1 and S_2 are disjoint and cover U, the diagram has the four nodes $\emptyset < S_1, S_2 < U$. If they overlap, $S_1 \cap S_2$ appears between $\emptyset$ and the two generators.

It is therefore natural to draw the generated references by containment. We place larger sets higher and smaller sets lower. The universe U contains every candidate, and $\emptyset$ is contained in every candidate. To make these two endpoints always available in the diagram, set $\overline{\mathscr{L}} = \mathscr{C}(\mathcal{S}) \cup \{U, \emptyset\}$. The input sets $S_1, \ldots, S_m$ appear as generator nodes between the endpoints, and each repeated union or intersection adds the reference forced by a common-parent or common-lower-reference relation.

For any two nodes $A, B \in \overline{\mathscr{L}}$, the diagram contains a smallest node above both of them and a largest node below both of them. These nodes are $A \cup B$ and $A \cap B$, respectively. In lattice terminology, they are the join and meet of A and B [57]. Since set union and intersection satisfy $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$ and $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$, the same identities hold inside $\overline{\mathscr{L}}$. Hence $\overline{\mathscr{L}}$ is a finite distributive lattice.

[57]: Ganter et al. (1999), *Formal concept analysis*

The bounded lattice and the reference labels differ only at the bottom. Under the convention $U = \bigcup_i S_i$ of Section 4.1, U already belongs to $\mathscr{C}(\mathcal{S})$. The empty set may or may not belong to $\mathscr{C}(\mathcal{S})$, and adding it lets intersections such as $A \cap B = \emptyset$ stay inside the lattice. We write $\mathscr{L} = \overline{\mathscr{L}} \setminus \{\emptyset\}$ for the nonempty reference candidates, because the empty set cannot act as a parent for a nonempty input set. We use $\overline{\mathscr{L}}$ for lattice statements and $\mathscr{L}$ for hierarchy labels.

A.1.1 Upset representation

To describe a closure reference without listing elements of U, we record which membership-pattern atoms it contains. By Definition 5.1.2, the nonempty sets A_τ partition U, and each generator S_i either contains all of A_τ or none of it. The same all or nothing property is preserved by intersection and union, so every set in $\overline{\mathscr{L}}$ is a union of whole atoms. Let

$$R = R(\mathcal{S}) = \{\, \sigma(x) : x \in U \,\} \subseteq 2^{[m]} \setminus \{\emptyset\},$$

be the realised membership patterns. We compare patterns by inclusion, because $\tau \subseteq \tau'$ means that every input set named by τ is also named by τ'. For every $Q \subseteq R$, define

$$X_Q = \bigcup_{\tau \in Q} A_\tau,$$

with $X_\emptyset = \emptyset$. Conversely, if $X \subseteq U$ is a union of atoms, set $Q_X = \{\tau \in R : A_\tau \subseteq X\}$. The atom partition gives $X = X_{Q_X}$, so subsets of R name references built as unions of atoms without listing their elements in U.

The indices of closure references are constrained by this comparison. For a generator S_i, the index contains exactly the patterns τ with $i \in \tau$. If it contains τ and $\tau \subseteq \tau'$, then it also contains τ', because $i \in \tau'$ as well. Thus generator indices, and the indices obtained from them by unions and intersections, are not arbitrary subsets of R. Once they contain a

pattern, they contain every realised pattern above it. We use the standard name for this closure property.

Definition A.1.2 (Upset) *An upset of an ordered set $(M, \leq)$ is a subset $Q \subseteq M$ such that $q \in Q$ and $q \leq q'$ imply $q' \in Q$.*

With this terminology, each generator S_i has the upset index $Q_i = \{\tau \in R : i \in \tau\}$. Intersections and unions of upsets are still upsets, so every set in $\mathscr{C}(\mathcal{S})$ has an upset index. The bottom $\emptyset$ has index $\emptyset$, and the top U has index R.

To count or list upset indices, it is enough to record where upward closure begins. In a finite upset, every element lies above at least one minimal element. Two distinct minimal elements cannot be comparable, because the larger one would not be minimal. We use the standard name for sets with this incomparability property.

Definition A.1.3 (Antichain [57]) *An antichain of an ordered set $(M, \leq)$ is a subset $A \subseteq M$ in which any two distinct elements are incomparable.*

Given a set of starting patterns, we need notation for all realised patterns forced above them. For one pattern $a \in R$, write $\uparrow a$ for $\{\tau \in R : a \subseteq \tau\}$. For a set $A \subseteq R$, take all such upper sets at once and write $\uparrow A = \{\tau \in R : \tau \supseteq a \text{ for some } a \in A\}$. The empty choice generates no pattern, so $\uparrow \emptyset = \emptyset$.

If Q is an upset of $(R, \subseteq)$, then the elements of $\min(Q)$ are exactly the elements of Q with no smaller element of Q below them. Since R is finite, every element of Q lies above at least one element of $\min(Q)$, and the elements of $\min(Q)$ are pairwise incomparable. Thus $\min(Q)$ is an antichain and $Q = \uparrow \min(Q)$. Conversely, if A is an antichain of R, then $\uparrow A$ is an upset and $\min(\uparrow A) = A$. The maps $Q \mapsto \min(Q)$ and $A \mapsto \uparrow A$ are mutual inverses, so antichains of R index upsets of R bijectively.

We write $J(R)$ for the total number of upsets of R, including both $\emptyset$ and R. Every closure reference has an upset index. The reverse direction is also available: every upset can be built from the generators by intersecting the sets named in each minimal pattern and then taking their union. The next theorem records the exact correspondence.

Theorem A.1.1 (Closure representation) *The map $Q \mapsto X_Q$ is a lattice isomorphism between the upsets of $(R, \subseteq)$ and $\overline{\mathscr{L}}$, where both lattices use intersection as meet and union as join. It sends the empty upset to $\emptyset$. Restricted to nonempty upsets, it is a poset isomorphism onto $\mathscr{L}$. In particular $|\overline{\mathscr{L}}| = J(R)$ and $|\mathscr{L}| = J(R) - 1$.*

Proof. Since the atoms are pairwise disjoint, for any $Q_1, Q_2 \subseteq R$ we have

$$X_{Q_1} \cap X_{Q_2} = X_{Q_1 \cap Q_2}, \qquad X_{Q_1} \cup X_{Q_2} = X_{Q_1 \cup Q_2}.$$

Both the intersection and the union of upsets are upsets, so $Q \mapsto X_Q$ preserves meet and join on the lattice of upsets.

