EFFICIENT SUCCINCT DATA STRUCTURES ON DIRECTED ACYCLIC GRAPHS

LUCA LOMBARDO

Tesi Triennale

Dipartimento di Matematica
Università di Pisa

Supervisor: Roberto Grossi

ABSTRACT

This work introduces a redefinition of the classic rank query, traditionally applied to bit-vectors and strings, for Directed Acyclic Graphs (DAGs). We present a novel succinct data structure that leverages the topology of the input DAG to efficiently support these generalized queries while maintaining space usage below the entropy of the original graph (as there is no need to keep the whole graph to answer the queries). Our approach targets node-weighted DAGs and incorporates a critical algorithmic component: solving a generalized prefix-sum problem along DAG paths. These prefix sums, computed recursively from the root, encapsulate essential meta-information that underpins the succinct representation. The outcome of a query is expressed as a collection of ranges $[l, r]$, each representing a contiguous span of values reachable from the queried node. These ranges capture all possible accumulations (sums) of information derived from paths that originate at the source of the DAG and end at the queried node.

CONTENTS

1 INTRODUCTION

1.1 WHY SUCCINCT DATA STRUCTURES?

The digital age is characterized by an exponential and seemingly boundless increase in data generation and collection. This phenomenon, while not new, has dramatically shifted in scale and nature over time. We have progressed from early structured databases and text digitization to the vast heterogeneity of the World Wide Web. Subsequently, major scientific endeavors began producing unprecedented data volumes: consider genomics projects sequencing DNA, large-scale climate simulations, or astronomical observatories surveying the cosmos. The rise of social networks then turned billions of users into continuous producers of content (text, images, video) and relational data. More recently, the Internet of Things (IoT) has deployed sensors capturing real-time environmental and operational data, while the ascent of Large Language Models (LLMs) and generative AI relies on training over astronomical datasets and necessitates efficient representations for the models themselves.

This relentless data production often outpaces our ability to process it effectively. While storage technology (hard drives, SSDs, cloud storage) continually improves, allowing us to archive petabytes and exabytes, a persistent bottleneck remains the main memory (RAM) capacity of computers. Accessing data in RAM is crucial for performance, being orders of magnitude faster (typically a factor of $\sim 10^5$) than accessing secondary storage [36]. Consequently, fitting the necessary data—and critically, the auxiliary structures built upon it—into RAM is vital for countless data-intensive applications.

Indeed, the challenge frequently lies not just with the size of the raw data, but with the footprint of the data structures required for efficient querying and manipulation. Classical indices, trees, graphs, and other structures, while enabling fast operations, can demand significantly more space than the data they represent—sometimes one or two orders of magnitude larger. The human genome remains a compelling illustration: the sequence of roughly 3.3 billion bases can be stored in under 800 megabytes using a simple 2-bit encoding. However, powerful structures like suffix trees, essential for many pattern matching and sequence analysis tasks, can easily consume tens of gigabytes, exceeding the RAM capacity of typical machines [36].

Several paradigms exist to cope with massive datasets: secondary-memory algorithms optimizing disk access, streaming algorithms processing data "on the fly" with limited memory, and distributed algorithms partitioning data and computation across clusters. Each has strengths and limitations, often involving trade-offs in performance, accuracy, or applicability. Furthermore, the proliferation of devices with constrained computational and memory resources (from smartphones to embedded sensors in edge computing) creates scenarios where space is a primary constraint.

Data compression offers a well-established method for reducing storage space. However, most compression algorithms require full or partial decompression before the data can be randomly accessed or queried, making them unsuitable for tasks requiring direct interaction with the compressed form. This is precisely where **Succinct Data Structures** emerge as a powerful alternative. Situated at the intersection of Data Structures and Information Theory, they aim to represent data using space close to its theoretical minimum (often related to information-theoretic measures like entropy) while *simultaneously* supporting efficient queries directly on the compressed representation [36].

Essentially, succinct data structures strive to achieve the best of both worlds: the space efficiency of compression and the operational efficiency of traditional data structures. By compressing both the data itself and the indexing overhead needed to query it, they enable larger datasets and more complex structures to fit within faster levels of the memory hierarchy (e.g., cache instead of RAM, RAM instead of disk). This not only facilitates solving larger problems on a single machine but can also yield performance improvements due to better memory locality and reduced data transfer costs.

1.2 RESULTS AND CONTRIBUTIONS

This thesis contributes to the field of succinct data structures by developing and analyzing a novel representation for node-weighted Directed Acyclic Graphs (DAGs), specifically designed to efficiently support path-based aggregation queries.

Building upon foundational concepts in data compression (Chapter 2) and established succinct techniques for sequences, particularly Rank and Select operations (Chapter 3), this work addresses the challenge of managing potentially complex path information in weighted DAGs. A key motivation stems from the observation that problems like querying occurrences within degenerate strings (Section 3.3) can be effectively modeled using weighted DAGs (Chapter 4), but the application extends to other domains involving path analysis in acyclic graph structures.

The core technical contribution, presented in Chapter 4, is a space-efficient DAG representation that supports a generalized rank query (Theorem 4.9). This query aims to characterize the set of possible cumulative weights achievable on paths from a source vertex to a target vertex N, considering the contribution of N's weight itself.

Our approach hinges on a strategic partitioning of the DAG's vertices into explicit nodes (V_E, typically sinks, whose path weight information is stored directly) and implicit nodes (V_I). For implicit nodes, the path weight information ($\mathcal{O}$-sets) is reconstructed on demand by following a path defined by a carefully chosen successor function σ (Theorem 4.11). This traversal relies on compact offset sequences ($\mathcal{I}_v$) stored for each implicit node v, which map indices in $\mathcal{O}_v$ to indices in the $\mathcal{O}$-set of its successor $\sigma(v)$. We provide algorithms (GetValue, GetOSet) for reconstructing $\mathcal{O}$-set information (Section 4.3.1) and computing the final rank query (Section 4.3.2) based on this representation.

A crucial aspect is the analysis of compression strategies for the structure's components (Section 4.4), including vertex weights, successor information, and the associated data sequences ($\mathcal{O}_v$ and $\mathcal{I}_v$). We leverage techniques like variable-length integer coding (potentially using implementations like those discussed in Appendix A), Elias-Fano coding for monotonic sequences (Section 2.6.4), and Run-Length Encoding (RLE) to minimize space.

Significantly, we demonstrate (Section 4.5) that the space usage of our proposed structure can be substantially lower than the theoretical 0^{th}-order entropy bound required for a lossless representation of the entire input graph $G = (V, E, w)$. This efficiency is achieved because our structure is specifically tailored to the rankquery and does not need to store the full graph topology (all edges E), thereby offering a highly space-efficient solution for its designated task compared to both general-purpose graph encodings and naive precomputation approaches.

1.3 STRUCTURE OF THE THESIS

This thesis is organized as follows. Chapter 1, the current chapter, provides the motivation for studying succinct data structures, outlines the main contributions of this thesis concerning succinct representations for weighted DAGs, and describes the overall structure of the document. Chapter 2 then lays the theoretical groundwork by reviewing fundamental concepts from information theory and essential compression techniques relevant for building compact representations, such as entropy, source coding, integer coding, and statistical coding. Following this, Chapter 3 introduces core building blocks used in many succinct data structures, focusing on Rank and Select operations, their implementation on bitvectors and Wavelet Trees, and their application to degenerate strings. The primary research contribution is presented in Chapter 4, which details the proposed succinct representation for weighted DAGs, including the mathematical framework, the structure itself, query algorithms, compression strategies, and space efficiency analysis. Chapter 5 summarizes the findings and contributions, discusses limitations, and proposes future research directions aimed at enhancing query time predictability. Finally, Appendix A provides practical implementation details for a compressed integer vector structure supporting efficient random access, complementing the theoretical discussion of compression techniques.

2 COMPRESSION PRINCIPLES AND METHODS

Entropy, in essence, represents the minimal quantity of bits required to unequivocally distinguish an object within a set. Consequently, it serves as a foundational metric for the space utilization in compressed data representations. The ultimate aim of compressed data structures is to occupy space nearly equivalent to the entropy required for object identification, while simultaneously enabling efficient querying operations. This pursuit lies at the core of optimizing data compression techniques: achieving a balance between storage efficiency and query responsiveness.

There are plenty of compression techniques, yet they share certain fundamental steps. In Figure 1 is shown the typical processes employed for data compression. These procedures depend on the nature of the data, and the arrangement or fusion of the blocks in 1 may differ. Numerical manipulation, such as predictive coding and linear transformations, is commonly employed for waveform signals like images and audio. Logical manipulation involves altering the data into a format more feasible to compression, including techniques such as run-length encoding, zero-trees, set-partitioning information, and dictionary entries. Then, source modeling is used to predict the data's behavior and structure, which is crucial for entropy coding.

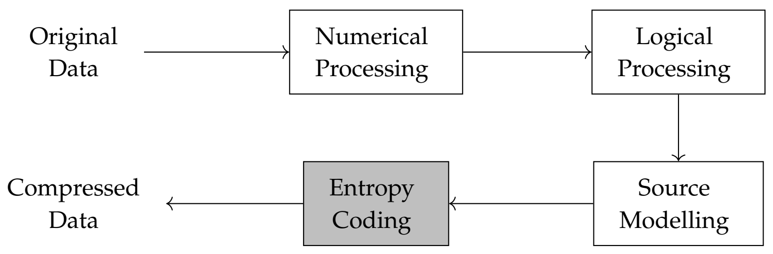


Figure 1: Typical processes in data compression

These initial numerical and logical processing stages typically aim to transform the data, exploiting specific properties like signal correlation or symbol repetition, to create a representation with enhanced statistical redundancy (e.g., more frequent symbols, predictable patterns). A common feature among most compression systems is the incorporation of *entropy coding* as the final process, wherein information is represented in the most compressed form possible. This stage may bear a significant impact on the overall compression ratio, as it is responsible for the final reduction in the data size. In this chapter we will delve into the principles of entropy coding, exploring the fundamental concepts and methods that underpin this crucial stage of data compression.

2.1 WORST CASE ENTROPY

In its simplest form, entropy can be seen as the minimum number of bits required by identifiers (*codes*, see Section 2.3), when each element of a set U has a unique code of identical length. This is called the *worst case entropy* of U and it's denoted by $H_{wc}(U)$. The worst case entropy of a set U is given by the formula:

$$H_{wc}(U) = \log |U| \tag{1}$$

where $|U|$ is the number of elements in U.

Remark 2.1. *If we used codes of length $l < H_{wc}(U)$, we would have only $2^l \leqslant 2^{H_{wc}(U)} = |U|$ possible codes, which is not enough to uniquely identify all elements in U.*

The reason behind the attribute *worst case* is that if all codes are of the same length, then this length must be at least $\lceil \log |U| \rceil$ bits to be able to uniquely identify all elements in U. If they all have different lengths, the longest code must be at least $\lceil \log |U| \rceil$ bits long.

Example 2.2 (Worst-case entropy of $\mathcal{T}_n$). *Let $\mathcal{T}_n$ denote the set of all general ordinal trees [4] with n nodes. In this scenario, each node can have an arbitrary number of children, and their order is distinguished. With n nodes, the number of possible ordinal trees is the $(n-1)$-th Catalan number, given by:*

$$|\mathcal{T}_n| = \frac{1}{n}\binom{2n-2}{n-1} \tag{2}$$

Using Stirling's approximation, we can estimate the worst-case entropy of $\mathcal{T}_n$ as:

$$|\mathcal{T}_n| = \frac{(2n-2)!}{n!(n-1)!} = \frac{(2n-2)^{2n-2}e^n e^{n-1}}{e^{2n-2}n^n(n-1)^{n-1}\sqrt{\pi n}}\left(1 + O\left(\frac{1}{n}\right)\right)$$

This simplifies to $\frac{4^n}{n^{3/2}} \cdot \Theta(1)$, hence

$$H_{wc}(\mathcal{T}_n) = \log |\mathcal{T}_n| = 2n - \Theta(\log n) \tag{3}$$

Thus, we have determined the minimum number of bits required to uniquely identify (encode) a general ordinal tree with n nodes.

2.2 ENTROPY

Let's introduce the concept of entropy as a measure of uncertainty of a random variable. While the worst-case entropy H_{wc}, discussed previously, provides a lower bound based solely on the set's cardinality (effectively assuming fixed-length codes or a uniform probability distribution over the elements), Shannon entropy offers a more refined measure. It accounts for the actual probability distribution of the elements, quantifying the *average* uncertainty or information content associated with the random variable. A deeper explanation can be found in [22, 36, 10]

Definition 2.3 (Entropy of a Random Variable). *Let X be a random variable taking values in a finite alphabet $\mathcal{X}$ with the probabilistic distribution $P_X(x) = Pr\{X = x\}$ $(x \in \mathcal{X})$. Then, the entropy of X is defined as*

This is also known as Shannon entropy, named after Claude Shannon, who introduced it in his seminal work [49]

$$H(X) = H(P_X) \overset{\text{def}}{=} E_{P_x}\{-\log P_X(x)\} = -\sum_{x\in\mathcal{X}} P_X(x)\log P_X(x) \qquad (1)$$

Where E_P denotes the expectation with respect to the probability distribution P. The log is taken to the base 2 and the entropy is expressed in bits. It is then clear that the entropy of a discrete random variable will always be nonnegative[1].

Example 2.4 (Toss of a fair coin). *Let X be a random variable representing the outcome of a toss of a fair coin. The probability distribution of X is $P_X(0) = P_X(1) = \frac{1}{2}$. The entropy of X is*

$$H(X) = -\frac{1}{2}\log\frac{1}{2} - \frac{1}{2}\log\frac{1}{2} = 1 \qquad (2)$$

This means that the toss of a fair coin has an entropy of 1 bit.

Remark 2.5. *Due to historical reasons, we are abusing the notation and using $H(X)$ to denote the entropy of the random variable X. It's important to note that this is not a function of the random variable: it's a functional of the distribution of X. It does not depend on the actual values taken by the random variable, but only on the probabilities of these values.*

The concept of entropy, introduced in definition 2.3, helps us quantify the randomness or uncertainty associated with a random variable. It essentially reflects the average amount of information needed to identify a specific value drawn from that variable. Intuitively, we can think of entropy as the average number of digits required to express a sampled value.

1 The entropy is null if and only if $X = c$, where c is a costant with probability one

2.2.1 *Properties*

In the previous section 2.2, we have introduced the entropy of a single random variable X. What if we have two random variables X and Y? How can we measure the uncertainty of the pair (X, Y)? This is where the concept of joint entropy comes into play. The idea is to consider (X, Y) as a single vector-valued random variable and compute its entropy. This is the joint entropy of X and Y. To quantify the total uncertainty associated with a pair of variables considered together, we define the joint entropy:

Definition 2.6 (Joint Entropy). *Let* (X, Y) *be a pair of discrete random variables* (X, Y) *with a joint distribution* $P_{XY}(x, y) = Pr\{X = x, Y = y\}$. *The joint entropy of* (X, Y) *is defined as*

$$H(X, Y) = H(P_{XY}) = -\sum_{x\in\mathcal{X}}\sum_{y\in\mathcal{Y}} P_{XY}(x, y) \log P_{XY}(x, y) \tag{3}$$

Which we can be extended to the joint entropy of n random variables $(X_1, X_2, \ldots, X_n)$ as $H(X_1, \ldots, X_n)$.

We also define the conditional entropy of a random variable given another as the expected value of the entropies of the conditional distributions, averaged over the conditioning random variable. Often, it's helpful to conceptualize the relationship between Y and X in terms of information transmission. Given X, we can determine the probability of observing Y through the conditional probability $W(y|x) = \Pr\{Y = y|X = x\}$ for all $x \in \mathcal{X}$ and $y \in \mathcal{Y}$. The collection W of these conditional probabilities effectively describes how information about X influences the outcome of Y, and is often referred to as a *channel* with *input alphabet* $\mathcal{X}$ and *output alphabet* $\mathcal{Y}$.

Definition 2.7 (Conditional Entropy). *Let* (X, Y) *be a pair of discrete random variables with a joint distribution* $P_{XY}(x, y) = Pr\{X = x, Y = y\}$. *The conditional entropy of* Y *given* X *is defined as*

$$\begin{aligned}
H(Y|X) = H(W|P_X) &\overset{\text{def}}{=} \sum_{x} P_X(x)H(Y|x) &(4)\\
&= \sum_{x\in\mathcal{X}} P_X(x)\Big\{-\sum_{y\in\mathcal{Y}} W(y|x)\log W(y|x)\Big\} &(5)\\
&= -\sum_{x\in\mathcal{X}}\sum_{y\in\mathcal{Y}} P_{XY}(x, y)\log W(y|x) &(6)\\
&= E_{P_{XY}}\{-\log W(Y|X)\} &(7)
\end{aligned}$$

Since entropy is always nonnegative, conditional entropy is likewise nonnegative; it has value zero if and only if Y can be entirely determined from X with certainty, meaning there exists a function $f(X)$ such that $Y = f(X)$ with probability one.

The connection between joint entropy and conditional is more evident when considering that the entropy of two random variables equals the entropy of one of them plus the conditional entropy of the other. This connection is formally proven in the following theorem.

Theorem 2.8 (Chain Rule). *Let (X, Y) be a pair of discrete random variables with a joint distribution $P_{XY}(x, y)$. Then, the joint entropy of (X, Y) can be expressed as*

This is also known as additivity of entropy.

$$H(X, Y) = H(X) + H(Y|X) \tag{8}$$

Proof. From the definition of conditional entropy (2.7), we have

$$\begin{aligned} H(X, Y) &= -\sum_{x,y} P_{XY}(x, y) \log W(y|x) \\ &= -\sum_{x,y} P_{XY}(x, y) \log \frac{P_{XY}(x, y)}{P_X(x)} \\ &= -\sum_{x,y} P_{XY}(x, y) \log P_{XY}(x, y) + \sum_{x,y} P_X(x) \log P_X(x) \\ &= H(XY) + H(X) \end{aligned}$$

Where we used the relation

$$W(y|x) = \frac{P_{XY}(x, y)}{P_X(x)} \tag{9}$$

When $P_X(x) \neq 0$. □

Corollary 2.9.

$$H(X, Y|Z) = H(X|Z) + H(Y|X, Z) \tag{10}$$

Proof. The proof is analogous to the proof of the chain rule. □

Corollary 2.10.

$$\begin{aligned} H(X_1, X_2, \ldots, X_n) = H(X_1) + H(X_2|X_1) + H(X_3|X_1, X_2) \\ + \ldots + H(X_n|X_1, X_2, \ldots, X_{n-1}) \end{aligned} \tag{11}$$

Proof. We can apply the two-variable chain rule in repetition obtain the result. □

2.2.2 *Mutual Information*

Given two random variables X and Y, the mutual information between them quantifies the reduction in uncertainty about one variable due to the knowledge of the other. It is defined as the difference between the entropy and the conditional entropy. Figure 2 illustrates the concept of mutual information between two random variables. We've seen how to measure the uncertainty of individual variables and pairs. But how much does knowing one variable tell us about the other? In other words, how much uncertainty about X is removed by knowing Y? This is quantified by the mutual information:

Definition 2.11 (Mutual Information)**.** *Let (X, Y) be a pair of discrete random variables with a joint distribution $P_{XY}(x, y)$. The mutual information between X and Y is defined as*

$$I(X;Y) = H(X) - H(X|Y) \tag{12}$$

Using the chain rule (2.8), we can rewrite it as

$$\begin{aligned}
I(X;Y) &= H(X) - H(X|Y) \\
&= H(X) + H(Y) - H(X,Y) \qquad (13) \\
&= -\sum_x P_X(x)\log P_X(x) - \sum_y P_Y(y)\log P_Y(y) \\
&\quad + \sum_{x,y} P_{XY}(x,y)\log P_{XY}(x,y) \qquad (14) \\
&= \sum_{x,y} P_{XY}(x,y)\log\frac{P_{XY}(x,y)}{P_X(x)P_Y(y)} \qquad (15) \\
&= E_{P_{XY}}\left\{\log\frac{P_{XY}(x,y)}{P_X(x)P_Y(y)}\right\} \qquad (16)
\end{aligned}$$

It follows immediately that the mutual information is symmetric, $I(X;Y) = I(Y;X)$.

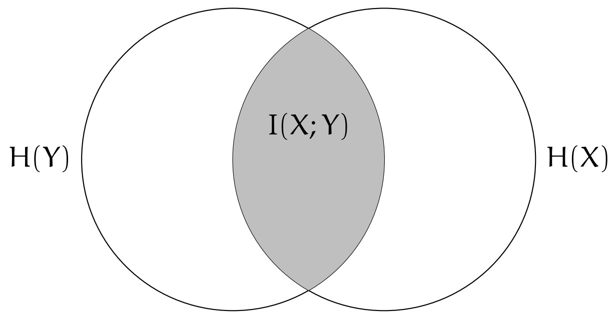


Figure 2: Mutual information between two random variables X and Y.

2.2.3 *Fano's inequality*

Information theory serves as a cornerstone for understanding fundamental limits in data compression. It not only allows us to prove the existence of encoders (Section 2.3) achieving demonstrably good performance, but also establishes a theoretical barrier against surpassing this performance. The following theorem, known as Fano's inequality, provides a lower bound on the probability of error in guessing a random variable X to it's conditional entropy H(X|Y), where Y is another random variable[2].

Theorem 2.12 (Fano's Inequality)**.** *Let* X *and* Y *be two discrete random variables with* X *taking values in some discrete alphabet* $\mathcal{X}$*, we have*

$$H(X|Y) \leqslant Pr\{X \neq Y\} \log(|\mathcal{X}| - 1) + h(Pr\{X \neq Y\}) \tag{17}$$

where $h(p) = -p \log p - (1-p) \log(1-p)$ *is the binary entropy function.*

Proof. Let Z be a random variable defined as follows:

$$Z = \begin{cases} 1 & \text{if } X \neq Y \\ 0 & \text{if } X = Y \end{cases} \tag{18}$$

We can then write

$$\begin{aligned} H(X|Y) = H(X|Y) + H(Z|XY) &= H(XZ|Y) \\ &= H(X|YZ) + H(Z|Y) \\ &\leqslant H(X|YZ) + H(Z) \end{aligned} \tag{19}$$

The last inequality follows from the fact that conditioning reduces entropy. We can then write

$$H(Z) = h(\Pr\{X \neq Y\}) \tag{20}$$

Since $\forall y \in \mathcal{Y}$, we can write

$$H(X|Y = y, Z = 0) = 0 \tag{21}$$

and

$$H(X|Y = y, Z = 1) \leqslant \log(|\mathcal{X}| - 1) \tag{22}$$

Combining these results, we have

$$H(X|YZ) \leqslant \Pr\{X \neq Y\} \log(|\mathcal{X}| - 1) \tag{23}$$

From equations 19, 20 and 23, we have Fano's inequality. □

2 We have seen in 2.7 that the conditional entropy of X given Y is zero if and only if X is a deterministic function of Y. Hence, we can estimate X from Y with zero error if and only if $H(X|Y) = 0$.

Fano's inequality thus provides a tangible link between the conditional entropy $H(X|Y)$, which quantifies the remaining uncertainty about X when Y is known, and the minimum probability of error achievable in any attempt to estimate X from Y. This inequality, along with the foundational concepts of entropy, joint entropy, conditional entropy, and mutual information introduced throughout this section, establishes a robust theoretical framework. These tools are not merely abstract measures; they allow us to quantify information, understand dependencies between data sources, and ultimately, to delineate the fundamental limits governing how efficiently data can be represented and compressed. Understanding these limits is essential as we delve deeper into specific encoding techniques.

2.3 source and code

In the previous section, we established information-theoretic limits based on the probabilistic nature of data sources. Now, we turn our attention to the practical mechanisms for achieving data compression: the interplay between a *source* of information and the *code* used to represent it. A source, in this context, can be thought of as any process generating a sequence of symbols drawn from a specific alphabet (e.g., letters of text, pixel values in an image, sensor readings). Source coding, or data compression, is the task of converting this sequence into a different, typically shorter, sequence of symbols from a target coding alphabet (often binary).

The core principle behind efficient coding is to exploit the statistical properties of the source. Symbols or patterns that occur frequently should ideally be assigned shorter representations (codewords), while less frequent ones can be assigned longer codewords. A classic, intuitive example is Morse code: the most common letter in English text, 'E', is represented by the shortest possible signal, a single dot ('.'), whereas infrequent letters like 'Q' ('–.-') receive much longer sequences.

2.3.1 *Codes*

A source characterized by a random process generates symbols from a specific alphabet at each time step. The objective is to transform this output sequence into a more concise representation. This data reduction technique, known as *source coding* or *data compression*, utilizes a code to represent the original symbols more efficiently. The device that performs this transformation is termed an *encoder*, and the process itself is referred to as *encoding*. [22]

Definition 2.13 (Source Code). *A source code for a random variable* X *is a mapping from the set of possible outcomes of* X*, called* $\mathcal{X}$*, to* $\mathcal{D}^*$*, the set of all finite-length strings of symbols from a* $\mathcal{D}$*-ary alphabet. Let* $C(X)$ *denote the codeword assigned to* x *and let* $l(x)$ *denote length of* $C(x)$

Definition 2.14 (Expected length). *The expected length* $L(C)$ *of a source code* C *for a random variable* X *with probability mass function* $P_X(x)$ *is defined as*

$$L(C) = \sum_{x \in \mathcal{X}} P_X(x) l(x) \tag{1}$$

where $l(x)$ *is the length of the codeword assigned to* x*.*

Let's assume from now for simplicity that the $\mathcal{D}$-ary alphabet is $\mathcal{D} = \{0, 1, \ldots, D-1\}$.

Example 2.15. *Let's consider a source code for a random variable* X *with* $\mathcal{X} = \{a, b, c, d\}$ *and* $P_X(a) = 0.5$*,* $P_X(b) = 0.25$*,* $P_X(c) = 0.125$ *and* $P_X(d) = 0.125$*. The code is defined as*

$$\begin{aligned} C(a) &= 0 \\ C(b) &= 10 \\ C(c) &= 110 \\ C(d) &= 111 \end{aligned}$$

The entropy of X *is*

$$H(X) = 0.5 \log 2 + 0.25 \log 4 + 0.125 \log 8 + 0.125 \log 8 = 1.75 \textit{ bits}$$

The expected length of this code is also 1.75*:*

$$L(C) = 0.5 \cdot 1 + 0.25 \cdot 2 + 0.125 \cdot 3 + 0.125 \cdot 3 = 1.75 \textit{ bits}$$

In this example we have seen a code that is optimal in the sense that the expected length of the code is equal to the entropy of the random variable.

Definition 2.16 (Nonsingular Code). *A code is nonsingular if every element of the range of* X *maps to a different element of* $\mathcal{D}^*$*. Thus:*

$$x \neq y \Rightarrow C(x) \neq C(y) \tag{2}$$

While a single unique code can represent a single value from our source X without ambiguity, our real goal is often to transmit sequences of these values. In such scenarios, we could ensure the receiver can decode the sequence by inserting a special symbol, like a "comma," between each codeword. However, this approach wastes the special symbol's potential. To overcome this inefficiency, especially when dealing with sequences of symbols from X, we can leverage the concept of self-punctuating or instantaneous codes. These codes possess a special property: the structure of the code itself inherently indicates the end of each codeword, eliminating the need for a separate punctuation symbol. The following definitions formalize this concept. [10]

Definition 2.17 (Extension of a Code)**.** *The extension C^* of a code C is the mapping from finite-length sequences of symbols from $\mathcal{X}$ to finite-length strings of symbols from the $\mathcal{D}$-ary alphabet defined by*

$$C^*(x_1x_2\dots x_n) = C(x_1)C(x_2)\dots C(x_n) \tag{3}$$

where $C(x_1)C(x_2)\dots C(x_n)$ denotes the concatenation of the codewords assigned to $x_1, x_2, \dots, x_n$.

Example 2.18. *If $C(x_1) = 0$ and $C(x_2) = 110$, then $C^*(x_1x_2) = 0110$.*

Definition 2.19 (Unique Decodability)**.** *A code C is uniquely decodable if its extension is nonsingular*

Thus, any encoded string in a uniquely decodable code has only one possibile source string that could have generated it.

Definition 2.20 (Prefix Code)**.** *A code is a prefix code if no codeword is a prefix of any other codeword.*

Also called instantaneous code

Imagine receiving a string of coded symbols. An *instantaneous code* allows us to decode each symbol as soon as we reach the end of its corresponding codeword. We don't need to wait and see what comes next. Because the code itself tells us where each codeword ends, it's like the code "punctuates itself" with invisible commas separating the symbols. This let us decode the entire message by simply reading the string and adding commas between the codewords without needing to see any further symbols. Consider the example 2.15 seen a the beginning of this section, where the binary string `01011111010` is decoded as `0,10,111,110,10` because the code used naturally separates the symbols. [10]. Figure 3 shows the relationship between different types of codes.

Example 2.21 (Morse Code)**.** *Morse code serves as a classic illustration of these concepts. Historically used for telegraphy, it represents text characters using sequences from a ternary alphabet: a short signal (dot, '.'), a longer signal (dash, '-'), and a space (pause used as a delimiter). Frequent letters like 'E' receive short codes ('.'), while less common ones like 'Q' get longer codes ('–.-'). Here are a few examples:*

Character/Sequence	*Code*
E	.
T	-
A	. -
N	- .
S	. . .
O	- -
SOS	. . . - - . . .

Let's evaluate Morse code based on our definitions:

- ***Nonsingular:*** *The code is nonsingular because each letter corresponds to a unique sequence of dots and dashes. For instance,* $E \neq T$*, and their respective codes* $C(E) = .$ *and* $C(T) = -$ *are distinct.*
- ***Prefix Code:*** *The code does not satisfy the prefix condition. Several codewords are prefixes of others. For example,* $C(E) = .$ *is a prefix of* $C(A) = .-$ *and* $C(S) = ...$ *Similarly,* $C(T) = -$ *is a prefix of* $C(N) = -.$ *and* $C(M) = -.$ *This lack of the prefix property means that receiving a sequence like '.–' is ambiguous without further information; it could represent 'A' or the sequence 'ET'.*
- ***Uniquely Decodable:*** *The code achieves unique decodability, but this relies critically on the use of pauses (spaces) inserted between letters and words according to specific timing rules. These pauses function as explicit delimiters. Without them, the inherent ambiguity due to the lack of the prefix property would make decoding impossible. This contrasts with true prefix codes (like Example 2.15), which are uniquely decodable based solely on their structure, without needing external delimiters. For example, the sequence '.—' is unambiguously decoded as 'ET' only when the timing correctly separates the 'E' ('.') from the 'T' ('-').*

2.3.2 *Kraft's Inequality*

We aim to construct efficient codes, ideally prefix codes (instantaneous codes), whose expected length approaches the source entropy. A fundamental constraint arises because we cannot arbitrarily assign short lengths to all symbols while maintaining the prefix property or even unique decodability. Kraft's inequality precisely quantifies this limitation. It establishes a *necessary* condition that the chosen codeword lengths $l(x)$ must satisfy for *any uniquely decodable* code to exist. Crucially, the same inequality also serves as a *sufficient* condition guaranteeing that a *prefix* code with these exact lengths can indeed

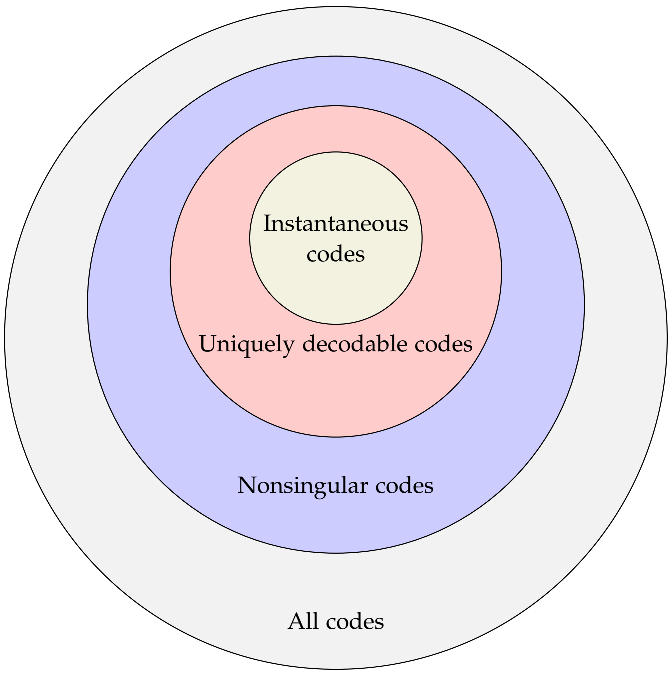


Figure 3: Relationship between different types of codes

be constructed. We will first state and prove the necessity part for uniquely decodable codes.

