EFFICIENT SUCCINCT DATA STRUCTURES ON DIRECTED ACYCLIC GRAPHS

LUCA LOMBARDO

Tesi Triennale

Dipartimento di Matematica
Università di Pisa

Supervisor: Roberto Grossi

ABSTRACT

In this thesis, we introduce a novel approach to constructing succinct data structures for Directed Acyclic Graphs (DAGs), with particular emphasis on optimizing query performance. Our research centers on redefining the conventional rank query, typically employed in bitvector contexts, for DAGs. We focus on a specialized class of DAGs where to each node is associated a string composed of characters drawn from a fixed alphabet, Σ. We begin by partitioning the original DAG into $|\Sigma|$ distinct DAGs, each corresponding to a character in the alphabet. Within these character-specific DAGs, nodes store the count of occurrences of the respective character in the string associated with each node. We then propose a succinct representation of these DAGs, specifically engineered to solve rank queries. Our proposed rank query takes a node and a character as input and efficiently returns a range $\langle l, r \rangle$. This range represents the minimum and maximum possible occurrences of the character along all paths originating from the root node to the specified node within the DAG.

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 depends 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.

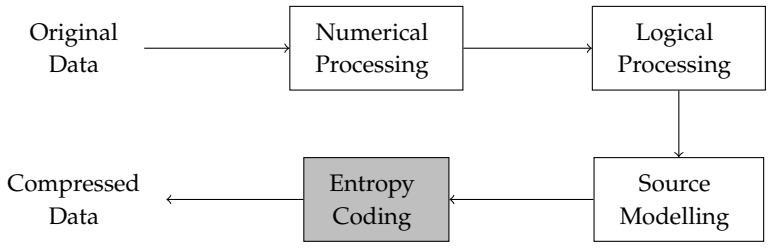

Figure 1: Typical processes in data compression

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.

WORST CASE ENTROPY

In its simplest form, entropy can be seen as the minimum number of bits required by identifiers (*codes*, see Section 2.2), 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 [5] 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.

2.1 ENTROPY

Let's introduce the concept of entropy as a measure of uncertainty of a random variable. A deeper explanation can be found in [22, 32, 9]

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 [41]

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.

2.1.1 *Properties*

In the previous section 2.1, 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.

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. Given

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

two random variables X and Y, we can define $W(y|x)$, with $x \in \mathcal{X}$ and $y \in \mathcal{Y}$, as the conditional probability of Y given X. The set W of those conditional probabilities is called *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*

$$H(Y|X) = H(W|P_X) \stackrel{\text{def}}{=} \sum_x P_X(x) H(Y|x) \tag{4}$$

$$= \sum_{x \in \mathcal{X}} P_X(x) \Big\{ - \sum_{y \in \mathcal{Y}} W(y|x) \log W(y|x) \Big\} \tag{5}$$

$$= - \sum_{x \in \mathcal{X}} \sum_{y \in \mathcal{Y}} P_{XY}(x, y) \log W(y|x) \tag{6}$$

$$= E_{P_{XY}}\{ - \log W(Y|X) \} \tag{7}$$

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. □

2.1.2 *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.

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, $I(X; Y) = I(Y; X)$.

2.1.3 *Fano's inequality*

Information theory serves as a cornerstone for understanding fundamental limits in data compression. It not only allows us to prove the

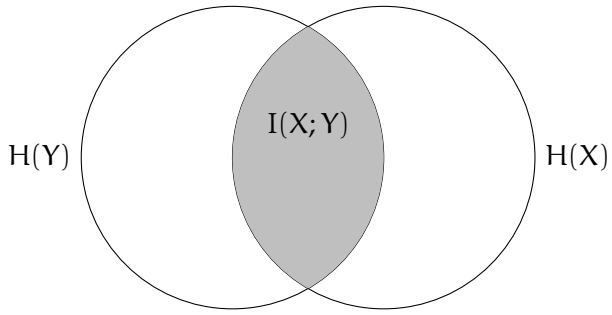

Figure 2: Mutual information between two random variables X and Y.

existence of encoders (Section 2.2) 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

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

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 $H(X|Y) = 0$.

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. □

Add some conclusions to this section.

2.2 SOURCE AND CODE

Some introduction about the source and coding, maybe with some very simple example like the morse code that uses a single dot to represent the most common symbol.

2.2.1 *Codes*

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

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

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

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

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

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

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

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

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.

Example 2.16 (Morse Code).

TODO from [9]

Definition 2.17 (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. [9]

Definition 2.18 (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 2.19. *If $C(x_1) = 0$ and $C(x_2) = 110$, then $C^*(x_1x_2) = 0110$.*

Definition 2.20 (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 2.21 (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. [9]. Figure 3 shows the relationship between different types of codes.