The image is exactly $\overline{\mathscr{L}}$. It contains $\emptyset = X_\emptyset$, $U = X_R$, and each generator $S_i = X_{Q_i}$, where $Q_i = \{\tau \in R : i \in \tau\}$. The identities above make the

image closed under union and intersection, so it contains $\overline{\mathscr{L}}$. For the reverse inclusion, let Q be an upset. If $Q = \emptyset$, then $X_Q = \emptyset$. Otherwise,

$$X_Q = \bigcup_{a \in \min(Q)} X_{\uparrow a} = \bigcup_{a \in \min(Q)} \bigcap_{i \in a} S_i,$$

because $x \in X_{\uparrow a}$ iff $\sigma(x) \supseteq a$ iff $x \in S_i$ for every $i \in a$. The right side is built from $\mathcal{S}$ by intersections and unions, so $X_Q \in \mathscr{C}(\mathcal{S}) \subseteq \overline{\mathscr{L}}$.

The map is injective. If $X_{Q_1} = X_{Q_2}$, then for every $\tau \in R$, $A_\tau \subseteq X_{Q_1}$ iff $\tau \in Q_1$, and $A_\tau \subseteq X_{Q_2}$ iff $\tau \in Q_2$. Thus $Q_1 = Q_2$. The empty upset is the only upset mapped to $\emptyset$, so restricting the isomorphism to nonempty upsets gives a poset isomorphism onto $\mathscr{L}$. The cardinality identities follow. □

Theorem A.1.1 reduces the size of the closure to the number of antichains of the realised pattern poset. Exact counting would require counting all antichains of R. We use two estimates instead. The first forgets the realised set R and compares it with the full Boolean lattice. The second keeps one parameter of R, its largest antichain.

Definition A.1.4 (Dedekind number) *For $m \geq 0$, the Dedekind number $D(m)$ is the number of antichains of the Boolean lattice $2^{[m]}$ ordered by inclusion. The empty antichain is counted.*

Kleitman studies this count in the form of Dedekind's problem and proves that its base two logarithm is asymptotic to the middle level of the Boolean lattice [58],

[58]: Kleitman (1969), *On Dedekind's problem: the number of monotone Boolean functions*

$$\lg D(m) \sim \binom{m}{\lfloor m/2 \rfloor}.$$

When every nonempty membership pattern is realised, this is the Boolean-lattice count against which no realised pattern set can be larger. If some patterns are absent, R supports fewer antichains, so the Dedekind number gives only the worst case.

Every antichain of R is an antichain of $2^{[m]} \setminus \{\emptyset\}$. The empty pattern is comparable with every other pattern, so the only antichain of $2^{[m]}$ that contains it is the singleton $\{\emptyset\}$. The nonempty part therefore has $D(m) - 1$ antichains. Hence

$$|\overline{\mathscr{L}}| = J(R) \leq D(m) - 1, \qquad |\mathscr{L}| \leq D(m) - 2.$$

Equality in the first inequality forces every singleton antichain $\{\tau\}$, with $\emptyset \neq \tau \subseteq [m]$, to come from R. Hence equality occurs only when $R = 2^{[m]} \setminus \{\emptyset\}$, the regime in which every nonempty Venn region is realised.

The Dedekind comparison ignores the shape of the realised pattern set. If R has no large family of pairwise incomparable patterns, then no antichain of R can be large. We name the maximum possible size of such a family.

Definition A.1.5 (Width) *The width of a finite ordered set $(M, \leq)$ is the maximum cardinality of an antichain in M.*

Let w be the width of $(R, \subseteq)$. Every antichain of R has size at most w, so the number of antichains of R is at most the number of subsets of R of size at most w. Thus

$$|\overline{\mathscr{L}}| \;=\; J(R) \;\le\; \sum_{j=0}^{w} \binom{|R|}{j}, \qquad |\mathscr{L}| \;\le\; \sum_{j=0}^{w} \binom{|R|}{j} - 1.$$

This estimate uses the realised order rather than the full Boolean lattice. If R has small width, few subsets of R can be antichains, and the lattice closure is correspondingly smaller.

These bounds count candidate reference labels. A hierarchy still has to choose edges among those labels and decide how many children each local encoding may combine. The next step is to make that arity choice explicit.

A.1.2 Arity chain

The closure gives many possible references, but a representation still has to connect them. A root reference is stored as a subset of U. Every lower reference is recovered from a parent by describing which part of the parent belongs to each child. If one parent has several children, one local table can describe their joint membership inside the parent. The number of children allowed in such a table is the arity parameter.

Fix a parent P and children $Q_1, \dots, Q_r$ with $Q_j \subseteq P$. Each element of P has a binary membership pattern across the children. The vector of counts over $\{0,1\}^r$ is the local Venn data of the split. Encoding this vector as one multinomial can be cheaper than encoding the r children independently as r marginal subsets of P.

The forest stores occurrences rather than unique set labels. Thus two different child occurrences may carry the same set label, and a child may carry the parent label when the containment is not strict. The local split primitive must accept both cases and must also say which joint Venn table is charged for that split.

Definition A.1.6 (Reference hyperarc) *A reference hyperarc is a local split written as a tuple*

$$e = (P; Q_1, \dots, Q_r)$$

with $P, Q_1, \dots, Q_r \in \mathscr{L}$, $r \ge 2$, and $Q_j \subseteq P$ for every j. We call P the parent label, $Q_1, \dots, Q_r$ the child labels, and r the arity of e. The displayed order of the child labels fixes the coordinates of the Venn table. Permuting the children only permutes those coordinates and does not change the cost. The child labels need not be distinct. We write $P_e = P$, $Q_j^e = Q_j$, and $r_e = r$.

For each bit vector $b = (b_1, \dots, b_{r_e}) \in \{0,1\}^{r_e}$, the local Venn count of b in e is

$$a_b^e \;=\; \big|\{\, x \in P_e : \mathbf{1}[x \in Q_j^e] = b_j \text{ for every } j \in [r_e] \,\}\big|.$$

These counts sum to $|P_e|$, since every element of P_e has exactly one membership vector across the child occurrences. The all-zero vector counts the part of the

parent that belongs to no child. The local cost of e is

$$\operatorname{cost}(e) \;=\; \lg \binom{|P_e|}{(a_b^e)_{b\in\{0,1\}^{r_e}}}.$$

It is the logarithm of the number of assignments of membership vectors to the elements of P_e with these fixed counts.