Let's denote the size of the source and code alphabets with $J = |\mathcal{X}|$ and $K = |\mathcal{D}|$, respectively. Different proofs of the following theorem can be found in [10, 22], here we report the one from [22], however the one proposed in [10] is also very interesting, based on the concept of a source tree.

Theorem 2.22 (Kraft's Inequality)**.** *The codeword lengths* $l(x)$*,* $x \in \mathcal{X}$*, of any uniquely decodable code* C *over a* K*-ary alphabet must satisfy the inequality*

$$\sum_{x \in \mathcal{X}} K^{-l(x)} \leqslant 1 \tag{4}$$

Proof. Consider the left hand side of the inequality 4 and consider its n-th power

$$\begin{aligned}\left(\sum_{x \in \mathcal{X}} K^{-l(x)}\right)^n &= \sum_{x_1 \in \mathcal{X}} \sum_{x_2 \in \mathcal{X}} \dots \sum_{x_n \in \mathcal{X}} K^{-l(x_1)} K^{-l(x_2)} \dots K^{-l(x_n)} \\ &= \sum_{x^n \in \mathcal{X}\backslash} K^{-l(x^n)} \end{aligned} \tag{5}$$

Where $l(x^n) = l(x_1) + l(x_2) + \ldots + l(x_n)$ is the length of the concatenation of the codewords assigned to $x_1, x_2, \ldots, x_n$. If we consider the all the extended codewords of length m we have

$$\sum_{x^n \in \mathcal{X}^\backslash} K^{-l(x^n)} = \sum_{m=1}^{nl_max} A(m)K^{-m} \tag{6}$$

where $A(m)$ is the number source sequences of length n whose codewords have length m and l_{max} is the maximum length of the codewords in the code. Since the code is separable, we have that $A(m) \leqslant K^m$ and therefore each term of the sum is less than or equal to 1. Hence

$$\left(\sum_{x \in \mathcal{X}} K^{-l(x)}\right)^n \leqslant nl_{max} \tag{7}$$

That is

$$\sum_{x \in \mathcal{X}} K^{-l(x)} \leqslant (nl_{max})^{1/n} \tag{8}$$

Taking the limit as n goes to infinity and using the fact that $(nl_{max})^{1/n} = e^{1/n \log(nl_{max})} \to 1$ we have that

$$\sum_{x \in \mathcal{X}} K^{-l(x)} \leqslant 1 \tag{9}$$

That concludes the proof. □

2.4 EMPIRICAL ENTROPY

Before digging into the concept of empirical entropy, let's begin with the notion of binary entropy. Consider an alphabet $\mathcal{U}$, where $\mathcal{U} = \{0, 1\}$. Let's assume it emits symbols with probabilities p_0 and $p_1 = 1 - p_0$. The entropy of this source can be calculated using the formula:

$$H(p_0) = -p_0 \log_2 p_0 - (1 - p_0) \log_2(1 - p_0)$$

We can extend this concept to scenarios where the elements are no longer individual bits, but sequences of these bits emitted by the source. Initially, let's assume the source is *memoryless* (or *zero-order*), meaning the probability of emitting a symbol doesn't depend on previously emitted symbols. In this case, we can consider chunks of n bits as our elements. Our alphabet becomes $\Sigma = \{0, 1\}^n$, and the Shannon Entropy of two independent symbols $x, y \in \Sigma$ will be the sum of their entropies. Thus, if the source emits symbols from a general alphabet Σ of size $|\Sigma| = \sigma$, where each symbol $s \in \Sigma$ has a probability p_s (with $\sum_{s \in \Sigma} p_s = 1$), the Shannon entropy of the source is given by:

$$H(P) = H(p_1, \dots, p_\sigma) = -\sum_{s\in\Sigma} p_s \log p_s = \sum_{s\in\Sigma} p_s \log \frac{1}{p_s}$$

Remark 2.23. *If all symbols have a probability of* $p_s = 1$*, then the entropy is* 0*, and all other probabilities are* 0*. If all symbols have the same probability* $\frac{1}{\sigma}$*, then the entropy is* $\log \sigma$*. So given a sequence of* n *elements from an alphabet* Σ*, belonging to* $\mathcal{U} = \Sigma^n$*, its entropy is straightforwardly* $n\mathcal{H}(p_1, \dots, p_\sigma)$

2.4.1 *Bit Sequences*

In many practical scenarios, however, we do not know the true probabilities p_s of the underlying source. Instead, we might only have access to a sequence generated by the source. The concept of empirical entropy allows us to estimate the information content based directly on the observed frequencies within that sequence. Let's first examine this for binary sequences.

Let's consider a bit sequence, $B[1, n]$, which we aim to compress without access to an explicit model of a known bit source. Instead, we only have access to B. Although lacking a precise model, we may reasonably anticipate that B exhibits a bias towards either more 0s or more 1s. Hence, we might attempt to compress B based on this characteristic. Specifically, we say that B is generated by a zero-order source emitting 0s and 1s. Assuming m represents the count of 1s in B, it's reasonable to posit that the source emits 1s with a probability of $p = m/n$. This leads us to the concept of zero-order empirical entropy:

Definition 2.24 (Zero-order empirical entropy)**.** *Given a bit sequence* $B[1, n]$ *with* m 1s *and* $n - m$ 0s*, the zero-order empirical entropy of* B *is defined as:*

$$\mathcal{H}_0(B) = \mathcal{H}\left(\frac{m}{n}\right) = \frac{m}{n} \log \frac{n}{m} + \frac{n-m}{n} \log \frac{n}{n-m} \tag{1}$$

The concept of zero-order empirical entropy carries significant weight: it indicates that if we attempt to compress B using a fixed code $C(1)$ for 1s and $C(0)$ for 0s, then it's impossible to compress B to fewer than $\mathcal{H}_0(B)$ bits per symbol. Otherwise, we would have $m|C(1)| + (n - m)|C(0)| < n\mathcal{H}_0(B)$, which violates the lower bound established by Shannon entropy.

CONNECTION WITH WORST CASE ENTROPY It is interesting to note a connection between the zero-order empirical entropy $\mathcal{H}_0(B)$ and the worst-case entropy H_{wc} previously introduced (Section 2.1). Consider the specific set $\mathcal{B}_{n,m}$ comprising all possible binary sequences of length n that contain exactly m ones, like our sequence B. The worst-case entropy necessary to assign a unique identifier to each sequence *within this set* is $H_{wc}(\mathcal{B}_{n,m}) = \log |\mathcal{B}_{n,m}| = \log \binom{n}{m}$. Using Stirling's approximation for the binomial coefficient, it can be demonstrated that this quantity is closely related to the total empirical entropy: $H_{wc}(\mathcal{B}_{n,m}) \approx n\mathcal{H}_0(B) - O(\log n)$. Thus, $n\mathcal{H}_0(B)$ approximates the minimum number of bits required, on average per sequence, to distinguish among all sequences sharing the same number of 0s and 1s, providing another perspective on the meaning of empirical entropy [36].

2.4.2 *Entropy of a Text*

The zero-order empirical entropy of a string $S[1,n]$, where each symbol s occurs n_s times in S, is similarly determined by the Shannon entropy of its observed probabilities:

Definition 2.25 (Zero-order empirical entropy of a text)**.** *Given a text* $S[1,n]$ *with* n_s *occurrences of symbol* s*, the zero-order empirical entropy of* S *is defined as:*

$$\mathcal{H}_0(S) = \mathcal{H}\left(\frac{n_1}{n}, \dots, \frac{n_\sigma}{n}\right) = \sum_{s=1}^{\sigma} \frac{n_s}{n} \log \frac{n}{n_s} \tag{2}$$

Example 2.26. *Let* $S =$ *"abracadabra". We have that* $n = 11$, $n_a = 5$, $n_b = 2$, $n_c = 1$, $n_d = 1$, $n_r = 2$*. The zero-order empirical entropy of* S *is:*

$$\mathcal{H}_0(S) = \frac{5}{11} \log \frac{11}{5} + 2 \cdot \frac{2}{11} \log \frac{11}{2} + 2 \cdot \frac{1}{11} \log \frac{11}{1} \approx 2.04$$

Thus, we could expect to compress S *to* $nH_0(S) \approx 22.44$ *bits, which is lower than the* $n \log \sigma = 11 \cdot \log 5 \approx 25.54$ *bits of the worst-case entropy of a general string of length* n *over an alphabet of size* $\sigma = 5$*.*

However, this definition falls short because in most natural languages, symbol choices aren't independent. For example, in English text, the sequence "*don'*" is almost always followed by "*t*". Higher-order entropy (Section 2.5) is a more accurate measure of the entropy of a text, as it considers the probability of a symbol given the preceding symbols. This principle was at the base of the development of the famous Morse Code and then the Huffman code (Section 2.7).

2.5 HIGHER ORDER ENTROPY

The zero-order empirical entropy $\mathcal{H}_0(S)$, discussed in the previous section, provides a useful baseline for compression by considering the frequency of individual symbols. However, it operates under the implicit assumption that symbols are generated independently, a condition seldom met in practice, especially for data like natural language text. For instance, the probability of encountering the letter 'u' in English text dramatically increases if the preceding letter is 'q'. To capture such dependencies and obtain a more accurate measure of the information content considering local context, we introduce the concept of *higher-order empirical entropy*. This approach conditions the probability of a symbol's occurrence on the sequence of k symbols that immediately precede it.

Definition 2.27 (Redundancy)**.** *For an information source X generating symbols from an alphabet Σ, the redundancy R is the difference between the maximum possible entropy per symbol and the actual entropy $H(X)$ of the source:*

$$R = \log_2 |\Sigma| - H(X) \tag{1}$$

This redundancy value, R, quantifies the degree of predictability or statistical structure inherent in the source. A high redundancy signifies that the source is far from random, exhibiting patterns (like non-uniform symbol probabilities or inter-symbol dependencies) that can potentially be exploited for compression. Conversely, a source with low redundancy behaves more randomly, leaving less room for compression beyond the theoretical minimum dictated by $H(X)$.

However, evaluating redundancy directly using Definition 2.27 often proves impractical, as determining the true source entropy $H(X)$ for the process generating a given string S is typically unfeasible. This limitation necessitates alternative, empirical approaches. To address this issue, we introduce the concept of the *k-th order empirical entropy* of a string S, denoted as $\mathcal{H}_k(S)$. In statistical coding (Section 2.7), we will see a scenario where $k = 0$, relying on symbol frequencies within the string. Now, with $\mathcal{H}_k(S)$, our objective is to extend the entropy concept by examining the frequencies of k-grams in string S. This requires analyzing subsequences of symbols with a length of k, thereby capturing the *compositional structure* of S [13].

Let S be a string of length $n = |S|$ over an alphabet Σ of size $|\Sigma| = \sigma$. Let ω denote a k-gram (a sequence of k symbols from Σ), and let n_ω be the number of occurrences of ω in S. Let $n_{\omega\sigma_i}$ be the number of times the k-gram ω is followed by the symbol $\sigma_i \in \Sigma$ in S. [3]

3 We use the notation $\omega \in \Sigma^k$ for a k-gram.

Definition 2.28 (k-th Order Empirical Entropy)**.** *The* k-th order empirical entropy *of a string* S *is defined as:*

$$\mathcal{H}_k(S) = \frac{1}{n} \sum_{\omega \in \Sigma^k} \left(\sum_{\sigma_i \in \Sigma} n_{\omega\sigma_i} \log_2 \left(\frac{n_\omega}{n_{\omega\sigma_i}} \right) \right) \tag{2}$$

where terms with $n_{\omega\sigma_i} = 0$ *contribute zero to the sum.*

This definition calculates the average conditional entropy based on the preceding k symbols. An equivalent and often more intuitive way to express this is by averaging the zero-order empirical entropies of the sequences formed by the symbols following each distinct k-gram context:

$$\mathcal{H}_k(S) = \sum_{\omega \in \Sigma^k, n_\omega > 0} \frac{n_\omega}{n} \cdot \mathcal{H}_0(S_\omega) \tag{3}$$

where S_ω is the string formed by concatenating all symbols that immediately follow an occurrence of the k-gram ω in S (its length is $|S_\omega| = n_\omega$). The sum is taken over all k-grams ω that actually appear in S (i.e., $n_\omega > 0$).

Example 2.29. *Consider the example 2.26, where* $S =$ *"abracadabra"* ($n = 11$) *and* $\Sigma = \{a, b, c, d, r\}$ ($\sigma = 5$)*. The zero-order empirical entropy is* $\mathcal{H}_0(S) \approx 2.04$*. Now, let's calculate the first-order* ($k = 1$) *empirical entropy using Equation 3. The contexts are the single characters:*

- *Context 'a'* ($n_a = 5$)*: Following symbols are 'b', 'c', 'd', 'b', '\$' (assuming end-of-string marker).* $S_a =$ *"bcdb\$".* $\mathcal{H}_0(S_a) \approx 1.922$ *bits/symbol (assuming \$ is a unique symbol).*
- *Context 'b'* ($n_b = 2$)*: Following symbols are 'r', 'r'.* $S_b =$ *"rr".* $\mathcal{H}_0(S_b) = 0$ *bits/symbol.*
- *Context 'c'* ($n_c = 1$)*: Following symbol is 'a'.* $S_c =$ *"a".* $\mathcal{H}_0(S_c) = 0$ *bits/symbol.*
- *Context 'd'* ($n_d = 1$)*: Following symbol is 'a'.* $S_d =$ *"a".* $\mathcal{H}_0(S_d) = 0$ *bits/symbol.*
- *Context 'r'* ($n_r = 2$)*: Following symbols are 'a', 'a'.* $S_r =$ *"aa".* $\mathcal{H}_0(S_r) = 0$ *bits/symbol.*

Therefore, the first-order empirical entropy of S *is:*

$$\mathcal{H}_1(S) = \frac{n_a}{n}\mathcal{H}_0(S_a) + \frac{n_b}{n}\mathcal{H}_0(S_b) + \frac{n_c}{n}\mathcal{H}_0(S_c) + \frac{n_d}{n}\mathcal{H}_0(S_d) + \frac{n_r}{n}\mathcal{H}_0(S_r)$$

$$\mathcal{H}_1(S) = \frac{5}{11} \cdot (1.922) + \frac{2}{11} \cdot 0 + \frac{1}{11} \cdot 0 + \frac{1}{11} \cdot 0 + \frac{2}{11} \cdot 0 \approx 0.874 \textit{ bits/symbol}$$

This value is significantly lower than the zero-order empirical entropy $\mathcal{H}_0(S)$*, reflecting the predictability introduced by considering the preceding character.*

The quantity $n\mathcal{H}_k(S)$ serves as a lower bound for the minimum number of bits attainable by any encoding of S, under the condition that the encoding of each symbol may rely only on the k symbols preceding it in S. Consistently, any compressor achieving fewer than $n\mathcal{H}_k(S)$ bits would imply the ability to compress symbols originating from the related k-th order Markov source to a level below its Shannon entropy.

The practical relevance of achieving compression close to this bound was demonstrated by Sadakane and Grossi [47], who presented a general technique based on LZ78 parsing to transform various succinct data structures, such as those supporting rank and select queries, towards the $n\mathcal{H}_k(S) + o(n)$ space limit while preserving their original query time complexities

Remark 2.30. *As k grows large (up to $k = n - 1$, and often sooner), the k-th order empirical entropy $\mathcal{H}_k(S)$ tends towards zero, given that most long k-grams appear only once, making their subsequent symbol perfectly predictable within the sequence S. This renders the model ineffective as a lower bound for practical compressors when k is very large relative to n. Even before reaching $\mathcal{H}_k(S) = 0$, achieving compression close to $n\mathcal{H}_k(S)$ bits becomes practically challenging for high k values. This is due to the necessity of storing or implicitly representing the conditional probabilities (or equivalent coding information) for all σ^k possible contexts, which requires significant space overhead ($\approx \sigma^{k+1} \log n$ bits in simple models). In theory, it is commonly assumed that S can be compressed up to $n\mathcal{H}_k(S) + o(n)$ bits for any k such that $k + 1 \leqslant \alpha \log_\sigma n$ for some constant $0 < \alpha < 1$. Under this condition, the overhead for storing the model ($\sigma^{k+1} \log n \leqslant n^\alpha \log n$) becomes asymptotically negligible compared to the compressed data size ($o(n)$ bits) [36].*

Definition 2.31 (Coarsely Optimal Compression Algorithm)**.** *A compression algorithm is* coarsely optimal *if, for every fixed value of $k \geqslant 0$, there exists a function $f_k(n)$ such that $\lim_{n\to\infty} f_k(n) = 0$, and for all sequences S of length n, the compression size achieved by the algorithm is bounded by $n(\mathcal{H}_k(S) + f_k(n))$ bits.*

The *Lempel-Ziv* algorithm family, particularly LZ78, serves as a prominent example of coarsely optimal compression techniques, as demonstrated by Plotnik et al. [43]. These algorithms typically rely on dictionary-based compression. However, as highlighted by Kosaraju and Manzini [28], the notion of coarse optimality does not inherently guarantee practical effectiveness across all scenarios. The additive term $n \cdot f_k(n)$ might still lead to poor performance on some sequences, especially if $f_k(n)$ converges slowly or if the sequence length n is not sufficiently large for the asymptotic behavior to dominate.

2.5.1 *Source Coding Theorem*

In Section 2.3 we have established the properties of different code types and the fundamental constraint imposed by Kraft's inequality (Theorem Theorem 2.22), we now arrive at a cornerstone result in information theory: the Source Coding Theorem. Attributed to Shannon [49], this theorem provides the definitive answer to the question of the ultimate limit of lossless data compression. It establishes that the entropy $\mathcal{H}(X)$ of the source (representing the underlying probability distribution, distinct from the empirical entropy $\mathcal{H}_k(S)$ of a specific string) is not just a theoretical measure of information, but the precise operational limit for the average length of any uniquely decodable code representing that source.

The theorem consists of two crucial parts: a lower bound on the achievable average length, and a statement about the existence of codes that approach this bound. We will state the theorem for a K-ary code alphabet $\mathcal{D}$, using $\mathcal{H}_K(X) = \mathcal{H}(X)/\log K$ to denote the theoretical source entropy measured in K-ary units (assuming $\mathcal{H}(X)$ is calculated, for instance, using base 2 logarithms unless otherwise specified).

Theorem 2.32 (Source Coding Theorem)**.** *Let X be a random variable generating symbols from an alphabet Σ with probability mass function $P_X(x)$. Let $\mathcal{D}$ be a code alphabet of size $K \geqslant 2$.*

1. ***(Lower Bound)** The expected length $L(C)$ of any uniquely decodable code $C : \Sigma \to \mathcal{D}^*$ for X satisfies*

$$L(C) \geqslant \mathcal{H}_K(X) = \frac{\mathcal{H}(X)}{\log K} \tag{4}$$

2. ***(Achievability)** There exists a prefix code $C : \Sigma \to \mathcal{D}^*$ such that its expected length $L(C)$ satisfies*

$$L(C) < \mathcal{H}_K(X) + 1 \tag{5}$$

Proof. **Part (i) - Lower Bound:** Let C be any uniquely decodable code with codeword lengths $l(x)$ for $x \in \Sigma$. By Theorem 2.22, these lengths must satisfy Kraft's inequality: $S = \sum_{x \in \Sigma} K^{-l(x)} \leqslant 1$. Let us define an auxiliary probability distribution $Q(x)$ over Σ as $Q(x) = K^{-l(x)}/S$. Note that $\sum_{x \in \Sigma} Q(x) = 1$, so Q is a valid probability distribution.

Now, consider the expected length $L(C)$:

$$\begin{aligned}
L(C) &= \sum_{x\in\Sigma} P_X(x)l(x) && (6)\\
&= \sum_{x\in\Sigma} P_X(x)\log_K\left(K^{l(x)}\right) && (7)\\
&= \sum_{x\in\Sigma} P_X(x)\log_K\left(\frac{S}{Q(x)}\right) \quad (\text{since } K^{-l(x)} = SQ(x)) && (8)\\
&= \sum_{x\in\Sigma} P_X(x)\left(\log_K S - \log_K Q(x)\right) && (9)\\
&= (\log_K S)\sum_{x\in\Sigma} P_X(x) - \sum_{x\in\Sigma} P_X(x)\log_K Q(x) && (10)\\
&= \log_K S - \sum_{x\in\Sigma} P_X(x)\log_K Q(x) && (11)
\end{aligned}$$

We can relate the last term to the relative entropy (Kullback-Leibler divergence) $D(P_X||Q)$ and the entropy $\mathcal{H}_K(X)$. Recall that:

$$\begin{aligned}
D(P_X||Q) &= \sum_{x\in\Sigma} P_X(x)\log_K\frac{P_X(x)}{Q(x)}\\
&= \sum_{x\in\Sigma} P_X(x)\log_K P_X(x) - \sum_{x\in\Sigma} P_X(x)\log_K Q(x)\\
&= -\mathcal{H}_K(X) - \sum_{x\in\Sigma} P_X(x)\log_K Q(x)
\end{aligned}$$

Thus, $-\sum P_X(x)\log_K Q(x) = D(P_X||Q) + \mathcal{H}_K(X)$. Substituting this back into the expression for $L(C)$:

$$L(C) = \log_K S + D(P_X||Q) + \mathcal{H}_K(X) \tag{12}$$

Since $S \leqslant 1$, we have $\log_K S \leqslant \log_K 1 = 0$. Also, the relative entropy is always non-negative, $D(P_X||Q) \geqslant 0$. Therefore,

$$L(C) \geqslant 0 + 0 + \mathcal{H}_K(X) = \mathcal{H}_K(X) \tag{13}$$

This establishes the lower bound (4). This line of proof closely follows [10].

Part (ii) - Achievability: We need to show the existence of a prefix code whose expected length satisfies (5). Consider choosing codeword lengths $l(x)$ for each $x \in \Sigma$ as:

$$l(x) = \lceil -\log_K P_X(x) \rceil \tag{14}$$

where $\lceil\cdot\rceil$ denotes the ceiling function (rounding up to the nearest integer). These lengths are positive integers (assuming $P_X(x) \leqslant 1$).

First, we verify that these lengths satisfy Kraft's inequality. From the definition of the ceiling function, we have:

$$-\log_K P_X(x) \leqslant l(x) < -\log_K P_X(x) + 1 \tag{15}$$

Exponentiating the left inequality with base K:

$$K^{-\log_K P_X(x)} \geqslant K^{-l(x)} \implies P_X(x) \geqslant K^{-l(x)} \tag{16}$$

Summing over all $x \in \Sigma$:

$$\sum_{x\in\Sigma} K^{-l(x)} \leqslant \sum_{x\in\Sigma} P_X(x) = 1 \tag{17}$$

Since the chosen lengths satisfy Kraft's inequality, the sufficiency part of Kraft's theorem guarantees that there exists a *prefix* code C with these exact lengths $l(x) = \lceil -\log_K P_X(x) \rceil$. (This existence is often demonstrated constructively, e.g., using code trees or interval mapping [10, 22]).

Now, let's calculate the expected length $L(C)$ for this prefix code:

$$\begin{aligned} L(C) &= \sum_{x\in\Sigma} P_X(x) l(x) && (18)\\ &= \sum_{x\in\Sigma} P_X(x) \lceil -\log_K P_X(x) \rceil && (19)\\ &< \sum_{x\in\Sigma} P_X(x) \left(-\log_K P_X(x) + 1 \right) \quad \text{(using } \lceil y \rceil < y+1\text{)} && (20)\\ &= \sum_{x\in\Sigma} -P_X(x) \log_K P_X(x) + \sum_{x\in\Sigma} P_X(x) \cdot 1 && (21)\\ &= \mathcal{H}_K(X) + 1 && (22) \end{aligned}$$

Thus, we have shown that there exists a prefix code C with $L(C) < \mathcal{H}_K(X) + 1$, proving the achievability part (5). □

The Source Coding Theorem is a profound result. It states that the source entropy $\mathcal{H}_K(X)$ is the fundamental lower limit on the average number of K-ary symbols required per source symbol for reliable (lossless) representation using any uniquely decodable code. Furthermore, it guarantees that we can always find a prefix code (which is easy to decode instantaneously) whose average length is within 1 symbol of this theoretical minimum.

The gap of "1" in the achievability part arises from the constraint that codeword lengths must be integers, while $-\log_K P_X(x)$ is generally not. This gap can be made arbitrarily small (per source symbol) by encoding blocks of source symbols together. If we consider encoding blocks $X^n = (X_1, \ldots, X_n)$ from an i.i.d. source, the entropy per symbol is $\mathcal{H}(X^n)/n = \mathcal{H}(X)$. Applying the theorem to the block source Σ^n, we can find a prefix code with expected length L_n such that $\mathcal{H}_K(X^n) \leqslant L_n < \mathcal{H}_K(X^n) + 1$. Dividing by n, the average length per original source symbol, L_n/n, satisfies:

$$\mathcal{H}_K(X) \leqslant \frac{L_n}{n} < \frac{\mathcal{H}_K(X^n)}{n} + \frac{1}{n} = \mathcal{H}_K(X) + \frac{1}{n} \tag{23}$$

As the block length n increases, the average codeword length per source symbol approaches the entropy $\mathcal{H}_K(X)$. This demonstrates that the entropy limit is asymptotically achievable. Practical codes like Huffman coding (Section 2.7.1) provide methods to construct optimal prefix codes for a given distribution, while techniques like arithmetic coding (Section 2.7.2) effectively approximate the block coding concept to get very close to the entropy bound even for moderate sequence lengths.

2.6 integer coding

This chapter examines methods for representing a sequence of positive integers, $S = \{x_1, x_2, \ldots, x_n\}$, potentially containing repetitions, as a compact sequence of bits [13]. The primary objective is to minimize the total bits used. A key requirement is that the resulting binary sequence must be *self-delimiting*: the concatenation of individual integer codes must be unambiguously decodable, allowing a decoder to identify the boundaries between consecutive codes.

The practical importance of efficient integer coding affects both storage space and processing speed in numerous applications. For example, *search engines* maintain large indexes mapping terms to lists of document identifiers (IDs). These *posting lists* can contain billions of integer IDs. Efficient storage is vital. A common approach involves sorting the IDs and encoding the differences (gaps) between consecutive IDs using variable-length integer codes, assigning shorter codes to smaller, more frequent gaps [13, 53]. The engineering considerations for building practical data structures based on these principles, such as providing random access capabilities, are discussed in detail in Appendix A, which describes a library developed as part of this work.

Another significant application occurs in the final stage of many *data compression algorithms*. Techniques such as LZ77, Move-to-Front (MTF), Run-Length Encoding (RLE), or Burrows-Wheeler Transform (BWT) often generate intermediate outputs as sequences of integers, where smaller values typically appear more frequently. An effective integer coding scheme is then needed to convert this intermediate sequence into a compact final bitstream [13]. Similarly, compressing natural language text might involve mapping words or characters to integer token IDs and subsequently compressing the resulting ID sequence using integer codes [13].

This chapter explores techniques for designing such variable-length, prefix-free binary representations for integer sequences, aiming for maximum space efficiency.

The central concern in this section revolves around formulating an efficient binary representation method for an indefinite sequence of integers. Our objective is to minimize bit usage while ensuring that the encoding remains prefix-free. In simpler terms, we aim to devise a binary format where the codes for individual integers can be concatenated without ambiguity, allowing the decoder to reliably identify the start and end of each integer's representation within the bit stream and thus restore it to its original uncompressed state.

2.6.1 *Unary Code*

We begin by examining the unary code, a straightforward encoding method that represents a positive integer $x \geqslant 1$[4] using x bits. It represents x as a sequence of $x-1$ zeros followed by a single one, denoted as $U(x)$. The correctness of this encoding is straightforward: the decoder identifies the end of the code upon encountering the first '1', and the value x is simply the total number of bits read.

This coding method requires x bits to represent x. While simple, this is exponentially longer than the $\lceil \log_2 x \rceil$ bits needed by its standard binary representation $B(x)$. Consequently, unary coding is efficient only for very small values of x and becomes rapidly impractical as x increases. This behavior aligns with the principles of Shannon's source coding theorem (Theorem 2.32), which suggests an ideal code length of $-\log_2 P(x)$ bits for a symbol x with probability $P(x)$. The unary code's length of x bits corresponds precisely to this ideal length if the integers follow the specific probability distribution $P(x) = 2^{-x}$ [13].

Theorem 2.33. *The unary code $U(x)$ of a positive integer x requires x bits, and it is optimal for the geometric distribution $P(x) = 2^{-x}$.*

Despite its theoretical optimality for the $P(x) = 2^{-x}$ distribution, the unary code faces practical challenges. Its implementation often involves numerous bit shifts or bit-level operations during decoding, which can be relatively slow on modern processors, especially for large x.

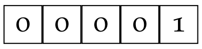


Figure 4: Unary code $U(5) =$ `00001`. It uses $x = 5$ bits, consisting of $x - 1 = 4$ zeros followed by a one.

2.6.2 *Elias Codes*

While unary code is simple, its inefficiency for larger integers motivates the development of *universal codes* like those proposed by Elias [11], building upon earlier work by Levenstein. The term universal signifies that the length of the codeword for an integer x grows proportionally to its minimal binary representation, specifically as $O(\log x)$, rather than, for instance, $O(x)$ as in the unary code. Compared to the standard binary code $B(x)$ (which requires $\lceil \log_2 x \rceil$ bits but is not prefix-free), the γ and δ codes are only a constant factor longer but possess the crucial property of being prefix-free.

4 This is not a strict condition, but we will assume it for clarity.

GAMMA (γ) CODE The γ code represents a positive integer x by combining information about its magnitude (specifically, the length of its binary representation) with its actual bits. First, determine the length of the standard binary representation of x, denoted as $l = \lfloor \log_2 x \rfloor + 1$. The γ code, $\gamma(x)$, is formed by concatenating the unary code of this length, $U(l)$, with the $l-1$ least significant bits of x (i.e., $B(x)$ excluding its leading '1' bit, which is implicitly represented by the '1' in $U(l)$). The decoding process mirrors this structure: read bits until the terminating '1' of the unary part is found to determine l, then read the subsequent $l-1$ bits and prepend a '1' to reconstruct x. The total length is $|U(l)| + (l-1) = l + (l-1) = 2l - 1 = 2(\lfloor \log_2 x \rfloor + 1) - 1$ bits. From Shannon's condition, it follows that this code is optimal for sources where integer probabilities decay approximately as $P(x) \approx 1/x^2$ [13].

