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

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.

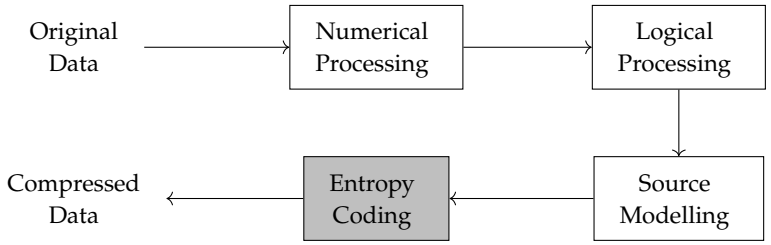


Figure 1: Typical processes in data compression

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 [25, 38, 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*

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

This is also known as Shannon entropy, named after Claude Shannon, who introduced it in his seminal work [50]

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*

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

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*

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

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

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

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

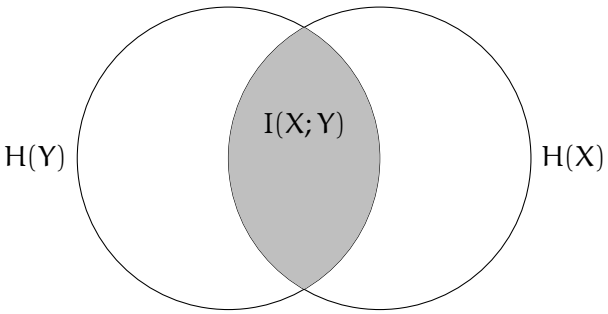


Figure 2: Mutual information between two random variables X and Y.

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*. [25]

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*

$$C(a) = 0$$
$$C(b) = 10$$
$$C(c) = 110$$
$$C(d) = 111$$

The entropy of X *is*

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

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

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

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*

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

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

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,* $\mathsf{E} \neq \mathsf{T}$*, and their respective codes* $\mathsf{C}(\mathsf{E}) =$. *and* $\mathsf{C}(\mathsf{T}) =$ - *are distinct.*
- ***Prefix Code:*** *The code does not satisfy the prefix condition. Several codewords are prefixes of others. For example,* $\mathsf{C}(\mathsf{E}) =$. *is a prefix of* $\mathsf{C}(\mathsf{A}) =$.- *and* $\mathsf{C}(\mathsf{S}) =$ .... *Similarly,* $\mathsf{C}(\mathsf{T}) =$ - *is a prefix of* $\mathsf{C}(\mathsf{N}) =$ -. *and* $\mathsf{C}(\mathsf{M}) =$ --. *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 $\mathsf{l}(\mathsf{x})$ 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

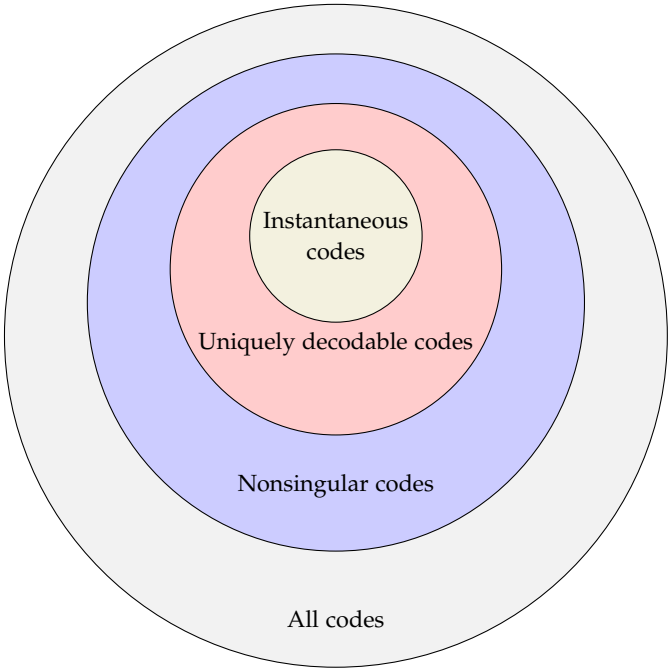


Figure 3: Relationship between different types of codes

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, 25], here we report the one from [25], 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

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

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

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

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

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

That is

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

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

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

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* $n\mathcal{H}(p_1, \ldots, p_\sigma)$

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 [38].

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:*

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

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:*

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

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

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.

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) [38].*

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. [45]. These algorithms typically rely on dictionary-based compression. However, as highlighted by Kosaraju and Manzini [30], 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 [50], 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 $L(C)$:

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

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, 25]).

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

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

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, 55]. 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.

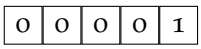


Figure 4: Unary code $U(5) = \texttt{00001}$. It uses $x = 5$ bits, consisting of $x - 1 = 4$ zeros followed by a one.

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 $B(6) = \texttt{110}$, length $l = 3$. The code consists of $U(3) = \texttt{001}$ followed by the $l-1 = 2$ trailing bits (`10`). Result: $\gamma(6) = \texttt{00110}$ (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 $B(x)$.*

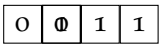


Figure 6: Elias δ code for $x = 6$. $B(6) = \texttt{110}$, length $l = 3$. First, encode $l = 3$ using γ: $\gamma(3) = \texttt{011}$. Then, append the $l - 1 = 2$ trailing bits (`10`). Result: $\delta(6) = \texttt{01110}$ (5 bits).

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 [47] (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, 55]. The fixed length of the remainder part facilitates faster decoding compared to Elias codes in certain implementations.

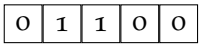


Figure 7: Rice code for $x = 13$ with parameter $k = 3$. Calculate $q = \lfloor (13-1)/2^3 \rfloor = 1$ and $r = (13-1) \pmod 8 = 4$. The code is $U(q+1) = U(2) = \texttt{01}$ followed by $r = 4$ in $k = 3$ bits, $B_3(4) = \texttt{100}$. Result: $R_3(13) = \texttt{01100}$ (5 bits).

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, 44]. This makes it highly suitable for applications such as inverted index compression in modern search engines [54, 41].

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

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

where $0 \leqslant s_0 < s_1 < \ldots < s_{n-1} < u$. 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: $L = L(s_0)L(s_1)\ldots L(s_{n-1})$. 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 $j = 0, 1, \ldots, 2^h - 1$, 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 [44].

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 $pos_0 = \texttt{select}_0(H, H(x)+1)$ (using 1-based index for select); then $p_1 = pos_0 - H(x)$. The ending index p_2 (exclusive) is similarly found using the $(H(x)+1)$-th zero: $pos'_0 = \texttt{select}_0(H, H(x)+1+1)$; then $p_2 = pos'_0 - (H(x)+1)$.
3. Perform a search (e.g., binary search, or linear scan if $p_2 - p_1$ is small) over the lower bits $L[p_1 \cdot l \ldots p_2 \cdot l - 1]$. 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 [44, 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 [26], 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, 49, 25, 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.

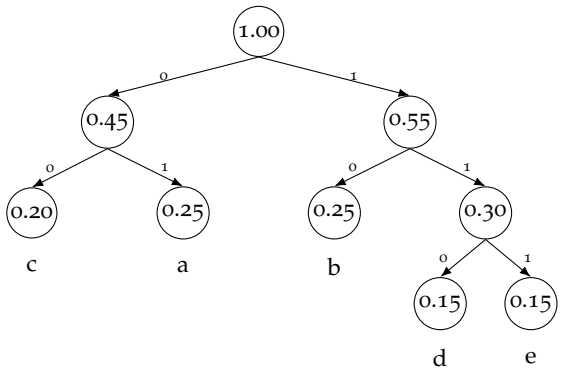


Figure 9: Huffman tree for the example probabilities ($P(a) = 0.25, P(b) = 0.25, P(c) = 0.2, P(d) = 0.15, P(e) = 0.15$). The resulting codes (0 for left, 1 for right) are: $C(a) = \texttt{01}$, $C(b) = \texttt{10}$, $C(c) = \texttt{00}$, $C(d) = \texttt{110}$, $C(e) = \texttt{111}$.

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, 49, 25, 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 $O(\log(1/p_{\min}))$ [38]. 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 [48] and Pasco [42] 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.

```
Compute cumulative probabilities C(σ) = Σ_{σ'<σ} P(σ') (Note: sum over σ' < σ)
l ← 0
s ← 1                                   ▷ Initial interval [0, 1), size 1
for i = 1 to n do
    l_new ← l + s · C(S[i])             ▷ Calculate start of sub-interval
    s_new ← s · P(S[i])                 ▷ Calculate size of sub-interval
    l ← l_new
    s ← s_new
end for
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 [37, 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 $S[1..n]$.

```
Compute cumulative probabilities C(σ) = Σ_{σ'<σ} P(σ')
l ← 0
s ← 1
S ← empty sequence
for i = 1 to n do
    Find symbol σ such that l + s · C(σ) ⩽ x < l + s · (C(σ) + P(σ))
    S.append(σ)
    l_new ← l + s · C(σ)
    s_new ← s · P(σ)
    l ← l_new
    s ← s_new
    ▷ Practical decoders also update x relative to the new interval
end for
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* $2 + nH$*, where* 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, 49, 25, 10]. The core idea, as outlined above, relates the required number of bits k to the final interval size $s_n = 2^{-nH}$ via $k \approx -\log_2 s_n = nH$. The additive constant accounts for representing a specific point within the interval. □

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 [37].*

3

RANK AND SELECT

In the preceding chapters, we explored foundational concepts related to data compression and information theory. We now transition to the domain of *compressed data structures*, which are designed to store data in a compact format while still permitting efficient query operations directly on the compressed representation. This paradigm often leads to what is sometimes termed *pointer-less programming*, where traditional memory pointers are eschewed in favor of structures built upon bit sequences (bitvectors) augmented with operations that implicitly handle navigation and access [13].

This chapter introduces the *bitvector* as a fundamental building block in this area. We will formally define the core operations associated with bitvectors, namely `rank` and `select`, and investigate techniques to support these operations efficiently, often in constant time, while maintaining low space overhead (succinctness). Subsequently, we will delve into methods for compressing bitvectors themselves, particularly exploiting skewed distributions of bits, and discuss practical considerations for implementing these structures effectively. We will also briefly touch upon generalizations like wavelet trees for handling larger alphabets later in the thesis. The efficient implementation of rank and select on bitvectors is crucial, as they underpin numerous advanced compressed data structures used throughout computer science [38].

3.1 BITVECTORS

Consider the following problem [13]: imagine a dictionary $\mathcal{D}$ containing n strings from an alphabet Σ. We can merge all strings in $\mathcal{D}$ into a single string $T[1, m]$, without any separators between them, where m is the total length of the dictionary. The task is to handle the following queries:

- `Read(i)`: retrieve the i-th string in $\mathcal{D}$.
- `Which_string(x)`: find the starting position of the string in T, including the character $T[x]$.