A reference hierarchy uses one such hyperarc at every internal occurrence. It is a rooted forest of occurrences, as in Definition 5.2.2, but an internal occurrence may have arity greater than two. Its label is any reference in $\mathscr{L}$ containing all child labels. If an occurrence labelled P has children labelled $Q_1, \ldots, Q_r$, then those labels form a reference hyperarc with parent P. Since $\mathscr{L}$ excludes $\emptyset$, the optimisation model below keeps only nonempty distinguished input labels. Empty input sets have zero independent support cost and are answered directly by Section 5.3, so they can be omitted from the hierarchy.

Definition A.1.7 (Bounded-arity reference hyperforest) *For $2 \le d \le m$, a d-ary reference hyperforest for $\mathcal{S}$ on $\mathscr{L}$ is a rooted forest $\mathcal{T}$ of occurrences with labels in $\mathscr{L}$. Each internal occurrence has between 2 and d children, and its label together with the labels of its children forms a reference hyperarc. For every input index i, there is a distinguished leaf occurrence a_i with label S_i. These leaf occurrences are reachable from the component roots and remain distinct even when two input sets are equal.*

The cost of $\mathcal{T}$ is

$$\operatorname{cost}(\mathcal{T}) \;=\; \sum_{\rho\in\operatorname{roots}(\mathcal{T})} \lg \binom{u}{|P_\rho|} \;+\; \sum_{e\in E(\mathcal{T})} \operatorname{cost}(e),$$

where P_ρ is the label of the root occurrence ρ and $E(\mathcal{T})$ is the multiset of reference hyperarcs induced by the internal occurrences of $\mathcal{T}$, with one hyperarc for each internal occurrence.

The definition deliberately permits child labels that coincide with the parent or with one another. This keeps the occurrence forest stable when set labels collapse. The cost still has the lower dimensional form obtained by deleting redundant coordinates. If $Q_j = P$ for some j, every element of P has indicator 1 in coordinate j, so a_b^e vanishes whenever $b_j = 0$, and the multinomial reduces to the multinomial of the remaining children inside P. If $Q_i = Q_j$ for two indices $i \neq j$, the two coordinates carry identical indicators on every element of P, and the multinomial reduces to the multinomial obtained by identifying coordinates i and j.

SUM already creates this situation. When a merge of Definition 5.2.2 combines labels $P_a \subseteq P_b$, the union label is P_b, so one child has the same label as the parent. The hyperarc view still charges the same local multinomial, because the coordinate belonging to that child contributes no choices. Repeated child labels are handled in the same way. Two occurrences may occupy different positions in the forest while carrying the same set label, and the local multinomial identifies their coordinates.

For $2 \le d \le m$, write $\mathscr{H}_d(\mathcal{S})$ for the family of all reference hyperforests for $\mathcal{S}$ on $\mathscr{L}$ whose internal occurrences have at most d children. This notation separates the arity constraint from the choices still left to the

representation. Once d is fixed, the hierarchy may still choose its roots, its intermediate references, repeated occurrence labels, and the shape of the occurrence forest. To compare arity bounds, we assign to each d the least cost achievable over all those choices.

Definition A.1.8 (Bounded-arity optimum) *For every $d \in \{2, 3, \dots, m\}$,*

$$\mathrm{OPT}_d(\mathcal{S}) \;=\; \min_{\mathcal{T} \in \mathcal{H}_d(\mathcal{S})} \mathrm{cost}(\mathcal{T}).$$

The arity range has two feasible endpoints. The trivial hyperforest has m singleton root occurrences labelled $S_1, \dots, S_m$ and no hyperarcs. It is feasible for every $d \geq 2$ and has cost $\sum_i \lg \binom{u}{|S_i|} = H_{wc}(\mathcal{S})$, the per-set baseline of Chapter 4. At the other end, the one-component hyperforest with root label U and a single m-ary hyperarc $U \to (S_1, \dots, S_m)$ is feasible at $d = m$.

At these endpoints, singleton roots and the one-root hyperforest charge different local objects. Encoding $S_1, \dots, S_m$ as singleton roots charges m independent binomials against U. Encoding them as siblings of one root charges one m-way Venn multinomial inside U. To use the same comparison at an arbitrary hyperarc, we compare one joint Venn table inside a parent with the independent marginal descriptions of the children. The multinomial term is the fixed-count type-class count used in enumerative coding [53]. Once the joint Venn labelling of the children is known, each marginal child subset is known, so the joint count cannot exceed the product of the marginal counts.

[53]: Cover et al. (2006), *Elements of Information Theory*

Lemma A.1.2 (Joint Venn bound) *Let $r \geq 1$, let P be a finite set, and let $Q_1, \dots, Q_r \subseteq P$. For $b \in \{0,1\}^r$, let*

$$a_b \;=\; \big|\{\, x \in P : \mathbf{1}[x \in Q_j] = b_j \text{ for all } j \in [r] \,\}\big|.$$

The counts $(a_b)_{b \in \{0,1\}^r}$ sum to $|P|$, with zero parts allowed. Then

$$\lg \binom{|P|}{(a_b)_{b \in \{0,1\}^r}} \;\leq\; \sum_{j=1}^{r} \lg \binom{|P|}{|Q_j|}.$$

Proof. The multinomial $\binom{|P|}{(a_b)}$ counts the labellings $\ell \colon P \to \{0,1\}^r$ whose joint counts are exactly (a_b). Each such labelling projects to r binary labellings $\ell_j \colon P \to \{0,1\}$, with ℓ_j carrying $|Q_j| = \sum_{b : b_j = 1} a_b$ ones. The tuple $(\ell_1, \dots, \ell_r)$ determines ℓ pointwise, so the projection map $\ell \mapsto (\ell_1, \dots, \ell_r)$ is injective. Its codomain is the set of all tuples of binary labellings whose j-th coordinate has $|Q_j|$ ones. This codomain has cardinality $\prod_j \binom{|P|}{|Q_j|}$, hence

$$\binom{|P|}{(a_b)} \;\leq\; \prod_{j=1}^{r} \binom{|P|}{|Q_j|}.$$

□

Lemma A.1.2 is the local inequality that lets a hierarchy replace separate sibling descriptions by one joint table without increasing the counting term. At arity m, the one-level hyperforest in $\mathcal{H}_m(\mathcal{S})$ has one local Venn

table over all input sets. Under the coverage convention, this table is the atom-count table of Proposition 5.1.1.