Theorem 2.34. *The γ code of a positive integer x takes $2(\lfloor \log_2 x \rfloor + 1) - 1$ bits. It is optimal for distributions where $P(x) \propto 1/x^2$ and its length is within a factor of two of the length of the standard binary code $B(x)$.*

0	0	1	1	0

Figure 5: Elias γ code for $x = 6$. Binary $B(6) =$ `110`, length $l = 3$. The code consists of $U(3) =$ `001` followed by the $l - 1 = 2$ trailing bits (`10`). Result: $\gamma(6) =$ `00110` (5 bits).

The inefficiency in the γ code resides in the unary encoding of the length l, which can become long for large x. The δ code addresses this.

DELTA (δ) CODE The δ code improves upon γ by encoding the length $l = \lfloor \log_2 x \rfloor + 1$ more efficiently using the γ code itself. The δ code, $\delta(x)$, is constructed by first computing $\gamma(l)$, the gamma code of the length l. Then, it appends the same $l-1$ least significant bits of x used in $\gamma(x)$ (i.e., $B(x)$ without its leading '1'). Decoding involves first decoding $\gamma(l)$ to find the length l, and then reading the next $l-1$ bits to reconstruct x. The total number of bits is $|\gamma(l)| + (l-1) = (2\lfloor \log_2 l \rfloor + 1) + (l-1) = 2\lfloor \log_2 l \rfloor + l$. Asymptotically, this is approximately $\log_2 x + 2\log_2 \log_2 x + O(1)$ bits, which is only marginally longer ($1 + o(1)$ factor) than the raw binary representation $B(x)$. This code achieves optimality for distributions where $P(x) \approx 1/(x(\log_2 x)^2)$ [13].

Theorem 2.35. *The δ code of a positive integer x takes $2\lfloor \log_2(\lfloor \log_2 x \rfloor + 1) \rfloor + \lfloor \log_2 x \rfloor + 1$ bits, approximately $\log_2 x + 2\log_2 \log_2 x$. It is optimal for distributions $P(x) \propto 1/(x(\log_2 x)^2)$ and is within a factor $1 + o(1)$ of the length of $B(x)$.*

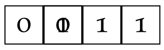


Figure 6: Elias δ code for $x = 6$. $B(6) =$ `110`, length $l = 3$. First, encode $l = 3$ using γ: $\gamma(3) =$ `011`. Then, append the $l - 1 = 2$ trailing bits (`10`). Result: $\delta(6) =$ `01110` (5 bits).

As with the unary code, decoding Elias codes often involves bit shifts, potentially impacting performance for very large integers compared to byte-aligned or word-aligned codes.

2.6.3 *Rice Code*

Elias codes offer universality but can be suboptimal if integers cluster around values far from powers of two. Rice codes [45] (a special case of Golomb codes) address this by introducing a parameter $k > 0$, chosen based on the expected distribution of integers. For an integer $x \geqslant 1$, the Rice code $R_k(x)$ is determined by calculating the quotient $q = \lfloor (x-1)/2^k \rfloor$ and the remainder $r = (x-1) \pmod{2^k}$. The code is then formed by concatenating the unary code of the quotient plus one, $U(q+1)$, followed by the remainder r encoded using exactly k bits (padding with leading zeros if necessary), denoted $B_k(r)$. This structure is efficient when integers often yield small quotients q, meaning they are close to (specifically, just above) multiples of 2^k.

The total number of bits required for $R_k(x)$ is $(q+1)+k$. Rice codes are optimal for geometric distributions $P(x) = p(1-p)^{x-1}$, provided the parameter k is chosen such that 2^k is close to the mean or median of the distribution (specifically, optimal when $2^k \approx -\frac{\ln 2}{\ln(1-p)} \approx 0.69 \times \text{mean}(S)$) [13, 53]. The fixed length of the remainder part facilitates faster decoding compared to Elias codes in certain implementations.

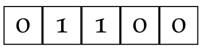


Figure 7: Rice code for $x = 13$ with parameter $k = 3$. Calculate $q = \lfloor (13-1)/2^3 \rfloor = 1$ and $r = (13-1) \pmod 8 = 4$. The code is $U(q+1) = U(2) =$ `01` followed by $r = 4$ in $k = 3$ bits, $B_3(4) =$ `100`. Result: $R_3(13) =$ `01100` (5 bits).

2.6.4 *Elias-Fano Code*

The Elias-Fano representation, conceived independently by Peter Elias [11] and Robert M. Fano [12], provides an elegant and practically effective method for compressing monotonically increasing sequences of integers. A key advantage of this technique is its ability to achieve near-optimal space occupancy, often requiring only a small overhead above the information-theoretic minimum, while simultaneously supporting efficient random access and search operations directly on the compressed form [13, 42]. This makes it highly suitable for applications such as inverted index compression in modern search engines [52, 39].

REPRESENTATION STRUCTURE Let's consider a sequence of n non-negative increasing integers:

$$S = \{s_0, s_1, \dots, s_{n-1}\}$$

where $0 \leqslant s_0 < s_1 < \dots < s_{n-1} < u$. The universe size is u, and we typically assume $u > n$. Each integer s_i can be represented using $b = \lceil \log_2 u \rceil$ bits. The Elias-Fano encoding strategy involves partitioning these b bits into two segments, based on a parameter l. The choice $l = \lfloor \log_2(u/n) \rfloor$ minimizes the total space requirement [11, 13] (if $u \leqslant n$, we set $l = 0$). The two parts are:

- The *lower bits*, $L(s_i)$, consisting of the l least significant bits of s_i.
- The *upper bits*, $H(s_i)$, consisting of the remaining $h = b - l$ most significant bits.

The representation then comprises two main components:

1. *Lower Bits Array (L):* This array is formed by concatenating the l-bit lower parts of all integers in the sequence: $L = L(s_0)L(s_1)\dots L(s_{n-1})$. The total size of this array is exactly $n \cdot l$ bits.
2. *Upper Bits Bitvector (H):* This bitvector encodes the distribution of the upper bits. For each possible value j (from 0 to $2^h - 1$) that the upper bits can assume, let c_j be the number of elements s_i in S for which $H(s_i) = j$. The bitvector H is constructed by concatenating, for $j = 0, 1, \dots, 2^h - 1$, a sequence of c_j ones followed by a single zero ($1^{c_j}0$). This structure results in a bitvector of length exactly $n + 2^h$, containing n ones (one for each element in S) and 2^h zeros (one acting as a delimiter for each possible upper bit value).

The space bound $n + 2^h \leqslant 2n$ holds if $2^h \leqslant n$, which corresponds to $u/n \leqslant n$. When u/n is small (dense sequences), l is small and h is large; when u/n is large (sparse sequences), l is large and h is small.

Theorem 2.36 (Elias-Fano Space Complexity [13])**.** *The Elias-Fano encoding of a strictly increasing sequence S of n integers in the range $[0, u)$ requires $n\lfloor \log_2(u/n) \rfloor + n + 2^h$ bits, where $h = \lceil \log_2 u \rceil - \lfloor \log_2(u/n) \rfloor$.*

This is upper bounded by $n \log_2(u/n) + 2n$ *bits, which is provably less than 2 bits per integer above the information-theoretic lower bound. The representation can be constructed in* $O(n)$ *time.*

Figure 8 illustrates the Elias-Fano encoding for the sequence $S = \{1, 4, 7, 18, 24, 26, 30, 31\}$. In this example, we have $n = 8$ integers in the universe $u = 32$. The total number of bits needed to represent any number in the universe is $b = \lceil \log_2 32 \rceil = 5$. Since $u/n = 32/8 = 4$, the number of lower bits is $l = \lfloor \log_2 4 \rfloor = 2$, and the number of upper bits is $h = b - l = 5 - 2 = 3$. The lower bits array L is formed by concatenating the $l = 2$ least significant bits of each s_i. The upper bits bitvector H is formed by counting the occurrences c_j of each upper bit value $j \in [0, 2^h - 1)$ and concatenating $1^{c_j}0$ for each j. For instance, $H(s_0) = 0$ occurs once ($c_0 = 1$), $H(s_1) = H(s_2) = 1$ occurs twice ($c_1 = 2$), $H = 2$ and $H = 3$ never occur ($c_2 = 0, c_3 = 0$), $H(s_3) = 4$ occurs once ($c_4 = 1$), $H = 5$ never occurs ($c_5 = 0$), $H(s_4) = H(s_5) = 6$ occurs twice ($c_6 = 2$), and $H(s_6) = H(s_7) = 7$ occurs twice ($c_7 = 2$). Concatenating 1^10 (for $H = 0$), 1^20 (for $H = 1$), 1^00 (for $H = 2$), 1^00 (for $H = 3$), 1^10 (for $H = 4$), 1^00 (for $H = 5$), 1^20 (for $H = 6$), and 1^20 (for $H = 7$) yields the final bitvector H.

i	s_i	$H(s_i)$ (val, 3b)	$L(s_i)$ (val, 2b)
0	1	0 (`000`)	1 (`01`)
1	4	1 (`001`)	0 (`00`)
2	7	1 (`001`)	3 (`11`)
3	18	4 (`100`)	2 (`10`)
4	24	6 (`110`)	0 (`00`)
5	26	6 (`110`)	2 (`10`)
6	30	7 (`111`)	2 (`10`)
7	31	7 (`111`)	3 (`11`)

$L =$ `0100111000101011` ($n \cdot l = 8 \times 2 = 16$ bits)
$H =$ `1011000100110110` ($n + 2^h = 8 + 2^3 = 16$ bits)

Figure 8: Elias-Fano encoding example for the sequence $S = \{1, 4, 7, 18, 24, 26, 30, 31\}$ with parameters $n = 8$, $u = 32$, $l = 2$, $h = 3$. The table shows the decomposition of each s_i into its upper $H(s_i)$ and lower $L(s_i)$ bits. Below the table are the resulting concatenated lower bits array L and the upper bits bitvector H.

QUERY OPERATIONS A significant advantage of the Elias-Fano representation is its support for direct queries on the compressed data. This requires augmenting the upper bits bitvector H with auxiliary data structures that enable constant-time calculation of *rank* and *select* queries (see Section 3.1 for details). These structures typically add a $o(n)$ bits to the overall space complexity. With these in place, the core operations are:

Access(i): This operation retrieves the i-th element s_i (using 0-based indexing for i, $0 \leqslant i < n$).

1. The lower l bits, $L(s_i)$, are directly read from the array L starting at bit position $i \cdot l$.

2. The position p in H corresponding to the end of the unary code for s_i is found using $p = \texttt{select}_1(H, i+1)$. The $\texttt{select}_1$ operation finds the position of the $(i+1)$-th bit set to 1.

3. The value of the upper h bits, $H(s_i)$, is determined by counting the number of preceding zeros in H up to position p. This count is precisely $H(s_i) = p - (i+1)$, as there are $i+1$ ones and $H(s_i)$ zeros up to that point. Alternatively, $H(s_i) = \texttt{rank}_0(H, p)$.

4. The original integer is reconstructed by combining the upper and lower parts: $s_i = (H(s_i) \ll l) \vee L(s_i)$, where $\ll$ denotes the bitwise left shift and $\vee$ denotes the bitwise OR.

Since reading from L and performing rank/select on H take constant time, *Access(i)* operates in $O(1)$ time [42].

Successor(x) (or *NextGEQ(x)*): This operation finds the smallest element s_i in S such that $s_i \geqslant x$, given a query value $x \in [0, u)$.

1. Determine the upper h bits $H(x)$ and lower l bits $L(x)$ of the query value x.

2. Identify the range of indices $[p_1, p_2)$ in S corresponding to elements whose upper bits are equal to $H(x)$. The starting index p_1 is the number of elements in S with upper bits strictly less than $H(x)$. This can be found by locating the $H(x)$-th zero in H using $\texttt{pos}_0 = \texttt{select}_0(H, H(x)+1)$ (using 1-based index for select); then $p_1 = \texttt{pos}_0 - H(x)$. The ending index p_2 (exclusive) is similarly found using the $(H(x)+1)$-th zero: $\texttt{pos}'_0 = \texttt{select}_0(H, H(x)+1+1)$; then $p_2 = \texttt{pos}'_0 - (H(x)+1)$.

3. Perform a search (e.g., binary search, or linear scan if $p_2 - p_1$ is small) over the lower bits $L[p_1 \cdot l \dots p_2 \cdot l - 1]$. The goal is to find the smallest index $k \in [p_1, p_2)$ such that the reconstructed value $(H(x) \ll l) \vee L(s_k)$ is greater than or equal to x.

4. If such a k is found within the range $[p_1, p_2)$, then s_k is the successor.

5. If no such element exists in the range (i.e., all elements with upper bits $H(x)$ are smaller than x), the successor must be the first element with upper bits greater than $H(x)$. This element is simply s_{p_2}, which can be retrieved using *Access(p_2)* (if $p_2 < n$).

The dominant cost is the search over the lower bits. Since there can be up to roughly u/n elements sharing the same upper bits in the worst case, the search step takes $O(\log(u/n))$ time using binary search. The select operations take $O(1)$ time. Thus, *Successor(x)* takes $O(1 + \log(u/n))$ time [42, 13].

Predecessor(x): Finding the largest element $s_i \leqslant x$ follows a symmetric logic, searching within the same index range $[p_1, p_2)$ identified using $H(x)$. If the search within the lower bits $L[p_1 \cdot l \ldots p_2 \cdot l - 1]$ yields a suitable candidate $s_k \leqslant x$, that is the answer (specifically, the largest such s_k). If all elements in the range $[p_1, p_2)$ are greater than x, or if the range is empty ($p_1 = p_2$), the predecessor must be the last element with upper bits less than $H(x)$, which is $s_{p_1 - 1}$ (if $p_1 > 0$). This can be retrieved using *Access(*$p_1 - 1$*)*. The time complexity is also $O(1 + \log(u/n))$.

2.7 STATISTICAL CODING

This section explores a technique called *statistical coding*: a method for compressing a sequence of symbols (*texts*) drawn from a finite alphabet Σ. The idea is to divide the process in two key stages: modeling and coding. During the modeling phase, statistical characteristics of the input sequence are analyzed to construct a model, typically estimating the probability $P(\sigma)$ for each symbol $\sigma \in \Sigma$. In the coding phase, this model is utilized to generate codewords for the symbols, which are then employed to compress the input sequence. We will focus on two popular statistical coding methods: Huffman coding and Arithmetic coding.

2.7.1 *Huffman Coding*

Compared to the methods seen in Section 2.6, Huffman Codes, introduced by David A. Huffman in his landmark 1952 paper [24], offer broader applicability. They construct optimal prefix-free codes for a given set of symbol probabilities, without requiring specific assumptions about the underlying distribution itself (beyond non-zero probabilities). This versatility makes them suitable for diverse data types, including text where symbol frequencies often lack a simple mathematical pattern.

For instance, in English text, the letter *e* is far more frequent than *z*, and simple integer codes based on alphabetical order would be highly inefficient. Huffman coding directly addresses this by assigning shorter codewords to more frequent symbols.

CONSTRUCTION OF HUFFMAN CODES The construction algorithm is greedy and builds a binary tree bottom-up. Each symbol $\sigma \in \Sigma$ initially forms a leaf node, typically weighted by its probability $P(\sigma)$ or its frequency count n_σ. The algorithm repeatedly selects the two nodes (initially leaves, later internal nodes representing merged subtrees) with the smallest current weights, merges them into a new internal node whose weight is the sum of the two merged weights, and places the two selected nodes as its children. This process continues until only one node, the root, remains.

The prefix-free code for each symbol σ is then determined by the path from the root to the leaf corresponding to σ. Conventionally, a 0 is assigned to traversing a left branch and a 1 to a right branch (or vice versa). The concatenation of these bits along the path forms the Huffman code for the symbol. More formal descriptions and variations can be found in [13, 48, 22, 10].

Example 2.37 (Huffman Coding Construction). *Let $\Sigma = \{a, b, c, d, e\}$ with probabilities $P(a) = 0.25$, $P(b) = 0.25$, $P(c) = 0.2$, $P(d) = 0.15$, $P(e) = 0.15$.*

1. *Initial nodes: (a: 0.25), (b: 0.25), (c: 0.2), (d: 0.15), (e: 0.15).*
2. *Merge smallest: d and e. Create node (de: 0.30). Current nodes: (a: 0.25), (b: 0.25), (c: 0.2), (de: 0.30).*
3. *Merge smallest: c and a. Create node (ca: 0.45). Current nodes: (b: 0.25), (de: 0.30), (ca: 0.45). (Note: Choosing c and a over c and b is arbitrary here, another valid tree exists).*
4. *Merge smallest: b and de. Create node (bde: 0.55). Current nodes: (ca: 0.45), (bde: 0.55).*
5. *Merge last two: ca and bde. Create root (root: 1.00).*

The resulting tree and codes (assigning 0 to left, 1 to right) are shown in Figure 9.

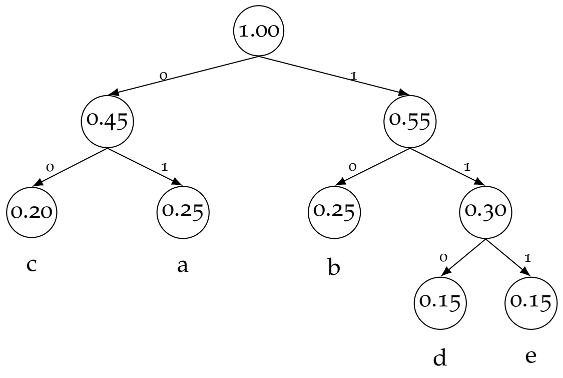


Figure 9: Huffman tree for the example probabilities ($P(a) = 0.25, P(b) = 0.25, P(c) = 0.2, P(d) = 0.15, P(e) = 0.15$). The resulting codes (0 for left, 1 for right) are: $C(a) = \texttt{01}$, $C(b) = \texttt{10}$, $C(c) = \texttt{00}$, $C(d) = \texttt{110}$, $C(e) = \texttt{111}$.

Let $L_C = \sum_{\sigma \in \Sigma} P(\sigma) \cdot l(\sigma)$ be the average codeword length for a prefix-free code C, where $l(\sigma)$ is the length of the codeword assigned to symbol σ. The Huffman coding algorithm produces a code C_H that is optimal among all possible prefix-free codes for the given probability distribution.

Theorem 2.38 (Optimality of Huffman Codes). *Let C_H be a Huffman code generated for a given probability distribution P over alphabet Σ. For any other prefix-free code C' for the same distribution, the average codeword length satisfies $L_{C_H} \leqslant L_{C'}$.*

This optimality signifies that no other uniquely decodable code assigning fixed codewords to symbols can achieve a shorter average length. The proof typically relies on induction or an exchange argument, demonstrating that any deviation from the greedy merging strategy cannot improve the average length [13, 48, 22, 10].

The length of individual Huffman codewords can vary. In the worst case, the longest codeword might approach $|\Sigma| - 1$ bits (in a highly skewed distribution). However, a tighter bound related to the minimum probability $p_{\min}$ exists: the maximum length is $O(\log(1/p_{\min}))$ [36]. If probabilities derive from empirical frequencies in a text of length n, then $p_{\min} \geqslant 1/n$, bounding the maximum codeword length by $O(\log n)$. The encoding process itself, once the tree (or equivalent structure) is built, is typically linear in the length of the input sequence S, i.e., $O(|S|)$.

Decoding uses the Huffman Tree (or an equivalent lookup structure). Bits are read sequentially from the compressed stream, traversing the tree from the root according to the bit values (e.g., 0 for left, 1 for right) until a leaf node is reached. The symbol associated with that leaf is output, and the process restarts from the root for the next symbol. The total decoding time is proportional to the total number of bits in the compressed sequence. Since individual codes have length $O(\log n)$ in the empirical case, decoding a single symbol takes at most $O(\log n)$ bit reads and tree traversals.

While optimal among prefix codes, Huffman coding still assigns an integer number of bits to each symbol. This leads to a slight inefficiency compared to the theoretical entropy limit, as quantified by the following theorem.

Theorem 2.39. *Let $\mathcal{H} = \sum_{\sigma \in \Sigma} P(\sigma) \log_2(1/P(\sigma))$ be the entropy of a source emitting symbols from Σ according to distribution P. The average length L_H of the corresponding Huffman code is bounded by $\mathcal{H} \leqslant L_H < \mathcal{H} + 1$.*

Proof. The lower bound $\mathcal{H} \leqslant L_H$ follows directly from Shannon's source coding theorem (Theorem 2.32), which states that $\mathcal{H}$ is the minimum possible average length for any uniquely decodable code. For the upper bound, consider assigning an "ideal" codeword length $l'_\sigma = -\log_2 P(\sigma)$ to each symbol σ. As we must use an integer number of bits, let $l_\sigma = \lceil -\log_2 P(\sigma) \rceil$. Clearly, $l_\sigma < -\log_2 P(\sigma) + 1$. Summing these l_σ satisfies Kraft's inequality ($\sum 2^{-l_\sigma} \leqslant \sum 2^{-(-\log_2 P(\sigma))} = \sum P(\sigma) = 1$), guaranteeing the existence of a prefix code C' with these lengths l_σ. Its average length is $L_{C'} = \sum P(\sigma) l_\sigma < \sum P(\sigma)(-\log_2 P(\sigma) + 1) = \mathcal{H} + 1$. Since Huffman coding produces the optimal prefix code (Theorem 2.38), its average length L_H must be less than or equal to $L_{C'}$. Therefore, $L_H \leqslant L_{C'} < \mathcal{H} + 1$. Combining bounds gives $\mathcal{H} \leqslant L_H < \mathcal{H} + 1$. □

This theorem highlights that the average Huffman code length is always within one bit of the source entropy. The gap $(L_H - \mathcal{H})$ represents the inefficiency due to the constraint of using integer bit lengths for each symbol's codeword. This gap is significant only when some symbol probabilities are very high (close to 1).

2.7.2 *Arithmetic Coding*

Introduced conceptually by Peter Elias in the 1960s and later developed into practical algorithms by Rissanen [46] and Pasco [40] in the 1970s, Arithmetic Coding offers a more powerful approach to statistical compression than Huffman coding. Its key advantage lies in its ability to approach the theoretical entropy limit more closely, often achieving better compression ratios, especially when dealing with skewed probability distributions or when encoding sequences rather than individual symbols.

Unlike Huffman coding, which assigns a distinct, fixed-length (integer number of bits) prefix-free code to each symbol, Arithmetic coding represents an entire sequence of symbols as a single fraction within the unit interval $[0, 1)$. The length of the binary representation of this fraction effectively corresponds to the information content (entropy) of the entire sequence, allowing for an average representation that can use a fractional number of bits per symbol. This overcomes the inherent inefficiency of Huffman coding, which is bounded by $\mathcal{H} \leqslant L_H < \mathcal{H} + 1$. Arithmetic coding aims to achieve a compressed size very close to $n\mathcal{H}$ bits for a sequence of length n.

2.7.2.1 *Encoding and Decoding Process*

Let $S = S[1]S[2] \dots S[n]$ be the input sequence of symbols drawn from alphabet Σ, and let $P(\sigma)$ be the probability of symbol σ according to the chosen statistical model.

The core idea of the encoding process (Algorithm 1) is to progressively narrow down a sub-interval of $[0, 1)$. Initially, the interval is $[l_0, h_0) = [0, 1)$. For each symbol $S[i]$ in the sequence, the current interval $[l_{i-1}, h_{i-1})$ of size $s_{i-1} = h_{i-1} - l_{i-1}$ is partitioned into smaller sub-intervals, one for each symbol $\sigma \in \Sigma$. The size of the sub-interval for σ is proportional to its probability, $s_{i-1} \cdot P(\sigma)$. The algorithm then selects the sub-interval corresponding to the actual symbol $S[i]$ and makes it the new current interval $[l_i, h_i)$ for the next step. The cumulative probability function $C(\sigma) = \sum_{\sigma' \leqslant \sigma} P(\sigma')$ is used to efficiently calculate the start (l_i) of the correct sub-interval. After processing all n symbols, the final interval is $[l_n, h_n) = [l_n, l_n + s_n)$, where $s_n = \prod_{i=1}^{n} P(S[i])$.

Algorithm 1 Arithmetic Coding (Conceptual)

```
Require: Sequence $S = S[1..n]$, Probabilities $P(\sigma)$ for $\sigma \in \Sigma$
Ensure: A sub-interval $[l_n, l_n + s_n)$ uniquely identifying S.
 1: Compute cumulative probabilities $C(\sigma) = \sum_{\sigma' < \sigma} P(\sigma')$ (Note: sum over $\sigma' < \sigma$)
 2: $l \leftarrow 0$
 3: $s \leftarrow 1$                                    ▷ Initial interval [0, 1), size 1
 4: for $i = 1$ to $n$ do
 5:     $l_{new} \leftarrow l + s \cdot C(S[i])$        ▷ Calculate start of sub-interval
 6:     $s_{new} \leftarrow s \cdot P(S[i])$            ▷ Calculate size of sub-interval
 7:     $l \leftarrow l_{new}$
 8:     $s \leftarrow s_{new}$
 9: end for
10: return $[l, l + s)$                     ▷ Final interval represents the sequence
```

The final output of the encoder is not the interval itself, but rather a binary fraction x that falls within this final interval $[l_n, l_n + s_n)$ and can be represented with the fewest possible bits. Practical implementations use techniques to incrementally output bits as soon as they are determined (i.e., when the interval lies entirely within $[0, 0.5)$ or $[0.5, 1)$) and rescale the interval to maintain precision using fixed-point arithmetic [35, 13].

The decoding process (Algorithm 2) essentially reverses the encoding. The decoder needs the compressed bitstream (representing the fraction x), the same probability model $P(\sigma)$, and the original sequence length n. It starts with the interval $[0, 1)$. In each step i, it determines which symbol σ's sub-interval $[l + s \cdot C(\sigma), l + s \cdot C(\sigma) + s \cdot P(\sigma))$ contains the encoded fraction x. That symbol σ must be $S[i]$. The decoder outputs σ and updates its current interval to be this sub-interval, just as the encoder did. This is repeated n times to reconstruct the original sequence S.

2.7.2.2 *Efficiency of Arithmetic Coding*

The final interval size $s_n = \prod_{i=1}^{n} P(S[i])$ is crucial. If we use empirical probabilities $P(\sigma) = n_\sigma / n$, where n_σ is the frequency of σ in S, then $s_n = \prod_{\sigma \in \Sigma} (n_\sigma / n)^{n_\sigma}$. As noted before, the number of bits required to uniquely specify a number within an interval of size s_n is approximately $-\log_2 s_n$.

Algorithm 2 Arithmetic Decoding (Conceptual)

```
Require: Encoded fraction $x$, Probabilities $P(\sigma)$, Sequence length $n$.
Ensure: Original sequence $S[1..n]$.
 1: Compute cumulative probabilities $C(\sigma) = \sum_{\sigma' < \sigma} P(\sigma')$
 2: $l \leftarrow 0$
 3: $s \leftarrow 1$
 4: $S \leftarrow$ empty sequence
 5: for $i = 1$ to $n$ do
 6:     Find symbol $\sigma$ such that $l + s \cdot C(\sigma) \leqslant x < l + s \cdot (C(\sigma) + P(\sigma))$
 7:     S.append($\sigma$)
 8:     $l_{new} \leftarrow l + s \cdot C(\sigma)$
 9:     $s_{new} \leftarrow s \cdot P(\sigma)$
10:     $l \leftarrow l_{new}$
11:     $s \leftarrow s_{new}$
12:     ▷ Practical decoders also update $x$ relative to the new interval
13: end for
14: return S
```

Calculating $-\log_2 s_n$ with empirical probabilities gives:

$$\begin{aligned} -\log_2 \left(\prod_{\sigma \in \Sigma} \left(\frac{n_\sigma}{n} \right)^{n_\sigma} \right) &= -\sum_{\sigma \in \Sigma} n_\sigma \log_2 \left(\frac{n_\sigma}{n} \right) \\ &= n \sum_{\sigma \in \Sigma} \frac{n_\sigma}{n} \log_2 \left(\frac{n}{n_\sigma} \right) \\ &= n\mathcal{H} \end{aligned}$$

where $\mathcal{H}$ is the empirical (0-th order) entropy of the sequence S. This demonstrates that the *ideal* number of bits needed by arithmetic coding matches the entropy of the sequence exactly.

The connection between the final interval size s_n and the actual number of output bits deserves clarification. The encoder needs to transmit a binary representation of *some* number x that lies within the final interval $[l_n, l_n + s_n)$. To ensure the decoder can uniquely identify this interval (and thus the sequence), the chosen number x must be distinguishable from any number lying in adjacent potential intervals. This requires a certain precision. The minimum number of bits k needed to represent such an x as a dyadic fraction (i.e., a number of the form $N/2^k$) must satisfy $2^{-k} \leqslant s_n$. This condition ensures that the precision 2^{-k} is fine enough to pinpoint a unique value within the target interval of size s_n. Taking logarithms, this implies $k \geqslant -\log_2 s_n$. To guarantee that such a fraction actually *exists* within the interval, and to handle the process of incrementally outputting bits, practical arithmetic coding requires slightly more bits than the theoretical minimum $-\log_2 s_n$. A careful analysis shows that at most 2 extra bits are needed beyond the ideal $n\mathcal{H}$ [13].

Theorem 2.40. *The number of bits emitted by arithmetic coding for a sequence* S *of* n *symbols, using probabilities* $P(\sigma)$ *derived from the empirical frequencies within* S*, is at most* $2 + n\mathcal{H}$*, where* $\mathcal{H}$ *is the empirical entropy of the sequence* S*.*

Proof. Formal proofs can be found in standard texts on information theory and data compression [13, 48, 22, 10]. The core idea, as outlined above, relates the required number of bits k to the final interval size $s_n = 2^{-n\mathcal{H}}$ via $k \approx -\log_2 s_n = n\mathcal{H}$. The additive constant accounts for representing a specific point within the interval. $\square$

Remark 2.41. *Practical arithmetic coders do not use floating-point numbers due to precision issues. They employ integer arithmetic, maintaining the interval bounds* $[L, H)$ *as large integers within a fixed range (e.g., 16 or 32 bits). As the conceptual interval shrinks, common leading bits of* L *and* H *are output, and the integer interval is rescaled (e.g., doubled) to occupy the full range again, effectively shifting the conceptual interval. Special handling ("underflow") is needed when the interval becomes very small but straddles the midpoint (e.g., 0.5), preventing immediate output of the next bit. These implementation details ensure correctness and efficiency with fixed-precision arithmetic [35].*

3 RANK AND SELECT

Building upon the concepts of data compression and information theory from the preceding chapters, we now address the construction of *succinct data structures*. As motivated in Section 1.1, the objective is to represent discrete structures using space close to their information-theoretic minimum, while supporting efficient query operations directly on their compressed representation.

This chapter focuses on the fundamental rank and select operations. Given a sequence, rank counts occurrences of specified elements up to a given position, whereas select finds the position of the i-th occurrence of a specified element. The efficient implementation of these queries is critical for the functionality of many succinct data structures. We will investigate methods to support rank and select efficiently, typically achieving constant query time, through the use of auxiliary structures whose space requirement is sublinear relative to the input size.