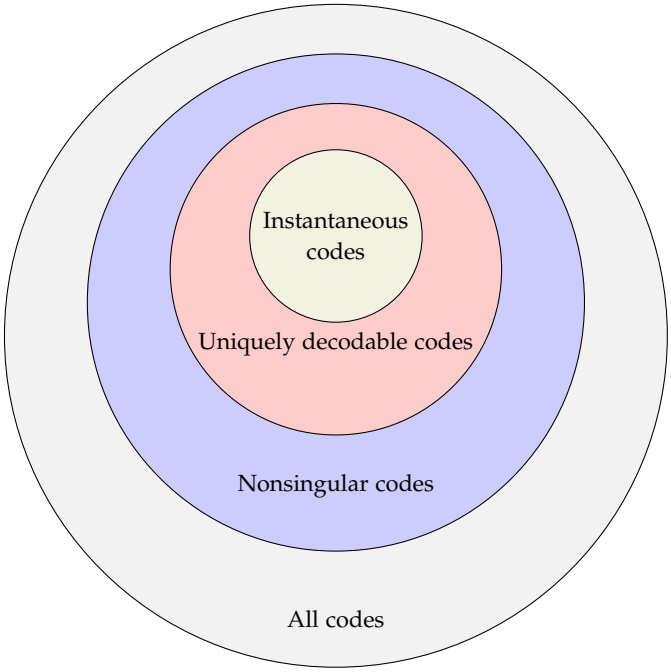

Figure 3: Relationship between different types of codes

2.2.2 *Kraft's Inequality*

We would like to construct instantaneous codes that are optimal in the sense that the expected length of the code is equal to the entropy of the random variable. However, we can't assign short codewords to all symbols and hope to be still prefix-free. Kraft's inequality provides a necessary and sufficient condition for the existence of a prefix code with given codeword lengths.

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 [9, 22], here we report the one from [22], however the one proposed in [9] is also very interesting, based on the concept of a source tree.

Theorem 2.22 (Kraft's Inequality). *The codeword length* $l(x)$, $x \in \mathcal{X}$, *of any separable code* C *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. □

2.2.3 *Source Coding Theorem*

Add some introduction from [9, 41, 1, 22]

Theorem 2.23 (Source Coding Theorem). *TODO from [9, 22]*

Proof. TODO from [9, 22] □

2.3 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 an alphabet $\Sigma = [0, \sigma]$ where each symbol has a probability p_s, the entropy of the source becomes:

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

Remark 2.24. *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* $n\mathcal{H}(p_1, \ldots, p_\sigma)$

2.3.1 *Bit 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 2.25 (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

TBD if to add this paragraph, from [32] 2.3.1

2.3.2 *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 2.26 (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 2.27. *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 2.4) 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 2.6).

2.4 HIGHER ORDER ENTROPY

TODO: A bit of introduction

Definition 2.28 (Redundacy)**.**

TODO: Give a formal definition of redundacy: informally is a measure of the distance between the source's entropy and the compression ration, and can thereby be seen as a measure of how fast the algorithm reaches the entropy of the source.

While using measures like 2.28 are certainly intriguing, their actual usability is questionable due to the inherent challenge of determining the entropy of the source generating the string we aim to compress. To address this issue, an alternative empirical approach is the concept of the *k-th order empirical entropy* of a string S, denoted as $\mathcal{H}_k(S)$. In statistical coding (Section 2.6), 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. [11]

Let S be a string over the alphabet $\Sigma = \{\sigma_1, \ldots, \sigma_n\}$. Denote with n_ω the number of occurrences of the k-gram ω in S. [3]

Definition 2.29 (k-th Order Empirical Entropy)**.** *The* k-th order empirical entropy *of a string* S *is defined as*

$$\mathcal{H}_k(S) = \frac{1}{|S|} \sum_{\omega \in \Sigma^k} \left(\sum_{i=1}^{h} n_{\omega\sigma_i} \log \left(\frac{n_\omega}{n_{\omega\sigma_i}} \right) \right) \tag{1}$$

where $|S|$ *is the length of the string* S*.*

When considering a sequence $S[1, n]$ we can compute the *empirical k-th entropy* of S by considering the frequencies of symbols depending on the k preceding symbols.

$$\mathcal{H}_k(S) = \sum_{\omega \in \Sigma^k} \frac{|S_\omega|}{n} \cdot \mathcal{H}_\prime(S_\omega) \tag{2}$$

where S_ω is a string formed by collecting the symbol that follows each occurrence of the k-gram $\omega = \sigma_1 \ldots \sigma_k$ in S.