Theorem A.1.3 (Atom endpoint) *Under the convention $U = \bigcup_i S_i$, the one-level m-ary hyperforest with root hyperarc $U \to (S_1, \dots, S_m)$ has cost*

$$\lg \binom{u}{(c_\tau)_{\tau \in R}} = L^*(\mathcal{S}).$$

Proof. The hyperforest has root U and one hyperarc $U \to (S_1, \dots, S_m)$. The root support cost is $\lg \binom{u}{u} = 0$. By Definition A.1.6, the hyperarc cost is $\lg \binom{u}{(a_b)_{b \in \{0,1\}^m}}$ with $a_b = |\{x \in U : (\mathbf{1}[x \in S_j])_{j=1}^m = b\}|$. Identifying $b \in \{0,1\}^m$ with $\tau = \{j : b_j = 1\} \subseteq [m]$ gives $a_b = c_\tau$, the atom count of Definition 5.1.2. The coverage convention $U = \bigcup_i S_i$ forces $c_\emptyset = 0$, so the multinomial reduces to $\binom{u}{(c_\tau)_{\tau \in R}}$, and $L^*(\mathcal{S}) = \lg \binom{u}{(c_\tau)_{\tau \in R}}$ by Proposition 5.1.1. □

Increasing d enlarges the feasible family, so the optimum cannot increase. The endpoint $d = m$ reaches the atom count by Theorem A.1.3. To prove the chain, it remains to show that no bounded-arity hyperforest, even with arbitrary references from $\mathcal{L}$, can beat the atom type-class count.

Theorem A.1.4 (Arity chain) *Assume $m \geq 2$. For every $d \in \{2, 3, \dots, m\}$,*

$$L^*(\mathcal{S}) = \mathrm{OPT}_m(\mathcal{S}) \leq \mathrm{OPT}_d(\mathcal{S}) \leq \mathrm{OPT}_2(\mathcal{S}) \leq H_{wc}(\mathcal{S}).$$

Proof. The trivial hyperforest with singleton roots $S_1, \dots, S_m$ is feasible at $d = 2$ with cost $H_{wc}(\mathcal{S})$, giving the rightmost inequality. The inclusions

$$\mathscr{H}_2(\mathcal{S}) \subseteq \mathscr{H}_3(\mathcal{S}) \subseteq \cdots \subseteq \mathscr{H}_m(\mathcal{S})$$

give the middle inequalities, since every d-ary hyperforest is also feasible when the arity bound is larger. By Theorem A.1.3, $\mathrm{OPT}_m(\mathcal{S}) \leq L^*(\mathcal{S})$. It remains to prove $L^*(\mathcal{S}) \leq \mathrm{OPT}_d(\mathcal{S})$ for every d.

Fix $\mathcal{T} \in \mathscr{H}_d(\mathcal{S})$. By Theorem A.1.1, each occurrence label is X_B for a nonempty upset $B \subseteq R$. For any collection $\mathcal{S}'$ in the atom type class $\mathscr{C}(R, (c_\tau))$, replace X_B by the union X'_B of the atoms of $\mathcal{S}'$ whose patterns lie in B. Containments are preserved because they are inclusions of upset indices. Root sizes and local Venn counts are sums of the fixed values c_τ over sets of patterns determined by $\mathcal{T}$.

Record each root subset and, for every internal occurrence, the Venn labelling of its parent set across its child occurrences. These records reconstruct all descendants from the roots, and hence reconstruct the distinguished leaves $S'_1, \dots, S'_m$. Distinct collections in the atom type class therefore give distinct records. The number of valid records is at most

$$\prod_{\rho \in \mathrm{roots}(\mathcal{T})} \binom{u}{|P_\rho|} \cdot \prod_{e \in E(\mathcal{T})} \binom{|P_e|}{(a_b^e)_b} = 2^{\mathrm{cost}(\mathcal{T})}.$$

The type-class count of Proposition 5.1.1 gives $|\mathscr{C}(R, (c_\tau))| = \binom{u}{(c_\tau)_{\tau \in R}} = 2^{L^*(\mathcal{S})}$. Thus $L^*(\mathcal{S}) \leq \mathrm{cost}(\mathcal{T})$, and minimising over $\mathcal{T}$ gives $L^*(\mathcal{S}) \leq \mathrm{OPT}_d(\mathcal{S})$. □

At $d = 2$, every local split is pairwise. This is the most restrictive endpoint of the model and the one compatible with SUM, whose query structure follows pairwise containment edges. We now turn from the arity comparison to $\mathrm{OPT}_2(\mathcal{S})$ itself.

A.2 Binary case

We now specialise the arity chain to $d = 2$. This endpoint is closest to SUM: each internal occurrence has two children, SUM chooses pairs of current roots, and the query structure of Section 5.3 follows pairwise containment edges. The binary case is still not just SUM, because OPT_2 may choose any reference from $\mathcal{L}$ as a binary parent. We study it separately to isolate the part of the problem that remains when local decisions are pairwise but reference labels are not fixed by the SUM rule.

A binary hyperarc has the form $(P; A, B)$ with $A, B \subseteq P$. To decode both children from the parent, we partition P according to membership in A and B. Let $k = |A \cap B|$, $l = |A \setminus B|$, $r = |B \setminus A|$, and $a_{(0,0)} = |P \setminus (A \cup B)|$. By Definition A.1.6, the cost of this split is

$$\lg \binom{|P|}{a_{(0,0)}, k, l, r},$$

and $\mathrm{OPT}_2(\mathcal{S})$ minimises the sum of these split costs together with the root support costs $\lg \binom{u}{|P_\rho|}$.

If we ignore the coupling between siblings and pay for containments one at a time, the containment order on $\mathcal{L}$ becomes a directed acyclic graph. Its vertices are the nonempty references, and it has an arc $P \to Q$ when Q is a strict subset of P. Each arc is then a possible one-child containment move. A one-root structure that starts at U and reaches $S_1, \ldots, S_m$ has the form of a rooted connection problem on this graph.