Our examination proceeds in three stages. First, we consider the foundational case of *bitvectors* (binary sequences). We will analyze techniques, including hierarchical decomposition methods, for constructing auxiliary structures that use $o(n)$ bits of space, where n is the bitvector length, enabling constant-time rank and select queries on the original bitvector. Second, we generalize these concepts to sequences defined over larger, finite alphabets. We will study *Wavelet Trees*, a structure that reduces rank and select operations on general strings to corresponding operations performed on underlying bitvectors. Finally, the chapter addresses the more recent case of *degenerate strings*, which are sequences where each position may represent a subset of characters from the alphabet. We will review approaches that extend rank and select capabilities to this setting, by adapting the principles established for standard strings and bitvectors.

3.1 BITVECTORS

We begin our study with the most fundamental sequence type, the *bitvector* $B[1..n]$, a sequence of n bits from $\{0, 1\}$. Our primary objective is to support two essential query operations on B efficiently: $\mathit{rank}_b(B, i)$, which counts the occurrences of bit b in the prefix $B[1..i]$, and $\mathit{select}_b(B, i)$, which finds the index of the i-th occurrence of bit b in B. While these operations can be answered by scanning B in $O(n)$ time, we seek constant-time solutions, $O(1)$, by augmenting B with *succinct* auxiliary data structures. These structures should occupy $o(n)$ bits, leading to a total space usage of $n + o(n)$ bits when storing B explicitly. We now formally define these operations.

Definition 3.1 (Rank)**.** *Given a bitvector* $B[1..n]$*, the* ***rank*** *of an index* i *(*$1 \leqslant i \leqslant n$*) relative to a bit* $c \in \{0, 1\}$ *is the number of occurrences of* c *in the prefix* $B[1..i]$*. We denote it as* $\mathrm{rank}_c(i)$*. Specifically, for* $c = 1$*:*

$$\mathrm{rank}_1(i) = \sum_{j=1}^{i} B[j]$$

The rank for $c = 0$ *can be derived as* $\mathrm{rank}_0(i) = i - \mathrm{rank}_1(i)$*.*

Definition 3.2 (Select)**.** *Given a bitvector* $B[1..n]$*, the* ***select*** *of the* i*-th occurrence of a bit* $c \in \{0, 1\}$ *is the index* j *such that* $B[j] = c$ *and* $\mathrm{rank}_c(j) = i$*. We denote it as* $\mathrm{select}_c(i)$*. If the* i*-th occurrence of* c *does not exist,* $\mathrm{select}_c(i)$ *is undefined (or returns a special value). Unlike rank,* $\mathrm{select}_0(i)$ *cannot generally be computed directly from* $\mathrm{select}_1(i)$ *in constant time.*

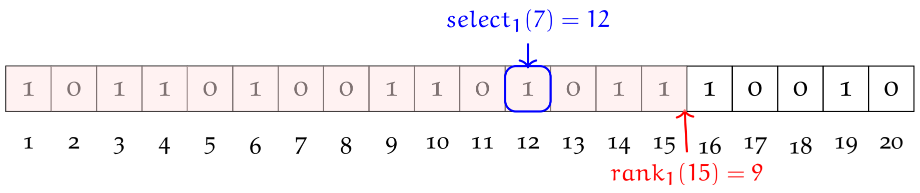


Figure 10: Example of a bitvector $B[1..20]$. The prefix $B[1..15]$ (shaded red) contains 9 ones, so $\mathrm{rank}_1(15) = 9$. The 7th 1 (circled blue) occurs at index 12, so $\mathrm{select}_1(7) = 12$.

Example 3.3 (Rank and Select on a plain bitvector)**.** *Let* B *be the bitvector of length* $n = 20$ *shown in Figure 10:*

$$B = 10110100110101110010$$

- $\mathrm{rank}_1(15)$*: We count the number of 1s in the prefix* $B[1..15]$*.*

  $$B[1..15] = \underbrace{101101001101011}_{15\ bits}$$

  By scanning, there are 9 ones. Therefore, $\mathrm{rank}_1(15) = 9$*.*

- $\mathrm{select}_1(7)$*: We find the index of the 7th occurrence of* 1 *in* B*. Scanning* B*: the 1st* 1 *is at index 1, 2nd at 3, 3rd at 4, 4th at 6, 5th at 9, 6th at 10, and the 7th* 1 *is at index 12. Therefore,* $\mathrm{select}_1(7) = 12$*.*

These manual calculations illustrate the definitions. Efficient data structures avoid such linear scans.

Bitvectors supporting efficient rank and select operations are indeed foundational components for many compressed and succinct data structures. Attaining high performance for these operations is therefore a central concern. The following sections will describe methods to construct the $o(n)$-bit auxiliary structures that achieve constant query times. Furthermore, it is often the case that bitvectors encountered in applications exhibit skewed distributions of 0s and 1s (i.e., they are sparse). While the $n + o(n)$ structures operate on the explicit bitvector, separate lines of research have explored compressing the bitvector B itself by leveraging these statistical properties, thereby reducing the initial n-bit storage requirement.

Significant research, exemplified by the work of Sadakane and Grossi [47], has addressed the integration of compression with query support. Such approaches aim to achieve space bounds related to the empirical entropy of the bitvector, for instance $n\mathcal{H}_k(B) + o(n)$ bits using k-th order entropy, while simultaneously maintaining constant query times for rank and select. This direction differs from the direct application of general-purpose compression algorithms presented in Chapter 2.

Remark 3.4. *Applying standard symbol-wise coding techniques from Chapter 2 (such as Huffman or Arithmetic coding) directly to a bitvector B typically yields a compressed size related to its zero-order entropy, approximately $n\mathcal{H}_0(B)$ bits. While potentially reducing space, especially for biased bitvectors, this form of compression generally obstructs efficient random access and the direct computation of* rank *and* select *queries without significant decompression overhead. The specialized structures detailed in this chapter are expressly designed to provide both space efficiency and fast query capabilities.*

3.1.1 *Rank*

A fundamental approach to support rank queries in constant time using sublinear additional space was introduced by Jacobson [25]. The technique relies on a hierarchical decomposition of the bitvector $B[1..n]$ and precomputation of ranks at different granularities. The auxiliary structures occupy $o(n)$ bits in total.

The structure typically employs two levels of blocking on top of the original bitvector B. First, B is conceptually divided into *superblocks* of size Z. Second, each superblock is further divided into *blocks* of size z. Common parameter choices yielding $o(n)$ overhead are $Z =$

$\Theta(\log^2 n)$ and $z = \Theta(\log n)$, for example $Z = \lfloor \log^2 n \rfloor$ and $z = \lfloor (1/2) \log n \rfloor$. We assume for simplicity that z divides Z and Z divides n.

Two auxiliary arrays store precomputed rank information. The first array, R_S, stores the absolute rank at the beginning of each superblock:

$$R_S[k] = \mathsf{rank}_1(B, k \cdot Z) \qquad k = 0, \ldots, n/Z - 1$$

The second array, R_B, stores the rank within a superblock at the beginning of each block, relative to the start of the superblock. Specifically, for the l-th block overall, which belongs to superblock $k = \lfloor l \cdot z/Z \rfloor$

$$R_B[l] = \mathsf{rank}_1(B, l \cdot z) - \mathsf{rank}_1(B, k \cdot Z)$$

Finally, a *lookup table*, often denoted T, is used to determine the rank within a small block of size z. For every possible z-bit pattern p, and every position $j \in [1, z]$, $T[p][j]$ stores the value $\mathsf{rank}_1(p, j)$, i.e., the number of set bits in the first j positions of the pattern p.

Figure 11 shows a visual representation of the hierarchical data structure.

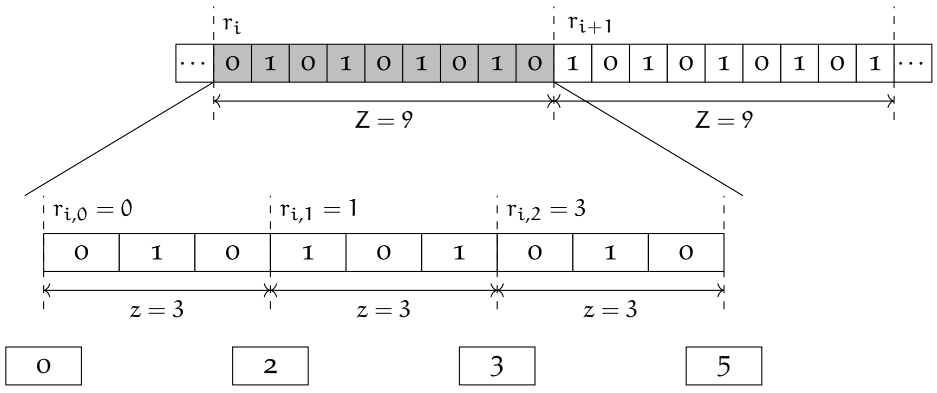


Figure 11: The hierarchical rank data structure. The first level is composed of superblocks of size Z (conceptually storing absolute ranks r_i), the second level of blocks of size z (storing relative ranks $r_{i,j}$), and the third level uses a lookup table (represented abstractly by the bottom row).

The lookup table T allows determining the number of set bits within any prefix of a z-bit block in constant time. Table 1 shows an example for $z = 3$. For a block p, the entry in column j (corresponding to $\mathsf{rank}_1(p, j)$) gives the precomputed result.

We can state the following theorem regarding the space and time complexity [25].

Theorem 3.5. *Given a bitvector* $B[1..n]$, *there exists an auxiliary data structure using* $o(n)$ *bits that allows computing* $\mathit{rank}_b(B, i)$ *for any* $i \in [1, n]$

block	$r_{i,0}$	$r_{i,1}$	$r_{i,2}$
000	0	0	0
001	0	0	1
010	0	1	1
011	0	1	2
100	1	1	1
101	1	1	2
110	1	2	2
111	1	2	3

Table 1: Example of a lookup table T for intra-block rank computation with $z = 3$. Each row corresponds to a possible z-bit pattern p. The cell for pattern p and index j stores $\mathsf{rank}_1(p, j)$.

and $b \in \{0, 1\}$ *in* $O(1)$ *time. The original bitvector* B *is accessed only in read mode. The total space required is* $n + o(n)$ *bits.*

Proof. We construct the auxiliary data structure using a standard two-level hierarchical decomposition of the bitvector B. We define a *superblock* size $Z = \lfloor \log^2 n \rfloor$ and a *block* size $z = \lfloor (1/2) \log n \rfloor$. For analytical simplicity, assume n is a multiple of Z, and Z is a multiple of z.

The structure comprises three main components designed to store precomputed rank information at different scales.

First, the *Superblock Rank Directory (*R_S*)* is an array storing the absolute rank at the start of each superblock. Specifically, for $k \in [0, n/Z - 1]$, $R_S[k] = \mathsf{rank}_1(B, k \cdot Z)$. Since each rank value is at most n, storing these requires $\lceil \log n \rceil$ bits per entry. The total space for R_S is:

$$\begin{aligned}\text{Space}(R_S) &= \frac{n}{Z}\lceil \log n \rceil = \frac{n}{\lfloor \log^2 n \rfloor}\lceil \log n \rceil \\ &= O\left(\frac{n \log n}{\log^2 n}\right) = O(n/\log n) \text{ bits.}\end{aligned}$$

Second, the *Block Rank Directory (*R_B*)* stores ranks relative to the start of the containing superblock. For each block index $l \in [0, n/z - 1]$, let $k = \lfloor l \cdot z/Z \rfloor$ be its superblock index. $R_B[l]$ stores the relative rank $\mathsf{rank}_1(B, l \cdot z) - \mathsf{rank}_1(B, k \cdot Z)$. This value is at most Z, requiring $\lceil \log Z \rceil$ bits per entry. The total space for R_B is:

$$\begin{aligned} \text{Space}(R_B) &= \frac{n}{z}\lceil \log Z = \frac{n}{\lfloor (1/2)\log n \rfloor}\lceil \log(\lfloor \log^2 n \rfloor)\rceil \\ &= O\left(\frac{n}{\log n}\log(\log^2 n)\right) \\ &= O\left(\frac{n \log\log n}{\log n}\right) \text{ bits.} \end{aligned}$$

We verify that the space complexity for both directories R_S and R_B is sublinear, i.e., $o(n)$. For R_S, we have $O(n/\log n)$. As $n \to \infty$, $(n/\log n)/n = 1/\log n \to 0$. For R_B, we have $O(n \log\log n/\log n)$. As $n \to \infty$, $(n \log\log n/\log n)/n = \log\log n/\log n \to 0$. Thus, both $\text{Space}(R_S)$ and $\text{Space}(R_B)$ are $o(n)$.

Third, the *Intra-Block Rank Mechanism (*T*)* is responsible for determining the rank within any z-bit block in constant time. A direct precomputation approach involves storing, for each of the 2^z possible z-bit patterns p and each position $j \in [1, z]$, the value $\mathsf{rank}_1(p, j)$. The space requirement for such a naive table would be:

$$\text{Space(Naive } T) = O(2^z \cdot z \cdot \log z).$$

Substituting $z = \lfloor (1/2)\log n \rfloor$, this becomes

$$O(2^{(1/2)\log n} \log n \log\log n) = O(\sqrt{n} \log n \log\log n)$$

which is not sublinear ($o(n)$). The crucial insight, originally provided by Jacobson [25], is that this intra-block rank functionality can indeed be implemented using auxiliary structures occupying only $o(n)$ bits while still supporting $O(1)$ query time. This often involves techniques like further table compression or specialized indexing strategies not detailed here, but whose existence and performance guarantees we rely upon.

The total auxiliary space is the sum of the space complexities for R_S, R_B, and the efficient implementation of T. This sum is $O(n/\log n) + O(n \log\log n/\log n) + o(n)$, which simplifies to $o(n)$. Therefore, the total space including the original bitvector is $n + o(n)$ bits. □

It is worth noting a crucial optimization for practical implementations, especially when the block size z is chosen such that it fits within a machine word (e.g., $z \leqslant 64$ on standard 64-bit architectures). In this scenario, the theoretical $o(n)$-bit lookup mechanism T for intra-block rank can often be replaced by direct computation using highly optimized hardware instructions.

Specifically, to compute the rank within the z-bit block $p = B[l \cdot z + 1..(l+1) \cdot z]$ up to position j, i.e., $\mathsf{rank}_1(p, j)$, one can perform bitwise operations. First, isolate the j-bit prefix of p[1]. Then, the number of set bits in this prefix can be computed efficiently using the processor's population count (popcount) instruction. Modern programming languages often expose this functionality directly; for instance, the Rust standard library provides the `count_ones()` method on primitive integer types. This operation is typically executed in constant time (often a single machine instruction) and can be significantly faster in practice than accessing a more complex precomputed table structure, especially for small values of z. We will discuss this in more detail in Section 3.1.4.

To answer a query $\mathsf{rank}_1(B, i)$, we perform the following constant-time steps: Calculate the relevant indices: superblock $k = \lfloor (i-1)/Z \rfloor$, block $l = \lfloor (i-1)/z \rfloor$, and intra-block position $j = (i-1) \pmod z + 1$. Retrieve the precomputed ranks: $rank_S = R_S[k]$ from the superblock directory and $rank_B = R_B[l]$ from the block directory. Access the z-bit block $p = B[l \cdot z + 1..(l+1) \cdot z]$ from the original bitvector B. This access takes $O(1)$ time on a Word RAM where the word size $w \geqslant \log n \geqslant z$. Compute the intra-block rank $rank_T = T(p, j)$ using the $o(n)$-space constant-time mechanism T. The final rank is obtained by summing these components:

$$\mathsf{rank}_1(B, i) = rank_S + rank_B + rank_T.$$

All steps involve only constant-time operations (arithmetic, array lookups, memory access of z bits, and the T lookup), hence the total query time is $O(1)$.

The query $\mathsf{rank}_0(B, i)$ is then computed simply as $i - \mathsf{rank}_1(B, i)$, also in constant time.

3.1.2 *Select*

The select operation serves as the inverse to rank. Formally, for a bitvector B and an integer i, $\mathsf{select}_b(B, i)$ returns the index j such that $B[j] = b$ and $\mathsf{rank}_b(B, j) = i$. This relationship can be expressed as:

$$\mathsf{rank}_b(B, \mathsf{select}_b(B, i)) = i$$

provided the i-th occurrence of b exists.

Supporting select queries efficiently requires a different auxiliary structure compared to the fixed-size blocking used for rank. An approach providing constant-time select within this space framework was developed by Clark [8]. It relies on a multi-level hierarchy guided by the number of set bits (specifically, 1s, as select_0 can be handled symmetrically or via select_1 on the complemented bitvector with additional

1 For example, using a bitmask like $p \,\&\, ((1 \ll j) - 1)$

rank structures). The core idea is to partition the bitvector B based on the cumulative count of 1s and use multiple levels of indexing structures to locate the i-th 1 quickly.

We begin by designing the first level of the select data structure. The bitvector B is conceptually divided into variable-length *chunks*, such that each chunk (except possibly the last) contains exactly K set bits. A typical choice is $K = \Theta(\log n \log\log n)$. We store an array P_1 containing the starting position (index in B) of each chunk. The number of chunks is $\lceil m/K \rceil$, where $m = \mathsf{rank}_1(B, n)$ is the total number of 1s. The space for P_1 is

$$O(m/K \cdot \log n) = O(n/(\log n \log\log n) \cdot \log n) = O(n/\log\log n)$$

which is $o(n)$. Given a query $\mathsf{select}_1(i)$, the relevant chunk index can be determined as $k = \lceil i/K \rceil$, and its starting position retrieved from P_1.

The second step addresses how to find the target 1 within its chunk. Let the length of a chunk be Z. We categorize chunks into two types based on their density. A chunk is considered *sparse* if its length Z is large compared to K, specifically $Z > K^2$. Conversely, a chunk is *dense* if $Z \leqslant K^2$. For sparse chunks, we can afford to store the relative positions (offsets from the chunk's start) of the K set bits explicitly. The total space required across all sparse chunks for these offsets can be shown to be $o(n)$ [8]. If the target 1 falls within a sparse chunk, its position is found by calculating its relative rank $i' = (i-1) \pmod K + 1$ and retrieving the i'-th stored offset. For dense chunks ($Z \leqslant K^2$), explicitly storing offsets is too costly. Instead, we introduce a second level of structure within these dense chunks.

This second level structure subdivides each dense chunk into smaller variable-length *sub-chunks*, each containing exactly k set bits, where $k = \Theta((\log\log n)^2)$. We store an array P_2 for each dense chunk, holding the starting positions of these sub-chunks relative to the start of the dense chunk. The total space for all P_2 structures across all dense chunks is $O(m/k \cdot \log Z_{max}) = O(n/k \cdot \log K^2) = O(n/(\log\log n)^2 \cdot \log(\log n \log\log n)) = o(n)$.

Finally, we need to handle the sub-chunks. Let a sub-chunk have length z. Similar to the chunk level, we distinguish between *sparse sub-chunks* ($z > k^2$) and *dense sub-chunks* ($z \leqslant k^2$). For sparse sub-chunks, we again store the relative positions of the k set bits explicitly; the total space across all sparse sub-chunks remains $o(n)$ [8]. For dense sub-chunks ($z \leqslant k^2$), we require a mechanism to find the i''-th 1 (where i'' is the rank relative to the sub-chunk start) within the z bits in constant time. It is known [8] that constant-time select within a block of size $z = O(\text{polylog } n)$ can be achieved using an auxiliary structure of size $o(z)$ bits associated with the block (e.g., using precomputed tables or other techniques). Summing over all dense sub-chunks, the total space for these Level 3 mechanisms is $o(n)$.

The overall query process involves navigating this hierarchy. The algorithm can be summarized by the following pseudocode.

Algorithm 3 Select_1 Algorithm (Conceptual)

```
function $\mathrm{Select}_1(B, i)$
    $k \leftarrow \lceil i/K \rceil$
    $pos_1 \leftarrow$ GetChunkStartPos$(k)$
    $ChunkInfo \leftarrow$ GetChunkInfo$(k)$
    if $ChunkInfo$ indicates sparse ($Z > K^2$) then
        $i' \leftarrow (i-1) \pmod K + 1$
        $offset \leftarrow$ GetSparseChunkOffset$(k, i')$
        return $pos_1 + offset$
    else
        $i' \leftarrow (i-1) \pmod K + 1$
        $l \leftarrow \lceil i'/k \rceil$
        $pos_2 \leftarrow$ GetSubChunkStartPos$(k, l)$
        $SubChunkInfo \leftarrow$ GetSubChunkInfo$(k, l)$
        if $SubChunkInfo$ indicates sparse ($z > k^2$) then
            $i'' \leftarrow (i'-1) \pmod k + 1$
            $offset \leftarrow$ GetSparseSubChunkOffset$(k, l, i'')$
            return $pos_1 + pos_2 + offset$
        else
            $i'' \leftarrow (i'-1) \pmod k + 1$
            $offset \leftarrow$ DenseSubChunkSelect$(k, l, i'')$
            return $pos_1 + pos_2 + offset$
        end if
    end if
end function
```

As for the rank data structure, we can state the following theorem:

Theorem 3.6. *Given a bitvector $B[1..n]$, there exists an auxiliary data structure using $o(n)$ bits that allows computing $\mathsf{select}_b(B, i)$ for any valid i and $b \in \{0, 1\}$ in $O(1)$ time. The original bitvector B is accessed only in read mode. The total space required is $n + o(n)$ bits.*

Proof. The constant query time follows from the algorithm described, where each step (calculating indices, accessing pointer structures P_1, P_2, retrieving stored offsets for sparse cases, or using the constant-time Level 3 dense mechanism) takes $O(1)$ time. The total auxiliary space is the sum of the space required for P_1, the relative offsets for sparse chunks, the P_2 arrays, the relative offsets for sparse sub-chunks, the Level 3 dense sub-chunk mechanisms, and the structures needed to distinguish between sparse and dense cases. As analyzed during the description, each component requires $o(n)$ bits with parameters $K = \Theta(\log n \log\log n)$ and $k = \Theta((\log\log n)^2)$, relying on established results [8]. Therefore, the total auxiliary space is $o(n)$ bits, yielding an overall space of $n + o(n)$ bits. □

Remark 3.7 (Practical Considerations)**.** *The threshold $k^2 = \Theta((\log \log n)^4)$ for handling the smallest dense blocks can be extremely small for practical values of n. In implementations, if k is small enough (e.g., fits within a machine word or cache line), scanning the dense sub-chunk directly to find the i''-th occurrence of 1 might be faster than using the more complex theoretical $o(n)$ mechanism or precomputed tables. This again relates to broadword programming techniques, similar to the optimization mentioned for* rank*.*

3.1.3 *Compressing Sparse Bitvectors with Elias-Fano*

The rank and select structures discussed previously (3.1.1, 3.1.2) operate on the plain bitvector $B[1..n]$, achieving a total space occupancy of $n + o(n)$ bits. However, in many practical scenarios, the bitvector B exhibits a skewed distribution, containing significantly fewer 1s than 0s (or vice-versa). Let $m = \text{rank}_1(B, n)$ be the total number of set bits. When $m \ll n$, storing the full n-bit vector is inefficient.

In such sparse settings, we can leverage compression techniques that exploit the low density of set bits. The Elias-Fano representation, previously introduced in Section 2.6.4, provides a highly effective method for this task. Recall that Elias-Fano encodes a monotonically increasing sequence of m integers up to a maximum value n. We can represent the bitvector B by encoding the sequence of indices $\{i \mid B[i] = 1\}$.

As detailed by Vigna [52] in the context of quasi-succinct indices for information retrieval, the Elias-Fano representation achieves a space complexity of approximately $m \log_2(n/m) + O(m)$ bits. This is remarkably close to the information-theoretic lower bound for representing a subset of size m from a universe of size n, often expressed as $n\mathcal{H}_0(B) + O(m)$ bits, where $\mathcal{H}_0(B)$ is the empirical zero-order entropy of the bitvector B. The crucial advantage is that the space depends primarily on m, the number of set bits, rather than the full length n, leading to significant compression when m is small.

This compressed representation directly supports efficient operations. The $\text{select}_1(i)$ operation, finding the position of the i-th set bit, can typically be implemented in constant time on average, often leveraging auxiliary pointers within the Elias-Fano structure as engineered in [52]. However, this space efficiency comes at the cost of potentially slower rank_1 and accessing $B[i]$ operations compared to the $n + o(n)$ structures. These operations usually involve decoding parts of the Elias-Fano structure and may take $O(\log(n/m))$ time or depend on the specific implementation details [36]. Therefore, Elias-Fano presents a compelling space-time trade-off, offering near-optimal compression for sparse bitvectors at the expense of rank_1 and access time complexity. The choice between plain bitvector structures and Elias-Fano depends critically on the sparsity of the data and the required query performance profile.

3.1.4 *Practical Implementation Considerations*

While the asymptotic analysis guarantees $O(1)$ query time and $o(n)$ extra space for the rank and select structures presented earlier, achieving high performance in practice requires careful consideration of architectural factors and constant overheads hidden in the $o(n)$ term. Memory latency, cache efficiency, and instruction-level parallelism often dominate the actual running time on modern processors.

A particularly effective approach for optimizing rank and select implementations leverages *broadword programming* (also known as SWAR - SIMD Within A Register). This technique treats machine registers as small parallel processors, performing operations on multiple data fields packed within a single word using standard arithmetic and logical instructions. Vigna [50] applied these techniques to rank and select queries, leading to highly efficient practical implementations.

The `rank9` structure proposed by Vigna [50] exemplifies this approach. It employs a two-level hierarchy, similar in concept to the structure in Section 3.1.1, but critically relies on broadword algorithms for the final rank computation within a machine word (specifically, sideways addition or population count). Instead of large precomputed lookup tables for small blocks, `rank9` uses carefully designed constants and bitwise operations (detailed in Algorithm 1 of [50]) to compute the rank within a 64-bit word quickly. This typically involves storing relative counts for sub-blocks (e.g., seven 9-bit counts within a 64-bit word) in the second level. The advantages of this approach include speed, resulting from the exploitation of fast register operations and the avoidance of large table lookups, often outperforming other methods in practice. It also offers space efficiency, requiring relatively low overhead (typically around 25% on top of the original bitvector B) mainly for the cumulative counts. Furthermore, broadword algorithms are generally branch-free, benefiting performance on modern pipelined processors by avoiding potential misprediction penalties.

Similarly, Vigna [50] developed broadword algorithms for selection within a word (Algorithm 2 in the paper). The companion `select9` structure integrates these intra-word selection capabilities with a multi-level inventory scheme. The objective of `select9` is to support high-performance selection queries, often achieving near constant-time execution, through hierarchical indexing combined with efficient broadword search for the final location. This capability involves an additional space cost, typically measured at approximately 37.5% relative to the `rank9` structure.

Furthermore, a major bottleneck in rank/select operations is often memory access latency. To mitigate this, *interleaving* the auxiliary data structures is highly recommended. For instance, storing a first-level (superblock) rank count immediately followed by its corresponding second-level (sub-block) counts increases the probability that all necessary auxiliary information for a query resides within the same cache line. This simple layout optimization can dramatically reduce cache misses compared to storing different levels of the hierarchy in separate arrays.

3.2 WAVELET TREES

Building upon the concepts established for bitvectors in Section 3.1, we now extend our focus to sequences over general alphabets using *Wavelet Trees*. Introduced by Grossi, Gupta, and Vitter [19], the wavelet tree is a highly versatile data structure. It functions as a self-index, capable of representing a sequence $S[1..n]$ drawn from an alphabet $\Sigma = \{1, \ldots, \sigma\}$ within space close to its compressed size, while supporting fundamental query operations directly on this representation and allowing reconstruction of the original sequence data. This characteristic makes wavelet trees a cornerstone in applications like compressed full-text indexing, for example within the FM-index [15], where they provide efficient support for the essential rank queries during the backward search phase [37].

While the core idea of hierarchical alphabet partitioning shares conceptual similarities with earlier structures, such as Chazelle's structure for geometric point grids [7] and Kärkkäinen's work on repetition-based indexing [26], the specific formulation and operational capabilities defined by Grossi et al. [19] established the wavelet tree as a broadly applicable tool for sequence processing [37]. Its flexibility derives from its capacity to represent data variously as sequences, permutations, or point grids [37, 21, 14].

We consider a sequence $S[1..n] = s_1 s_2 \ldots s_n$, with $s_i \in \Sigma = \{1, \ldots, \sigma\}$. The primary operations of interest are generalizations of those defined for bitvectors:

- `Access(S, i)`: $S \times [1..n] \to \Sigma$. Returns the symbol $S[i]$.
- $\mathsf{rank}_c(S, i)$: $S \times \Sigma \times [0..n] \to [0..n]$. Returns the number of occurrences of symbol c in the prefix $S[1..i]$, i.e., $|\{j \mid 1 \leqslant j \leqslant i, S[j] = c\}|$. We define $\mathsf{rank}_c(S, 0) = 0$.
- $\mathsf{select}_c(S, j)$: $S \times \Sigma \times [1..n] \to [1..n] \cup \{\perp\}$. Returns the position (index) k such that $S[k] = c$ and $\mathsf{rank}_c(S, k) = j$. If the j-th occurrence does not exist, it returns a special symbol $\perp$.

A naive approach using σ distinct bitvectors B_c (where $B_c[i] = 1$ if and only if $S[i] = c$) reduces these operations to bitvector rank/select ($\mathsf{rank}_c(S, i) = \mathsf{rank}_1(B_c, i)$ and $\mathsf{select}_c(S, j) = \mathsf{select}_1(B_c, j)$). However, augmenting these bitvectors for $O(1)$ time queries using the methods of Section 3.1 requires $n\sigma + o(n\sigma)$ total bits, which is inefficient compared to the $n\lceil\log\sigma\rceil$ bits of the plain representation, especially for large σ [36]. Wavelet trees offer a significantly more space-conscious solution.

3.2.1 *Structure and construction*

The standard wavelet tree for a sequence $S[1..n]$ over $\Sigma = \{1, \ldots, \sigma\}$ is a balanced binary tree structure. Each node v corresponds to an alphabet sub-range $[a_v, b_v] \subseteq \Sigma$ and implicitly represents the subsequence S_v containing all characters s_i from the original sequence S such that $s_i \in [a_v, b_v]$, maintaining their relative order.

The construction proceeds recursively:

- The root node v_{root} corresponds to the full alphabet range $[1, \sigma]$ and the entire sequence $S_{v_{\text{root}}} = S$.
- For an internal node v representing the range $[a_v, b_v]$ with $a_v < b_v$:
    1. Define a midpoint $m_v = \lfloor (a_v + b_v)/2 \rfloor$.
    2. Store a bitmap $B_v[1..|S_v|]$ where $B_v[k] = 0$ if the k-th character of S_v is in $[a_v, m_v]$, and $B_v[k] = 1$ if it is in $[m_v + 1, b_v]$.
    3. Recursively construct the left child v_l for the alphabet range $[a_v, m_v]$ and the subsequence S_{v_l} formed by characters $s \in S_v$ with $s \leqslant m_v$.
    4. Recursively construct the right child v_r for the alphabet range $[m_v + 1, b_v]$ and the subsequence S_{v_r} formed by characters $s \in S_v$ with $s > m_v$.
- A leaf node v represents a single symbol alphabet range $[a_v, a_v]$ (i.e., $a_v = b_v$) and stores no bitmap.

The height of this tree is $h = \lceil \log \sigma \rceil$. The construction process, detailed in Algorithm 4, takes $O(n \log \sigma)$ time, as each symbol from S is processed at each level of the tree.