The conventional solution involves employing an array of pointers $A[1, n]$ to the strings in $\mathcal{D}$, represented by their offsets in $T[1, m]$, requiring $\Theta(n \log n)$ bits. Consequently, `Read(i)` simply returns $A[i]$,

while `Which_string(x)` involves locating the predecessor of x in A. The first operation is instantaneous, whereas the second one necessitates $O(\log n)$ time using binary search.

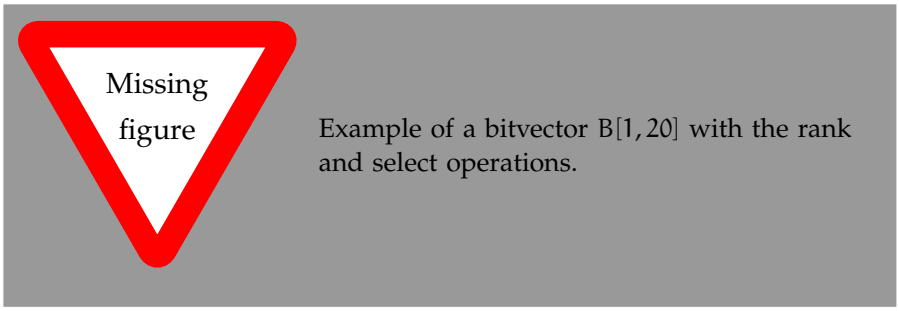
We can address the problem by employing a compressed representation of the offsets in A via a binary array $B[1, m]$ of m bits, where $B[i] = 1$ if and only if i is the starting position of a string in T. In this case then `Access_string(i)` searches for the i-th 1 in B, while `Which_string(x)` counts the number of 1s in the prefix $B[1, x]$.

In modern literature this two operations are well known as *rank* and *select* queries, respectively.

Definition 3.1 (Rank and Select)**.** *Given $B[1, n]$ a binary array of n bits (a bitvector), we define the following operations:*

- *The **rank** of an index i in B relative to a bit b is the number of occurrences of b in the prefix $B[1, i]$. We denote it as $\texttt{rank}_1(i) = \sum_{j=1}^{i} B[j]$. Similarly we can compute $\texttt{rank}_0(i) = i - \texttt{rank}_1(i)$ in constant time.*
- *The **select** of the i-th occurrence of a bit b in B is the index of the i-th occurrence of b in B. We denote it as $\texttt{select}_b(i)$. Opposite to rank, we can't derive select of 0 from select of 1 in constant time.*

Example 3.2 (Rank and Select on a plain bitvector)**.**



Example of a bitvector $B[1, 20]$ with the rank and select operations.

As stated before, bitvectors are the fundamental piece in the implementation of compressed data structures. Therefore, an efficient implementation is crucial. In the following sections, our aim is to built structures of size $o(n)$ bits that can be added on top either the bit array or the compressed representation of B to facilitate rank and select operations. We will see that will often encounter skewed distributions of 0s and 1s in B, and we will exploit this property to achieve higher order compression.

Remark 3.3. *If we try to compress bitvectors with the techniques seen in Chapter 2, we would need to encode each bit individually, requiring at least n bits.*

3.1.1 *Rank*

In their seminal paper [46] Raman et al. introduced a hierarchical succinct data structure that supports the rank operation in constant time, while only using only extra $o(n)$ bits of space. The structure is based on the idea of splitting the binary array $B[1, n]$ into big and small blocks of fixed length, and then encoding the number of bits set to 1 in each block.

More precisely, the structure is composed of three levels: in the first one we (logically) split $B[1, n]$ into blocks of size Z each, where at the beginning of each superblock we store the number (*class number*) of bits set to 1 in the corresponding block, i.e the output of the query $\mathtt{rank}_1(i)$ for i being the starting position of the block. In the second level, we split the superblocks into blocks of size z bits each[1] with the same meta-information stored at the beginning of each block. Finally the third level is a lookup table that is indexed by the small blocks and queried positions. In other words, for each possible small block and each possible position within that block, the lookup table stores the result of the $\mathtt{rank}_1$ operation. This pre-computed information allows for constant time retrieval of the $\mathtt{rank}_1$ operation results, as the result can be directly looked up in the table instead of having to be computed each time. This is the key to the efficiency of the data structure. In this way, the $i-\mathtt{th}$ block, of size Z, can be accessed as

$$B[i \cdot Z + 1, (i+1) \cdot Z]$$

while the small block j of size z in the $i-\mathtt{th}$ superblock is

$$B[i \cdot Z + j \cdot z + 1, i \cdot Z + (j+1) \cdot z] \qquad \forall j \in [0, Z/z), \forall i \in [0, n/Z)$$

We will denote with r_i and call it *absolute rank* the number of bits set to 1 in the $i-\mathtt{th}$ block, and with $r_{i,j}$ (*relative rank*) the number of bits set to 1 in the $j-\mathtt{th}$ small block of the $i-\mathtt{th}$ superblock. Figure 10 shows a visual representation of the RRR data structure.

Let's focus on the third level: the lookup table. Along with the value of the absolute and relative ranks, we also store an offset that serves as an index[2] into the table. To be precise, this table is a table of tables: one for each possible value of r_i and $r_{i,j}$. The table T is then indexed by the values of r_i and $r_{i,j}$. For every possibile value of r_i and $r_{i,j}$, the sub-table stores an array of prefix sums. Thus, since we have $\binom{Z}{z}$ possible values for r_i and $r_{i,j}$ (and consequently entries in the considered sub-table), the lookup table has a size of $\binom{Z}{z} \log Z$ bits. In Table 1 we show an example of a lookup table for the RRR data structure.

We can now state the following theorem [13]:

1 For simplicity, we assume that z divides Z

2 I we imagine that the blocks are sorted lexically, the offset is position of the block in that order

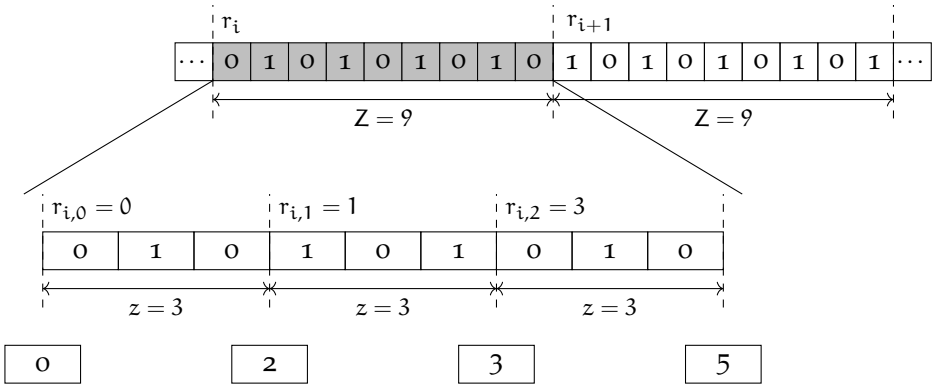


Figure 10: The RRR Rank data structure. The first level is composed of blocks of size Z, the second level of blocks of size z, and the third level is an entry of the lookup table.

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 the RRR data structure. The table stores the result of the rank operation for all possible small blocks with $z = 3$. The cell $T[b, r_{i,j}]$ stores the result of the rank operation for the block b inside the $i-th$ superblock and the $j-th$ small block.

Theorem 3.4. *The space occupancy of the Rank data structure is* $o(n)$ *bits, and thus it is asymptotically sublinear in the size of the binary array* $B[1, n]$*. The Rank algorithm takes constant time in the worst case, and accesses the array* B *only in read-mode*

Proof. The space occupancy of all the big blocks can be computed by multiplying the number of big blocks by the number of bits needed to store the *absolute rank* of each block. Thus, the space occupancy of the big blocks is $O(\frac{n}{Z} \log m)$ bits, since each block can store at most m bits. The same reasoning can be applied to the small blocks, which occupy $O(\frac{n}{z} \log Z)$ bits, since each block can store at most Z bits. So the space complexity is

$$O\left(\frac{n}{Z} \log m + \frac{n}{z} \log Z\right) \tag{1}$$

Let's set $Z = (\log n)^2$ and $z = 1/2 \log n$, then the space complexity becomes

$$= O\left(\frac{n}{(\log n)^2} \log m + \frac{n}{\frac{1}{2}\log n} \log(\log n)^2\right) \tag{2}$$

$$= O\left(\frac{n}{\log^2 n} \log m + \frac{n}{\log n} \log\log n\right) \tag{3}$$

$$= O\left(\frac{n \log\log n}{\log n}\right) = o(n) \tag{4}$$

□

The $o(n)$ space complexity highlighted in Theorem 3.4 signifies that the auxiliary structures consume asymptotically less space than the bitvector itself. Significant research effort has been dedicated to minimizing the constant factors hidden within this $o(n)$ term and understanding the inherent space-time tradeoffs. Works such as [24] explore techniques to further reduce this redundancy, striving for implementations that are both theoretically efficient and practically performant, often achieving space bounds closer to the information-theoretic minimum $B(n, m)$ (the space needed just to represent the bitvector) plus a smaller redundancy term, especially for certain ranges of n and m.

The current explanation of this data structure only clarifies how to respond to rank queries for indices located at the end of a block (or superblock). This can be achieved efficiently, taking constant time, either by directly accessing the value in the lookup table or by calculating the cumulative rank of preceding blocks along with the relative rank within the current block.

However, we also need to address the non-trivial case where the index i is located in the middle of a block[3]. Differently from the previous case, if we want to compute the $\texttt{rank}_1$ operation over an arbitrary position x, we would need to compute $r_i + r_{i,j} + \texttt{popcount}(B_{i,j}[1, x])$, where the last term is an operation that counts the number of bits set to 1 in the prefix $B_{i,j}[1, x]$. While the first two terms can be computed in constant time, the last term requires $O(\log n)$ time[4] in the worst case.

If the size of the small blocks doesn't fit in a single memory word, we can pre-process in our lookup table (the third level of the data structure) all the results of the `popcount` operation for all possible blocks and then use this table to answer rank queries in constant time (as shown in table 1). Let's denote this table as T and see how to use it to answer rank queries in constant time. In order to retrive the

3 For the sake of simplicity, we will assume that $B[x]$ is included in the $j-th$ small block of the $i-th$ superblock

4 It actually grows log-logarithmically with the size of the small blocks

result of $\texttt{popcount}(B_{i,j}[1, x])$ we can access the table T at the position $T[B_{i,j}, o]$. Where o is the offset of the bit $B[x]$ in $B_{i,j}$, and $B_{i,j}$. The offset o can be computed as $o = 1 + ((x - 1) \mod z)$. Thus we only need to perform three atomic operations, two memory accesses and one addition, to retrieve the result of the rank operation in constant time.