If each containment arc could be paid for independently, we would put root U at the top, mark $S_1, \ldots, S_m$ as terminals, and give each arc $P \to Q$ the cost $\lg \binom{|P|}{|Q|}$. Under this independent-arc objective, a feasible solution is a directed arborescence that starts at U and reaches every terminal, which is a Directed Steiner Tree instance on the containment graph [59].

[59]: Charikar et al. (1999), *Approximation algorithms for directed Steiner problems*

The independent arc model separates the choices below a parent. If an arborescence contains one outgoing arc from P, it still treats every other outgoing arc as a separate purchase. A binary hyperarc behaves differently. Choosing $(P; A, B)$ stores one Venn table for the pair inside P. That table has entries for the four regions determined by both children, so it has no standalone price for only the branch $P \to A$ or only the branch $P \to B$. Both children are part of the same split. The Directed Steiner instance above captures only reachability from U to the input sets through containments. The objective of OPT_2 is different, because each local move buys one two-child split with one joint Venn cost. We therefore keep the hyperforest formulation for the binary model.

The comparison with Directed Steiner used a single root U. For the actual binary objective, we first show that the number of component roots can also be removed as a choice. Each component root pays its own support

cost. If two roots carry labels A and B, the closure also contains their union $M = A \cup B$. We can create a new root labelled M and attach the old roots below it with the binary hyperarc $(M; A, B)$. This move replaces two independent root descriptions with one root description and one joint Venn description of A and B inside M. The lemma proves that this replacement never increases the cost.

Lemma A.2.1 (Root merging) *Let $A, B \subseteq U$, let $M = A \cup B$, and set $k = |A \cap B|$, $l = |A \setminus B|$, and $r = |B \setminus A|$. Then*

$$\lg \binom{u}{|M|} + \lg \binom{|M|}{k, l, r} \;\le\; \lg \binom{u}{|A|} + \lg \binom{u}{|B|}.$$

Proof. The multinomial chain identity gives

$$\binom{u}{|M|}\binom{|M|}{k, l, r} \;=\; \binom{u}{u - |M|, k, l, r}.$$

The tuple $(u - |M|, k, l, r)$ is the Venn count of A and B inside U. Applying Lemma A.1.2 of Section A.1 with parent U and children A, B gives $\binom{u}{u-|M|,k,l,r} \le \binom{u}{|A|}\binom{u}{|B|}$. Taking logarithms proves the claim. □

We can now merge component roots one pair at a time. Pick two roots with labels A and B, insert a new root with label $M = A \cup B$, and make the old roots its two children. Since $\mathcal{L}$ is closed under union, M is an admissible reference. The move is valid even when $A \subseteq B$ or $B \subseteq A$, because Definition A.1.6 permits a child label to coincide with its parent. Existing descendant costs do not change, and Lemma A.2.1 shows that the new root charge plus the new split cost is no larger than the two old root charges. Repeating the operation preserves the invariant that every distinguished leaf lies below a current root, and gives a binary hyperforest with one root and no larger cost. Under the coverage convention $U = \bigcup_i S_i$, that root is labelled U, since every element of U appears in some distinguished leaf and all labels lie inside U.

A.2.1 Union-only dynamic programme

Lemma A.2.1 lets us merge all components and study one rooted binary tree. The full OPT_2 model still asks us to choose a reference at every internal node. Suppose the two child subtrees use references A and B. We may choose as parent any reference $P \in \mathcal{L}$ with $A \cup B \subseteq P$, and this choice changes the residue part of the Venn table inside P. Searching the full model therefore means choosing both the binary shape and the labels from the lattice closure.

To get a dynamic programme over input subsets, we now impose a stronger rule that removes the label choice. Every occurrence must lie above at least one distinguished leaf. For an occurrence v, let

$$X_v \;=\; \{\, i \in [m] : a_i \text{ lies below } v \,\}$$

be its nonempty descendant input set. In the union-only restriction, this set determines the occurrence label. Once $X_v = X$, the label of v is forced

to be

$$U_X = \bigcup_{i \in X} S_i.$$

The remaining search chooses only how the input indices are split across the binary tree.

Let $\mathscr{H}_2^{\cup}(\mathcal{S})$ be the family of binary reference hyperforests on $\mathcal{L}$ in which every occurrence v has a nonempty descendant input set $X_v \subseteq [m]$ and label U_{X_v}. We define

$$\mathrm{OPT}_2^{\cup}(\mathcal{S}) = \min_{\mathcal{T} \in \mathscr{H}_2^{\cup}(\mathcal{S})} \mathrm{cost}(\mathcal{T}),$$

and the inclusion $\mathscr{H}_2^{\cup}(\mathcal{S}) \subseteq \mathscr{H}_2(\mathcal{S})$ gives $\mathrm{OPT}_2(\mathcal{S}) \le \mathrm{OPT}_2^{\cup}(\mathcal{S})$. The restriction may exclude feasible binary labels, because an occurrence with descendant set X_v is forced to use U_{X_v} even though the full model may choose any reference in $\mathcal{L}$ that contains its child labels. By Lemma A.2.1, we may restrict the minimum in $\mathscr{H}_2^{\cup}(\mathcal{S})$ to one binary tree rooted at $U_{[m]}$.

At a node whose descendant input set is X, a binary union tree has no choice when X is a singleton. The node is one of the required leaves and has no split cost. If $|X| \ge 2$, the first decision below that node is an unordered bipartition of X into two nonempty parts. Write one part as Y and the other as $X \setminus Y$. The union-only rule fixes the two child labels as U_Y and $U_{X \setminus Y}$. After paying the Venn cost of this split, the left subtree depends only on Y, and the right subtree depends only on $X \setminus Y$. The same problem has reappeared on two smaller input subsets, so we compute the optimum values for smaller subsets first and reuse them.

To turn this recursive description into a cost recurrence, we separate the cost of storing a root from the cost of splitting an internal node. If a union tree on input indices $X \subseteq [m]$ were stored as a component root, it would pay the support cost

$$\rho(X) = \lg \binom{u}{|U_X|}.$$

In the one-component optimum, this cost is paid only for $X = [m]$. If a split sends the nonempty disjoint sets Y and Z to the two children, the parent label is $U_{Y \cup Z}$ and the child labels are U_Y and U_Z. Let

$$\mu(Y, Z) = \lg \binom{|U_{Y \cup Z}|}{k, l, r},$$

where $k = |U_Y \cap U_Z|$, $l = |U_Y \setminus U_Z|$, and $r = |U_Z \setminus U_Y|$. Since $U_{Y \cup Z} = U_Y \cup U_Z$, every element of the parent lies in at least one child. The residue count $a_{(0,0)}$ of Definition A.1.6 is zero, and $\mu(Y, Z)$ is exactly the cost of the binary hyperarc $U_{Y \cup Z} \to (U_Y, U_Z)$.