The space usage of this pointer-based wavelet tree consists of the bitmaps and the pointers. The bitmaps across all nodes at any given level collectively store exactly n bits. With $h = \lceil \log \sigma \rceil$ levels, the total space for bitmaps is $n \lceil \log \sigma \rceil$. The tree has σ leaves and $\sigma - 1$ internal nodes. Storing child and parent pointers (e.g., $3(\sigma - 1)$ pointers, each $\approx \log \sigma$ bits) adds an overhead of $O(\sigma \log \sigma)$ bits (or $O(\sigma \log n)$ if indices into an array are used), which can dominate for large σ. Each bitmap B_v must also be augmented with $o(|S_v|)$ bits to support constant-time rank/select (Section 3.1), contributing an additional $o(n \log \sigma)$ bits overall.

Tracking symbols

The wavelet tree supports the primary sequence operations by translating them into traversals involving bitvector rank/select queries.

Algorithm 4 Building a wavelet tree

```
function BUILD_WT($S, n$)
    $T \leftarrow \mathrm{build}(S, n, 1, \sigma)$
    return $T$
end function
function BUILD($S, n, a, b$)                          ▷ Takes a string $S[1, n]$ over $[a, b]$
    if $a = b$ then
        Free S
        return null
    end if
    $v \leftarrow$ new node
    $m \leftarrow \lfloor (a+b)/2 \rfloor$
    $z \leftarrow 0$                                   ▷ number of elements in S that are $\leqslant m$
    for $i \leftarrow 1$ to $n$ do
        if $S[i] \leqslant m$ then
            $z \leftarrow z + 1$
        end if
    end for
    Allocate strings $S_{left}[1, z]$ and $S_{right}[1, n - z]$
    Allocate bitmap $v.B[1, n]$
    $z \leftarrow 0$
    for $i \leftarrow 1$ to $n$ do
        if $S[i] \leqslant m$ then
            bitclear($v.B, i$)                        ▷ set $i$-th bit of $v.B$ to 0
            $z \leftarrow z + 1$
            $S_{left}[z] \leftarrow S[i]$
        else
            bitset($v.B, i$)                          ▷ set $i$-th bit of $v.B$ to 1
            $S_{right}[i - z] \leftarrow S[i]$
        end if
    end for
    Free S
    $v.\mathrm{left} \leftarrow \mathtt{build}(S_{left}, z, a, m)$
    $v.\mathrm{right} \leftarrow \mathtt{build}(S_{right}, n - z, m + 1, b)$
    Pre-process $v.B$ for rank and select queries
    return $v$
end function
```

3.2.1.1 *Access*

Computing $\mathsf{access}(S, i)$ involves a top-down traversal from the root v_{root}. At each internal node v (representing range $[a_v, b_v]$ with midpoint m_v), we examine the bit $B_v[i_v]$, where i_v is the current index within the node's implicit sequence S_v. If $B_v[i_v] = 0$, we proceed to the left child v_l (range $[a_v, m_v]$) and update the index to $i_{v_l} = \mathsf{rank}_0(B_v, i_v)$. If $B_v[i_v] = 1$, we proceed to the right child v_r (range $[m_v + 1, b_v]$) and update the index to $i_{v_r} = \mathsf{rank}_1(B_v, i_v)$. This repeats until a leaf node representing a single symbol $[a, a]$ is reached;

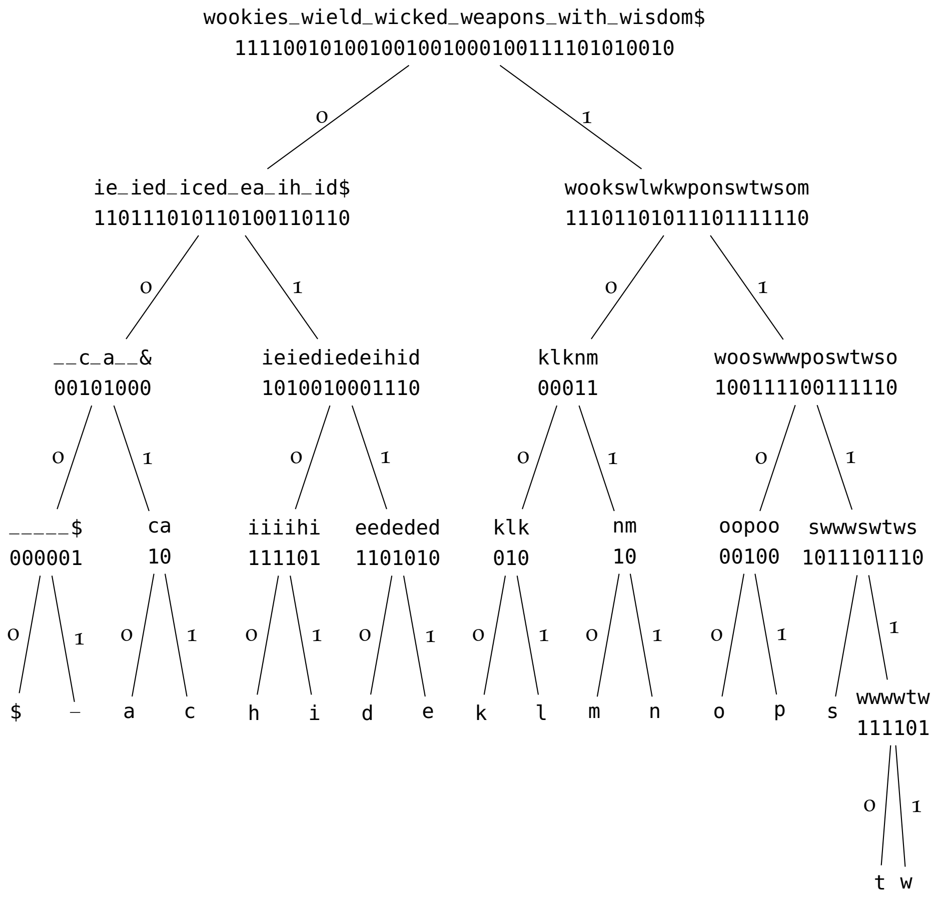


Figure 12: Wavelet tree for the sequence wookies_wield_...

this a is $S[i]$. The process performs $\lceil \log \sigma \rceil$ bitvector rank operations in $O(\log \sigma)$ time. Algorithm 5 details this.

Algorithm 5 access queries on a wavelet tree

```
function ACCESS(T, i)                ▷ T is the sequence S seen as a wavelet tree
    $v \leftarrow T_{root}$                                    ▷ start at the root node
    $[a, b] \leftarrow [1, \sigma]$
    while $a \neq b$ do
        if $access(v.B, i) = 0$ then                     ▷ i-th bit of the bitmap of v
            $i \leftarrow rank_0(v.B, i)$
            $v \leftarrow v.left$                       ▷ move to the left child of node v
            $b \leftarrow \lfloor (a+b)/2 \rfloor$
        else
            $i \leftarrow rank_1(v.B, i)$
            $v \leftarrow v.right$                     ▷ move to the right child of node v
            $a \leftarrow \lfloor (a+b)/2 \rfloor + 1$
        end if
    end while
    return $a$
end function
```

3.2.1.2 *Select*

Computing $\mathsf{select}_c(S, j)$ involves an upward traversal from the leaf node u corresponding to symbol c. Let the index within the leaf level be $j_u = j$. To move to the parent v, we determine if u is the left (v_l) or right (v_r) child. If $u = v_l$, the corresponding index in the parent's bitmap B_v is $j_v = \mathsf{select}_0(B_v, j_u)$. If $u = v_r$, the index is $j_v = \mathsf{select}_1(B_v, j_u)$. We repeat this process, updating j at each level, until the root node is reached. The final index j_{root} is the position of the j-th c in S. This requires $\lceil \log \sigma \rceil$ bitvector select operations in $O(\log \sigma)$ time. Algorithm 6 shows the recursive structure.

Algorithm 6 `Select` queries on a wavelet tree

```
function SELECT_c(S, j)
    return select(T.root, 1, σ, c, j)
end function
function SELECT(v, a, b, c, j)
    if a = b then
        return j
    end if
    if c ⩽ ⌊(a + b)/2⌋ then
        j ← select(v.left, a, ⌊(a + b)/2⌋, c, j) return select_0(v.B, j)
    else
        j ← select(v.right, ⌊(a + b)/2⌋ + 1, b, c, j)
        return select_1(v.B, j)
    end if
end function
```

3.2.1.3 *Rank*

Computing $\mathsf{rank}_c(S, i)$ uses a top-down traversal similar to access. We start at the root v_{root} with index $i_{root} = i$. At each node v (range $[a_v, b_v]$, midpoint m_v), we check if the target symbol c belongs to the left sub-range ($c \leqslant m_v$) or the right sub-range ($c > m_v$). If $c \leqslant m_v$, we descend to the left child v_l and update the index to $i_{v_l} = \mathsf{rank}_0(B_v, i_v)$. If $c > m_v$, we descend to the right child v_r and update the index to $i_{v_r} = \mathsf{rank}_1(B_v, i_v)$. The traversal continues until the leaf node u corresponding to symbol c is reached. The final index i_u at this leaf is the value $\mathsf{rank}_c(S, i)$. This requires $\lceil \log \sigma \rceil$ bitvector rank operations in $O(\log \sigma)$ time. Algorithm 7 formalizes this.

Pointerless Representation: To remove the $O(\sigma \log \sigma)$ pointer overhead, the pointerless wavelet tree concatenates bitmaps level-wise [32, 31]. Let B_l be the concatenated bitmap for level l. If a node v at level l

Algorithm 7 Rank queries on a wavelet tree

```
function RANK_c(S, i)
    $v \leftarrow T_{root}$                              ▷ start at the root node
    $[a, b] \leftarrow [1, \sigma]$
    while $a \neq b$ do
        if $c \leqslant \lfloor (a+b)/2 \rfloor$ then
            $i \leftarrow rank_0(v.B, i)$
            $v \leftarrow v.left$                        ▷ move to the left child of node $v$
            $b \leftarrow \lfloor (a+b)/2 \rfloor$
        else
            $i \leftarrow rank_1(v.B, i)$
            $v \leftarrow v.right$                       ▷ move to the right child of node $v$
            $a \leftarrow \lfloor (a+b)/2 \rfloor + 1$
        end if
    end while
    return $i$
end function
```

corresponds to the range $[s_v, e_v]$ in B_l, its left child's range in B_{l+1} is

$$[(l+1)n + \mathsf{rank}_0(B_l, s_v - 1) + 1, (l+1)n + \mathsf{rank}_0(B_l, e_v)]$$

Its right child's range is

$$[(l+1)n + z_l + \mathsf{rank}_1(B_l, s_v - 1) + 1, (l+1)n + z_l + \mathsf{rank}_1(B_l, e_v)]$$

where $z_l = \mathsf{rank}_0(B_l, n)$ is the total 0-count at level l. This achieves $n\lceil \log \sigma \rceil + o(n \log \sigma)$ total space. Navigation requires extra rank operations per level compared to the pointer-based version (as seen [9]), which can impact practical performance, although the asymptotic query time remains $O(\log \sigma)$.

3.2.2 *Compressed Wavelet Trees*

The standard wavelet tree representations (pointer-based and pointerless) achieve space proportional to $n \log \sigma$. However, when the sequence S itself is compressible (i.e., its empirical entropy is lower than $\log \sigma$), it is desirable for the index structure to reflect this compressibility. Wavelet trees can be adapted to achieve space bounds related to the empirical entropy of S, primarily through two main strategies: directly compressing the bitmaps stored within the nodes, or reshaping the tree based on symbol frequencies.

3.2.2.1 *Compressing the Bitvectors*

This approach maintains the balanced binary structure of the wavelet tree but replaces the plain bitmaps B_v at each internal node v with compressed representations that still support efficient rank and select queries. As explored in Section 3.1.3, various techniques exist for compressing sparse or biased bitvectors. If we employ a representation for each B_v (of length $N_v = |S_v|$) that uses $N_v\mathcal{H}_0(B_v) + o(N_v)$ bits while maintaining $O(1)$ query times for rank and select (e.g., using structures based on Raman et al. [44] or improved variants [41]), the total space complexity aggregates advantageously.

Theorem 3.8 (Entropy-Compressed WT Space (Bitmaps) [19, 36, 16])**.** *A wavelet tree for* $S[1..n]$ *over* $\Sigma = \{1, \dots, \sigma\}$*, using node bitmaps compressed to their zero-order entropy with* $O(1)$*-time rank/select support, can be stored in*

$$n\mathcal{H}_0(S) + o(n \log \sigma) \text{ bits.}$$

The query times for access*,* rank*, and* select *remain* $O(\log \sigma)$*.*

Proof sketch. The core idea relies on the property that the sum of zero-order entropies of the bitmaps at any level l is upper bounded by the sum at level $l - 1$. Summing across all internal nodes, the total space for the compressed bitmaps relates directly to the zero-order entropy of the original sequence S:

$$\sum_{v \text{ internal}} |S_v|\mathcal{H}_0(B_v) = n\mathcal{H}_0(S)$$

The $o(n \log \sigma)$ term arises from aggregating the sublinear $o(N_v)$ overheads from each node's rank/select structure across all levels. Since rank/select on each compressed bitmap is $O(1)$, the total query time remains determined by the tree height, $O(\log \sigma)$. □

While theoretically achieving optimal space relative to $H_0(S)$, practical implementations using compressed bitmaps (like those based on RRR found in libraries such as SDSL [18]) can exhibit higher query latency compared to using plain, uncompressed bitmaps due to the computational overhead of performing rank/select on the compressed format [9].

3.2.2.2 *Huffman-Shaped Wavelet Trees*

Instead of compressing the content (bitmaps), this strategy compresses the structure itself by adapting the tree shape to the symbol frequencies $f_c = \mathsf{rank}_c(S, n)$ in S. The wavelet tree is given the shape of a Huffman tree [23] constructed for the symbols based on their frequencies [20, 30]. In this structure, a symbol c occurring f_c times resides at a leaf at depth $|h(c)|$, where $h(c)$ is its Huffman code. The bitmaps B_v are stored uncompressed at the internal nodes.

The total number of bits stored in the plain bitmaps across the entire tree is precisely the length of the Huffman-encoded sequence:

$$\sum_{c \in \Sigma} f_c |h(c)| \quad \leqslant \quad n(\mathcal{H}_0(S) + 1) \text{ bits.}$$

Adding the overhead for rank/select support on these plain bitmaps ($o(n(\mathcal{H}_0(S) + 1))$ total) and the space to store the Huffman model itself (e.g., $O(\sigma \log \sigma)$ bits, reducible for canonical codes [9]), gives the total space.

Theorem 3.9 (Huffman-Shaped WT Performance [30, 36]). *A Huffman-shaped wavelet tree using plain bitmaps (with $O(1)$ rank/select support) occupies space bounded by*

$$n(\mathcal{H}_0(S) + 1) + o(n(\mathcal{H}_0(S) + 1)) + O(\sigma \log \sigma) \textit{ bits.}$$

It supports access, rank, *and* select. *The worst-case query time is $O(d_{max})$, where d_{max} is the maximum depth (potentially $O(n)$, but typically bounded to $O(\log \sigma)$ [38]). The average query time, assuming queries are distributed according to symbol frequencies (f_c/n), is $O(1 + \mathcal{H}_0(S))$.*

This approach achieves compression related to $\mathcal{H}_0(S)$ through structural adaptation rather than bitmap compression, often leading to faster average query times ($O(1 + \mathcal{H}_0(S))$ vs $O(\log \sigma)$) for statistically skewed query distributions, while using potentially simpler and faster plain bitmap rank/select structures. Pointerless versions based on canonical Huffman codes are also possible [9].

3.2.2.3 *Higher-Order Entropy Compression*

To capture statistical dependencies beyond symbol frequencies, wavelet trees are often used on the Burrows-Wheeler Transformed [6] sequence S^{BWT}. The BWT groups symbols preceded by the same context of length k, making S^{BWT} highly compressible by methods sensitive to local statistics. The k-th order empirical entropy, $H_k(S)$, quantifies this context-dependent compressibility (see Section 2.5). It is known that $\sum_{A \in \Sigma^k} |S_A| \mathcal{H}_0(S_A) = n\mathcal{H}_k(S)$, where S_A is the sequence of symbols following context A [33].

Building a single wavelet tree over the entire S^{BWT} and using zero-order entropy compression on its internal bitmaps (as in Section 3.2.2.1) allows the structure to achieve space related to $H_k(S)$ for the original sequence S.

Theorem 3.10 (H_k-Compressed WT [38, 14]). *Let S^{BWT} be the Burrows-Wheeler Transform of S. A wavelet tree built over S^{BWT} using node bitmaps* *compressed to their zero-order entropy (requiring* $N_v \mathcal{H}_0(B_v) + o(N_v)$ *bits per node* v*) occupies a total space of*

$$n\mathcal{H}_k(S) + o(n \log \sigma) \text{ bits}$$

for any $k \leqslant \alpha \log_\sigma n$ *(*$0 < \alpha < 1$*).*

Proof sketch. The space bound follows because the zero-order entropy of the wavelet tree bitmaps over S^{BWT} sums to $n\mathcal{H}_0(S^{\text{BWT}})$, and $\mathcal{H}_0(S^{\text{BWT}})$ effectively captures $\mathcal{H}_k(S)$ due to the context-grouping property of the BWT [33, 38]. The $o(n \log \sigma)$ term accumulates the overheads of the compressed rank/select structures. □

This structure supports the rank operations on S^{BWT} needed for FM-index pattern matching in $O(\log \sigma)$ time per operation.

This connection is crucial for compressed full-text indexing. Practical variants exist, such as partitioning S^{BWT} and using Huffman-shaped wavelet trees on the blocks [27], which may offer different practical performance trade-offs. The choice between RLE and GE for bitmap encoding in this context was analyzed in [14], favoring RLE-like approaches for achieving H_k.

3.2.3 *Wavelet Matrices and Quad Vectors*

Further refinements address practical performance bottlenecks, primarily memory latency during tree traversal and the overhead associated with large alphabets in pointerless or Huffman representations.

3.2.3.1 *The Wavelet Matrix*

The wavelet matrix [9] provides an alternative layout for the pointerless wavelet tree that simplifies navigation and improves speed. It maintains the level-wise concatenated bitmaps $B_0, B_1, \dots, B_{h-1}$ (where $h = \lceil \log \sigma \rceil$) and stores the counts $z_l = \mathsf{rank}_0(B_l, n)$ for each level l. The key difference lies in the mapping between levels: a position i in B_l corresponding to a 0-bit maps to position $\mathsf{rank}_0(B_l, i)$ in B_{l+1}, while a position corresponding to a 1-bit maps to position $z_l + \mathsf{rank}_1(B_l, i)$ in B_{l+1}.

Theorem 3.11 (Wavelet Matrix Performance [9]). *The wavelet matrix represents* $S[1..n]$ *using*

$$n\lceil \log \sigma \rceil + o(n \log \sigma) \text{ bits}$$

(where $o(n \log \sigma)$ *is for rank/select support on the* h *bitmaps, and the* z_l *values require negligible* $O(\log \sigma \log n)$ *bits). It supports* access*,* rank*, and* select *queries in* $O(\log \sigma)$ *time, executing exactly one bitvector rank or select operation per level traversed.*

This simplified navigation avoids the extra rank operations needed in the strict pointerless WT and eliminates pointer overhead, making it practically faster than pointerless trees and competitive with pointer-based trees, especially for large σ [9]. It can also be combined with bitmap compression or specialized Huffman shaping [9].

3.2.3.2 *4-ary (Quad) Wavelet Trees*

To combat the $O(\log \sigma)$ cache misses inherent in traversing a binary tree, 4-ary wavelet trees reduce the tree height to $h' = \lceil (\log \sigma)/2 \rceil$ [34]. Each internal node partitions its alphabet range $[a_v, b_v]$ into four sub-ranges based on the next two bits in the symbols' binary representation. Instead of a bitmap, each node stores a *quad vector* Q_v (a sequence over $\{0, 1, 2, 3\}$ or $\{00, 01, 10, 11\}$) indicating to which of the four children each symbol belongs.

Supporting queries requires specialized rank/select operations on these quad vectors. Structures achieving $O(1)$ time for quad vector rank/select with $o(N_v \log 4) = o(N_v)$ bits of overhead per node (where N_v is the quad vector length) have been developed [34].

Theorem 3.12 (4-ary WT Performance [34])**.** *A 4-ary wavelet tree using* $O(1)$*-time rank/select structures on quad vectors represents* $S[1..n]$ *in*

$$n\lceil \log \sigma \rceil + o(n \log \sigma) \text{ bits}$$

and supports access*,* rank*, and* select *in* $O(\log \sigma)$ *time, specifically requiring* $O(\lceil (\log \sigma)/2 \rceil)$ *quad vector rank or select operations.*

Proof sketch. Each level effectively stores 2 bits per original symbol. With $\approx (\log \sigma)/2$ levels, the total space for the quad vectors is $n \times 2 \times (\log \sigma)/2 = n \log \sigma$ bits. The $o(n \log \sigma)$ term accounts for the aggregated overhead of the quad vector rank/select structures. The query time is determined by the reduced tree height. □

The principal advantage lies in the potential halving of cache misses during queries, leading to significant practical speedups in latency-bound scenarios, as demonstrated experimentally in [34], particularly when implemented as a 4-ary wavelet matrix.

3.3 RANK AND SELECT ON DEGENERATE STRINGS

The concepts of rank and select queries, fundamental tools in string processing and succinct data structures as explored earlier in this chapter (Section 3.1, Section 3.2), can be extended to the domain of *degenerate strings*. This representation is often used to model uncertainty or variability in sequences, particularly in biological contexts.

Recall that a standard *string* S of length n over a finite non-empty alphabet Σ is a sequence $S = s_1 s_2 \dots s_n$ where each $s_i \in \Sigma$. A degenerate string generalizes this:

Definition 3.13 (Degenerate String [cf. 17]). *A* degenerate string *is a sequence $X = X_1 X_2 \dots X_n$, where each X_i is a subset of the alphabet Σ (i.e., $X_i \subseteq \Sigma$). The value n is termed the* length *of X. The* size *of X, denoted by N, is defined as $N = \sum_{i=1}^{n} |X_i|$. We denote the number of empty sets ($\emptyset$) among $X_1, \dots, X_n$ by n_0.*

Degenerate strings were introduced by Fischer and Paterson [17] and are frequently employed in bioinformatics, for example, to represent DNA sequences with ambiguities (using IUPAC codes) or to model sequence variations within a population [2, 3].

Alanko et al. [2] introduced the counterparts of rank and select for degenerate strings:

Definition 3.14 (Subset Rank and Select [2]). *Given a degenerate string $X = X_1 \dots X_n$ over alphabet Σ, a character $c \in \Sigma$, an index $i \in \{1, \dots, n\}$, and a rank $j \in \mathbb{N}^+$, we define:*

- *subset-rank$_X(i, c)$: Returns the number of sets among the first i sets $X_1, \dots, X_i$ that contain the symbol c. Formally, $|\{k \mid 1 \leqslant k \leqslant i \text{ and } c \in X_k\}|$.*
- *subset-select$_X(j, c)$: Returns the index k such that X_k is the j-th set in the sequence X (from left to right) that contains the symbol c. If fewer than j such sets exist, the result is undefined or signals an error.*

Example 3.15. *Let $X = \{T\}\{G\}\{A, C, G, T\}\{\}\{\}\{C, G\}\{\}\{A\}\{\}\{A\}\{A, C\}\{\}\{\}\{A\}\{A\}$ be a degenerate string of length $n = 15$ over $\Sigma = \{A, C, G, T\}$. Then:*

- *subset-rank$_X(8, A) = 2$, as the sets containing 'A' up to index 8 are X_3 and X_8.*
- *subset-select$_X(2, G) = 3$, as the sets containing 'G' are X_2 (index 2, 1st), X_3 (index 3, 2nd), and X_6 (index 6, 3rd). The index of the 2nd such set is 3.*

The motivation for studying these queries in [2] arose from pangenomics applications, particularly for fast membership queries on de Bruijn graphs represented via the Spectral Burrows-Wheeler Transform (SBWT) [1]. In the SBWT framework, a k-mer query translates to $2k$ subset-rank queries on a specific degenerate string.

A naive solution involves storing a bitvector B_c of length n for each $c \in \Sigma$, marking the presence of c in X_k at $B_c[k]$. Standard $O(1)$-time rank and select on these bitvectors suffice, but the total space is $O(\sigma n)$ bits, which is impractical for large σ or n.

3.3.1 *The Subset Wavelet Tree Approach*

To address the space issue, Alanko et al. [2] introduced the *Subset Wavelet Tree (SWT)*, generalizing the standard Wavelet Tree (Section 3.2) to degenerate strings.

STRUCTURE The SWT is a balanced binary tree over Σ. Each node v represents an alphabet partition A_v (root is Σ), with children representing halves of A_v. A sequence Q_v at node v contains subsets X_k from the original string that intersect with A_v (plus empty sets at the root). Each node v stores two bitvectors, L_v and R_v, of length $|Q_v|$, preprocessed for rank/select:

- $L_v[k] = 1 \iff$ the k-th set in Q_v intersects the *first* half of A_v.
- $R_v[k] = 1 \iff$ the k-th set in Q_v intersects the *second* half of A_v.

Often, L_v and R_v are combined into a base-4/base-3 sequence requiring specialized rank support.

QUERIES $\mathsf{subset\text{-}rank}_X(i, c)$ is answered by traversing from the root to the leaf for c. At node v, if c is in the left partition, update $i \leftarrow \mathsf{rank}_{L_v}(i, 1)$ and go left; otherwise, update $i \leftarrow \mathsf{rank}_{R_v}(i, 1)$ and go right. The final i is the result (Algorithm 8).

$\mathsf{subset\text{-}select}_X(j, c)$ is answered by traversing from the leaf for c up to the root. Moving from child v to parent u, if v is the left child, update $j \leftarrow \mathsf{select}_{L_u}(j, 1)$; otherwise, update $j \leftarrow \mathsf{select}_{R_u}(j, 1)$. The final j is the result (Algorithm 9).

COMPLEXITY With constant-time rank/select on node bitvectors, SWT queries take $O(\log \sigma)$ time. The general space complexity is $2n(\sigma - 1) + o(n\sigma)$ bits. For *balanced* degenerate strings ($n = N$), this improves.

Algorithm 8 Subset-Rank Query using SWT [cf. 2]

```
Require: Character $c$ from $[1, \sigma]$, index $i$
Ensure: The number of subsets $X_k$ such that $k \leqslant i$ and $c \in X_k$
 1: function SUBSET-RANK($c, i$)
 2:     $v \leftarrow$ root
 3:     $[l, r] \leftarrow [1, \sigma]$                    ▷ Range of alphabet indices
 4:     while $l \neq r$ do
 5:         $\mathrm{mid} \leftarrow \lfloor (l + r - 1)/2 \rfloor$
 6:         if $c \leqslant \mathrm{mid}$ then
 7:             $r \leftarrow \mathrm{mid}$
 8:             $i \leftarrow \mathsf{rank}_{L_v}(i, 1)$   ▷ Assumes Rank operation on node's bitvector
 9:             $v \leftarrow$ left child of $v$
10:         else
11:             $l \leftarrow \mathrm{mid} + 1$
12:             $i \leftarrow \mathsf{rank}_{R_v}(i, 1)$   ▷ Assumes Rank operation on node's bitvector
13:             $v \leftarrow$ right child of $v$
14:         end if
15:     end while
16:     return $i$
17: end function
```

Algorithm 9 Subset-Select Query using SWT [cf. 2]

```
Require: Character $c$ from $[1, \sigma]$, rank $j$
Ensure: The index $k$ such that $X_k$ is the $j$-th set containing $c$
 1: function SUBSET-SELECT($c, j$)
 2:     $v \leftarrow$ leaf node corresponding to $c$
 3:     while $v \neq$ root do
 4:         $u \leftarrow$ parent of $v$
 5:         if $v =$ left child of $u$ then
 6:             $j \leftarrow \mathsf{select}_{L_u}(j, 1)$        ▷ Assumes Select operation on parent's bitvector
 7:         else
 8:             $j \leftarrow \mathsf{select}_{R_u}(j, 1)$        ▷ Assumes Select operation on parent's bitvector
 9:         end if
10:         $v \leftarrow u$
11:     end while
12:     return $j$
13: end function
```

Theorem 3.16 (SWT Space Complexity for Balanced Strings [2]). *The subset wavelet tree of a balanced degenerate string takes* $2n \log \sigma + o(n \log \sigma)$ *bits of space and supports* subset-rank *and* subset-select *queries in* $O(\log \sigma)$ *time.*

The $o(n \log \sigma)$ term covers the overhead for rank/select structures on the node bitvectors. Practical performance hinges on efficient rank (especially rank-pair queries [2]) on the internal base-3/4 sequences.

3.3.2 *Improved Reductions and Bounds*

While the SWT offered a valuable first step, Bille et al. [5] revisited the subset rank-select problem, achieving significant theoretical and practical advances by focusing on reductions to standard rank-select operations on regular strings.

They made three significant contributions in this context. First, they introduced the parameter N and revisited the problem through reductions to the regular rank-select problem, deriving flexible complexity bounds based on existing rank-select structures, as detailed in Theorem 3.17. Second, they established a worst-case lower bound of $N \log \sigma - o(N \log \sigma)$ bits for structures supporting `subset-rank` or `subset-select`, and demonstrated that, by leveraging standard rank-select structures, their bounds often approach this lower limit while maintaining optimal query times (Theorem 3.18). Lastly, they implemented and compared their reductions to prior implementations, achieving twice the query speed of the most compact structure from [2] while maintaining comparable space usage. Additionally, they designed a vectorized structure that offers a 4-7x speedup over compact alternatives, rivaling the fastest known solutions.

Theorem 3.17 (General Upper Bound). *Let* X *be a degenerate string of length* n*, size* N*, and containing* n_0 *empty sets over an alphabet* $[1, \sigma]$*. Let* $\mathcal{D}$ *denote a* $\mathcal{D}_b(\ell, \sigma)$*-bit data structure for a length-*ℓ *string over* $[1, \sigma]$*, supporting:*

- rank *queries in* $\mathcal{D}_r(\ell, \sigma)$ *time, and*
- select *queries in* $\mathcal{D}_s(\ell, \sigma)$ *time.*

The subset rank-select problem on X *can be solved under the following conditions:*

(i) ***Case*** $n_0 = 0$***:*** *The structure requires:*

$$\mathcal{D}_b(N, \sigma) + N + o(N) \text{ bits,}$$

and supports:

$$\textit{subset-rank in } \mathcal{D}_r(N, \sigma) + O(1) \textit{ time,}$$

$$\textit{subset-select in } \mathcal{D}_s(N, \sigma) + O(1) \textit{ time.}$$

(ii) ***Case*** $n_0 > 0$***:*** *The bounds from case (i) apply with the following substitutions:*

$$N' = N + n_0 \quad \textit{and} \quad \sigma' = \sigma + 1.$$

(iii) ***Alternative Bound:*** *The structure uses additional* $\mathcal{B}_b(n, n_0)$ *bits of space and supports:*

$$\textit{subset-rank in } \mathcal{D}_r(N, \sigma) + \mathcal{B}_r(n, n_0) \textit{ time,}$$

$$\textit{subset-select in } \mathcal{D}_s(N, \sigma) + \mathcal{B}_s(n, n_0) \textit{ time.}$$

Here, $\mathcal{B}$ *refers to a data structure for a length-*n *bitstring containing* n_0 *1s, which:*

- *uses* $\mathcal{B}_b(n, n_0)$ *bits,*
- *supports rank*$(\cdot, 1)$ *in* $\mathcal{B}_r(n, n_0)$ *time, and*
- *supports select*$(\cdot, 0)$ *in* $\mathcal{B}_s(n, n_0)$ *time.*

In theorem 3.17, (i) and (ii) extend prior reductions from [1], while (iii) introduces an alternative strategy to handle empty sets using an auxiliary bitvector. By applying succinct rank-select structures to these bounds, they achieved improvements in query times without increasing space usage. For instance, substituting their structure into Theorem 3.17 (i) results in a data structure occupying $N \log \sigma + N + o(N \log \sigma + N)$ bits, supporting subset-rank in $O(\log\log \sigma)$ time and subset-select in constant time. This improves the space constant from 2 to $1 + 1/\log \sigma$ compared to Alanko et al. [2], while exponentially reducing query times.