Storing this table requires $O(\sqrt{n} \log \log n)$ bits[5], which is asymptotically sub-linear in the size of the binary array $B[1, n]$ and allows the popcount operation in a block of $O(\log n)$ bits in constant time. Thus, if we consider the word length as $\log n$ and still maintain the $o(n)$ space occupancy stated in 3.4

3.1.2 *Select*

The select operation can be seen as the inverse of the rank operation, i.e given a binary array B and an integer i, the select operation returns the index of the i-th occurrence of a bit b in B. More formally, we have that:

$$\mathrm{rank}_c(B, \mathrm{select}_c(B, i)) = i$$

Maybe talk about monoids and how rank and select are inverses of each other

The implementation of the select operation heavily relies on the three level data structure discussed before (3.1.1). The difference lies in the fact that, in this case, the bitmap B doesn't get split into blocks of fixed size, but rather into blocks of variable size that are determined by the rank of the block. We start by designing the first level of the select data structure: we split the bitmap B into blocks of size Z bitvectors each containing K bits set to 1.

Remark 3.5 (Notation and assumptions)**.** *In the following,* Z *will represent, as before, the size in bits of the big blocks containing* K *bits set to* 1*, where* $K = \log n$*. We will use always the same notation* Z *even if the size of the blocks is variable, clarifying the context in which it is used.*

Since $K \leqslant Z$, we can easily derive that space occupance of all the starting positions of the blocks $O(\frac{n}{K} \log n) = o(n)$ bits. The first step of our search in then clear: since each block contains K bits set to 1, we can find the block containing the i-th occurrence of 1 in B by computing i/K.

The second step is to find the i-th occurrence of 1 in the block. This could be done by scanning the block from the beginning and counting the number of bits set to 1 until we reach the i-th occurrence,

5 We have 2^z rows and z columns and each cell stores a value in $[0, z]$.

but this would require $O(K)$ time making it highly un-efficient for our purposes. To address this issue, we introduce the second level of the select data structure where we divide the big blocks into smaller blocks and categorize them into two types: *dense* and *sparse* blocks. A big block is considered *dense* if $Z \leqslant K^2$ and *sparse* otherwise. When dealing with a sparse block, we can store the positions of the bits set to 1 in the block in a separate array, allowing us to access the i-th occurrence of 1 in constant time. Due to it's small number of bits set to 1, we can store the positions in $O(\frac{n}{K^2} K \log n) = O(\frac{n}{\log^2 n} \log n) = o(n)$ bits.

Dealing with the dense blocks is not as straightforward as with the sparse ones. In this case, we can't afford to store the positions of the bits set to 1 in the block, as it would require too much space. We introduce then the third level of the select data structure, where we split the dense blocks into smaller blocks of length[6] z, each containing $k = (\log\log m)^2$ bits set to 1. Thus storing all the starting positions of the smalls blocks and relative beginning of the dense blocks requires $O(\frac{n}{k} \log K^2) = O(\frac{n}{(\log\log n)^2} \log\log^4 n) = o(n)$ bits[7].

The only remaining issue is to keep track of the positions of the bits set to 1 in the small blocks. We can follow the idea introduced for the big blocks and divide them into *dense* and *sparse* small blocks. The sparse small blocks are those with length less then $k^2 = (\log\log m)^4$, and we can store the positions of the bits set to 1 in the block relative to the beginning of its enclosing block in

$$O\left(\frac{n}{k^2} k \log K^2\right) = O\left(\frac{n}{(\log\log n)^2} \log\log^4 n\right) = o(n)$$

bits[8]. Following the idea of the third level of the rank data structure, we can store the positions of the bits set to 1 in the dense small blocks in a lookup table, allowing us to access the i-th occurrence of 1 in constant time. This table will store all the pre-computed results of the select operation for all possible small blocks and, since $z \leqslant k^2$, having 2^z columns and z rows, it will require $O(z2^z \log z) = o(n)$ bits[9].

Remark 3.6 (Pratical Considerations)**.** *The value* $(\log\log m)^4$ *can be very small for practical values of* m*, thus we could avoid dividing the small blocks into dense and sparse blocks and just scan the block from the beginning to find the* i*-th occurrence of* 1*.*

6 The same assumptions made before apply as well: z can vary but we will use the same notation for simplicity and clarify the context in which it is used idf necessary

7 We exploited the fact that each small block has at least length k and the length of its enclosing dense block is at most K^2.

8 We exploited the fact that each sparse small block has length $z > k^2$, thus their number is $O(\frac{n}{k^2})$. We also note that the length of the enclosing dense block is at most K^2.

9 Each cell of the table stores a value in $[0, z]$, thus the $\log z$ factor.

In algorithm 3 are outlined the steps of the $\texttt{select}_1$ (the $\texttt{select}_0$ works in the same way) algorithm, which takes as input the binary array B and an index i, and returns the index of the i-th occurrence of a bit b in B.

Algorithm 3 $\texttt{Select}_1$ Algorithm

```
function Select_1(B, i)
    j = 1 + ⌊(i−1)/K⌋                                  ▷ index of big block
    B_j ← big block j
    if B_j is sparse then
        S ← array of positions of bits set to 1 in B_j
        return S[i mod K]
    else
        s_j ← starting position of B_j
        i' ← 1 + (i − 1 mod K)          ▷ Relative select index in the block
        j' ← 1 + ⌊(i'−1)/k⌋                              ▷ index of small block
        B_{j,j'} ← small block j' in big block j
        s_{j,j'} ← starting position of B_{j,j'}
        if B_{j,j'} is sparse then
            S ← array of positions of bits set to 1 in B_{j,j'}
            return s_j + S[i' mod k]
        else
            o ← 1 + (i' − 1 mod k^2)                 ▷ offset in the small block
            return s_j + s_{j,j'} + T[B_{j,j'}, o]
        end if
    end if
end function
```

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

Theorem 3.7. *The space occupancy of the Select data structure is* $o(n)$ *bits, and thus it is asymptotically sublinear in the size of the binary array* $B[1, n]$*. The Select algorithm takes constant time in the worst case, and accesses the array* B *only in read-mode*

Proof. Follows from the previous discussion. □

For dense small blocks whose size z is very small (e.g., $z \leqslant k^2 = (\log\log m)^4$), direct scanning can indeed be faster than accessing the precomputed table T.

3.1.3 *Compressing Sparse Bitvectors with Elias-Fano*

The rank and select structures discussed before (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

os (or vice-versa). Let $m = \mathrm{rank}_1(n)$ 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 [54] 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 $\mathrm{select}_1(i)$ 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 [54]. However, this space efficiency comes at the cost of potentially slower rank_1 and access (checking the value of $B[i]$) 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 [38]. Therefore, Elias-Fano presents a compelling space-time trade-off, offering near-optimal compression for sparse bitvectors at the expense of rank 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 [13].

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 [52] applied these techniques to rank and select queries, leading to highly efficient practical implementations.

The `rank9` structure proposed by Vigna [52] 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 [52]) 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 include:

- Speed: Exploits fast register operations and avoids large table lookups, often outperforming other methods in practice.
- Space Efficiency: Requires relatively low space overhead, typically around 25% on top of the original bitvector B, mainly for storing the cumulative rank counts.
- Branch Avoidance: Broadword algorithms are generally branch-free, which benefits performance on modern pipelined processors by avoiding potential misprediction penalties.

Similarly, Vigna [52] 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

Wavelet trees, introduced in 2003 by Grossi, Gupta, and Vitter [20], represent a significant development in the field of succinct data structures. They function as self-indexing structures, capable of supporting fundamental queries like `rank` and `select` directly on the compressed representation, while still permitting access to the original sequence data. This blend of capabilities makes them exceptionally useful, particularly in the construction of compressed full-text indexes such as the FM-index [15], where they are often employed to efficiently handle rank queries during pattern matching [39].

Interestingly, the core idea behind wavelet trees shares resemblance to earlier structures; notably, it can be seen as a generalization of a data structure developed by Chazelle in 1988 [7] for computational geometry problems, designed to represent point grids and manage their reshuffling based on coordinates. Furthermore, Kärkkäinen utilized a related concept in 1999 [28] within the context of repetition-based text indexing. However, the specific formulation and the range of applications envisioned by Grossi et al. were distinct and led to a broader impact [39].

The versatility of wavelet trees stems from their ability to be interpreted in multiple ways: (i) as a direct representation of a sequence, (ii) as an encoding of a permutation or reordering of elements, and (iii) as a representation of points on a grid. Since their inception, these perspectives, along with their interactions, have fueled innovative solutions across a surprisingly wide array of problems, extending far beyond their origins in text indexing and computational geometry [39, 23, 14].

An introduction to the problem

Consider a sequence $S[1, n]$ as a generalization of bitvectors whose elements $S[i]$ are drawn from an alphabet Σ[10]. We are interested in the following operations on the sequence S:

- `Access(i)`: return the i-th element of S.
- `Rank(c,i)`: return the number of occurrences of character c in the prefix $S[1, i]$.
- `Select(c,i)`: return the position of the i-th occurrence of character c in S.

10 The size of the alphabet varies depending on the application. For example, in DNA sequences, the alphabet is $\Sigma = \{A, C, G, T\}$, while in other case it could be of millions of characters, such as in natural language processing.