Because only the whole root pays support in the one-component optimum, the subproblem value should not charge the root support cost at every level. We let $B(X)$ count only the internal split cost of the best binary union tree whose distinguished leaves are exactly the input indices in X. For a singleton, no internal split is needed, so $B(\{i\}) = 0$. For $|X| \ge 2$, a top split chooses a nonempty proper subset $Y \subsetneq X$ and puts $X \setminus Y$ on the other side. That choice contributes the two smaller split costs plus

the local cost $\mu(Y, X \setminus Y)$, so

$$B(X) \;=\; \min_{\substack{\emptyset \neq Y \subsetneq X \\ \min(X) \in Y}} \big[\, B(Y) + B(X \setminus Y) + \mu(Y, X \setminus Y) \,\big].$$

The constraint $\min(X) \in Y$ selects one representative of the symmetric split $(Y, X \setminus Y)$. Only the root of the whole tree pays a support cost, so the minimum union-only cost is $\rho([m]) + B([m])$. Under the coverage convention, $\rho([m]) = \lg \binom{u}{u} = 0$.

The recurrence for $B(X)$ refers only to proper subsets of X. We can therefore evaluate it by increasing subset size. Algorithm 8 first computes every union U_X, since each evaluation of $\mu(Y, Z)$ needs the three sets U_Y, U_Z, and $U_{Y \cup Z}$. It then initializes the singleton values and fills larger subsets. After finishing all subsets of size s, it has computed the correct value $B(X)$ for every nonempty X with $|X| \leq s$.

Algorithm 8: Computing $\mathrm{OPT}_2^{\cup}$.

Input: Collection $\mathcal{S} = \{S_1, \ldots, S_m\}$ of nonempty sets over universe U.
Output: The value $\mathrm{OPT}_2^{\cup}(\mathcal{S})$.

```
compute U_X = ⋃_{i∈X} S_i for every X ⊆ [m];
foreach i ∈ [m] do
    B({i}) ← 0;
for s = 2, 3, ..., m do
    foreach X ⊆ [m] with |X| = s do
        B(X) ← +∞;
        foreach Y ⊊ X with Y ≠ ∅ and min(X) ∈ Y do
            Z ← X \ Y;
            compute μ(Y, Z) from U_Y, U_Z, and U_X;
            B(X) ← min{B(X), B(Y) + B(Z) + μ(Y, Z)};
return ρ([m]) + B([m]);
```

For a subset X, the loop tries every admissible first split and combines the local split cost with the best values already computed for the two sides. The table therefore represents all binary union trees through their recursive top splits, without listing the trees explicitly. The proposition turns this invariant into the exact value of $\mathrm{OPT}_2^{\cup}$ and bounds the split evaluations needed to compute it.

Proposition A.2.2 (Subset dynamic programme) *For every collection $\mathcal{S}$ of $m \geq 2$ nonempty sets over a universe of size u,*

$$\mathrm{OPT}_2^{\cup}(\mathcal{S}) \;=\; \rho([m]) + B([m]).$$

The dynamic programme in Algorithm 8 computes this value. If each U_X is stored as a characteristic bitvector of length u, then the bitset work is $O(3^m \cdot u/\omega)$ word operations after precomputing the unions $\{U_X : X \subseteq [m]\}$ in $O(2^m \cdot u/\omega)$ word operations on a word RAM with word size ω. This count treats arithmetic on the resulting cost values as unit-cost.

Proof. Any union-only optimum can be taken to have one component. If two roots have disjoint descendant index sets X and Y, then their

labels are U_X and U_Y. Replacing them by a new root labelled $U_{X\cup Y}$ with children U_X and U_Y remains in $\mathscr{H}_2^{\cup}(\mathcal{S})$, preserves all descendant costs, and does not increase the root-plus-split cost by Lemma A.2.1. Repeating the replacement leaves one tree rooted at $U_{[m]}$.

We prove by induction on $|X|$ that $B(X)$ is the minimum internal split cost of a binary union tree with leaf set X. The singleton case is $B(\{i\}) = 0$. For $|X| \geq 2$, every valid tree has a top split into two nonempty parts. Orient that bipartition as Y and $X \setminus Y$ with $\min(X) \in Y$. The union-only rule fixes the corresponding hyperarc cost to $\mu(Y, X \setminus Y)$, and the induction hypothesis gives a lower bound $B(Y) + B(X \setminus Y) + \mu(Y, X \setminus Y)$. Conversely, any subset Y considered by the recurrence can be realised by joining optimal child trees for Y and $X \setminus Y$ below the hyperarc $U_X \to (U_Y, U_{X\setminus Y})$. The condition $\min(X) \in Y$ keeps exactly one orientation of each unordered bipartition. Thus the recurrence is exact for every X, and the one-root optimum is $\rho([m]) + B([m])$. The algorithm evaluates this recurrence in increasing subset size, so every referenced proper subset has already been computed.

A subset X has at most $2^{|X|-1}$ candidates satisfying $\min(X) \in Y$. Summing this upper bound over all nonempty subsets gives

$$\sum_{j=1}^{m} \binom{m}{j} 2^{j-1} \;=\; \frac{3^m - 1}{2} \;=\; O(3^m).$$

The unions U_X are precomputed by iterating over the subset lattice and applying $U_X = U_{X\setminus\{i\}} \cup S_i$ for any $i \in X$, in $O(u/\omega)$ word operations per subset and $O(2^m \cdot u/\omega)$ word operations in total. Each evaluation of $\mu(Y, Z)$ scans the bitvectors for U_Y, U_Z, and $U_{Y\cup Z}$ to count the three Venn regions in $O(u/\omega)$ word operations. The stated bound counts these scans and treats arithmetic on the resulting cost values as unit-cost. □

The result is best read as an exact baseline for small values of m. Once m is fixed, the dependence on the universe is polynomial through the bitset operations on the precomputed unions, while the dynamic programme evaluates $O(3^m)$ split candidates. It optimises a restricted reference-hyperarc model whose internal labels are the union labels carried by SUM nodes. It does not optimise SUM itself, because SUM also fixes a greedy level rule and uses the header terms of Definition 5.2.1. The dynamic programme does not give a polynomial-time algorithm in the full input size. It also leaves the full OPT_2 separate, since that problem may choose internal labels anywhere in $\mathscr{L}$.