For $n_0 > 0$, Theorem 3.17 (ii) modifies the bounds to $(N + n_0)\log(\sigma + 1) + (N + n_0) + o(n_0 \log \sigma + N \log \sigma + N + n_0)$ bits, maintaining the same improved query times. When $n_0 = o(N)$ and $\sigma = \omega(1)$, the space matches the $n_0 = 0$ case. Alternatively, Theorem 3.17(iii) allows for tailored bitvector structures sensitive to n_0.

Theorem 3.18 (Space Lower Bound)**.** *Let* X *be a degenerate string of size* N *over an alphabet* $[1, \sigma]$*. Any data structure supporting subset-rank or subset-select on* X *must use at least* $N \log \sigma - o(N \log \sigma)$ *bits in the worst case.*

In Theorem 3.18 we aim to establish a lower bound on the space required to represent X while supporting subset-rank or subset-select. Since these operations allow us to reconstruct X fully, any valid data structure must encode X completely. Our approach is to determine the number L of distinct degenerate strings possible for given parameters N and σ, and to show that distinguishing between these instances necessitates at least $\log_2 L$ bits.

Proof. Let N be sufficiently large, and let $\sigma = \omega(\log N)$. Without loss of generality, assume $\log N$ and $N/\log N$ are integers. Consider the class of degenerate strings $X_1, \ldots, X_n$ where $|X_i| = \log N$ for each i and $n = N/\log N$. The number of such strings is given by

$$\binom{\sigma}{\log N}^{N/\log N} \tag{1}$$

This is because each X_i can be formed by choosing $\log N$ characters from σ symbols, and there are n such subsets. The number of bits required to represent any degenerate string X must be at least:

$$\begin{aligned}\log \binom{\sigma}{\log N}^{N/\log N} &= \frac{N}{\log N} \log \binom{\sigma}{\log N} \\ &\geqslant \frac{N}{\log N} \log \left(\frac{\sigma - \log N}{\log N}\right)^{\log N} \\ &= N \log \left(\frac{\sigma - \log N}{\log N}\right) \\ &= N \log \sigma - o(N \log \sigma).\end{aligned}$$

Thus, any representation of X that supports subset-rank or subset-select must use at least $N \log \sigma - o(N \log \sigma)$ bits in the worst case, concluding the proof. □

3.3.2.1 *Reductions*

Let $X, \mathcal{D}, \mathcal{B}$ be as in Theorem 3.17 and consider $\mathcal{V}$ a data structure (for example the one described by Jacobson in [25]), which uses $n + o(n)$ bits for a bitstring of length n and supports rank in constant time and select in $O(1)$ time.

The reductions in Theorem 3.17 rely on the construction of two auxiliary strings S and R derived from the sets X_i. When $n_0 = 0$, each S_i is the concatenation of elements in X_i, and R_i is a single 1 followed by $|X_i| - 1$ 0s. The global strings S and R are formed by concatenating these, appending a 1 after R_n. The data structure consists of $\mathcal{D}$ built over S and Jacobson's structure $\mathcal{V}$ over R, using $\mathcal{D}(N, \sigma) + N + o(N)$ bits. Figure 13 from [5] illustrates this reduction for $n_0 = 0$.

$$X = \underset{X_1}{\begin{Bmatrix}\mathtt{A}\\ \mathtt{C}\\ \mathtt{G}\end{Bmatrix}} \underset{X_2}{\begin{Bmatrix}\mathtt{A}\\ \mathtt{T}\end{Bmatrix}} \underset{X_3}{\begin{Bmatrix}\mathtt{C}\end{Bmatrix}} \underset{X_4}{\begin{Bmatrix}\mathtt{T}\\ \mathtt{G}\end{Bmatrix}} \qquad \begin{array}{rllll} S = & \mathtt{ACG} & \mathtt{AT} & \mathtt{C} & \mathtt{TG} \\ R = & \mathtt{100} & \mathtt{10} & \mathtt{1} & \mathtt{10}\ \mathtt{1} \\ & S_1 & S_2 & S_3 & S_4 \end{array}$$

Figure 13: *Left:* A degenerate string X over the alphabet $\{\mathtt{A},\mathtt{C},\mathtt{G},\mathtt{T}\}$ where $n = 4$ and $N = 8$. *Right:* The reduction from Theorem 3.17 (i) on X. White space is for illustration purposes only.

Queries are supported as follows: subset-rank computes the start position of S_{i+1} using select_R, then evaluates the rank in S. Conversely, subset-select determines the ith occurrence of c in S and identifies the corresponding set via rank_R. Let's consider the pratical example in Figure 13: to compute $\mathsf{subset\text{-}rank}(2, \mathtt{A})$, we first compute $\mathsf{select}_R(3, 1) = 6$. Now we know that S_2 ends at position 5, so we return $\mathsf{rank}_S(5, \mathtt{A}) = 2$. To compute $\mathsf{subset\text{-}select}(2, \mathtt{G})$ we compute $\mathsf{select}_S(2, \mathtt{G}) = 8$, and compute $\mathsf{rank}_R(8, 1) = 4$ to determine that position 8 corresponds to X_4.

Since rank and select on R are constant time, these operations achieve $\mathcal{D}_r(N, \sigma) + O(1)$ and $\mathcal{D}_s(N, \sigma) + O(1)$ time, as required by Theorem 3.17 (i).

For $n_0 \neq 0$, empty sets are replaced by singletons containing a new character $\sigma + 1$, effectively reducing the problem to the $n_0 = 0$ case with $N' = N + n_0$ and $\sigma' = \sigma + 1$. This achieves the bounds of Theorem 3.17 (ii).

ALTERNATIVE BOUND Let E be a bitvector of length n, where $E[i] = 1$ if $X_i = \emptyset$ and $E[i] = 0$ otherwise. Define X'' as the simplified string derived from X by removing all empty sets. The data structure consists in a reduction (i) applied to X'', along with a bitvector structure $\mathcal{B}$ built on E. This requires $\mathcal{D}_b(N, \sigma) + N + o(N) + \mathcal{B}_b(n, n_0)$ bits of space.

To support $\mathsf{subset\text{-}rank}_X(i, c)$, calculate $k = i - \mathsf{rank}_E(i, 1)$, which maps X_i to its corresponding set X''_k. Then, return $\mathsf{subset\text{-}rank}_{X''}(k, c)$. This operation runs in $\mathcal{B}_r(n, n_0) + \mathcal{D}_r(N, \sigma) + O(1)$ time.

To support $\mathsf{subset\text{-}select}_X(i, c)$, first determine $k = \mathsf{subset\text{-}select}_{X''}(i, c)$, and then return $\mathsf{select}_E(k, \mathtt{0})$, which identifies the position of the k-th zero in E (i.e., the k-th non-empty set in X). This operation runs in $\mathcal{B}_s(n, n_0) + \mathcal{D}_s(N, \sigma) + O(1)$, achieving the stated performance bounds.

Empirical Results

The authors conducted a comprehensive evaluation of various data structures for subset rank queries on genomic datasets. Their work emphasizes both space efficiency and query performance, benchmarking methods from [2] alongside their proposed designs. The experiments utilized two primary datasets: a pangenome of 3,682 *E. coli* genomes and a human metagenome containing 17 million sequence reads. Testing was conducted in two modes: integrating the subset rank-select structures into a k-mer query index and isolating these structures to evaluate their performance on 20 million subset-rank queries, which were randomly generated for controlled comparison. Each result reflects an average over five iterations to ensure robustness.

The study introduces the *dense-sparse decomposition* (DSD) as a novel method extending the principles of subset wavelet trees. This decomposition refines the classic split representation by categorizing sets into empty, singleton, and larger subsets, with optimized handling for each category. The authors incorporated advanced rank-select techniques into this framework, including SIMD-based optimizations. Compared to subset wavelet trees and their modern implementations, the DSD structures consistently demonstrated significant improvements. For example, the SIMD-enhanced DSD achieved query times that were 4 to 7 times faster than Concat (ef), a competitive baseline, while maintaining similar space efficiency. Furthermore, the DSD (rrr) variant provided comparable space usage to the compact Concat (ef) structure but offered double the query speed.

The experiments revealed nuanced trade-offs between space and time across all tested structures. While subset wavelet trees, such as Split (ef) and Split (rrr), remain strong contenders, the authors' DSD approach often outperformed them in both dimensions. The DSD (scan) structure, for example, provided a competitive balance, achieving space usage close to entropy bounds while delivering faster query times than comparable subset wavelet tree configurations. The SIMD-enhanced DSD design was particularly noteworthy, achieving near-optimal space efficiency with remarkable query performance.

4 SUCCINCT WEIGHTED DAGS FOR PATH QUERIES

The preceding chapters have established a foundation in data compression (Chapter 2) and succinct data structures, particularly focusing on Rank and Select operations over sequences (Chapter 3). These sequence-based tools provide efficient ways to handle queries on linear data.

Building upon this foundation, we now shift our focus to graph structures, specifically directed acyclic graphs (DAGs) where nodes carry weights. A key motivation for this shift comes from revisiting the *degenerate string problem* introduced in Section 3.3. This problem can be viewed through a different lens, that of graph representation. As we detail below, a degenerate string and a target character can be naturally modeled as a specific type of weighted DAG.

Degenerate Strings as DAGs

Given a degenerate string $X = X_1X_2 \dots X_n$ over an alphabet Σ (as defined in Section 3.3), we construct a weighted DAG $G_c = (V_c, E_c, w_c)$ for a specified character $c \in \Sigma$. This construction provides a mapping from the sequence structure to a graph structure.

First, we define the set of vertices V_c. Let s be a unique source vertex. For each index k ($1 \leqslant k \leqslant n$) and each character $a \in X_k$, we introduce a unique vertex, denoted abstractly as $v_{k,a}$. The vertex set V_c is the union of the source and all such vertices:

$$V_c = \{s\} \cup \{v_{k,a} \mid 1 \leqslant k \leqslant n, a \in X_k\}.$$

These vertices $v_{k,a}$ represent the choice of character a at position k of the degenerate string.

The weight function $w_c : V_c \to \mathbb{N}_0$ is defined as follows: the weight of the source vertex s is $w_c(s) = 0$. For any other vertex $v_{k,a} \in V_c \setminus \{s\}$, its weight depends on whether the character a matches the target character c:

$$w_c(v_{k,a}) = \begin{cases} 1 & \text{if } a = c \\ 0 & \text{if } a \neq c \end{cases}.$$

This function assigns a positive weight only to vertices corresponding to the specific character c we are focusing on.

The edge set E_c connects the source to the vertices representing the first set X_1, and subsequently connects vertices between adjacent positions k and $k+1$:

$$E_c = \{(s, v_{1,a}) \mid a \in X_1\} \cup \{(v_{k,a}, v_{k+1,b}) \mid 1 \leqslant k < n, a \in X_k, b \in X_{k+1}\}.$$

Since edges only connect vertices associated with index k to vertices associated with index $k+1$, the graph (V_c, E_c) contains no directed cycles and is therefore a DAG.

Figure 14 shows an example degenerate string. The weighted DAG G_A derived from this degenerate string for character $c = A$, following the construction detailed above, is illustrated in Figure 15. In the figure, the notation (k, a) inside a node identifies the vertex $v_{k,a}$.

$$X = \underset{X_1}{\left\{\begin{matrix}\mathtt{A}\\ \mathtt{C}\\ \mathtt{G}\end{matrix}\right\}} \underset{X_2}{\left\{\begin{matrix}\mathtt{A}\\ \mathtt{T}\end{matrix}\right\}} \underset{X_3}{\left\{\begin{matrix}\mathtt{T}\\ \mathtt{C}\\ \mathtt{A}\end{matrix}\right\}} \underset{X_4}{\left\{\begin{matrix}\mathtt{A}\\ \mathtt{G}\end{matrix}\right\}}$$

Figure 14: An example degenerate string $X = X_1X_2X_3X_4$ over $\Sigma = \{A, C, G, T\}$.

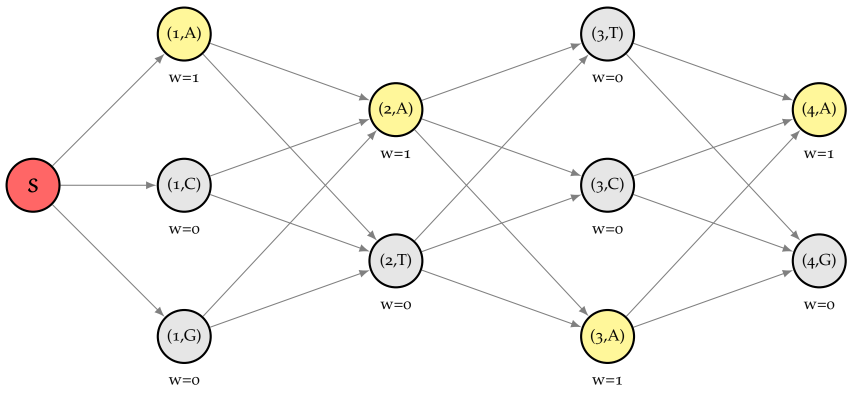


Figure 15: The weighted DAG G_A derived from the degenerate string in Figure 14 for character $c = A$. Nodes visually labeled (k, a) represent the abstract vertices $v_{k,a}$. Nodes with $w_A(v_{k,a}) = 1$ are yellow; those with $w_A(v_{k,a}) = 0$ are gray. Edges represent the connections defined in E_A.

This graph-based perspective on degenerate strings serves as a concrete starting point for the core topic of this chapter: the development of succinct data structures for general node-weighted DAGs to support path-based queries. We address the challenge of representing an arbitrary DAG $G = (V, E, w)$, where each vertex $v \in V$ carries a non-negative integer weight $w(v)$, in a compressed format that efficiently supports queries related to cumulative path weights. Such weighted DAGs model various phenomena beyond degenerate strings. For example, in bioinformatics, pangenome graphs[1] can be interpreted through this lens: if each node corresponds to a DNA sequence (a string over $\{A, C, G, T\}$), the weight $w(v)$ could represent the count of a specific nucleotide (e.g., *A*) within that sequence; similarly for *C*, *G*, and *T*.

Our primary focus is on generalizing the rank query to this graph setting; the select query, while definable, will not be treated further in this work. For a given vertex N, the rank query aims to describe the set of possible cumulative weights achievable on paths originating from a designated source vertex s and terminating at N.

The combinatorial complexity of paths in a DAG — potentially exponential in the number of vertices — makes naive approaches based on explicit path enumeration or storage infeasible for large graphs. This necessitates the development of a *succinct* data structure. Our approach involves partitioning the vertices based on how their path weight information is represented: some vertices (*explicit*) will store this information directly, while others (*implicit*) will rely on indirect derivation through references facilitated by a carefully defined *successor* relationship, as detailed in Section 4.2.

4.1 MATHEMATICAL FRAMEWORK

To develop our data structure and associated algorithms, we first establish the necessary mathematical definitions and properties concerning weighted DAGs and path weights.

Weighted Directed Acyclic Graphs

We begin with the formal definition of the combinatorial structure central to this chapter.

Definition 4.1 (Weighted DAG). *A node-weighted Directed Acyclic Graph (weighted DAG) is a triple* $G = (V, E, w)$*, where:*

- V *is a finite set of vertices. We typically identify* V *with the set* $\{0, 1, \ldots, n-1\}$ *where* $n = |V|$*, thereby implicitly defining a total order on the vertices.*
- $E \subseteq V \times V$ *is a set of directed edges such that the graph* (V, E) *contains no directed cycles.*

1 Pangenome graphs may contain cycles. These cycles can be addressed by either removing them or by utilizing path information provided by modern pangenome graph formats.

- *$w : V \to \mathbb{N}_0$ is a weight function assigning a non-negative integer $w(v)$ to each vertex $v \in V$, where $\mathbb{N}_0 = \{0, 1, 2, \dots\}$.*

For a vertex $v \in V$, we denote the set of its direct predecessors as

$$\mathtt{Pred}(v) = \{u \in V \mid (u, v) \in E\}$$

and the set of its direct successors as

$$\mathtt{Succ}(v) = \{u \in V \mid (v, u) \in E\}.$$

A vertex v with $\mathtt{Succ}(v) = \emptyset$ is termed a sink vertex.

This definition provides the fundamental object of study. We will often rely on the acyclic property, which guarantees the existence of topological orderings of the vertices.

Assumption 4.2 (Unique Source). *Without loss of generality, we assume that the DAG G possesses a unique source vertex $s \in V$ characterized by $\mathtt{Pred}(s) = \emptyset$. If multiple sources exist in the original graph, a standard preprocessing step involves introducing a virtual source vertex s' with $w(s') = 0$ and adding edges (s', v) for all original source vertices v. Throughout this work, we assume such a transformation has been applied if necessary, and we identify the unique source with vertex $s = 0$, setting $w(s) = 0$.*

Having defined the graph structure, we now define paths and their associated weights, which are central to the queries we aim to support.

Definition 4.3 (Path in a DAG). *A path P from a vertex u to a vertex v in G is a sequence of vertices $P = (v_0, v_1, \dots, v_k)$ such that $v_0 = u$, $v_k = v$, and $(v_{j-1}, v_j) \in E$ for all $1 \leqslant j \leqslant k$. The length of the path P is k, the number of edges. Let $\mathtt{Path}(s, v)$ denote the set of all paths originating from the unique source s and terminating at v.*

Definition 4.4 (Cumulative Path Weight). *The cumulative weight, denoted $W(P)$, for a path $P = (v_0 = s, \dots, v_k = v)$ as defined in 4.3, is the sum of the weights of the vertices along the path:*

$$W(P) = \sum_{j=1}^{k} w(v_j).$$

4.1.1 *Path Weight Aggregation*

A key challenge lies in efficiently representing the potentially vast collection of path weights terminating at each vertex. We introduce a set associated with each vertex to capture precisely this information.

Definition 4.5 ($\mathcal{O}$-Set)**.** *For each vertex $v \in V$ in a weighted DAG $G = (V, E, w)$ with source s, the $\mathcal{O}$-set, denoted $\mathcal{O}_v \subseteq \mathbb{N}_0$, is defined recursively. Let us assume a topological ordering of the vertices in V. The sets are constructed as follows:*

- *For the source vertex $v = s$:*

$$\mathcal{O}_s = \{0\}.$$

- *For any other vertex $v \neq s$:*

$$\mathcal{O}_v = \bigcup_{u \in \mathrm{Pred}(v)} \{y + w(v) \mid y \in \mathcal{O}_u\}.$$

This definition implies that $\mathcal{O}_v$ contains only the distinct values generated by this union process. We consider $\mathcal{O}_v$ to be the set of these unique values, represented as a sorted sequence.

The following proposition establishes the semantic meaning of the $\mathcal{O}$-set, confirming that it correctly captures the set of all possible path weights (as defined in 4.4) ending at a vertex.

Proposition 4.6 (Semantic Interpretation of $\mathcal{O}$-Set)**.** *For any vertex $v \in V$, the set $\mathcal{O}_v$ is precisely the set of cumulative path weights from the source s to v:*

$$\mathcal{O}_v = \{W(P) \mid P \in \mathrm{Path}(s, v)\}.$$

Proof. The proof proceeds by induction on the vertices $v \in V$, ordered according to a topological sort of G. *Base Case:* For the source vertex $v = s$, the only path in $\mathrm{Path}(s, s)$ is the trivial path $P = (s)$. According to Definition 4.4, $W(P) = 0$. By Definition 4.5, $\mathcal{O}_s = \{0\}$. Thus, the proposition holds for s. *Inductive Step:* Assume the proposition holds for all vertices u that strictly precede v in the topological order. In particular, this assumption holds for all $u \in \mathrm{Pred}(v)$, since the existence of an edge (u, v) implies u precedes v in any topological sort. We must show that $\mathcal{O}_v = \{W(P) \mid P \in \mathrm{Path}(s, v)\}$.

($\subseteq$) Let $x \in \mathcal{O}_v$. By Definition 4.5, since $v \neq s$, there must exist a predecessor $u \in \mathrm{Pred}(v)$ and a value $y \in \mathcal{O}_u$ such that $x = y + w(v)$. By the inductive hypothesis applied to u, $y = W'(P')$ for some path $P' = (v_0 = s, \ldots, v_{k-1} = u) \in \mathrm{Path}(s, u)$. Consider the path $P =$

$(v_0, \dots, v_{k-1}, v_k = v)$ formed by appending the edge (u, v) to P'. This is a valid path in $\mathsf{Path}(s, v)$. Its weight according to Definition 4.4 is

$$\begin{aligned} W(P) &= \sum_{j=1}^{k} w(v_j) = \left(\sum_{j=1}^{k-1} w(v_j) \right) + w(v_k) \\ &= W(P') + w(v) = y + w(v) = x. \end{aligned}$$

Therefore, any element $x \in \mathcal{O}_v$ corresponds to the weight of some path in $\mathsf{Path}(s, v)$.

($\supseteq$) Let $P = (v_0 = s, \dots, v_k = v)$ be an arbitrary path in $\mathsf{Path}(s, v)$. Since $v \neq s$, the path must have length $k \geqslant 1$. Let $u = v_{k-1}$ be the vertex immediately preceding v on this path; thus, $u \in \mathsf{Pred}(v)$. Let $P' = (v_0, \dots, v_{k-1})$ be the subpath of P ending at u. P' is a path in $\mathsf{Path}(s, u)$. By the inductive hypothesis, the weight $W(P') = \sum_{j=1}^{k-1} w(v_j)$ must be an element of $\mathcal{O}_u$. Let $y = W(P')$. By Definition 4.5, the value $y + w(v)$ is included in the construction of $\mathcal{O}_v$. We observe that

$$\begin{aligned} y + w(v) &= W(P') + w(v_k) = \left(\sum_{j=1}^{k-1} w(v_j) \right) + w(v_k) \\ &= \sum_{j=1}^{k} w(v_j) = W(P). \end{aligned}$$

Thus, the weight $W(P)$ of any path in $\mathsf{Path}(s, v)$ is contained in $\mathcal{O}_v$.

Since both inclusions hold, we conclude that $\mathcal{O}_v = \{W(P) \mid P \in \mathsf{Path}(s, v)\}$. $\square$

An important property related to the sizes of these $\mathcal{O}$-sets is monotonicity along edges, which plays a fundamental role in the design of our succinct structure.

Lemma 4.7 ($\mathcal{O}$-Set Cardinality Monotonicity along Edges). *Let $v \in V$ and let $u \in \mathsf{Succ}(v)$ be any direct successor of v. Then, the cardinality of the $\mathcal{O}$-set of v is less than or equal to the cardinality of the $\mathcal{O}$-set of u, i.e., $|\mathcal{O}_v| \leqslant |\mathcal{O}_u|$.*

Proof. From Definition 4.5, the $\mathcal{O}$-set for u is given by

$$\mathcal{O}_u = \bigcup_{p \in \mathsf{Pred}(u)} \{y + w(u) \mid y \in \mathcal{O}_p\}.$$

Since u is a successor of v, it follows that v is a predecessor of u, i.e., $v \in \mathsf{Pred}(u)$. Therefore, the set $S_v = \{y + w(u) \mid y \in \mathcal{O}_v\}$ contributes to the union forming $\mathcal{O}_u$. Specifically

$$S_v \subseteq \bigcup_{p \in \mathsf{Pred}(u)} \{y + w(u) \mid y \in \mathcal{O}_p\}$$

Consider the mapping $f : \mathcal{O}_v \to S_v$ defined by $f(y) = y + w(u)$. Since $w(u) \geqslant 0$, this mapping is injective. If $y_1 \neq y_2$, then $y_1 + w(u) \neq y_2 + w(u)$. Consequently, the number of distinct elements in S_v is exactly equal to the number of distinct elements in $\mathcal{O}_v$, so $|S_v| = |\mathcal{O}_v|$. The set $\mathcal{O}_u$ is formed by taking the union of sets like S_v for all predecessors $p \in \texttt{Pred}(u)$ and retaining only the unique values. Since the elements generated from v's contribution (namely, S_v) are a subset of the elements considered for $\mathcal{O}_u$, the total number of unique elements in $\mathcal{O}_u$ must be at least the number of unique elements contributed by v. Therefore, $|\mathcal{O}_u| \geqslant |S_v| = |\mathcal{O}_v|$. □

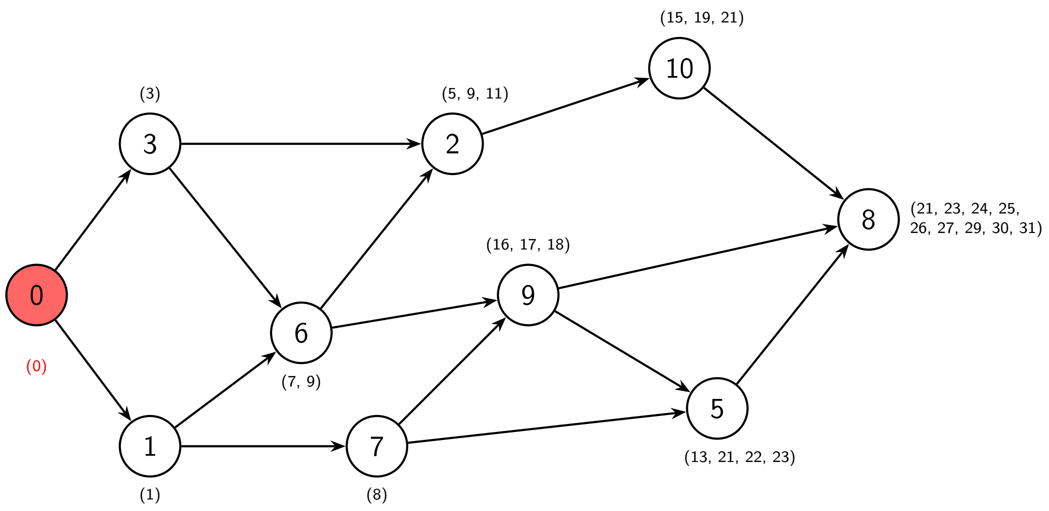


Figure 16: Example of a node-weighted DAG. Each node v contains its weight $w(v)$. The label associated with each node represents its calculated $\mathcal{O}$-set, $\mathcal{O}_v$, considered as a sorted sequence.

Example 4.8 ($\mathcal{O}$-Set Calculation)**.** *Consider the weighted DAG shown in Figure 16. Each node v is labelled with its weight $w(v)$ inside the circle. The label associated with each node displays its corresponding $\mathcal{O}$-set, calculated according to 4.5.*

4.1.2 *The Rank Query*

Having defined the $\mathcal{O}$-set, which precisely captures the set of all possible cumulative path weights terminating at a given vertex N, we now introduce the rank query. This query builds upon the $\mathcal{O}$-set to provide a richer description related to the path weights.

Intuitively, each value $x \in \mathcal{O}_N$ represents the total accumulated weight along some path from the source s ending exactly at N. We can think of the weight $w(N)$ of the node N itself as the contribution or cost associated with the final step or "activity" performed at N. The rank query, $\text{rank}_G(N)$, aims to capture not just the final cumulative weights $x \in \mathcal{O}_N$, but rather the set of all possible cumulative values that could be considered "active" or relevant *during* the activity represented by node N.

Specifically, for a path P reaching N with total weight $W(P) = x$, the query considers the range of cumulative values from the point just before incorporating N's full weight up to the final value x. This corresponds mathematically to the integer interval

$$[\max(0, x - w(N) + 1), x]$$

This interval represents all possible integer cumulative weights observed during the *processing* of node N along that specific path. The rank query then aggregates these intervals over all possible paths ending at N.

This intuition leads to the following formal definition:

Definition 4.9 (Rank Query on Weighted DAG). *Given a vertex $N \in V$ in a weighted DAG $G = (V, E, w)$, the Rank query, denoted $\text{rank}_G(N)$, returns a representation of a specific set of integers derived from the $\mathcal{O}$-set $\mathcal{O}_N$. The target set, $S_N \subseteq \mathbb{N}_0$, is defined as the union of intervals generated from each element $x \in \mathcal{O}_N$:*

$$S_N = \bigcup_{x \in \mathcal{O}_N} \{z \in \mathbb{N}_0 \mid \max(0, x - w(N) + 1) \leqslant z \leqslant x\}.$$

These intervals are then maximally merged, meaning $r_k < l_{k+1} - 1$ for all $k = 1, \ldots, p-1$. The query result is specified as a minimal collection of disjoint, closed integer intervals,

$$\mathcal{R}_N = \{[l_1, r_1], [l_2, r_2], \ldots, [l_p, r_p]\}$$

such that their union exactly covers S_N,

$$\bigcup_{k=1}^{p} [l_k, r_k] = S_N$$

We now illustrate the rankquery with an example.

Example 4.10 (Rank Query Calculation with Disjoint Result). *Let us compute the rank query for vertex $N = 2$ in the DAG shown*[2] *in Figure 16. From Example 4.8, we know that the weight of node 2 is $w(2) = 2$, and its $\mathcal{O}$-set is $\mathcal{O}_2 = \{5, 9, 11\}$. These are the possible total weights of paths ending at node 2. Applying Definition 4.9, we associate an interval with each $x \in \mathcal{O}_2$, representing the values active during the processing of node 2:*

2 In this example, the weight function w assigns to each vertex v a weight equal to its identifier, i.e., $w(v) = v$. This is possible since it's just a small graph and we have taken unique weights

- *For path weight* $x = 5$*: Interval is* $[\max(0, 5-2+1), 5] = [4, 5]$*. These are the values active while accumulating the weight* $w(2) = 2$ *to reach 5.*
- *For path weight* $x = 9$*: Interval is* $[\max(0, 9-2+1), 9] = [8, 9]$*.*
- *For path weight* $x = 11$*: Interval is* $[\max(0, 11-2+1), 11] = [10, 11]$*.*

The target set S_2 *is the union of these intervals:* $S_2 = [4, 5] \cup [8, 9] \cup [10, 11]$*. We merge these intervals to obtain the minimal disjoint representation* $\mathcal{R}_2$*.*

- *Compare* $[4, 5]$ *and* $[8, 9]$*. Since* $8 > 5 + 1$*, they remain separate.*
- *Compare* $[8, 9]$ *and* $[10, 11]$*. Since* $10 \leqslant 9 + 1$*, they are merged into* $[8, \max(9, 11)] = [8, 11]$*.*

The final minimal collection of disjoint intervals is:

$$\mathrm{Rank}_G(2) = \{[4, 5], [8, 11]\}.$$

This collection represents the set $S_2 = \{4, 5\} \cup \{8, 9, 10, 11\}$*, which encompasses all possible integer cumulative weights that could be considered active during the processing phase associated with node 2, across all possible paths leading to it.*

4.2 THE SUCCINCT DAG REPRESENTATION

As established, the $\mathcal{O}$-sets can grow significantly in size, rendering their explicit storage for all vertices prohibitive for large graphs. This section details our proposed succinct representation strategy, designed to mitigate this space complexity while still enabling efficient query evaluation. The core idea is to partition the vertices and utilize indirect references guided by a successor relationship.