However, dealing with sequences is much more complex than dealing with bitvectors (as we have seen in Section 3.1). Navarro [38] shows how a naive approach to solve this problem would require $n\sigma + o(n\sigma)$ bits of space, which is not space-efficient. Consider σ bitvectors of length n, one for each symbol in the alphabet such that the i-th bit of the c-th bitvector is 1 if $S[i] = c$ and 0 otherwise. Then answering a $rank$ and $select$ query would be done by this simple transformation

$$rank_c(S, i) = rank_1(B_c, i)$$
$$select_c(S, j) = select_1(B_c, j)$$

If we try to use the techniques from Section 3.1 to compress the bitvectors, we would end up with a constant time complexity for the $rank$ and $select$ queries, but with the downside of a space occupancy of $n\sigma + o(n\sigma)$ bits. This is not space-efficient considering that the plain representation of the string requires $n \log \sigma + o(n)$ bits.[11]

Remark 3.8 (Notation). *From now on, let* $S[1, n] = s_1 s_2 \dots s_n$ *be a sequence of length* n *over an alphabet* Σ *that for simplicity we write as* $\Sigma = \{1, \dots, \sigma\}$. *In this way, the string can be represented using* $n\lceil \log \sigma \rceil = n \log \sigma + o(n)$ *bits in plain form.*

3.2.1 *Structure and construction*

In the beginning of this section we showed that storing one bitvector per symbol is not space-efficient. The wavelet tree is a data structure that solves this problem by using a recursive hierarchical partitioning of the alphabet. Consider the subset $[a, b] \subset [1, \dots, \sigma]$, then a wavelet tree over $[a, b]$ is a balanced binary tree with $b - a + 1$ leaves[12]. The root node v_{root} is associated with the whole sequence $S[1, n]$, and stores a bitmap $B_{v_{root}}[1, n]$ defined as follows: $B_{v_{root}}[i] = 0$ if $S[i] \leqslant (a + b)/2$ and $B_{v_{root}}[i] = 1$ otherwise. The tree is then recursively built by associating the subsequence $S_0[1, n_0]$ of elements in $[a, \dots, \lfloor (a + b)/2 \rfloor]$ to the left child of v, and the subsequence $S_1[1, n_1]$ of elements in $[\lfloor (a + b)/2 \rfloor + 1, \dots, b]$ to the right child of v. This process is repeated until the leaves are reached. In this way the left child of the root node, is a wavelet tree for $S_0[1, n_0]$ over the alphabet $[a, \dots, \lfloor (a + b)/2 \rfloor]$, and the right child is a wavelet tree for $S_1[1, n_1]$ over the alphabet $[\lfloor (a + b)/2 \rfloor + 1, \dots, b]$. [39]

Building a wavelet tree is a recursive process that takes $O(n \log \sigma)$ time by processing each node of the tree in linear time. The steps

11 Even if we use a compressed representation of the bitvectors, the space occupancy would still have the dominant term $n\sigma$, which is at least $\Omega(n\sigma \log \log n / \log n)$ bits if constant-time $rank$ and $select$ queries are still required.

12 if $a = b$ then the tree is just a leaf

are outlined in Algorithm 4. Excluding the sequence S and the final wavelet tree T, the algorithm uses $n \log \sigma$ bits of space [13].

Algorithm 4 Building a wavelet tree

```
function BUILD_WT(S, n)
    T ← build(S, n, 1, σ)
    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 ← new node
    m ← ⌊(a + b)/2⌋
    z ← 0                                     ▷ number of elements in S that are ⩽ m
    for i ← 1 to n do
        if S[i] ⩽ m then
            z ← 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 ← 0
    for i ← 1 to n do
        if S[i] ⩽ m then
            bitclear(v.B, i)                  ▷ set i-th bit of v.B to 0
            z ← z + 1
            S_left[z] ← S[i]
        else
            bitset(v.B, i)                    ▷ set i-th bit of v.B to 1
            S_right[i − z] ← S[i]
        end if
    end for
    Free S
    v.left ← build(S_left, z, a, m)
    v.right ← build(S_right, n − z, m + 1, b)
    Pre-process v.B for rank and select queries
    return v
end function
```

Remark 3.9. *The wavelet tree described has σ leaves and $\sigma - 1$ internal nodes, and the height of the tree is $\lceil \log \sigma \rceil$. The space occupancy of each level it's exactly n bits, while we have at most n bits for the last level. The total number of bits stored by the wavelet tree is then upper bounded by $n\lceil \log \sigma \rceil$ bits. [39]. However, if we also interested in storing the topology of the wavelet tree, then another $O(\sigma \log n)$ bits are needed, which can be critical for large alphabets. In [9, 51] are presented some techniques to build wavelet tree in a space-efficient way.*

13 While building the wavelet tree, we can store the sequence S on disk to free memory.

Example 3.10 (Building a wavelet tree)**.** *Consider the sentence*

`wookies_wield_wicked_weapons_with_wisdom$`

where spaces are replaced by underscores and the sentence ends with a special character. The sorted alphabet for this example is

$$\Sigma = \{\$, \_, a, c, d, e, h, i, k, l, m, n, o, p, s, t, w\}$$

where we assume that in the lexicon the special character comes before the underscore. We now assign a bit to each symbol in the alphabet, where 0 is assigned to the first half of the alphabet and 1 to the second half.

$	_	a	c	d	e	h	i	k	l	m	n	o	p	s	t	w
0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1

We can now build the wavelet tree for this sequence, recursively partitioning the alphabet and assigning a bit to each symbol. The resulting wavelet tree is shown in Figure 11

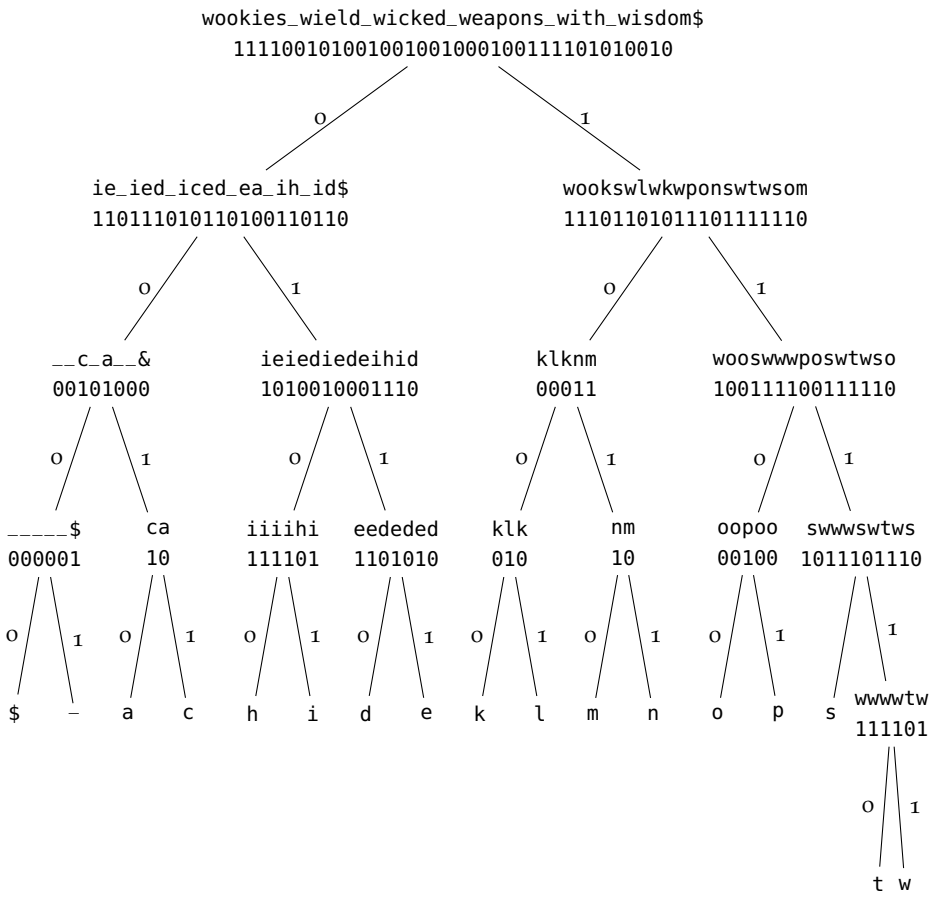


Figure 11: Wavelet tree for the sequence `wookies_wield_...`

Tracking symbols

We have seen how the wavelet tree serves as a representation for a string S, but more than that it is a succinct data structure for the string. Thus, it takes space asymptotically close to the plain representation of the string and allows us to access the i-th symbol of the string in $O(\log \sigma)$ time.

3.2.1.1 *Access*

In algorithm 5 we show how extract the i-th symbol of the string S using a wavelet tree T, this operation is called `Access`. In order to find $S[i]$, we first look at the bitmap associated with the root node of the wavelet tree, and depending on the value of the i-th bit of the bitmap, we move to the left or right child of the root node and continue recursively. However, the problem is to determine where our i has been mapped to: if we move to the left child, then we need to find the i-th 0 in the bitmap of the left child, and if we move to the right child, then we need to find the i-th 1 in the bitmap of the right child. This is done by the $\mathtt{rank}_0$ and $\mathtt{rank}_1$ functions, respectively. We continue this process until we reach a leaf node, and then we return the value of the leaf node.

Algorithm 5 `Access` queries on a wavelet tree

```
function ACCESS(T, i)          ▷ T is the sequence S seen as a wavelet tree
    v ← T_root                                          ▷ start at the root node
    [a, b] ← [1, σ]
    while a ≠ b do
        if access(v.B, i) = 0 then                      ▷ i-th bit of the bitmap of v
            i ← rank_0(v.B, i)
            v ← v.left                                  ▷ move to the left child of node v
            b ← ⌊(a + b)/2⌋
        else
            i ← rank_1(v.B, i)
            v ← v.right                                 ▷ move to the right child of node v
            a ← ⌊(a + b)/2⌋ + 1
        end if
    end while
    return a
end function
```

3.2.1.2 *Select*

In addition to retrieving the i-th symbol of the string, we might also need to perform the inverse operation. That is, given a symbol's position at a leaf node, we aim to determine the position of the symbol in

the string. This operation is referred to as `Select` and is outlined in Algorithm 6. Assume we start at a given leaf node v and want to find the position of the j-th occurrence of symbol c in the string. We recursively move to the left or right child of the node v: if the leaf is the right child of its parent, then we need to find the j-th 1 in the bitmap of the parent node, and if the leaf is the left child of its parent, then we need to find the j-th 0 in the bitmap of the parent node. This is done by the $select_0$ and $select_1$ functions, respectively. We continue this process until we reach the root node, and then we return the position of the symbol in the string. As we have seen in Section 3.1, this two single operations can be solved in constant time if we use the `RRR` data structure [46] on each bitmap. Thus, the time complexity to perform a `Select` query on a wavelet tree is $O(\log \sigma)$.

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*

During the select algorithm, we track upwards the path from the leaf to the root. The process for solving a $rank$ query is similar, but instead of moving from the leaf to the root, we move from the root to the leaf. Algorithm 5 also gives us the number of occurrences of a symbol $S[i]$ in the prefix $S[1, i]$, i.e $rank_{S[i]}(S, i)$. We now want to generalize this operation to solve any rank query $rank_c(S, i)$, where c is a symbol in the alphabet. This procedure is shown in algorithm 7.

As mentioned in Remark 3.9, storing the explicit tree topology can incur an $O(\sigma \log n)$ bit overhead, problematic for large alphabets. This redundancy can be eliminated by adopting a *pointer-less* representation [39]. The balanced tree shape is slightly modified to ensure levels

are fully populated (except potentially the last, which is filled left-to-right). This allows concatenating all the bitmaps at each level into a single large bitmap B_{level} for each level, and then concatenating these level bitmaps into one global bitmap B_{global}. Navigation replaces explicit child pointers with calculations: knowing the bitmap segment $[l, r]$ for a node v at level k, the segment for its left child (at level $k+1$) is determined by the range $[1 + (k+1)n + rank_0(B_{global}, l-1), (k+1)n + rank_0(B_{global}, r)]$ within B_{global}, and similarly for the right child using $rank_1$. This eliminates the pointers, reducing the extra space to the $o(n)$ bits per level required by the rank/select structures on the bitmaps (as seen in 3.1), achieving a total space of $n\lceil \log \sigma \rceil + o(n \log \sigma)$ bits [34, 33]. While space-efficient, this approach might slightly increase query times in practice due to the additional rank operations needed for navigation compared to directly following pointers.