Bibliography

[1] Alessio Conte et al. 'From Strings to Graphs, and Back Again: A Festschrift for Roberto Grossi's 60th Birthday'. In: (2025) (cited on page iii).

[2] Jarno N Alanko et al. 'Compact data structures for collections of sets'. In: *23rd Symposium on Experimental Algorithms*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik. 2025, p. 6 (cited on pages iii, 3, 4, 53–57, 88).

[3] Luca Lombardo. 'Efficient Succinct Data Structures on Directed Acyclic Graphs'. Bachelor's Thesis. University of Pisa, May 2025 (cited on page iii).

[4] Ming Li, Paul Vitányi, et al. *An introduction to Kolmogorov complexity and its applications*. Vol. 3. Springer, 2008 (cited on page 1).

[5] Paolo Boldi and Sebastiano Vigna. 'The webgraph framework I: compression techniques'. In: *Proceedings of the 13th international conference on World Wide Web*. 2004, pp. 595–602 (cited on pages 2, 61).

[6] Giulio Ermanno Pibiri and Rossano Venturini. 'Techniques for inverted index compression'. In: *ACM Computing Surveys (CSUR)* 53.6 (2020), pp. 1–36 (cited on page 2).

[7] Jarno N Alanko et al. 'Themisto: a scalable colored k-mer index for sensitive pseudoalignment against hundreds of thousands of bacterial genomes'. In: *Bioinformatics* 39.Supplement_1 (2023), pp. i260–i269 (cited on page 2).

[8] Fatemeh Almodaresi et al. 'An efficient, scalable, and exact representation of high-dimensional color information enabled using de Bruijn graph search'. In: *Journal of Computational Biology* 27.4 (2020), pp. 485–499 (cited on pages 2, 61).

[9] Travis Gagie, Meng He, and Gonzalo Navarro. 'Compressed Set Representations based on Set Difference'. In: *arXiv preprint arXiv:2601.23240* (2026) (cited on pages 3, 4, 43, 47, 49, 53, 54, 57–60, 62–68, 87, 88).

[10] Guy Joseph Jacobson. *Succinct static data structures*. Carnegie Mellon University, 1988 (cited on pages 7, 33).

[11] Guy Jacobson. 'Space-efficient static trees and graphs'. In: *30th annual symposium on foundations of computer science*. IEEE. 1989, pp. 549–554 (cited on pages 7, 10, 28, 29, 33).

[12] David Clark. 'Compact pat trees'. In: (1997) (cited on pages 7, 9, 10).

[13] Peter Miltersen. 'Lower bounds on the size of selection and rank indexes.' In: Jan. 2005, pp. 11–12 (cited on pages 11, 12).

[14] Alexander Golynski. 'Optimal lower bounds for rank and select indexes'. In: *Theoretical Computer Science* 387.3 (2007), pp. 348–359 (cited on pages 12, 13).

[15] Paul Beame and Faith E Fich. 'Optimal bounds for the predecessor problem and related problems'. In: *Journal of Computer and System Sciences* 65.1 (2002), pp. 38–72 (cited on page 13).

[16] Mihai Patrascu. 'Succincter'. In: *2008 49th Annual IEEE Symposium on Foundations of Computer Science*. IEEE. 2008, pp. 305–313 (cited on pages 13, 14).

[17] Roberto Grossi et al. 'More haste, less waste: Lowering the redundancy in fully indexable dictionaries'. In: *arXiv preprint arXiv:0902.2648* (2009) (cited on page 13).

[18] Venkatesh Raman and S Srinivasa Rao. 'Static dictionaries supporting rank'. In: *International Symposium on Algorithms and Computation*. Springer. 1999, pp. 18–26 (cited on page 14).

[19] Rajeev Raman, Venkatesh Raman, and Srinivasa Rao Satti. 'Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets'. In: *ACM Transactions on Algorithms (TALG)* 3.4 (2007), 43–es (cited on pages 14, 15).

[20] Michael L Fredman, János Komlós, and Endre Szemerédi. 'Storing a sparse table with 0 (1) worst case access time'. In: *Journal of the ACM (JACM)* 31.3 (1984), pp. 538–544 (cited on page 14).

[21] Peter Elias. 'Efficient storage and retrieval by content and address of static files'. In: *Journal of the ACM (JACM)* 21.2 (1974), pp. 246–260 (cited on page 16).

[22] Robert Mario Fano. *On the number of bits required to implement an associative memory*. Massachusetts Institute of Technology, Project MAC, 1971 (cited on page 16).

[23] Roberto Grossi and Jeffrey Scott Vitter. 'Compressed suffix arrays and suffix trees with applications to text indexing and string matching'. In: *Proceedings of the thirty-second annual ACM symposium on Theory of computing*. 2000, pp. 397–406 (cited on pages 16, 17).

[24] Daisuke Okanohara and Kunihiko Sadakane. 'Practical entropy-compressed rank/select dictionary'. In: *2007 Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM. 2007, pp. 60–70 (cited on pages 18, 19, 53).

[25] Sebastiano Vigna. 'Quasi-succinct indices'. In: *Proceedings of the sixth ACM international conference on Web search and data mining*. 2013, pp. 83–92 (cited on page 19).

[26] Kunihiko Sadakane and Roberto Grossi. 'Squeezing succinct data structures into entropy bounds'. In: *Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*. 2006, pp. 1230–1239 (cited on page 20).

[27] Bernard Chazelle. 'A functional approach to data structures and its use in multidimensional searching'. In: *SIAM Journal on Computing* 17.3 (1988), pp. 427–462 (cited on page 21).

[28] Roberto Grossi, Ankur Gupta, Jeffrey Scott Vitter, et al. 'High-order entropy-compressed text indexes'. In: *SODA*. Vol. 3. 2003, pp. 841–850 (cited on pages 21, 23–25, 48).

[29] Gonzalo Navarro. 'Wavelet trees for all'. In: *Journal of Discrete Algorithms* 25 (2014), pp. 2–20 (cited on page 24).