Successor Selection Heuristic

For an implicit representation of path information, we define a function σ that designates a specific successor for each non-sink vertex. This choice is guided by a simple heuristic aimed at minimizing the overall space required for storing the succinct representation.

Definition 4.11 (Successor Function σ)**.** *For each vertex* $v \in V$ *that is not a sink (i.e.,* $\mathrm{Succ}(v) \neq \emptyset$*), we select a designated successor* $\sigma(v) \in \mathrm{Succ}(v)$ *according to the following heuristic rule:*

$$\sigma(v) \in \operatorname*{argmin}_{u \in \mathrm{Succ}(v)} \{|\mathcal{O}_u|\}.$$

In case of ties (multiple successors minimize the $\mathcal{O}$-set cardinality) we select the successor with the smallest vertex ID among the candidates. The function σ is thus well-defined for all non-sink vertices.

Node Partitioning

The successor function σ induces a natural partitioning of the graph's vertices, which dictates how path information is stored or derived for each vertex.

Definition 4.12 (Explicit and Implicit Vertices)**.** *The set of vertices* V *is partitioned into two disjoint sets:*

- V_E*: The set of explicit vertices. This set comprises all sink vertices of the graph* G*:*

$$V_E = \{v \in V \mid \mathrm{Succ}(v) = \emptyset\}.$$

- V_I*: The set of implicit vertices. This set includes all non-sink vertices:*

$$V_I = V \setminus V_E = \{v \in V \mid \mathrm{Succ}(v) \neq \emptyset\}.$$

Vertices in V_E serve as base cases in our representation; their associated path weight information ($\mathcal{O}$-sets) is stored directly. For vertices in V_I, this information is represented implicitly, relying on references to their designated successor $\sigma(v)$ as defined above.

4.2.1 *Structure Components*

We now outline the main components of our data structure designed to represent the weighted DAG $G = (V, E, w)$ succinctly. These components are typically implemented as arrays indexed by vertex ID (from 0 to $n-1$), enabling a Structure of Arrays (SoA) memory layout [3].

1. *Weights $\mathcal{W}$:* An array storing the weight $w(v)$ for each vertex $v \in V$, $\mathcal{W}[v] = w(v)$.
2. *Successor Information Σ:* An array encoding the successor function σ and identifying explicit nodes. For an implicit vertex $v \in V_I$, $\Sigma[v]$ stores the identifier (ID) of its designated successor $\sigma(v)$. For an explicit vertex $v \in V_E$, $\Sigma[v]$ contains a special marker indicating its status.

3 The SoA organization can be advantageous for cache performance and allows for independent compression strategies for different data types (weights, pointers, path data).

3. *Associated Data $\mathcal{D}$:* An array or structure holding the core path weight information, structured differently depending on whether a vertex is explicit or implicit. For a vertex v, $\mathcal{D}[v]$ stores either its full $\mathcal{O}$-set (if $v \in V_E$) or an offset sequence $\mathcal{I}_v$ (if $v \in V_I$) that enables reconstruction of $\mathcal{O}_v$ via reference to the data associated with $\sigma(v)$ (and eventually all its successors up to a node in V_E).

The content stored within the Associated Data component $\mathcal{D}$ is determined by the classification of the vertex according to the partitioning defined in 4.12. Specifically:

For $v \in V_E$ (*Explicit Vertex*), $\mathcal{D}[v]$ stores the sorted sequence $\mathcal{O}_v$. We denote this stored data representation as $\mathcal{D}_E(v)$. In practice, $\mathcal{D}_E(v)$ would be implemented using a suitable (potentially compressed) representation of the sequence $\mathcal{O}_v$.

For $v \in V_I$ (*Implicit Vertex*), let $u = \sigma(v)$ be the designated successor. $\mathcal{D}[v]$ stores the *offset sequence* $\mathcal{I}_v$. This sequence $\mathcal{I}_v = (j_0, j_1, \dots, j_{m-1})$ is a strictly monotonically increasing sequence of $m = |\mathcal{O}_v|$ indices. The index j_k (stored at position k in $\mathcal{I}_v$) indicates that the k-th element of $\mathcal{O}_v$ (let it be x_k) can be computed from the j_k-th element of $\mathcal{O}_u$ (let it be y_{j_k}) using the reconstruction rule: $x_k = y_{j_k} - w(u)$. We denote this stored sequence representation as $\mathcal{D}_I(v)$.

Proposition 4.13. *For any implicit node $v \in V_I$, let $u = \sigma(v)$ be its designated successor chosen according to Definition 4.11. Then $|\mathcal{O}_v| \leqslant |\mathcal{O}_u|$.*

Proof. This is a direct consequence of Lemma 4.7. Since $\sigma(v)$ is chosen from the set $\mathrm{Succ}(v)$ of direct successors of v, the lemma applies, yielding $|\mathcal{O}_v| \leqslant |\mathcal{O}_{\sigma(v)}|$. □

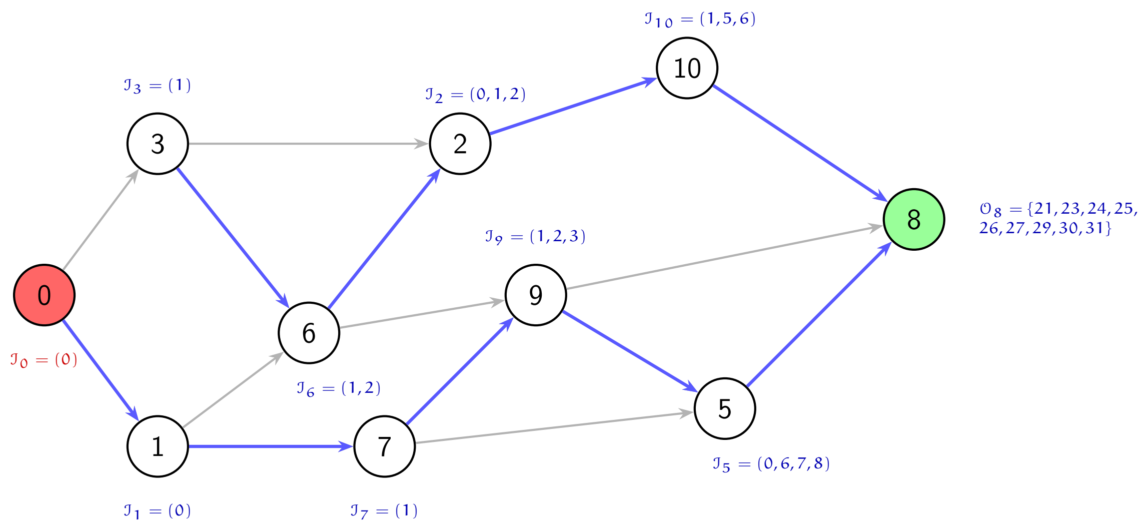
This inequality guarantees that the length of the offset sequence $\mathcal{I}_v$ (which is $m = |\mathcal{O}_v|$) is no greater than the length of the sequence $\mathcal{O}_u$ (which is $|\mathcal{O}_u|$) from which values are derived. Furthermore, the reconstruction rule $x_k = y_{j_k} - w(u)$ implies that each $x_k \in \mathcal{O}_v$ originates from some element in $\mathcal{O}_u$ (shifted by $-w(u)$). The offset sequence $\mathcal{I}_v$ stores the index j_k in $\mathcal{O}_u$ corresponding to the k-th element x_k in $\mathcal{O}_v$.

The heuristic choice of $\sigma(v)$ (4.11) aims to minimize $|\mathcal{O}_{\sigma(v)}|$. While this is a greedy, local optimization, the intuition is that choosing a successor with a smaller $\mathcal{O}$-set might lead to offset sequences $\mathcal{I}_v$ that are structurally simpler or involve smaller index values, potentially improving the compressibility of the $\mathcal{D}_I(v)$ component. In Chapter 5, we will discuss alternative strategies for selecting successors.

Remark 4.14 (Unique Sink Transformation). *A DAG with multiple sinks can be transformed into one with a unique sink by adding a virtual sink* t' *(*$w(t') = 0$*) and connecting all original sinks to it. This transformation preserves the* $\mathcal{O}$*-sets for all original vertices. The main advantage for our succinct structure is space efficiency: instead of storing potentially numerous and large* $\mathcal{O}$*-sets for all original sinks (explicit nodes), we only need to store the* $\mathcal{O}$*-set for the single virtual sink* t'*. This significantly reduces the cost associated with explicit nodes, which are generally more space-intensive than implicit nodes, especially for sinks deep within the DAG. We can thus assume, without loss of generality, that our DAG has a unique sink.*

Example of the Structure

To illustrate the succinct representation, let us apply the principles from Section 4.2.1 to the example DAG introduced in Figure 16. The resulting structure is visualized in Figure 17.



$\mathcal{I}_{10} = (1,5,6)$
$\mathcal{I}_3 = (1)$
$\mathcal{I}_2 = (0,1,2)$
$\mathcal{O}_8 = \{21,23,24,25,26,27,29,30,31\}$
$\mathcal{I}_9 = (1,2,3)$
$\mathcal{I}_0 = (0)$
$\mathcal{I}_6 = (1,2)$
$\mathcal{I}_5 = (0,6,7,8)$
$\mathcal{I}_1 = (0)$
$\mathcal{I}_7 = (1)$
0
1
2
3
5
6
7
8
9
10

Figure 17: Illustration of the succinct DAG representation for the graph in Figure 16. Implicit nodes are shown with standard borders, while the explicit sink node (8) is shaded green. The source node (0) is red. Highlighted blue edges indicate the chosen successor $\sigma(v)$ for each implicit node v, selected according to 4.11. Labels show the stored associated data $\mathcal{D}[v]$: the full $\mathcal{O}$-set for the explicit node 8, and the offset sequence $\mathcal{I}_v$ for all implicit nodes.

The construction proceeds in three main steps:

1. *Partitioning:* Vertices are partitioned into explicit sinks $V_E = \{v \mid \mathrm{Succ}(v) = \emptyset\}$ and implicit non-sinks $V_I = V \setminus V_E$. In the example (Figure 17), $V_E = \{8\}$.
2. *Successor Selection:* For each $v \in V_I$, a successor $\sigma(v)$ is chosen from $\mathrm{Succ}(v)$ to minimize $|\mathcal{O}_{\sigma(v)}|$. The selected successors are

shown as blue edges in Figure 17. For example, $\sigma(0) = 1$, $\sigma(1) = 7$, $\sigma(3) = 6$, etc.

3. *Associated Data Determination:* The data $\mathcal{D}[v]$ depends on the vertex type: For $v \in V_E$, the full $\mathcal{O}$-set is stored: $\mathcal{D}[v] = \mathcal{D}_E(v) = \mathcal{O}_v$. E.g., $\mathcal{D}[8] = \mathcal{O}_8$ in the figure.

   For $v \in V_I$, the offset sequence $\mathcal{J}_v$ is stored: $\mathcal{D}[v] = \mathcal{D}_I(v) = \mathcal{J}_v$. Let $u = \sigma(v)$. The sequence $\mathcal{J}_v = (j_0, \ldots, j_{m-1})$ where $m = |\mathcal{O}_v|$, provides the indices j_k such that the k-th element x_k of $\mathcal{O}_v$ is reconstructed from the j_k-th element y_{j_k} of $\mathcal{O}_u$ via the rule $x_k = y_{j_k} - w(u)$.

   For example, consider $v = 3$. We have $\sigma(3) = 6$, $w(6) = 6$, $\mathcal{O}_3 = \{3\}$ (so $x_0 = 3$), and $\mathcal{O}_6 = \{7, 9\}$ (so $y_0 = 7, y_1 = 9$). We need j_0 such that $x_0 = y_{j_0} - w(6)$, i.e., $3 = y_{j_0} - 6$. This gives $y_{j_0} = 9$, which corresponds to index $j_0 = 1$ in $\mathcal{O}_6$. Thus, $\mathcal{J}_3 = (1)$.

   The offset sequences for all implicit nodes are computed similarly and displayed as labels in Figure 17.

This structure stores $\mathcal{O}$-sets only for sinks, relying on successor paths and offset sequences for implicit nodes, enabling query evaluation via traversal as shown in Figure 18.

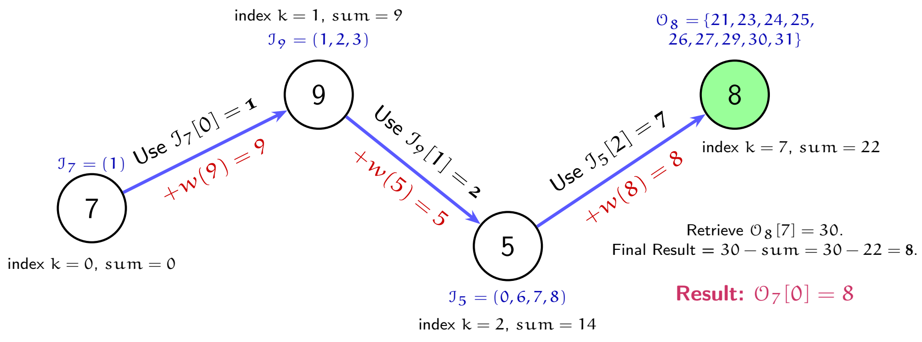

index $k = 1$, $sum = 9$
$\mathcal{J}_9 = (1, 2, 3)$
$\mathcal{O}_8 = \{21, 23, 24, 25, 26, 27, 29, 30, 31\}$
9
8
7
5
$\mathcal{J}_7 = (1)$
Use $\mathcal{J}_7[0] = 1$
$+w(9) = 9$
Use $\mathcal{J}_9[1] = 2$
$+w(5) = 5$
Use $\mathcal{J}_5[2] = 7$
$+w(8) = 8$
index $k = 7$, $sum = 22$
index $k = 0$, $sum = 0$
Retrieve $\mathcal{O}_8[7] = 30$.
Final Result $= 30 - sum = 30 - 22 = 8$.
$\mathcal{J}_5 = (0, 6, 7, 8)$
index $k = 2$, $sum = 14$
Result: $\mathcal{O}_7[0] = 8$

Figure 18: Visualization of the query process for retrieving the element at index $k = 0$ of $\mathcal{O}_7$ using the succinct representation. The query follows the successor path $7 \to 9 \to 5 \to 8$. At each step from an implicit node v to its successor $u = \sigma(v)$, the current index $k_{current}$ is updated using the stored offset $k_{next} = \mathcal{J}_v[k_{current}]$, and the weight $w(u)$ is added to an accumulator. The path starts with $k = 0$ at node 7. It proceeds to node 9 using index $\mathcal{J}_7[0] = 1$, accumulating $w(9) = 9$. Then to node 5 using index $\mathcal{J}_9[1] = 2$, accumulating $w(5) = 5$ (total sum 14). Then to node 8 using index $\mathcal{J}_5[2] = 7$, accumulating $w(8) = 8$ (total sum 22). Node 8 is explicit, so the value at the final index 7, $\mathcal{O}_8[7] = 30$, is retrieved. The result is obtained by subtracting the total accumulated weight: $30 - 22 = 8$. This correctly reconstructs $\mathcal{O}_7[0]$.

This representation avoids storing the full $\mathcal{O}$-sets for implicit nodes. Instead, it stores references (via σ) and transformations (via $\mathcal{J}_v$ and weights $\mathcal{W}$). Queries for implicit nodes require traversing the successor path until an explicit node is reached, as illustrated next.

The example in Figure 18 illustrates how a specific value from an implicit node's $\mathcal{O}$-set can be reconstructed. The query for $\mathcal{O}_7[0]$ triggers a traversal along the σ-defined path $7 \to 9 \to 5 \to 8$. During this traversal, the offset sequences $(\mathcal{I}_7, \mathcal{I}_9, \mathcal{I}_5)$ are used to update the target index within the respective successor's $\mathcal{O}$-set representation, and the weights of the successors $(w(9), w(5), w(8))$ are accumulated. Once the explicit node (8) is reached, the value at the final computed index (7) is retrieved from the stored $\mathcal{O}_8$. Subtracting the accumulated weight sum (22) from this base value (30) yields the desired result (8).

4.3 QUERY ALGORITHMS

This section details the algorithms operating on the succinct DAG representation introduced in Section 4.2. We first present the algorithm for reconstructing elements of the $\mathcal{O}$-set for any given vertex, which forms the basis for computing the rank query.

4.3.1 *Reconstructing $\mathcal{O}$-Sets*

The core operation is to determine the value of the k-th element (0-indexed) of $\mathcal{O}_v$ for an arbitrary vertex v. If v is explicit ($v \in V_E$), this involves directly accessing the stored representation $\mathcal{D}_E(v)$. If v is implicit ($v \in V_I$), the value must be reconstructed by traversing the successor path defined by σ until an explicit vertex is reached, accumulating weights and applying the offset indices stored in the $\mathcal{I}$ sequences along the path.

We define the function $\textsc{GetValue}(v, k)$ to perform this task. It relies on accessing the structure components: weights $\mathcal{W}$, successor information Σ (to get $\sigma(v)$ and check for explicit nodes), and associated data $\mathcal{D}$ (to retrieve $\mathcal{I}$ sequences or $\mathcal{O}$-sets).

The correctness of Algorithm 10 follows inductively from the definition of the offset sequence $\mathcal{I}_v$. Let $x_k^{(v)}$ denote the k-th element of $\mathcal{O}_v$. If v is implicit with successor $u = \sigma(v)$, then by construction $x_k^{(v)} = x_{j_k}^{(u)} - w(u)$, where $j_k = \mathcal{I}_v[k]$. The algorithm iteratively applies this relation: if $v \to u \to \cdots \to e$ is the successor path ending at an explicit node e, and the indices transform as

$$k \to j_k^{(v)} \to j_{j_k^{(v)}}^{(u)} \to \cdots \to K$$

Algorithm 10 GETVALUE(v, k): Compute the k-th element of $\mathcal{O}_v$

```
Require: Vertex ID $v$, index $k \in \{0, \dots, |\mathcal{O}_v| - 1\}$.
Ensure: The value of the k-th smallest element in $\mathcal{O}_v$.
 1: $\mathtt{current\_node} \leftarrow v$
 2: $\mathtt{current\_index} \leftarrow k$
 3: $\mathit{weight\_sum} \leftarrow 0$
 4: while $\mathtt{current\_node} \notin V_E$ do
 5:     $\mathtt{successor} \leftarrow \Sigma[\mathtt{current\_node}]$
 6:     $\mathit{weight\_sum} \leftarrow \mathit{weight\_sum} + \mathcal{W}[\mathtt{successor}]$
 7:     $\mathcal{I}_{\mathtt{current\_node}} \leftarrow \mathcal{D}[\mathtt{current\_node}]$
 8:     $\mathtt{next\_index} \leftarrow \mathcal{I}_{\mathtt{current\_node}}[\mathtt{current\_index}]$
 9:     $\mathtt{current\_node} \leftarrow \mathtt{successor}$
10:     $\mathtt{current\_index} \leftarrow \mathtt{next\_index}$
11: end while
12: $\mathcal{O}_{\mathtt{explicit}} \leftarrow \mathcal{D}[\mathtt{current\_node})$
13: $\mathtt{base\_value} \leftarrow \mathcal{O}_{\mathtt{explicit}}[\mathtt{current\_index}]$
14: return $\mathtt{base\_value} - \mathit{weight\_sum}$
```

then the algorithm computes

$$x_K^{(e)} - \sum_{z \in \text{path } v \to e, z \neq v} w(z)$$

which correctly yields $x_k^{(v)}$. The visualization in Figure 18 provides a concrete example of this process.

To reconstruct the entire $\mathcal{O}$-set for a vertex v, we can repeatedly call GETVALUE(v, k) for $k = 0, 1, \dots, |\mathcal{O}_v| - 1$. This requires knowing the size $|\mathcal{O}_v|$. We assume this size information is either stored explicitly for each node or can be efficiently derived (e.g., potentially stored alongside $\mathcal{I}_v$ or $\mathcal{O}_v$ in the $\mathcal{D}$ component). Let LENGTH(v) be an operation that returns $|\mathcal{O}_v|$. This procedure is implemented in Algorithm 11.

Algorithm 11 GETOSET(v): Reconstruct the $\mathcal{O}$-set for vertex v

```
Require: Vertex ID $v$.
Ensure: The sorted sequence $\mathcal{O}_v$.
 1: $\mathtt{size} \leftarrow$ LENGTH$(v)$
 2: Initialize an empty list O_list of size size.
 3: for $k$ from $0$ to $\mathtt{size} - 1$ do
 4:     $\mathit{value} \leftarrow$ GETVALUE$(v, k)$
 5:     $\mathtt{O\_list}[k] \leftarrow \mathit{value}$
 6: end for
 7: return O_list
```

4.3.2 *Computing the Rank Query*

Equipped with the ability to reconstruct $\mathcal{O}_N$ for any query vertex N via Algorithm 11, we can now implement the rank query as specified in 4.9. The procedure involves two main steps: first, generate the initial set of intervals based on the elements of $\mathcal{O}_N$ and the weight $w(N)$; second, merge these potentially overlapping or adjacent intervals into a minimal set of disjoint intervals.

The interval merging step is a standard procedure. Given a list of intervals sorted by their starting points, we can merge them in linear time. Algorithm 12 describes this process.

Algorithm 12 MergeIntervals($\mathrm{Intervals}$): Merge sorted intervals

```
Require: A list $\mathrm{Intervals} = \{[l_1, r_1], [l_2, r_2], \dots\}$ sorted by start points $l_i$.
Ensure: A minimal list $\mathrm{MergedIntervals}$ of disjoint intervals covering the
    same union.
 1: Initialize an empty list $\mathrm{MergedIntervals}$.
 2: if $\mathrm{Intervals}$ is not empty then
 3:     $\mathrm{current\_interval} \leftarrow \mathrm{Intervals}[0]$.
 4:     for $i$ from $1$ to $\mathrm{length}(\mathrm{Intervals}) - 1$ do
 5:         $\mathrm{next\_interval} \leftarrow \mathrm{Intervals}[i]$.
 6:         if $\mathrm{next\_interval}.l \leqslant \mathrm{current\_interval}.r + 1$ then
 7:             $\mathrm{current\_interval}.r \leftarrow \max(\mathrm{current\_interval}.r, \mathrm{next\_interval}.r)$
 8:         else
 9:             Append $\mathrm{current\_interval}$ to $\mathrm{MergedIntervals}$.
10:             $\mathrm{current\_interval} \leftarrow \mathrm{next\_interval}$.
11:         end if
12:     end for
13:     Append $\mathrm{current\_interval}$ to $\mathrm{MergedIntervals}$.
14: end if
15: return $\mathrm{MergedIntervals}$.
```

Algorithm 13 implements the Rank query defined in 4.9. It first reconstructs the necessary path weight information ($\mathcal{O}_N$), then applies the specified transformation to generate the initial set of intervals

$$S_N = \bigcup_{x \in \mathcal{O}_N} [\max(0, x - w_N + 1), x]$$

Finally, it employs the standard interval merging algorithm to produce the canonical representation $\mathcal{R}_N$ as a minimal set of disjoint intervals. Note that the sorting step (line 9) is necessary because intervals generated from consecutive elements in $\mathcal{O}_N$ are not guaranteed to have non-decreasing start points.

Algorithm 13 $\text{Rank}_G(N)$: Compute the Rank query for vertex N

```
Require: Vertex ID $N$.
Ensure: A minimal set of disjoint intervals $\mathcal{R}_N$ representing $S_N$
1: $\mathcal{O}_N \leftarrow \text{GetOSet}(N)$
2: $w_N \leftarrow \mathcal{W}[N]$
3: Initialize an empty list $\mathit{InitialIntervals}$.
4: for each $x \in \mathcal{O}_N$ do
5:     $l \leftarrow \max(0, x - w_N + 1)$
6:     $r \leftarrow x$
7:     Append the interval $[l, r]$ to $\mathit{InitialIntervals}$.
8: end for
9: Sort $\mathit{InitialIntervals}$ based on the starting point $l$.
10: $\mathit{MergedIntervals} \leftarrow \text{MergeIntervals}(\mathit{InitialIntervals})$
11: return $\mathit{MergedIntervals}$.
```

Example: Rank Query Computation

Let's trace the execution of Algorithm 13 to compute $\text{Rank}_G(N)$ for the only node with weight 2, in the representation already shown in Figure 16 and Figure 17.

First, the algorithm requires the $\mathcal{O}$-set for node 2. It calls $\text{GetOSet}(2)$ (Algorithm 11), which in turn relies on $\text{GetValue}(2, k)$ (Algorithm 10) for $k = 0, 1, \dots, |\mathcal{O}_2| - 1$. GetValue traverses the successor path determined by σ. For node 2, the successor path is $2 \to 10 \to 8$. Following this path, applying the stored offset indices $\mathcal{I}_2, \mathcal{I}_{10}$, accumulating weights $w(10)$ and $w(8)$, and finally retrieving the base value from the explicit node $\mathcal{O}_8$, yields the set $\mathcal{O}_2 = \{5, 9, 11\}$.

Next, the weight of the query node is retrieved: $w_N = w(2) = 2$.

The algorithm then proceeds to generate the initial set of intervals. Each element x from $\mathcal{O}_2$ maps to an interval $[\max(0, x - w_2 + 1), x]$:

$$\begin{aligned} x = 5 &\quad\longrightarrow\quad [\max(0, 5-2+1), 5] = [4,5] \\ x = 9 &\quad\longrightarrow\quad [\max(0, 9-2+1), 9] = [8,9] \\ x = 11 &\quad\longrightarrow\quad [\max(0, 11-2+1), 11] = [10,11] \end{aligned}$$

This yields the initial list $\mathit{Intervals} = \{[4,5], [8,9], [10,11]\}$.

Finally, MergeIntervals (12) is applied to this sorted list. The intervals $[8,9]$ and $[10,11]$ merge into $[8,11]$. The interval $[4,5]$ remains separate. The final result returned by $\text{Rank}_G(2)$ is the minimal set of disjoint intervals $\mathcal{R}_2 = \{[4,5], [8,11]\}$.

4.4 COMPRESSION STRATEGIES

The succinct representation detailed in Section 4.2 comprises three main components: the weights array $\mathcal{W}$, the successor information array Σ, and the associated data array $\mathcal{D}$. While the partitioning strategy itself reduces the need to store large $\mathcal{O}$-sets explicitly for all nodes, the components $\mathcal{W}$, Σ, and especially $\mathcal{D}$ can still consume significant space. This section explores strategies for further compressing these components, leveraging the techniques discussed in Chapter 2 and Chapter 3, as well as the implementation concepts from Appendix A. Our goal is to minimize the space occupancy while maintaining efficient query performance, particularly for the reconstruction step (10) underlying the Rank query (13).

4.4.1 *Compressing Weights and Successor Information*

The components $\mathcal{W}$ and Σ are arrays of integers.

- $\mathcal{W}$: An array of length $n = |V|$, where $\mathcal{W}[v] = w(v) \in \mathbb{N}_0$. The values are non-negative integers representing vertex weights, without any guarantee of monotonicity or specific distribution patterns a priori.
- Σ: An array of length n, where $\Sigma[v]$ stores either the integer ID $\sigma(v) \in \{0, \dots, n-1\}$ if $v \in V_I$, or a special marker (which can also be represented as an integer) if $v \in V_E$. This is also a sequence of integers.

Given that both $\mathcal{W}$ and Σ are sequences of integers, the variable-length integer coding schemes discussed in Section 2.6 are natural candidates for compression. Techniques such as Unary, Elias Gamma/Delta, or Rice codes can be employed. The optimal choice depends heavily on the empirical distribution of the weights $w(v)$ and the successor IDs $\sigma(v)$ (and the chosen marker value) within the specific DAG instance.

A practical approach to implement this compression while preserving the necessary random access capability (i.e., efficiently retrieving $\mathcal{W}[v]$ or $\Sigma[v]$ for any v) is provided by the engineered compressed-intvec structure described in Appendix A. This structure allows selecting an appropriate integer codec (e.g., Gamma, Delta, Rice) based on the data distribution and employs a sampling mechanism to ensure $O(1)$ expected time access to any element v, at the cost of a typically sublinear space overhead for the samples.

Alternatively, if the range of possible integer values in $\mathcal{W}$ or Σ (i.e., the maximum weight or n) is sufficiently small to be considered a small alphabet α, one could choose to use Wavelet Trees (Section 3.2), particularly efficient variants like Wavelet Matrices or Quad Wavelet Trees (Section 3.2.3). These structures provide efficient Access (retrieving the element at a given position) in $O(\log \alpha)$ time, where α is the size of the alphabet (the number of distinct values). However, for general graphs where weights or vertex counts n can be large, the alphabet size may not be small, making direct integer coding via compressed-intvec a more generally applicable starting point.

4.4.2 *Compressing Associated Data Sequences*

The associated data component $\mathcal{D}$ holds the core path weight information, represented as sequences tied to each vertex. For an explicit vertex $v \in V_E$, $\mathcal{D}$ contains the $\mathcal{O}$-set $\mathcal{O}_v = (x_0, x_1, \ldots, x_{m-1})$, where $m = |\mathcal{O}_v|$. This sequence consists of non-negative integers and, by Definition 4.5, is strictly increasing ($0 \leqslant x_0 < x_1 < \cdots < x_{m-1}$). For an implicit vertex $v \in V_I$, $\mathcal{D}$ contains the offset sequence $\mathcal{I}_v = (j_0, j_1, \ldots, j_{m-1})$, where $m = |\mathcal{O}_v|$. As established by the reconstruction rule $x_k = y_{j_k} - w(u)$ and the fact that $\mathcal{O}_v$ and $\mathcal{O}_u$ (where $u = \sigma(v)$) are themselves strictly increasing sequences, the offset sequence $\mathcal{I}_v$ is also strictly increasing ($0 \leqslant j_0 < j_1 < \cdots < j_{m-1}$).

The strictly monotonic nature of both $\mathcal{O}_v$ and $\mathcal{I}_v$ makes them highly suitable for compression methods designed for such sequences. Elias-Fano encoding (Section 2.6.4), already discussed, provides excellent compression guarantees and support for efficient random access and predecessor/successor queries.

Alternatively, especially if sequences exhibit significant clustering (long stretches of consecutive integers), Run-Length Encoding (RLE) can offer substantial compression. We focus here on describing the RLE approach as applied to these strictly monotonic sequences.

Run-Length Encoding (RLE) Representation

Run-Length Encoding exploits consecutive values within a monotonic sequence. Consider a generic strictly increasing sequence $Y = (y_0, y_1, \ldots, y_{m-1})$. A *run* is defined as a maximal contiguous subsequence of the form $(y_i, y_{i+1}, \ldots, y_{i+l-1})$ such that $y_{j+1} = y_j + 1$ for all j where $i \leqslant j < i + l - 1$. Note that a run can have length $l = 1$ if an element does not have a consecutive successor in the sequence. The RLE approach encodes Y by representing each run by its starting value and its length.