Example 2.30. *Consider the example 2.27, where* $S =$ *"abracadabra" and* $\Sigma = \{a, b, c, d, r\}$*. The zero-order empirical entropy of* S *is* $\mathcal{H}_0(S) \approx 2.04$*. Now, let's calculate the first-order empirical entropy of* S*. We have that*

3 We will use the notation $\omega \in \Sigma^k$ to denote a k-gram, i.e., a subsequence of k symbols in the string S.

S_a = *"bcdb$" (where $ is the end-of-string symbol),* S_b = *"rr",* S_c = *"a",* S_d = *"a", and* S_r = *"aa". Thus,* $H_0(S_a) \approx 1.922$, $H_0(S_b) = H_0(S_c) = H_0(S_d) = H_0(S_r) = 0$. *Therefore, the first-order empirical entropy of S is:*

$$\mathcal{H}_1(S) = \frac{5}{11} \cdot \mathcal{H}_0(S_a) \approx 0.874$$

That is much lower than the zero-order empirical entropy of S.

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 on itself and the k symbols preceding it in S. Consistently, any compressor that surpasses this threshold would also have the capability to compress symbols originating from the related kth-order source to a level lower than its Shannon entropy.

Remark 2.31. *As k grows large (up to $k = n - 1$, and often sooner), the k-th order empirical entropy of S reaches null, given that each k-gram appears only once. This renders our model ineffective as a lower bound for compressors. Even before reaching the k value where $\mathcal{H}_k(S) = 0$, compressors face practical difficulties in achieving the target of $n\mathcal{H}_k(S)$ bits, particularly for high k values. This is due to the necessity of storing the set of σ^{k+1} probabilities or codes, adding complexity to compression. Likewise, adaptive compressors must incorporate σ^{k+1} escape symbols into the compressed file, further complicating the process. In theory, it is commonly assumed that S can be compressed up to $n\mathcal{H}k(S) + o(n)$ bits for any $k + 1 \leqslant \alpha \log \sigma n$ and any constant $0 < \alpha < 1$. In such cases, storing σ^{k+1} numbers within the range $[1, n]$ (such as the frequencies of the k-grams) requires $\sigma^{k+1} \log n \leqslant n^{\alpha} \log n = o(n)$ bits. [32]*

Definition 2.32 (Coarsely Optimal Compression Algorithm)**.** *A compression algorithm is coarsely optimal if, for every value of k, there exists a function $f_k(n)$ that tends to zero as the length of the sequence n approaches infinity, such that for all sequences S of increasing length, the compression ratio achieved by the algorithm remains within $\mathcal{H}_k(S) + f_k(|S|)$.*

The *Lempel-Ziv* algorithm (LZ78) serves as an example of a coarsely optimal compression technique, as outlined by Plotnik et al. in [37]. This algorithm relies on the idea of dictionary-based compression. However, as highlighted by Manzini and Korařaju [27], the notion of coarse optimality doesn't necessarily guarantee the effectiveness of an algorithm. Even when the entropy of the string is extremely low, the algorithm might still perform inadequately due to the presence of the supplementary term $f_k(|S|)$.

FURTHER COMMENTS ON LZ77 AND LZ78

TBD if to include this section, but I think it's not relevant for the thesis. If included, it should discuss very briefly the LZ77 and LZ78 algorithms, and the differences between them. [11]. And then prove two lemmas: one about the compression ration achieved by LZ78 and the other about LZ77 not being coarsely optimal. [11], end of chapter 13

2.5 INTEGER CODING

TODO: Introduce the following problem: given $S = \{x_1, x_2, \ldots, x_n\}$, where $x_i \in \mathbb{N}$, we want to represent the integers of S as a sequence of bits that are self-delimiting. The goal is to minimize the space occupancy of the representation [11]. Add here some examples of where this problem appears in practice [43]

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.

2.5.1 *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. The correctness of this encoding is straightforward to verify: the decoder can identify the end of the integer by detecting the first one in the sequence, and the number of zeros preceding it determines the value of x.