Algorithm 7 Rank queries on a wavelet tree

```
function RANK_c(S, i)
    v ← T_root                                  ▷ start at the root node
    [a, b] ← [1, σ]
    while a ≠ b do
        if c ⩽ ⌊(a + b)/2⌋ then
            i ← rank_0(v.B, i)
            v ← v.left                          ▷ move to the left child of node v
            b ← ⌊(a + b)/2⌋
        else
            i ← rank_1(v.B, i)
            v ← v.right                         ▷ move to the right child of node v
            a ← ⌊(a + b)/2⌋ + 1
        end if
    end while
    return i
end function
```

3.2.2 *Compressed Wavelet Trees*

In order to make the wavelet tree more space efficient, we ask ourselves if we can compress this data structure. The answer is yes, and in this section we will see how a wavelet tree can be compressed to the zero-order entropy of the input string, while still being able to answer rank and select queries in $O(\log \sigma)$ time. The literature on this topic mainly focus one two different approaches: compressing the bitvectors and altering the shape of the wavelet tree itself.

3.2.2.1 *Compressing the bitvectors*

In [20] Grossi et. al showed that if the bitvectors of each single node are compressed to their zero-order entropy, then their overall space occupance is $nH_0(S)$. So if we suppose that the bitmap associated to the root node has a skewed distribution of 0s and 1s, then the zero-order compressing it yields a space of

$$n_0 \log \frac{n}{n_0} + n_1 \log \frac{n}{n_1} \tag{1}$$

where n_0 and n_1 are the number of 0s and 1s in the bitmap, respectively. This is the same as the zero-order entropy of the bitmap. The same reasoning can be applied to the bitmaps of the children of the root node, and so on. This way, one can easy prove by induction [38] that the overall space of the wavelet tree is

$$\sum_{c \in \Sigma} n_c \log(\frac{n}{n_c}) = nH_0(S) \tag{2}$$

We can now choose from the literature any zero-order entropy coding method for the bitvectors that supports rank and select queries in $O(1)$ time. Some of the most popular methods are RRR [46] that we have vastly discussed in Section 3.1, that for each bitvector of length n uses $nH_0(B) + o(n \log\log n / \log n)$ bits. In [43] the authors showed[14] that this value can be further reduced to $nH_0(B) + o(n/\log^c n)$ for any positive constant c.

3.2.2.2 *Huffman-Shaped Wavelet Trees*

Since working in practice with compressed bitvectors can be less efficient then in theory, we want a method for still obtaining nearly zero-order entropy compression, but while maintaining the bitvectors in plain form. The key idea for the compression method that we are going to analyze is that, as noted by Grossi et. al in [21], the shape of the wavelet tree has no impact on the space occupance of the structure. They proposed to use this fact to alter the shape of the tree in order to optimize the average query time. Recalling how we built an Huffman Tree in 2.7.1, we can adapt the same idea to the wavelet tree: given the frequencies f_c with which each leaf node appears in the tree, we can create an Huffman-shaped wavelet tree, obtaining an average access time of

$$\sum_{c \in \Sigma} f_c \log \frac{1}{f_c} \leqslant \log \sigma \tag{3}$$

14 In this case, the time complexity of rank and select queries is $O(c)$.

A counter effect noted by the authors in [21] is that in the worst case, we could and up with a time complexity of $O(\log n)$, for example in the case of a very infrequent symbol.However, if we choose i uniformly at random from $[1, n]$ then the average access[15] time is

$$O(\frac{1}{n}\sum_{c} f_c|h(c)| = O(1 + H_0(S)) \tag{4}$$

That is better than the $O(\log \sigma)$ time of the original balanced wavelet tree.

Remark 3.11. *On a further note, if we bound the depth of the Huffman Tree, we can keep worst care access time to* $O(\log \sigma)$*, with extra* $O(n/\sigma)$ *bits of redundancy*

Another possibile approach following the same idea of a Huffman-shaped wavelet tree, proposed in [32], is to use the frequencies with which the symbols appear in the string. If we use this frequencies to build the Huffman Tree, we can then attach to each node v a bitvector B_v in the same way that we would do for the balanced wavelet tree (4). In this way, the bits of B_v are the bits of of the path from the root to v in the Huffman Tree, i.e the Huffman codes of the symbols. Let's see the space occupance of this structure. Consider a leaf corresponding to a symbol c, at depth $|h(c|)$ (where $h(c)$ is the bitwise Huffman code for c), representing f_c symbols. Each of this occurrences leads to a bit in each bitvector that is in the path from the root to the leaf; that is $|h(c)|$ bits. Thus, the occurrences of c lead to $f_c|h(c)|$ bits in total. If we add this values to all the leaves we obtain same number of bits outputted by the Huffman coding of the string, that is

$$\sum_{c\in\Sigma} f_c|h(c)| \leqslant n(H_0(S) + 1) \tag{5}$$

If we also want to add the space to support the rank and select queries and the tree pointers needed to navigate the tree, we arrive to a space occupance of

$$n(H_0(S) + 1) + o(n(H_0(S) + 1)) + O(n \log \sigma) \tag{6}$$

For the sake of completeness, we also mention that the shape of an Huffman tree is not the only one that can be given to a wavelet tree. In [22] Grossi and Ottaviano gave the wavelet the shape of a trie, making it possible to handle a sequence of strings.

15 And also for $rank_c(S, j)$ or $select_c(S, j)$ with $c = S = [1]$

3.2.2.3 *Higher Order Entropy Coding*

While compressing wavelet trees to their zero-order entropy $H_0(S)$ is effective, many sequences exhibit further compressibility captured by higher-order entropy $H_k(S)$. This measure leverages the statistical dependencies between symbols by considering the context of k preceding symbols (as defined in Section 2.5). Achieving H_k compression often involves the Burrows-Wheeler Transform (BWT) [6], a reversible permutation that groups symbols with similar preceding contexts together in the transformed string, denoted S^{BWT}. Compressing the resulting runs or substrings (S_A for each context A) within S^{BWT} to their respective zero-order entropies effectively achieves $H_k(S)$ for the original sequence S [35].

Early applications connecting wavelet trees and H_k followed this logic. Grossi et al. [20], compressed context-based subsequences using wavelet trees. Ferragina et al. [16] explicitly used the BWT output and partitioned it, applying wavelet trees to compress the parts corresponding to different contexts, thereby reaching $nH_k(S) + o(n \log \sigma)$ bits.

A significant advancement came with the realization, notably explored by Grossi et al. in 2004 [21], that partitioning S^{BWT} might not be necessary. A single wavelet tree built over the entire S^{BWT}, equipped with appropriate compression schemes for its internal bitmaps, could achieve H_k compression directly. Their initial approach using run-length encoding on the bitmaps yielded approximately $2nH_k(S)$ bits plus redundancy.

Subsequently, Mäkinen and Navarro [40] demonstrated that using more sophisticated bitmap representations, such as the fully indexable dictionaries of Raman et al. [46] (mentioned in Section 3.2.2.1), within the single wavelet tree over S^{BWT} allowed reaching the theoretically tighter bound of $nH_k(S) + o(n \log \sigma)$ bits, while still supporting efficient rank/select operations needed for BWT-based indexing.

Ferragina and Manzini [14] provided a thorough analysis of wavelet tree variants for compression, comparing bitmap encodings like Run-Length Encoding (RLE) and Gap Encoding (GE). Their findings supported the suitability of RLE-based approaches (like the one in [21]) for achieving H_k in conjunction with the BWT. More recently, simpler strategies have also proven effective; Kärkkäinen and Puglisi [29] showed that partitioning S^{BWT} into fixed blocks and using Huffman-shaped wavelet trees (Section 3.2.2.2) on each block can practically achieve H_k compression, offering a potentially faster alternative.

3.2.3 *Wavelet Matrices and Quad Vectors*

While the pointerless and Huffman-shaped wavelet trees offer significant space advantages over the basic pointer-based structure, they can still face challenges. Pointerless trees, especially the strict variant, require extra rank operations for navigation, potentially slowing down queries compared to pointer-based trees in practice, despite having the same asymptotic complexity [8]. Furthermore, for very large alphabets σ, even the overhead associated with Huffman models or the $o(n \log \sigma)$ term in pointerless structures might be undesirable. Moreover, a primary bottleneck for query performance in practice, especially with large sequences that do not fit in cache, is memory latency. Each level traversal in a standard wavelet tree typically incurs cache misses, leading to a query time proportional to $\log \sigma$ times the latency of a cache miss.

Two main approaches have emerged to address these issues: the Wavelet Matrix and the use of higher-arity trees (specifically 4-ary or Quad trees).

3.2.3.1 *The Wavelet Matrix*

Introduced by Claude, Navarro, and Ordóñez [8], the wavelet matrix offers an alternative representation that retains the functionality of the pointerless wavelet tree but simplifies navigation and often improves practical performance, particularly for large alphabets.

The core idea is to decouple the alignment between parent and child bitmap segments. Instead of ensuring that the bits corresponding to a node v at level l map to a contiguous block within the level-$(l+1)$ bitmap B_{l+1} that aligns with v's conceptual children, the wavelet matrix uses a global reordering at each level. All the 0-bits from the level-l bitmap B_l are mapped to the beginning of the level-$(l+1)$ bitmap B_{l+1}, followed by all the 1-bits from B_l. To navigate, we only need to know the total number of 0s at level l, denoted z_l.

- If $B_l[i] = 0$, the corresponding position at level $l+1$ is $\mathrm{rank}_0(B_l, i)$.
- If $B_l[i] = 1$, the corresponding position at level $l+1$ is $z_l + \mathrm{rank}_1(B_l, i)$.

This mapping eliminates the need to calculate node boundaries $[s_v, e_v]$ during traversal. Access and rank operations still require one rank query per level, just like the standard pointer-based tree, but without the pointer overhead or the extra rank operations of the strict pointerless variant. Select operations can also be implemented efficiently using this mapping.

The space requirement is identical to the pointerless wavelet tree: $n\lceil \log \sigma \rceil$ bits for the bitmaps plus $o(n \log \sigma)$ for rank/select support (the z_l values require negligible $O(\log \sigma \log n)$ bits total). Experiments in [8] show that the wavelet matrix is significantly faster than pointerless wavelet trees in practice and competitive with, or sometimes faster than, pointer-based wavelet trees, while using substantially less space, especially for large σ. It can also be combined with bitmap compression or Huffman shaping (requiring a specific code assignment strategy [8] to maintain the necessary properties).