Formally, the RLE of Y generates two sequences:

- The *run starts sequence*, $S = (s_1, s_2, \ldots, s_p)$, where s_i is the first value ($y_{i'}$) of the i-th run identified in Y. Due to the maximality of runs and the strict monotonicity of Y, S is also a strictly increasing sequence: $s_1 < s_2 < \cdots < s_p$.

- The *run lengths sequence*, $L = (l_1, l_2, \dots, l_p)$, where $l_i \geqslant 1$ is the number of elements (length) of the i-th run. L is a sequence of positive integers, possessing no inherent monotonicity property.

The pair (S, L) uniquely determines the original sequence Y. Algorithm 14 provides the procedure for generating S and L.

Algorithm 14 ENCODERLE(Y): Generate RLE components for a monotonic sequence

```
Require: Monotonic sequence $Y = (y_0, y_1, \dots, y_{m-1})$, where $m = |Y|$.
Ensure: Run starts sequence $S$, Run lengths sequence $L$.
 1: Initialize $S \leftarrow \emptyset$, $L \leftarrow \emptyset$.
 2: if $m = 0$ then
 3:     return $(S, L)$
 4: end if
 5: $i \leftarrow 0$
 6: while $i < m$ do
 7:     $\mathtt{current\_start} \leftarrow y_i$
 8:     $\mathtt{current\_length} \leftarrow 1$
 9:     while $i + 1 < m$ and $y_{i+1} = y_i + 1$ do
10:         $\mathtt{current\_length} \leftarrow \mathtt{current\_length} + 1$
11:         $i \leftarrow i + 1$
12:     end while
13:     Append current_start to $S$.
14:     Append current_length to $L$.
15:     $i \leftarrow i + 1$
16: end while
17: return $(S, L)$
```

COMPRESSION OF RLE COMPONENTS The space efficiency of RLE hinges on effectively compressing the resulting sequences S and L. The run starts sequence $S = (s_1, \dots, s_p)$ is strictly monotonic. Consequently, it is an ideal candidate for Elias-Fano encoding (Section 2.6.4).

The run lengths sequence $L = (l_1, \dots, l_p)$ is a sequence of positive integers without guaranteed structure. Standard variable-length integer codes (Section 2.6), such as Elias Gamma or Delta codes, can be applied. In practice, employing a structure like the compressed-intvec (Appendix A) allows choosing an appropriate codec and provides efficient random access.

RANDOM ACCESS IN RLE Retrieving the element y_k (the element at index k in the original sequence Y, $0 \leqslant k < m$) from the RLE representation (S, L) requires identifying the run to which y_k belongs. This necessitates finding the unique run index i^* (where $1 \leqslant i^* \leqslant p$) such that:

$$\sum_{j=1}^{i^*-1} l_j \leqslant k < \sum_{j=1}^{i^*} l_j$$

where the sum is defined as 0 if $i^* = 1$. Once i^* is found, the value y_k is given by:

$$y_k = s_{i^*} + \left(k - \sum_{j=1}^{i^*-1} l_j \right)$$

Computing the prefix sums $\sum l_j$ on the fly requires iterating through L, potentially leading to $O(p)$ time complexity for access, which is inefficient if the number of runs p is large.

To accelerate random access, one can precompute and store the sequence of prefix sums of the run lengths:

$$P = (p_1, p_2, \dots, p_p) \qquad p_i = \sum_{j=1}^{i} l_j.$$

Since $l_j \geqslant 1$ for all j, the sequence P is strictly increasing. As a strictly monotonic sequence, P itself can be compressed effectively, for instance, using Elias-Fano encoding.

With the prefix sum sequence P available, finding the index i^* corresponding to the query index k reduces to searching for the smallest i^* such that $p_{i^*} > k$. This is a successor search problem on the monotonic sequence P. Using binary search on P (if stored as an array) takes $O(\log p)$ time. If P is stored in a structure supporting faster searches (like rank/select structures built upon certain Elias-Fano constructions), this lookup time might be further improved. Algorithm 15 formalizes access using prefix sums.

Algorithm 15 GetValueRLE(S, L, P, k): Retrieve element from RLE representation

```
Require: Run starts $S = (s_1, \dots, s_p)$, Run lengths $L = (l_1, \dots, l_p)$, Prefix
    sums $P = (p_1, \dots, p_p)$, index $k \in [0, m-1]$.
Ensure: The value $y_k$ of the element at index $k$.
1: Find smallest $i^* \in \{1, \dots, p\}$ such that $P[i^*] > k$.
2: if $i^* = 1$ then
3:     $previous\_cumulative\_length \leftarrow 0$
4: else
5:     $previous\_cumulative\_length \leftarrow P[i^* - 1]$
6: end if
7: $offset \leftarrow k - previous\_cumulative\_length$
8: $start\_value \leftarrow S[i^*]$
9: return $start\_value + offset$
```

METHOD SELECTION The choice between direct Elias-Fano encoding and Run-Length Encoding (RLE) for representing the strictly increasing sequences $\mathcal{O}_v$ and $\mathcal{I}_v$ depends on their structure. RLE is advantageous when the sequence exhibits significant clustering, meaning the number of runs p is substantially smaller than the total number of elements m ($p \ll m$). In such cases, the combined compressed size of the run starts S and run lengths L (and potentially the prefix sums P) might be less than direct Elias-Fano encoding of the original sequence.

On the other hand, if sequences are sparse or lack significant runs (p is close to m), direct Elias-Fano is likely more straightforward and potentially more space-efficient.

Regardless of the chosen compression method, representing the entire associated data component $\mathcal{D}$ practically involves concatenating the compressed representations of all sequences ($\{\mathcal{D}_E(v)\}_{v \in V_E}$ and $\{\mathcal{D}_I(v)\}_{v \in V_I}$) into a single buffer. An auxiliary index structure is then needed to map each vertex ID v to the starting position and metadata of its compressed sequence. The space overhead for this index is typically negligible compared to the compressed data itself

4.5 BELOW THE ENTROPY LOWER BOUND

To evaluate the space efficiency of our proposed structure, we establish a baseline based on the information content in the weighted DAG itself. Drawing upon the principles of information theory outlined in Chapter 2, we can define a measure of entropy for the graph $G = (V, E, w)$.

Any *lossless* representation of the graph G must, at a minimum, encode the information required to uniquely specify its structure (the edges E) and the associated weights (the function w). We formulate a 0^{th}-order entropy measure, denoted $\mathcal{H}_0(G)$, as a lower bound on the number of bits required to represent these components, assuming no prior knowledge about correlations or higher-order statistical properties.

The information content required to specify the sequence of vertex weights $(w(v))_{v \in V}$ constitutes the first component. A fundamental lower bound, denoted $\mathcal{H}_W(G)$, can be established by considering the minimal binary representation length for each individual weight:

$$\mathcal{H}_W(G) = \sum_{v \in V} \lceil \log_2(w(v) + 1) \rceil \quad \text{bits.}$$

This measure reflects the space needed assuming each weight is encoded independently, using the minimal bits for its value (handling $w(v) = 0$), without leveraging potential statistical correlations or distribution patterns suitable for techniques like variable-length integer coding (Section 2.6).

The second component relates to the graph's topology, specifically the set of edges E. With $n = |V|$ vertices, there exist $n(n-1)$ possible directed edges (excluding self-loops). Encoding the topology requires specifying which $m = |E|$ of these potential edges are present. Assuming all directed graphs with n vertices and m edges are equally probable a priori, the information content is determined by the number of ways to choose these m edges. This leads to the topological information component, $\mathcal{H}_E(G)$:

$$\mathcal{H}_E(G) = \log_2 \binom{n(n-1)}{m} \quad \text{bits.}$$

This quantity can be approximated using Stirling's formula, $\log_2 \binom{N}{k} \approx k \log_2(N/k) + O(k)$, yielding $\mathcal{H}_E(G) \approx m \log_2 \left(\frac{n(n-1)}{m}\right) + O(m)$ bits.

Combining these components gives us a formal definition for the 0^{th}-order entropy of the weighted DAG.

Definition 4.15 (0^{th}-Order Weighted DAG Entropy)**.** *For a weighted DAG $G = (V, E, w)$ with $n = |V|$ vertices and $m = |E|$ edges, its 0^{th}-order entropy $\mathcal{H}_0(G)$ is defined as the sum of the information content required for the weights and the topology:*

$$\mathcal{H}_0(G) = \sum_{v \in V} \lceil \log_2(w(v) + 1) \rceil + \log_2 \binom{n(n-1)}{m} \quad \textit{bits.}$$

The value $\mathcal{H}_0(G)$ represents a theoretical lower bound on the space required by any lossless encoding of the graph (V, E, w) based solely on its zero-order statistics.

Our proposed succinct data structure (Section 4.2), however, is designed differently. While lossless for rank query computation (4.9), it is inherently *lossy* concerning the reconstruction of the original graph G, as it does not store the complete edge set E. It retains only vertex weights $\mathcal{W}$, successor information Σ, and associated data $\mathcal{D}$. This distinction allows the structure's space usage, $S(G)$, to potentially fall below the $\mathcal{H}_0(G)$ bound.

To illustrate this, we analyze performance on a weighted DAG derived from unrolling a Bitcoin Peer-to-Peer temporal network graph (details of the unrolling process are beyond the scope of this thesis), having $n = 22,210$ vertices and $m = 50,514$ edges. For this specific DAG, the calculated 0^{th}-order entropy $\mathcal{H}_0(G)$ amounts to 1,525,730 bits, comprising $\mathcal{H}_W(G) = 60,824$ bits for weights and $\mathcal{H}_E(G) = 1,464,906$ bits for topology according to 4.15.

We compare this theoretical lower bound $\mathcal{H}_0(G)$ against theoretical estimates of the space required by our succinct structure and alternative approaches based on precomputation.

For sequences composed of general non-negative integers x, such as the vertex weights $\mathcal{W}$, the successor identifiers Σ, or the interval endpoints in baseline precomputation results, the space estimation is based on summing the minimal binary representation cost for each integer independently. This cost is calculated as $\lceil \log_2(x+1) \rceil$ bits per integer x.

When representing strictly monotonic sequences, like the run start values in RLE or interval endpoints compressed using Elias-Fano (Section 2.6.4), our space estimation relies on its established theoretical complexity. Theorem 2.36 guarantees an upper bound of

$$n \log_2(u/n) + 2n$$

bits for encoding n integers within the range $[0, u)$ using Elias-Fano.

Finally, the space for the offset sequences $\mathcal{I}_v$ when stored using the Run-Length Encoding (RLE) scheme described in Section 4.4 is estimated by combining the previous principles. It requires the sum of the space estimate for the strictly monotonic sequence of run start values (using the Elias-Fano estimation) and the space estimate for the sequence of corresponding run lengths (using the minimal binary representation cost for each length).

Component / Method	Estimated Bits
Theoretical Lower Bound ($\mathcal{H}_0(G)$)	
0^{th}-Order Entropy Total	1,525,730
Weights Component ($\mathcal{H}_W(G)$)	60,824
Topology Component ($\mathcal{H}_E(G)$)	1,464,906
Precomputed Rank Queries	
Explicit Storage (Minimal Binary)	4,854,533
Elias-Fano Compressed Storage	2,211,849
Succinct DAG Representation (RLE)	
Total Space ($S(G)$)	602,808
Weights $\mathcal{W}$ (Minimal Binary)	60,824
Successors Σ (Minimal Binary)	297,700
Associated Data $\mathcal{D}$ (RLE Offsets)	244,284

Table 2: Theoretical space estimates (in bits) for the example Bitcoin DAG ($n = 22,210$, $m = 50,514$).

Table 2 presents the results of this theoretical space estimation for the example DAG. The total estimated space for our succinct DAG representation using RLE for the offsets, $S(G)$, is 602,808 bits. This value is notably less than half of the 0^{th}-order entropy $\mathcal{H}_0(G)$ (1,525,730 bits). This result highlights the advantage of our approach: by selectively discarding information required for full graph reconstruction (specifically, the complete edge set E) while preserving all information necessary for rank query computation, the structure achieves a space footprint smaller than the theoretical minimum for lossless graph encoding.

Moreover, the analysis reveals the significant space overhead associated with precomputing rank query results. Storing the interval sets for all nodes explicitly, even using minimal binary representations, requires an estimated 4,854,533 bits. Applying Elias-Fano compression to these precomputed results reduces the space to 2,211,849 bits. While this represents a substantial saving over the explicit form, it remains considerably larger than both the 0^{th}-order entropy bound and, crucially, the space achieved by our succinct DAG representation ($S(G)$). Therefore, our structure provides not only a mechanism to answer complex path queries but does so with remarkable space efficiency, far surpassing precomputation strategies and falling below the conventional entropy bounds associated with lossless graph representation due to its targeted, query-specific information retention.

5 CONCLUSION AND FUTURE DIRECTIONS

This thesis began by covering succinct data structures, data compression, and sequence queries. Building on that foundation, Chapter 4 introduced our primary contribution: a space-efficient method for representing node-weighted Directed Acyclic Graphs (DAGs). This representation is specifically designed to support rank (4.9) queries, which aggregate cumulative path weights ending at a particular vertex.

The core contribution presented in Chapter 4 is a succinct representation strategy for weighted DAGs. Motivated initially by the reinterpretation of the degenerate string problem as a graph problem, we generalized the approach to arbitrary weighted DAGs. The key idea involves partitioning the vertices into two sets: explicit vertices (V_E), typically comprising the sink nodes, for which path weight information ($\mathcal{O}$-sets) is stored directly; and implicit vertices (V_I), for which this information is derived indirectly. This derivation relies on a carefully defined successor function, σ, which designates a specific successor for each implicit node, guiding a traversal path. Associated with each implicit node v, an offset sequence $\mathcal{I}_v$ stores the necessary indices to reconstruct its $\mathcal{O}$-set elements from the $\mathcal{O}$-set of its designated successor $\sigma(v)$. Query algorithms, notably GetValue and GetOSet, were presented, demonstrating how to reconstruct the path weight information by traversing these successor paths until an explicit node is reached.

Furthermore, we investigated compression strategies for the components of our structure,vertex weights $\mathcal{W}$, successor information Σ, and the associated data $\mathcal{D}$ (containing $\mathcal{O}$-sets or $\mathcal{I}$ sequences). Techniques such as variable-length integer coding and Run-Length Encoding, coupled with methods like Elias-Fano encoding for monotonic sequences, were discussed to minimize space occupancy. A significant finding highlighted in Section 4.5 is that the space usage of our proposed structure can fall below the established 0^{th}-order entropy lower bound for lossless graph representation. This is possible because our structure is tailored specifically for the rank query and does not retain the full topological information (the complete edge set E) of the original DAG, thereby achieving high space efficiency for its designated task compared to both lossless graph encodings and methods based on precomputing query results.

While the developed succinct DAG representation offers substantial space savings, a potential performance consideration arises from the query evaluation process itself. The time required to compute a query for an implicit node v depends directly on the length of the successor path that must be traversed from v until an explicit node is encountered (as seen in Algorithm 10). In large or deep DAGs, these paths could potentially become long, leading to variability and potentially slow query times in the worst case for certain nodes. This observation motivates a primary direction for future research: enhancing the query time predictability and efficiency by ensuring that all implicit nodes are reasonably close to an explicit node within the successor path structure.

To address this, instead of relying solely on the sink nodes as the base cases for path traversal, we propose a strategy based on ensuring a maximum traversal distance, denoted by a predefined integer k. The core idea is to augment the original set of explicit nodes V_E (initially, the sinks) with a carefully selected subset of currently implicit nodes, resulting in a new, larger set of explicit nodes $V'_E \supseteq V_E$. This target set V'_E must satisfy a specific property related to the successor function σ: every vertex v that remains implicit (i.e., $v \in V \setminus V'_E$) must be able to reach some node $u \in V'_E$ by following the successor path defined by σ, using at most k steps. More formally, let the successor path starting from v be the sequence $v_0 = v, v_1 = \sigma(v_0), v_2 = \sigma(v_1), \dots$. Then, for every $v \in V \setminus V'_E$, there must exist an index j, where $0 \leqslant j \leqslant k$, such that $v_j \in V'_E$.

The challenge then becomes selecting such a set V'_E that is as small as possible, in order to minimize the additional space overhead associated with storing the $\mathcal{O}$-sets explicitly for these newly designated explicit nodes. This optimization problem is conceptually analogous to finding a minimum distance-k dominating set in a graph. In the standard definition, a distance-k dominating set D is a subset of vertices such that every vertex not in D is within a distance of k (measured by the number of edges in a shortest path) from at least one vertex in D. Our formulation adapts this concept: the "distance" is measured specifically along the directed paths induced by the successor function σ, and the goal is to find a set V'_E of minimum cardinality that "dominates" all other vertices within k steps along these σ-paths. Finding a minimum distance-k dominating set is known to be an NP-hard problem for general graphs. Given the added constraint of using only σ-paths, it is highly probable that determining an optimal set V'_E of minimum size remains computationally intractable for large DAGs.

Consequently, a practical direction for future investigation involves the design and analysis of efficient heuristics. Such heuristics would aim to construct a set V'_E satisfying the k-distance requirement along

σ-paths, while keeping its size reasonably small, even if not guaranteeing absolute minimality. The choice of heuristic would need to balance the quality of the solution (size of V'_E) with the computational cost of finding it.

Once a suitable set V'_E has been determined (whether optimally or via a heuristic), the successor selection function σ needs to be redefined to leverage this structure. For an implicit node $v \in V \setminus V'_E$, the choice of its designated successor $\sigma(v)$ from the set $Succ(v)$ should prioritize reaching *any* node within the target set V'_E as quickly as possible along the subsequent σ-path. That is, $\sigma(v)$ should ideally be selected as the node $u \in Succ(v)$ which minimizes the length of the σ-path starting from u to the nearest node in V'_E. Ties in path length could be resolved using secondary criteria, such as minimizing the $\mathcal{O}$-set cardinality $|\mathcal{O}_u|$ (as in the original heuristic) or simply selecting the successor with the minimum vertex identifier.

Implementing this refined strategy directly imposes an upper bound of k on the number of iterations performed by the main loop in Algorithm 10. This provides a worst-case guarantee on the time complexity of the path traversal component for any rank query, making the overall query performance significantly more predictable and uniform across all nodes, regardless of their position within the DAG. The main trade-off shifts to the selection of the parameter k: smaller values of k yield faster worst-case query times but likely necessitate larger (and thus more space-consuming) sets V'_E, whereas larger values of k may permit smaller sets V'_E at the expense of a higher query time bound.

A ENGINEERING A COMPRESSED INTEGER VECTOR

This appendix outlines the design principles and engineering considerations behind *compressed-intvec*, a software library that we developed for the efficient storage and retrieval of integer sequences [29]. The library leverages the variable-length integer coding techniques discussed in Section 2.6 to achieve significant space savings compared to standard fixed-width representations, while providing mechanisms for acceptably fast data access.

motivation and bitstream abstraction Storing sequences of integers, particularly when many values are small or follow predictable patterns, using standard fixed-width types (such as 64-bit integers) is inherently wasteful. Variable-length integer codes, such as Unary, Gamma, Delta, and Rice codes (Section 2.6), offer a solution by representing integers using a number of bits closer to their information content, assigning shorter codes to smaller or more frequent values.

However, these codes produce binary representations of varying lengths, not necessarily aligned to byte or machine word boundaries. Therefore, storing a sequence of integers compressed with these methods requires packing their binary codes contiguously into a single, undifferentiated sequence of bits, commonly referred to as a bitstream. This necessitates the use of specialized bitstream reading and writing capabilities, abstracting away the complexities of bit-level manipulation. The implementation described here relies on the `dsi-bitstream` library for this purpose [51], ensuring that the variable-length codes can be written to and read from memory efficiently. The fundamental requirement for correctly decoding the concatenated sequence is the prefix-free (self-delimiting) property of the chosen integer code, which guarantees that the end of one codeword can be determined without ambiguity before reading the next.

addressing random access via sampling While bitstream concatenation enables compression, it introduces a significant challenge for random access. Retrieving the i-th integer from the original sequence cannot be done by calculating a simple memory offset, as the bit lengths of preceding elements are variable. A naive approach would require sequentially decoding the first i integers from the beginning of the bitstream, resulting in an unacceptable $O(i)$ access time.

To provide efficient random access, the *compressed-intvec* library employs a *sampling* technique. During the encoding phase, the absolute starting bit position of every k-th integer in the sequence is recorded. These positions, or samples, are stored in an auxiliary data structure, typically a simple array. The value k is a user-configurable sampling parameter that dictates a trade-off between random access speed and the memory overhead incurred by storing the samples.

To retrieve the i-th integer, the library first determines the index of the sample corresponding to the block containing the i-th element: `sample_idx` $= \lfloor i/k \rfloor$. It retrieves the bit offset `start_bit` associated with this sample. The bitstream reader can then jump directly to this position. From `start_bit`, the decoder only needs to perform $i \pmod k$ sequential decoding operations to reach and return the desired i-th integer. If k is considered a constant (e.g., 32 or 64), this reduces the expected time complexity for random access to $O(1)$[1]. The space overhead for the samples is approximately $O((n/k)\log(\text{total\_bits}))$, which is generally sub-linear in the size of the compressed data for practical values of k. The choice of k allows tuning the balance between faster access (smaller k) and lower memory usage (larger k).

CODEC FLEXIBILITY AND DATA DISTRIBUTION The theoretical discussion in Section 2.6 highlights that the efficiency of different integer codes is highly dependent on the statistical distribution of the integers being compressed. For example, Gamma code is suited for distributions decaying roughly as $1/x^2$, Rice codes excel for geometrically distributed values (especially when integers cluster around multiples of 2^k), and minimal binary coding is optimal for uniformly distributed data within a known range [13].

Recognizing this dependency, the *compressed-intvec* library is designed with flexibility in mind. It employs generic programming paradigms (specifically, Rust traits) to allow the user to select the most appropriate integer coding scheme (*codec*) for their specific data distribution at compile time. The library provides implementations for several standard codecs, including Gamma, Delta, Rice, and Minimal Binary [29]. Some codecs, like Rice or Minimal Binary, require additional parameters (the Rice parameter k or the universe upper bound u, respectively), which are also managed by the data structure. Selecting

1 The underlying bitstream operations and single-integer decoding are sufficiently fast to assume that

a codec poorly matched to the data can significantly degrade compression performance, potentially even increasing the storage size compared to an uncompressed representation [29]. This flexibility is therefore crucial for achieving optimal results in practice.

BIBLIOGRAPHY

[1] Jarno N Alanko, Simon J Puglisi, and Jaakko Vuohtoniemi. “Small searchable κ-spectra via subset rank queries on the spectral burrows-wheeler transform.” In: *SIAM Conference on Applied and Computational Discrete Algorithms (ACDA23)*. SIAM. 2023, pp. 225–236.

[2] Jarno N. Alanko et al. “Subset Wavelet Trees.” In: *21st International Symposium on Experimental Algorithms (SEA 2023)*. Ed. by Loukas Georgiadis. Vol. 265. Leibniz International Proceedings in Informatics (LIPIcs). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023, 4:1–4:14.

[3] Mai Alzamel et al. “Degenerate string comparison and applications.” In: *WABI 2018-18th Workshop on Algorithms in Bioinformatics*. Vol. 113. 2018, pp. 1–14.

[4] David Benoit et al. “Representing trees of higher degree.” In: *Algorithmica* 43 (2005), pp. 275–292.

[5] Philip Bille, Inge Li Gørtz, and Tord Stordalen. *Rank and Select on Degenerate Strings*. 2023.

[6] Michael Burrows. “A block-sorting lossless data compression algorithm.” In: *SRS Research Report* 124 (1994).

[7] 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.

[8] David Clark. “Compact pat trees.” In: (1997).

[9] Francisco Claude, Gonzalo Navarro, and Alberto Ordónez. “The wavelet matrix: An efficient wavelet tree for large alphabets.” In: *Information Systems* 47 (2015), pp. 15–32.

[10] T.M. Cover and J.A. Thomas. *Elements of Information Theory*. Wiley, 2012.

[11] P. Elias. “Universal codeword sets and representations of the integers.” In: *IEEE Transactions on Information Theory* 21.2 (1975), pp. 194–203.

[12] Robert Mario Fano. *On the number of bits required to implement an associative memory*. Massachusetts Institute of Technology, Project MAC, 1971.

[13] P. Ferragina. *Pearls of Algorithm Engineering*. Cambridge University Press, 2023.

[14] Paolo Ferragina, Raffaele Giancarlo, and Giovanni Manzini. "The myriad virtues of Wavelet Trees." In: *Information and Computation* 207.8 (2009), pp. 849–866.

[15] Paolo Ferragina and Giovanni Manzini. "Opportunistic data structures with applications." In: *Proceedings 41st annual symposium on foundations of computer science*. IEEE. 2000, pp. 390–398.

[16] Paolo Ferragina et al. "Compressed representations of sequences and full-text indexes." In: *ACM Transactions on Algorithms (TALG)* 3.2 (2007), 20–es.

[17] Michael J Fischer and Michael S Paterson. "String-matching and other products." In: (1974).

[18] Simon Gog et al. "From theory to practice: Plug and play with succinct data structures." In: *Experimental Algorithms: 13th International Symposium, SEA 2014, Copenhagen, Denmark, June 29–July 1, 2014. Proceedings 13*. Springer. 2014, pp. 326–337.

[19] Roberto Grossi, Ankur Gupta, and Jeffrey Vitter. "High-Order Entropy-Compressed Text Indexes." In: *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms* (Nov. 2002).

[20] Roberto Grossi, Ankur Gupta, Jeffrey Scott Vitter, et al. "When indexing equals compression: experiments with compressing suffix arrays and applications." In: *SODA*. Vol. 4. 2004, pp. 636–645.

[21] Roberto Grossi, Jeffrey Scott Vitter, and Bojian Xu. "Wavelet Trees: From Theory to Practice." In: *2011 First International Conference on Data Compression, Communications and Processing*. 2011, pp. 210–221.

[22] T.S. Han and K. Kobayashi. *Mathematics of Information and Coding*. Fields Institute Monographs. American Mathematical Society, 2002.

[23] David A Huffman. "A method for the construction of minimum-redundancy codes." In: *Proceedings of the IRE* 40.9 (1952), pp. 1098–1101.

[24] David A. Huffman. "A Method for the Construction of Minimum-Redundancy Codes." In: *Proceedings of the IRE* 40.9 (1952), pp. 1098–1101.

[25] G. Jacobson. "Space-efficient static trees and graphs." In: *30th Annual Symposium on Foundations of Computer Science*. 1989, pp. 549–554.

[26] J Kärkkäinen. "Repetition-based text indexing." PhD thesis. Ph. D. thesis, Department of Computer Science, University of Helsinki, Finland, 1999.

[27] Juha Kärkkäinen and Simon J Puglisi. “Fixed block compression boosting in FM-indexes.” In: *International Symposium on String Processing and Information Retrieval*. Springer. 2011, pp. 174–184.

[28] S Rao Kosaraju and Giovanni Manzini. “Compression of low entropy strings with Lempel–Ziv algorithms.” In: *SIAM Journal on Computing* 29.3 (2000), pp. 893–911.

[29] Luca Lombardo. *compressed-intvec: Compressed Integer Vector Library*. Rust Crate. URL: https://crates.io/crates/compressed-intvec.

[30] Veli Mäkinen and Gonzalo Navarro. “New search algorithms and time/space tradeoffs for succinct suffix arrays.” In: *Technical rep. C-2004-20 (April). University of Helsinki, Helsinki, Finland* (2004).

[31] Veli Mäkinen and Gonzalo Navarro. “Position-Restricted Substring Searching.” In: *LATIN 2006: Theoretical Informatics*. Ed. by José R. Correa, Alejandro Hevia, and Marcos Kiwi. Springer Berlin Heidelberg, 2006, pp. 703–714.

[32] Veli Mäkinen and Gonzalo Navarro. “Rank and select revisited and extended.” In: *Theoretical Computer Science* 387.3 (2007), pp. 332–347.

[33] Giovanni Manzini. “An analysis of the Burrows—Wheeler transform.” In: *Journal of the ACM (JACM)* 48.3 (2001), pp. 407–430.

[34] Rossano Venturini Matteo Ceregini Florian Kurpicz. *Faster Wavelet Trees with Quad Vectors*. 2024.

[35] Alistair Moffat, Radford M Neal, and Ian H Witten. “Arithmetic coding revisited.” In: *ACM Transactions on Information Systems (TOIS)* 16.3 (1998), pp. 256–294.

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

[37] Gonzalo Navarro. “Wavelet trees for all.” In: *Journal of Discrete Algorithms* 25 (2014). 23rd Annual Symposium on Combinatorial Pattern Matching, pp. 2–20. ISSN: 1570-8667.

[38] Gonzalo Navarro and Veli Mäkinen. “Compressed full-text indexes.” In: *ACM Computing Surveys (CSUR)* 39.1 (2007), 2–es.

[39] Giuseppe Ottaviano and Rossano Venturini. “Partitioned Elias-Fano indexes.” In: *Proceedings of the 37th International ACM SIGIR Conference on Research & Development in Information Retrieval*. SIGIR ’14. New York, NY, USA: Association for Computing Machinery, 2014, pp. 273–282. ISBN: 9781450322577. DOI: 10.1145/2600428.2609615.

[40] Richard Clark Pasco. “Source coding algorithms for fast data compression.” PhD thesis. Stanford University CA, 1976.

[41] Mihai Patrascu. “Succincter.” In: *2008 49th Annual IEEE Symposium on Foundations of Computer Science*. 2008, pp. 305–313.

[42] Giulio Ermanno Pibiri and Rossano Venturini. “Dynamic Elias-Fano Representation.” In: *28th Annual Symposium on Combinatorial Pattern Matching (CPM 2017)*. Ed. by Juha Kärkkäinen, Jakub Radoszewski, and Wojciech Rytter. Vol. 78. Leibniz International Proceedings in Informatics (LIPIcs). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017, 30:1–30:14.

[43] Eli Plotnik, Marcelo J Weinberger, and Jacob Ziv. “Upper bounds on the probability of sequences emitted by finite-state sources and on the redundancy of the Lempel-Ziv algorithm.” In: *IEEE transactions on information theory* 38.1 (1992), pp. 66–72.

[44] 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* 3.4 (Nov. 2007), p. 43.

[45] Robert F Rice. *Some practical universal noiseless coding techniques*. Tech. rep. 1979.

[46] Jorma J Rissanen. “Generalized Kraft inequality and arithmetic coding.” In: *IBM Journal of research and development* 20.3 (1976), pp. 198–203.

[47] 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.

[48] K. Sayood. *Lossless Compression Handbook*. Communications, Networking and Multimedia. Elsevier Science, 2002, pp. 55–64.

[49] C. E. Shannon. “A mathematical theory of communication.” In: *The Bell System Technical Journal* 27.3 (1948), pp. 379–423.

[50] Sebastiano Vigna. “Broadword implementation of rank/select queries.” In: *International Workshop on Experimental and Efficient Algorithms*. Springer. 2008, pp. 154–168.

[51] Sebastiano Vigna et al. *Dsi-bitstream: Bitstream readers/writers for the DSI utilities*. Rust Crate. URL: https://crates.io/crates/dsi-bitstream.

[52] Sebastiano Vigna. “Quasi-succinct indices.” In: *Proceedings of the Sixth ACM International Conference on Web Search and Data Mining*. WSDM ’13. Association for Computing Machinery, 2013, pp. 83–92.

[53] Ian H Witten, Alistair Moffat, and Timothy C Bell. *Managing gigabytes: compressing and indexing documents and images*. Morgan Kaufmann, 1999.