This coding method requires x bits to represent the integer x, which is way more than the $\lceil \log_2(x) \rceil$ bits needed by a fixed-length binary code. In fact, it is very efficient for small values of x but becomes increasingly inefficient as x grows. This is a direct consequence of Theorem 2.23, which states that the ideal code length $L(c)$ for a symbol c is $-\log_2 P(c)$, where $P(c)$ is the probability of symbol c. In the case of the unary code, where we are considering positive integers, the ideal code for x would be $-\log_2 P(x) = -\log_2 2^{-x} = x$ bits. The following theorem formalizes this observation. [11]

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

It is important to note that implementing a unary code requires a lot of bit shifts and bitwise operations, which are computationally expensive on modern processors. This makes the unary code impractical for large values of x

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

2.5.2 *Elias Codes*

First introduced by Levenstein in the 1960s and later refined by Elias [10] in the 1970s, the γ and δ codes are two of the most popular *universal codes* for integers. The term *universal code* refers to the characteristic of these codes to have fixed-length of $O(\log x)$ for any integer x. Compared to the binary code that requires $\lceil \log_2(x+1) \rceil$ bits, the γ and δ codes are just a constant factor away from it, while having the advantage of being prefix-free.

GAMMA CODE The γ code represents a positive integer x is divided into two parts: given $|B(x)|$ as the number of bits needed to represent x in binary, the first part is a sequence of $|B(x)|-1$ zeros followed by the binary representation of x. The γ code of x is then the concatenation of these two parts, delimited by the first one bit. The decoding process is therefore very simple: the decoder reads the bits until it finds the first one, and the number of zeros preceding it determines the length of the binary representation of x. From Shannon's condition of ideal codes (Theorem 2.23), we can see that the γ code is optimal for the distribution $P(x) \approx 1/x^2$. [11]

Theorem 2.34. *The γ code of a positive integer x takes $2\lceil \log_2(x+1) \rceil - 1$ bits, and thus it is optimal for the distribution $P(x) = 1/x^2$. This is within a factor of two from the bit length $|B(x)| = \lceil \log_2(x) \rceil$ of the fixed-length binary code.*

The γ code is inefficient due to the large number of zeros that need to be stored in the prefix, that becomes increasingly large as x grows. The δ code, introduced by Elias in 1975, addresses this issue by using a more efficient prefix.

DELTA CODE The δ code is a variation of the γ code that uses a more efficient prefix. It represents a positive integer x by first encoding the binary length of x using the γ code (we can write it as $\gamma(|B(x)|)$) and then appending the binary representation of x itself. The δ code is thus the concatenation of these two parts that do not share any bits. The decoding process is similar to the γ code: the decoder reads the bits until it finds the first one, and the number of zeros preceding it determines the length of the binary representation of x, then we fetch the next $|B(x)|$ bits to get the binary representation of x. This code takes $|\gamma(|B(x)|)| + |B(x)| = 2\lceil \log_2(|B(x)|+1) \rceil - 1 + |B(x)| \approx 2\log\log x + 1 + \log x$ bits, which is $1 + o(1)$ factor away from the bit length of the fixed-length binary code. [11]

Theorem 2.35. *The δ code of $x \geqslant 0$ takes $1 + \log_2 x + 2\log_2\log_2 x$ bits, and thus it is optimal for the distribution $P(x) \approx 1/x(\log x)^2$. This is within a factor of $1 + o(1)$ from the bit length $|B(x)| = \lceil \log_2(x) \rceil$ of the fixed-length binary code. [11]*

As for the Unary Code, implementing these codes requires a lot of bit shifts during the decoding process, making them impractical for large values of x.

2.5.3 *Rice Code*

Rice codes [39] are a family of codes parameterized by a positive integer k. Their representation is the concatenation of $(1 + x/2^k)$ as a unary code and the binary representation of the integer $(x \mod 2^k)$. Rice codes are very efficient when the values iof x are close to 2^k. When dealing with large values of x, the efficiency of the γ and δ codes decreases, and Rice codes become a better alternative providing fast compression and decompression.

The fist part takes $1 + \lceil \log_2(x/2^k) \rceil$ bits, and the second part takes k bits (since it's in the range $[0, 2^k)$). Thus, the first part is encoded in variable-length unary code, and the second part is encoded in fixed-length binary code. The closer the values of x are to 2^k, the shorter the first part becomes, making the decoding process faster.

2.5.4 *Elias-Fano Code*

Add a detailed section about Elias Fano since it will be crucial in the implementation of the succinct DAG [11]

.

2.6 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. In the coding phase, this model is utilized to generate codewords for the symbols of Σ, which are then employed to compress the input sequence. We will focus on two popular statistical coding methods: Huffman coding and Arithmetic coding.

2.6.1 *Huffman Coding*

Compared to the methods seen in Section 2.5, Huffman Codes (introduced by Huffman in his landmark paper [23] in the 1950s) offer a broader applicability as they do not require any specific assumptions about the probability distribution, only that all probabilities are non-zero. This versatility makes them suitable for all distributions, including those where there is no clear relationship between symbol number and probability, such as in text data.