3.2.3.2 *4-ary (Quad) Wavelet Trees*

Addressing the latency bottleneck caused by cache misses during the $\log \sigma$ levels of traversal, Venturini et al. [36] proposed using a 4-ary wavelet tree structure. By increasing the arity from 2 to 4, the height of the tree is roughly halved to $\lceil (\log \sigma)/2 \rceil$.

In this structure, each internal node has four children, corresponding to partitioning the current alphabet range $[a, b]$ into four sub-ranges based on the two most significant bits differentiating symbols in that range. Consequently, the data stored at each node is not a bitmap but a quad vector - a sequence over the alphabet $\{00, 01, 10, 11\}$ (or equivalently, $\{0, 1, 2, 3\}$).

To support the standard wavelet tree operations, rank and select queries must now operate on these quad vectors. Venturini et al. [36] developed space-efficient data structures for quad vector rank/select, adapting block-based techniques used for binary vectors. Their implementation achieves constant query times with a space overhead slightly larger than that for binary vectors (e.g., $\approx$ 6.25%-7.81% overhead in their experiments compared to $\approx$ 3.12% for binary rank structures like [19]).

The primary benefit is significantly reduced query latency. By halving the number of traversal steps (levels), the number of dependent memory accesses and potential cache misses is also halved. Experimental results in [36] demonstrate that their 4-ary wavelet tree (specifically, a 4-ary wavelet matrix implementation, combining both ideas) improves the latency of rank and select queries by a factor of approximately 2 compared to widely used binary wavelet tree implementations from libraries like SDSL [18].

The space complexity for a 4-ary wavelet tree using quad vectors is $n\lceil \log \sigma \rceil$ bits (since $n \times 2$ bits are stored per level, but there are only $\approx (\log \sigma)/2$ levels) plus the overhead for quad rank/select support, which is $o(n \log \sigma)$.

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 3.12 (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 3.13 (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 3.14. *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 $\text{subset-rank}_X(i, c)$ is answered by traversing from the root to the leaf for c. At node v, if c is in the left partition, update $i \leftarrow \text{rank}_{L_v}(i, 1)$ and go left; otherwise, update $i \leftarrow \text{rank}_{R_v}(i, 1)$ and go right. The final i is the result (Algorithm 8).

$\text{subset-select}_X(j, c)$ 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 $j \leftarrow \text{select}_{L_u}(j, 1)$; otherwise, update $j \leftarrow \text{select}_{R_u}(j, 1)$. 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 $c \in X_k$

function SUBSET-RANK(c, i)
  $v \leftarrow$ root
  $[l, r] \leftarrow [1, \sigma]$ ▷ Range of alphabet indices
  while $l \neq r$ **do**
    $mid \leftarrow \lfloor (l + r - 1)/2 \rfloor$
    if $c \leqslant mid$ **then**
      $r \leftarrow mid$
      $i \leftarrow \mathsf{rank}_{L_v}(i, 1)$ ▷ Assumes Rank operation on node's bitvector
      $v \leftarrow$ left child of v
    else
      $l \leftarrow mid + 1$
      $i \leftarrow \mathsf{rank}_{R_v}(i, 1)$ ▷ Assumes Rank operation on node's bitvector
      $v \leftarrow$ right child of v
    end if
  end while
  return i
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

function SUBSET-SELECT(c, j)
  $v \leftarrow$ leaf node corresponding to c
  while $v \neq$ root **do**
    $u \leftarrow$ parent of v
    if $v =$ left child of u **then**
      $j \leftarrow \mathsf{select}_{L_u}(j, 1)$ ▷ Assumes Select operation on parent's bitvector
    else
      $j \leftarrow \mathsf{select}_{R_u}(j, 1)$ ▷ Assumes Select operation on parent's bitvector
    end if
    $v \leftarrow u$
  end while
  return j
end function

Theorem 3.15 (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 3.16. 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 3.17). 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 3.16 (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:

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

subset-select in $\mathcal{D}_s(N, \sigma) + O(1)$ *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:*

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

subset-select in $\mathcal{D}_s(N, \sigma) + \mathcal{B}_s(n, n_0)$ *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* $\mathsf{rank}(\cdot, \mathtt{1})$ *in* $\mathcal{B}_r(n, n_0)$ *time, and*
- *supports* $\mathsf{select}(\cdot, \mathtt{0})$ *in* $\mathcal{B}_s(n, n_0)$ *time.*

In theorem 3.16, (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 3.16 (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 3.16 (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 3.16(iii) allows for tailored bitvector structures sensitive to n_0.

Theorem 3.17 (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 3.17 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 3.16 and consider $\mathcal{V}$ a data structure (for example the one described by Jacobson in [27]), 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 3.16 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 $|X_i| - 1$ `0`s. 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 12 from [5] illustrates this reduction for $n_0 = 0$.

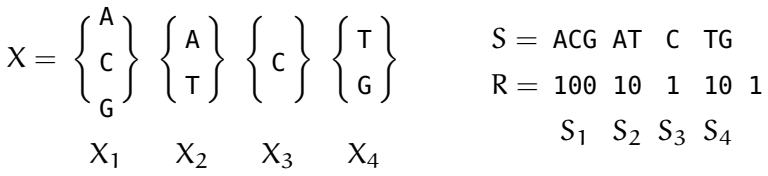


Figure 12: *Left:* A degenerate string X over the alphabet $\{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$ where $n = 4$ and $N = 8$. *Right:* The reduction from Theorem 3.16 (i) on X. White space is for illustration purposes only.

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 12: 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 3.16 (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 3.16 (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 $\mathsf{select}_E(k, 0)$, 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 formal mapping from the sequence structure to a graph structure, illustrating how the degenerate string problem can be viewed as a specific instance within a potentially broader graph-based framework.

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

$$X = \underset{X_1}{\begin{Bmatrix} A \\ C \\ G \end{Bmatrix}} \quad \underset{X_2}{\begin{Bmatrix} A \\ T \end{Bmatrix}} \quad \underset{X_3}{\begin{Bmatrix} T \\ C \\ A \end{Bmatrix}} \quad \underset{X_4}{\begin{Bmatrix} A \\ G \end{Bmatrix}}$$

Figure 13: An example degenerate string $X = X_1X_2X_3X_4$ over $\Sigma = \{A, C, G, T\}$.

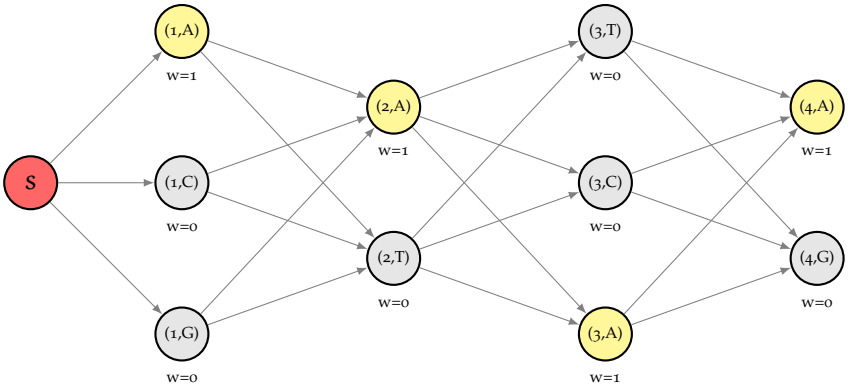


Figure 14: The weighted DAG G_A derived from the degenerate string in Figure 13 for character $c = A$. Nodes visually labeled (k, a) represent the abstract vertices $v_{k,a}$. Nodes with $w_A(v_{k,a}) = 1$ are yellow; those with $w_A(v_{k,a}) = 0$ are gray. Edges represent the connections defined in 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 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 concept 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. We introduce a novel representation strategy that partitions the vertices into two categories: *explicit* vertices, for which path weight information is stored directly, and *implicit* vertices, whose information is derived indirectly through references to other vertices via a carefully defined *successor* relationship.

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

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 (Paths and Path Weights)**.** *A path P from a vertex u to a vertex v in G is a sequence of vertices $P = (v_0, v_1, \ldots, v_k)$ 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. The cumulative weight relevant for our Rank query, denoted $W'(P)$, for a path $P = (v_0 = s, \ldots, v_k = v)$ is defined as the sum of the weights of the vertices along the path, excluding the source vertex:*

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

If the path consists only of the source vertex (i.e., $k = 0$ and $P = (s)$), its weight is defined as $W'(P) = 0$.

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 4.4 ($\mathcal{O}$-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 $v \neq s$:*

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

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 4.3) ending at a vertex.

Proposition 4.5 (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:*

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

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 $\mathtt{Path}(s, s)$ is the trivial path $P = (s)$. According to Definition 4.3, $W'(P) = 0$. By Definition 4.4, $\mathcal{O}_s = \{0\}$. 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 $u \in \mathtt{Pred}(v)$, since the existence of an edge (u, v) implies u precedes v in any topological sort. We must show that $\mathcal{O}_v = \{W'(P) \mid P \in \mathtt{Path}(s, v)\}$.

($\subseteq$) Let $x \in \mathcal{O}_v$. By Definition 4.4, since $v \neq s$, there must exist a predecessor $u \in \mathtt{Pred}(v)$ 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 $P' = (v_0 = s, \ldots, v_{k-1} = u) \in \mathtt{Path}(s, u)$. Consider the path $P = (v_0, \ldots, v_{k-1}, v_k = v)$ formed by appending the edge (u, v) to P'. This is a valid path in $\mathtt{Path}(s, v)$. Its weight according to Definition 4.3 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 $\mathtt{Path}(s, v)$.