[30] Paolo Ferragina et al. 'Compressed representations of sequences and full-text indexes'. In: *ACM Transactions on Algorithms (TALG)* 3.2 (2007), 20–es (cited on pages 25, 50).

[31] J Ian Munro and Venkatesh Raman. 'Succinct representation of balanced parentheses and static trees'. In: *SIAM Journal on Computing* 31.3 (2001), pp. 762–776 (cited on pages 27, 28, 32).

[32] Gonzalo Navarro and Kunihiko Sadakane. 'Fully functional static and dynamic succinct trees'. In: *ACM Transactions on Algorithms (TALG)* 10.3 (2014), pp. 1–39 (cited on pages 28, 29, 36, 39, 40).

[33] David Benoit et al. 'Representing trees of higher degree'. In: *Algorithmica* 43.4 (2005), pp. 275–292 (cited on pages 29, 34, 35).

[34] J Ian Munro. 'Tables'. In: *International Conference on Foundations of Software Technology and Theoretical Computer Science*. Springer. 1996, pp. 37–42 (cited on page 33).

[35] Vladimir L Arlazarov et al. 'On economical construction of the transitive closure of a directed graph'. In: *Dokl. Akad. Nauk SSSR*. Vol. 194. 11. 1970, pp. 1209–1210 (cited on page 33).

[36] Michael A Bender and Martin Farach-Colton. 'The LCA problem revisited'. In: *Latin American Symposium on Theoretical Informatics*. Springer. 2000, pp. 88–94 (cited on pages 33, 38).

[37] Meng He, J Ian Munro, and S Srinivasa Rao. 'Succinct ordinal trees based on tree covering'. In: *International Colloquium on Automata, Languages, and Programming*. Springer. 2007, pp. 509–520 (cited on pages 36, 38, 39).

[38] Richard F Geary, Rajeev Raman, and Venkatesh Raman. 'Succinct ordinal trees with level-ancestor queries'. In: *ACM Transactions on Algorithms (TALG)* 2.4 (2006), pp. 510–534 (cited on pages 36–39, 42).

[39] Arash Farzan and J Ian Munro. 'A uniform approach towards succinct representation of trees'. In: *Scandinavian Workshop on Algorithm Theory*. Springer. 2008, pp. 173–184 (cited on page 39).

[40] Meng He, J Ian Munro, and Gelin Zhou. 'A framework for succinct labeled ordinal trees over large alphabets'. In: *Algorithmica* 70.4 (2014), pp. 696–717 (cited on pages 41–47, 82).

[41] Kuo-Chung Tai. 'The tree-to-tree correction problem'. In: *Journal of the ACM (JACM)* 26.3 (1979), pp. 422–433 (cited on page 43).

[42] Philip Bille. 'A survey on tree edit distance and related problems'. In: *Theoretical computer science* 337.1-3 (2005), pp. 217–239 (cited on page 43).

[43] Meng He and Munro. 'Path queries in weighted trees'. In: *International Symposium on Algorithms and Computation*. Springer. 2011 (cited on page 43).

[44] Meng He, J Ian Munro, and Gelin Zhou. 'Data structures for path queries'. In: *ACM Transactions on Algorithms (TALG)* 12.4 (2016), pp. 1–32 (cited on pages 43, 44, 48–51, 82).

[45] Djamal Belazzougui and Gonzalo Navarro. 'Optimal lower and upper bounds for representing sequences'. In: *ACM Transactions on Algorithms (TALG)* 11.4 (2015), pp. 1–21 (cited on page 47).

[46] Prosenjit Bose et al. 'Succinct orthogonal range search structures on a grid with applications to text indexing'. In: *Workshop on Algorithms and Data Structures*. Springer. 2009, pp. 98–109 (cited on page 51).

[47] Timothy M Chan, Kasper Green Larsen, and Mihai Pătraşcu. 'Orthogonal range searching on the RAM, revisited'. In: *Proceedings of the twenty-seventh annual symposium on Computational geometry*. 2011, pp. 1–10 (cited on page 51).

[48] Abraham Bookstein and Shmuel T. Klein. 'Compression of correlated bit-vectors'. In: *Information Systems* 16.4 (1991), pp. 387–400 (cited on page 61).

[49] Adam L Buchsbaum et al. 'Engineering the compression of massive tables: an experimental approach'. In: *Symposium on Discrete Algorithms: Proceedings of the eleventh annual ACM-SIAM symposium on Discrete algorithms*. Vol. 9. 11. 2000, pp. 175–184 (cited on page 61).

[50] Andreas Björklund and Andrzej Lingas. 'Fast Boolean matrix multiplication for highly clustered data'. In: *Workshop on Algorithms and Data Structures*. Springer. 2001, pp. 258–263 (cited on page 61).

[51] João NF Alves et al. 'Accelerating graph neural networks using a novel computation-friendly matrix compression format'. In: *2025 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE. 2025, pp. 1091–1103 (cited on page 62).

[52] Paul Halmos and Steven Givant. *Introduction to Boolean algebras*. Springer, 2009 (cited on page 70).

[53] Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory*. 2nd ed. Wiley-Interscience, 2006 (cited on pages 72, 97).

[54] Harold N Gabow. 'Data structures for weighted matching and extensions to b-matching and f-factors'. In: *ACM Transactions on Algorithms (TALG)* 14.3 (2018), pp. 1–80 (cited on page 78).

[55] Andrei Z Broder. 'On the resemblance and containment of documents'. In: *Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No. 97TB100171)*. IEEE. 1997, pp. 21–29 (cited on page 89).

[56] Aristides Gionis, Piotr Indyk, Rajeev Motwani, et al. 'Similarity search in high dimensions via hashing'. In: *Vldb*. Vol. 99. 6. 1999, pp. 518–529 (cited on page 89).

[57] Bernhard Ganter, Rudolf Wille, and Rudolf Wille. *Formal concept analysis*. Vol. 150. Springer, 1999 (cited on pages 92, 93).

[58] Daniel Kleitman. 'On Dedekind's problem: the number of monotone Boolean functions'. In: *Proceedings of the American Mathematical Society* 21.3 (1969), pp. 677–682 (cited on page 94).

[59] Moses Charikar et al. 'Approximation algorithms for directed Steiner problems'. In: *Journal of Algorithms* 33.1 (1999), pp. 73–91 (cited on page 99).