($\supseteq$) Let $P = (v_0 = s, \ldots, v_k = v)$ be an arbitrary path in $\mathtt{Path}(s, v)$. 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, $u \in \mathtt{Pred}(v)$. Let $P' = (v_0, \ldots, v_{k-1})$ be the subpath of P ending at u. P' is a path in $\mathtt{Path}(s, u)$. 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 4.4, the value $y + w(v)$ is included in the construction of $\mathcal{O}_v$. We observe that

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

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

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

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

Lemma 4.6 ($\mathcal{O}$-Set Cardinality Monotonicity along Edges)**.** *Let $v \in V$ and let $u \in \mathtt{Succ}(v)$ 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 4.4, the $\mathcal{O}$-set for u is given by

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

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

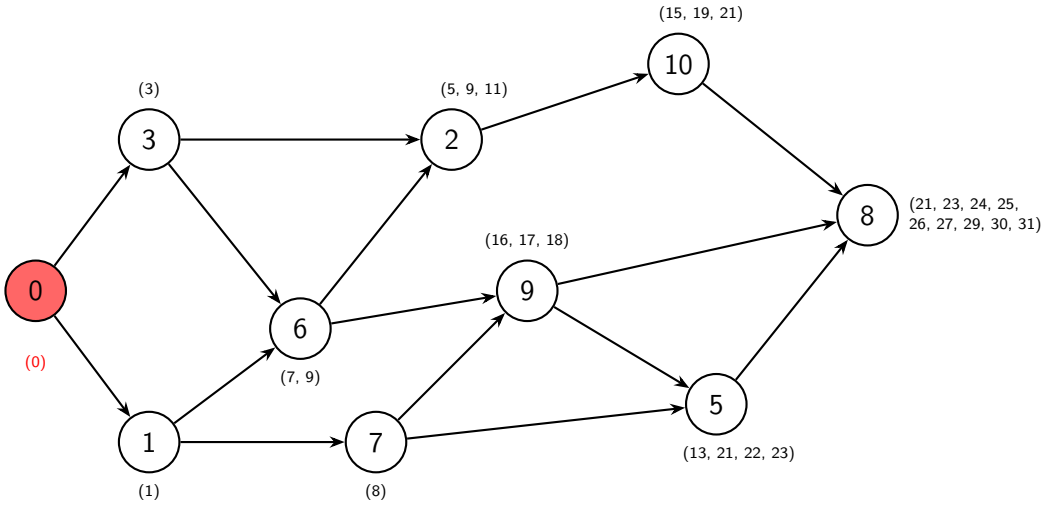


Figure 15: Example of a node-weighted DAG. Each node v contains its weight $w(v)$. The label associated with each node represents its calculated $\mathcal{O}$-set, $\mathcal{O}_v$, considered as a sorted sequence.

Example 4.7 ($\mathcal{O}$-Set Calculation)**.** *Consider the weighted DAG shown in Figure 15. 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 4.4. Let us trace the computation for several nodes, respecting a topological order.*

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 4.8 (Rank Query on Weighted DAG)**.** *Given a vertex $N \in V$ in a weighted DAG $G = (V, E, w)$, the Rank query, denoted $\mathrm{Rank}_G(N)$, 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 $x \in \mathcal{O}_N$:*

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

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 equals S_N,

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

and the intervals are maximally merged, meaning $r_k < l_{k+1} - 1$ for all $k = 1, \ldots, p-1$

Remark 4.9. *As motivated above, the transformation from $\mathcal{O}_N$ to the set S_N via the interval generation rule $\{z \mid \max(0, x - w(N) + 1) \leqslant z \leqslant x\}$ formalizes the idea of capturing the range of values associated with the final node N for each possible incoming path weight x. The length of the generated interval is typically $w(N)$ (unless $x < w(N) - 1$, in which case it starts from 0), ending precisely at x. This mathematical formulation directly implements the intuition of considering values accumulated during the phase associated with node N. The subsequent union and merging into disjoint intervals provide a canonical and compact representation of the overall set S_N.*

Example 4.10 (Rank Query Calculation with Disjoint Result)**.** *Let us compute the Rank query for vertex $N = 2$ in the DAG shown in Figure 15. From Example 4.7, we have:*

- *The weight of node 2 is $w(2) = 2$.*
- *The $\mathcal{O}$-set for node 2 is $\mathcal{O}_2 = \{5, 9, 11\}$. These are the possible total weights of paths ending at node 2.*

Applying Definition 4.8, we associate an interval with each $x \in \mathcal{O}_2$, representing the values active during the processing of node 2:

- *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* $\mathcal{R}_2$. *The intervals are already sorted by starting point.*

- *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:

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

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.,* $\text{Succ}(v) \neq \emptyset$*), we select a designated successor* $\sigma(v) \in \text{Succ}(v)$ *according to the following heuristic rule:*

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

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 [1].

1. *Weights* $\mathcal{W}$*:* An array storing the weight $w(v)$ for each vertex $v \in V$. Conceptually, $\mathcal{W}[v] = w(v)$.
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.

1 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, $\mathcal{D}[v]$ conceptually 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).

Let's focus on the content of the Associated Data $\mathcal{D}$ based on the vertex partitioning:

EXPLICIT VERTEX DATA ($\mathcal{D}_E$) For $v \in V_E$ (Explicit Vertex), $\mathcal{D}[v]$ conceptually 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 $\mathcal{O}_v$.

IMPLICIT VERTEX DATA ($\mathcal{D}_I$) For $v \in V_I$ (Implicit Vertex), let $u = \sigma(v)$ be the designated successor. $\mathcal{D}[v]$ conceptually stores the *offset sequence* $\mathcal{I}_v$. This sequence $\mathcal{I}_v = (j_0, j_1, \ldots, j_{m-1})$ is a 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 $\mathcal{D}_I(v)$.

The successor information $\Sigma[v]$ provides the link $u = \sigma(v)$, and the weight $w(u)$ needed for the reconstruction is retrieved from $\mathcal{W}[u]$.

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 4.6. 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.

Example of the Structure

To concretely illustrate the succinct representation, let us apply the principles from Section 4.2.1 to the example DAG introduced in Figure 15. The resulting structure is visualized in Figure 16.

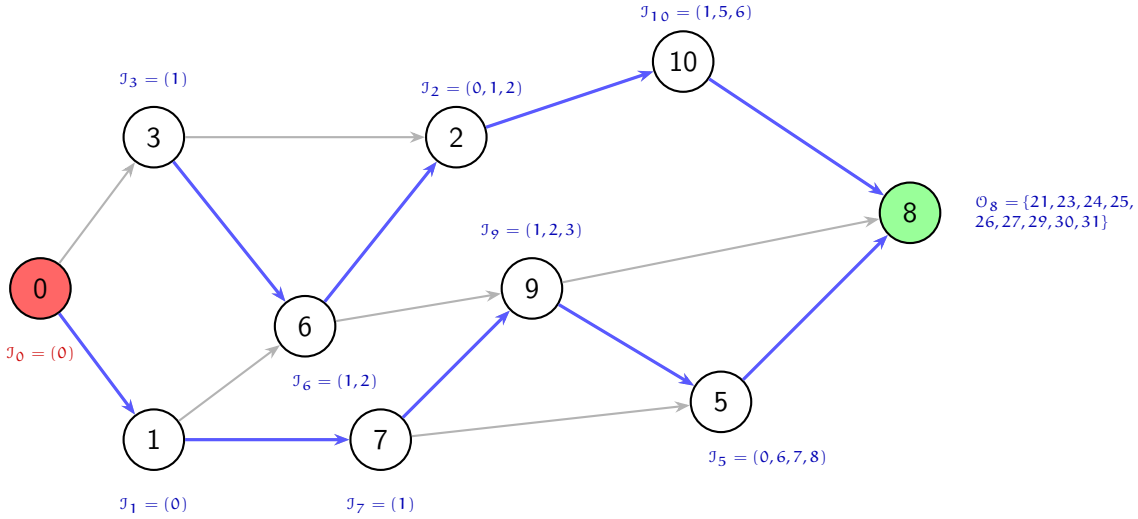


Figure 16: Illustration of the succinct DAG representation for the graph in Figure 15. Implicit nodes are shown with standard borders, while the explicit sink node (8) is shaded green. The source node (0) is red. Highlighted blue edges indicate the chosen successor $\sigma(v)$ for each implicit node v, selected according to Definition 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 $V_E = \{v \mid \text{Succ}(v) = \emptyset\}$ and implicit non-sinks $V_I = V \setminus V_E$. In the example (Figure 16), $V_E = \{8\}$.

2. *Successor Selection:* For each $v \in V_I$, a successor $\sigma(v)$ is chosen from $\text{Succ}(v)$ to minimize $|\mathcal{O}_{\sigma(v)}|$. The selected successors are shown as blue edges in Figure 16. 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 $\mathcal{I}_v$ is stored: $\mathcal{D}[v] = \mathcal{D}_I(v) = \mathcal{I}_v$. Let $u = \sigma(v)$. The sequence $\mathcal{I}_v = (j_0, \ldots, j_{m-1})$ 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, $\mathcal{I}_3 = (1)$.

The offset sequences for all implicit nodes are computed similarly and displayed as labels in Figure 16.

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 17.

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

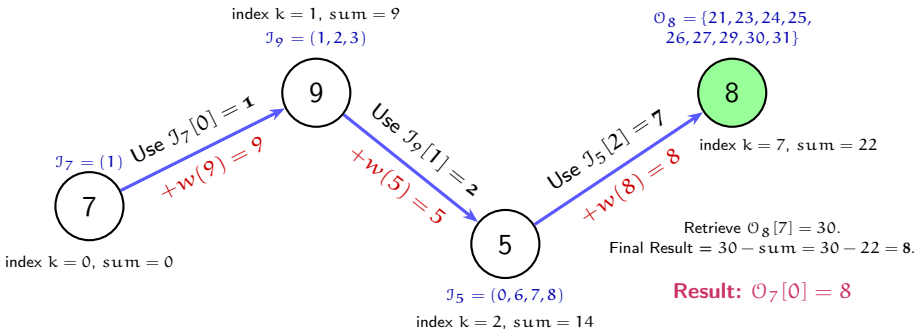


Figure 17: Visualization of the query process for retrieving the element at index $k = 0$ of $\mathcal{O}_7$ using the succinct representation. The query follows the successor path $7 \to 9 \to 5 \to 8$. At each step from an implicit node v to its successor $u = \sigma(v)$, the current index $k_{current}$ is updated using the stored offset $k_{next} = \mathcal{I}_v[k_{current}]$, 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 $\mathcal{I}_7[0] = 1$, accumulating $w(9) = 9$. Then to node 5 using index $\mathcal{I}_9[1] = 2$, accumulating $w(5) = 5$ (total sum 14). Then to node 8 using index $\mathcal{I}_5[2] = 7$, 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 $\mathcal{O}_7[0]$.

The example in Figure 17 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 (o-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 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).

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

Require: Vertex ID v, index $k \in \{0, \ldots, |\mathcal{O}_v| - 1\}$.
Ensure: The value of the k-th smallest element in $\mathcal{O}_v$.

```
current_node ← v
current_index ← k
weight_sum ← 0
while current_node ∉ V_E do
    successor ← σ(current_node)
    weight_sum ← weight_sum + W[successor]
    I_current_node ← D_I(current_node)
    next_index ← I_current_node[current_index]
    current_node ← successor
    current_index ← next_index
end while
O_explicit ← D_E(current_node)
base_value ← O_explicit[current_index]
return base_value − weight_sum
```

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

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

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 17 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 $k = 0, 1, \ldots, |\mathcal{O}_v| - 1$. 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 $|\mathcal{O}_v|$.

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

Require: Vertex ID v.
Ensure: The sorted sequence $\mathcal{O}_v$.
1: $size \leftarrow$ Length(v) ▷ Obtain the cardinality of $\mathcal{O}_v$
2: Initialize an empty list O_list of size $size$.
3: **for** k from 0 to $size - 1$ **do**
4: $value \leftarrow$ GetValue(v, k) ▷ Compute the k-th element
5: O_list$[k] \leftarrow value$ ▷ Store in list
6: **end for**
7: **return** O_list ▷ The list contains $\mathcal{O}_v$ as a sorted sequence

4.3.2 *Computing the Rank Query*

Equipped with the capability to reconstruct $\mathcal{O}_N$ for any query vertex N via Algorithm 11, we can now implement the Rank query as specified in 4.8. 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(Intervals): Merge sorted intervals

Require: A list $\mathit{Intervals} = \{[l_1, r_1], [l_2, r_2], \dots\}$ sorted by start points l_i.
Ensure: A minimal list MergedIntervals of disjoint intervals covering the same union.

```
1: Initialize an empty list MergedIntervals.
2: if Intervals is not empty then
3:     current_interval ← Intervals[0].
4:     for i from 1 to length(Intervals) − 1 do
5:         next_interval ← Intervals[i].
6:         if next_interval.l ⩽ current_interval.r + 1 then
7:             current_interval.r ← max(current_interval.r, next_interval.r)
8:         else
9:             Append current_interval to MergedIntervals.
10:            current_interval ← next_interval.
11:        end if
12:    end for
13:    Append current_interval to MergedIntervals.
14: end if
15: return MergedIntervals.
```

The overall Rank query algorithm combines $\mathcal{O}$-set reconstruction and interval merging.

Algorithm 13 $\mathrm{Rank}_G(N)$: 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: O_N ← GETOSET(N)
2: w_N ← W[N]
3: Initialize an empty list InitialIntervals.
4: for each x ∈ O_N do
5:     l ← max(0, x − w_N + 1)
6:     r ← x
7:     Append the interval [l, r] to InitialIntervals.
8: end for
9: Sort InitialIntervals based on the starting point l.
10: MergedIntervals ← MERGEINTERVALS(InitialIntervals)
11: return MergedIntervals.
```

Algorithm 13 implements the Rank query defined in 4.8. 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.

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 15 and Figure 16.

First, the algorithm requires the $\mathcal{O}$-set for node 2. It calls GetOSet(2) (Algorithm 11), which in turn relies on GetValue(2, k) (Algorithm 10) for $k = 0, 1, \ldots, |\mathcal{O}_2| - 1$. 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 $[\max(0, x - w_2 + 1), x]$:

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

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 conceptually 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 $\sigma(v) \in \{0, \ldots, n-1\}$ 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 fundamentally 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. For example, if weights are typically small, Gamma or Delta codes might be effective. If successor IDs cluster or follow certain patterns, other codes might be preferable.

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 sub-linear 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 $\mathcal{O}_v = (x_0, x_1, \ldots, x_{m-1})$. This sequence consists of non-negative integers and 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 $\mathcal{I}_v = (j_0, j_1, \ldots, j_{m-1})$. This sequence also consists of non-negative integers, but it is non-decreasing ($0 \leqslant j_0 \leqslant j_1 \leqslant \cdots \leqslant j_{m-1}$). The ordered nature of these sequences (monotonicity) makes them suitable for specialized compression techniques that outperform general-purpose integer coding. The first method is Elias-Fano encoding (Section 3.1.3), which is particularly effective for monotonic sequences. The second method is Run-Length Encoding (RLE), which is effective for sequences with long runs of consecutive values. We will only focus on RLE since we have already covered Elias-Fano encoding in **??**.

Run-Length Encoding (RLE) Representation

Run-Length Encoding (RLE) exploits consecutive values within the monotonic sequence. Consider a generic monotonic 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 $i \leqslant j < i + l - 1$. 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 of the i-th run identified in Y. Due to the maximality of runs and the monotonicity of Y, S is a strictly increasing sequence: $s_1 < s_2 < \cdots < s_p$.
- The *run lengths sequence*, $L = (l_1, l_2, \ldots, l_p)$, where $l_i \geqslant 1$ is the number of elements (length) of the i-th run. L is a sequence of positive integers, but it possesses no inherent monotonicity property.

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

COMPRESSION OF RLE COMPONENTS The space efficiency of RLE depends on effectively compressing the resulting sequences S and L.

Algorithm 14 ENCODE(Y)

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

The run starts sequence $S = (s_1, \ldots, s_p)$ is strictly monotonic. Consequently, it is an ideal candidate for Elias-Fano encoding. If S contains p values from the universe $[0 \ldots U)$, its Elias-Fano representation would require approximately $p \log_2(U/p) + O(p)$ bits.

The run lengths sequence $L = (l_1, \ldots, l_p)$ is a sequence of positive integers. Standard variable-length integer codes (Section 2.6), such as Elias Gamma or Delta codes, can be applied. In practice, we can use the compressed-intvec structure (Appendix A) to store L efficiently.

RANDOM ACCESS IN RLE Retrieving the element y_k (the element at index k in the original sequence Y) 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:

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

Directly computing the prefix sums $\sum l_j$ on demand requires iterating through L, leading to an $O(p)$ time complexity for access, which can be inefficient if p is large.

To make this process faster, one can precompute and store the sequence of prefix sums of the lengths:

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

Since $l_j \geqslant 1$ for all j, the sequence P is strictly increasing ($p_1 < p_2 < \cdots < p_p = m$). As a monotonic sequence, P itself can be compressed, for example, using Elias-Fano.

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 predecessor or successor search problem on the monotonic sequence P. Using standard binary search on P takes $O(\log p)$ time. If P is stored in a structure supporting faster searches (like certain Elias-Fano constructions), this lookup time might be further reduced. Algorithm 15 formalizes the access using prefix sums.

Algorithm 15 GET(S, L, P, k)

Require: Run starts $S = (s_1, \ldots, s_p)$, Run lengths $L = (l_1, \ldots, l_p)$, Prefix sums $P = (p_1, \ldots, p_p)$, index $k \in [0, m-1]$.
Ensure: The value y_k of the element at index k.

```
Find smallest i* ∈ {1, ..., p} such that P[i*] > k.
if i* = 1 then
    previous_cumulative_length ← 0
else
    previous_cumulative_length ← P[i* − 1]
end if
offset ← k − previous_cumulative_length
start_value ← S[i*]
return start_value + offset
```

METHOD SELECTION The choice between direct Elias-Fano encoding and RLE for representing the monotonic sequences $\mathcal{O}_v$ and $\mathcal{I}_v$ depends on the expected structure of these sequences. If long runs of consecutive integers are anticipated (e.g., due to specific weight patterns or graph structures), RLE offers potential for higher compression. If the sequences are expected to be sparse or lack significant runs, direct Elias-Fano is likely the more direct and potentially more space-efficient approach. The decision might also depend on the relative importance of compression ratio versus random access speed (considering the overhead and benefit of the prefix sum structure in RLE).

Finally, regardless of the chosen compression method (Elias-Fano or RLE) for individual sequences, representing the entire associated data

component $\mathcal{D}$ requires a mechanism to manage this collection. Conceptually, this involves concatenating the compressed representations of all sequences ($\{\mathcal{O}_v\}_{v \in V_E}$ and $\{\mathcal{I}_v\}_{v \in V_I}$) and maintaining an auxiliary index structure that maps each vertex ID v to the starting position and relevant metadata (like length or universe size, if needed) of its corresponding compressed sequence within the concatenated data. The space complexity of this auxiliary index can be considered negligible compared to the compressed data itself[2].

2 If we use a standard Rust Vec, the overhead is minimal: 64 bits for storing the length of the vector, 64 bits for storing the capacity and 64 bits for the pointer to the data.

4.5 BELOW THE ENTROPY LOWER BOUND

In the preceding sections, we introduced a succinct representation for weighted DAGs designed to efficiently support path-based queries, along with compression strategies (Section 4.4) aimed at minimizing its space footprint. To rigorously 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 amenable to 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, which relates the topological information content to the graph's density.

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

Definition 4.14 (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 (Theorem 4.8), 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 $\mathcal{H}_{\mathcal{W}}(G) = 60,824$ bits for weights and $\mathcal{H}_E(G) = 1,464,906$ bits for topology according to Theorem 4.14.

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. These estimates are derived using foundational principles applied according to the nature of the data sequence being represented.

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 $[0, u)$. Consequently,

for the specific calculations in this analysis, we employ the closely related estimate of $n\lceil \log_2(u/n) \rceil + 2n$ bits, which aligns with this theoretical upper limit while accounting for practical implementation considerations.

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 ($S(G)$)	699,779
Weights $\mathcal{W}$ (Minimal Binary)	60,824
Successors Σ (Minimal Binary)	274,015
Associated Data $\mathcal{D}$ (RLE Offsets)	364,940

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

Table ?? 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 699,779 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.

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 [31]. 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 [53], 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: $\texttt{sample\_idx} = \lfloor i/k \rfloor$. 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 [31]. 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 [31]. 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] 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.

[9] Francisco Claude, Patrick K Nicholson, and Diego Seco. "Space efficient wavelet tree construction." In: *String Processing and Information Retrieval: 18th International Symposium, SPIRE 2011, Pisa, Italy, October 17-21, 2011. Proceedings 18*. Springer. 2011, pp. 185–196.

[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. "An alphabet-friendly FM-index." In: *International Symposium on String Processing and Information Retrieval*. Springer. 2004, pp. 150–160.

[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] Rodrigo González et al. "Practical implementation of rank and select queries." In: *Poster Proc. Volume of 4th Workshop on Efficient and Experimental Algorithms (WEA)*. CTI Press and Ellinika Grammata Greece. 2005, pp. 27–38.

[20] 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).

[21] 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.

[22] Roberto Grossi and Giuseppe Ottaviano. "The wavelet trie: maintaining an indexed sequence of strings in compressed space." In: *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. PODS '12. New York, NY, USA: Association for Computing Machinery, 2012, pp. 203–214.

[23] 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.

[24] Roberto Grossi et al. *More Haste, Less Waste: Lowering the Redundancy in Fully Indexable Dictionaries*. 2009.

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

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

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

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

[29] 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.

[30] 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.

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

[32] 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).

[33] 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.

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

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

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

[37] 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.

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

[39] 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.

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

[41] 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.

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

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

[44] 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.

[45] 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.

[46] 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.

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

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

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

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

[51] German Tischler. "On wavelet tree construction." In: *Annual Symposium on Combinatorial Pattern Matching*. Springer. 2011, pp. 208–218.

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

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

[54] 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.

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