For example, in text, characters typically range from "*a*" to "*z*" and are often mapped to a contiguous range, such as 0 to 25 or 97 to 122 in ASCII. However, there is no direct correlation between a symbol's number and its frequency rank.

CONSTRUCTION OF HUFFMAN CODES The construction of Huffman codes is a greedy algorithm based on the idea of building a binary tree, where each leaf corresponds to a symbol in the alphabet Σ. The tree is built in a bottom-up fashion, starting with the symbols as leaves (we define their *size* as the number of occurrences) and iteratively merging the two nodes with the smallest probabilities until a single node is left. The code for each symbol is then obtained by traversing the tree from the root to the leaf, assigning a 0 for each left branch and a 1 for each right branch (or vice-versa). The resulting code is the path from the root to the leaf. More details can be found in [11, 40, 22, 9]

Example 2.36. *TODO: Classic example of Huffman coding with for example* $\mathcal{X} = \{a, b, c, d, e\}$ *and* $P(a) = 0.25$, $P(b) = 0.25$, $P(c) = 0.2$, $P(d) = 0.15$, $P(e) = 0.15$. *Do a nice tree and show the encoding of each symbol.*

Let $L_C = \Sigma_{\sigma \in \Sigma} L(\sigma) \cdot P[\sigma]$ be the average length of the codewords produced by a prefix-free code C, that encodes every symbol $\sigma \in \Sigma$ with a codeword of length $L(\sigma)$. The Huffman coding produces optimal prefix codes (not in the sense that produces an optimal encoding, but

in the sense that no prefix code can have a smaller average length). This is formalized in the following theorem.

Theorem 2.37 (Optimality of Huffman Codes)**.** *Let* C *be an Huffman Code and* L_C *is the shortest possible average length among all prefix-free codes* C'*. That is,* $L_C \leqslant L_{C'}$

This can also be interpreted as the *the minimality of the average depth* of the Huffman tree. A proof can be found in most information theory books [11, 40, 22, 9].

In the worst case, an Huffman Code can have a length of $|\Sigma| - 1$ bits, which is the same as the number of internal nodes in the tree. However, its length is limited also by $\lfloor \log_\Phi \frac{1}{p_{min}} \rfloor$, where p_{min} is the smallest probability in the set and Φ is the golden ratio. [32]. Thus, if the probabilities come from the observed frequencies of the symbols in the text, let's say n symbols, then $p_{min} \geqslant \frac{1}{n}$ and the maximum length of the code is $\log_\Phi n$. In particular, the encoding process is linear in the size of the input text [5].

The decoding process uses the Huffman Tree. It starts by reading consecutive bits from the stream and traversing the tree from the root towards a leaf based on the read bits. Upon reaching a leaf, we output the symbol it represents and then reset back to the root of the tree. Consequently, the overall decoding duration scales proportionally with the length of the compressed sequence in bits, denoted as $O(n(H(Pr) + 1))$. Since the codes are of length $O(\log n)$, it follows that any symbol can be decoded within $O(\log n)$ time.

Theorem 2.38. *Let* H *be the entropy of a source emitting the symbols of an alphabet* Σ*, hence* $H = \sum_{\sigma \in \Sigma} P(\sigma) \log_2 \left(\frac{1}{P(\sigma)} \right)$*. Then, the average length of the Huffman code is bounded by* $H < L_H < H + 1$*, where* L_H *is the average length of the Huffman code.*

Proof. The first inequality comes from Shannon's source coding theorem (Theorem 2.23). Let's define $l_\sigma = \lceil -\log_2 P(\sigma) \rceil$ as the length of the code for symbol σ, which is the smallest integer such upper bounding Shannon's optimal codeword length. We can easily derive that $\sum_{\sigma \in \Sigma} 2^{-l_\sigma} \leqslant 1$. Thus, recalling Kraft's inequality (Theorem 2.22), we have that exists a binary tree with $|\Sigma|$ leaves and depths l_σ for each leaf. This tree is a prefix code, and its average codeword length is $L_C = \sum_{\sigma \in \Sigma} P(\sigma) \cdot l_\sigma$. By optimality of the Huffman code (2.37), we have that $L_H \leqslant L_C$; thus from the definition of entropy H and from the inequality $l_\sigma < 1 + \log_2 \left(\frac{1}{P(\sigma)} \right)$, we have that $H < L_H < H + 1$. □

TBD: Do I talk about canonical Huffman codes? Do I talk more about the optimality of Huffman codes?

5 In the RAM model, $O(\log n)$ bits can be manipulated in $O(1)$, so the this is true also in practice

2.6.2 *Arithmetic Coding*

Introduced by Elias in the 1960s and then refined by Rissanen and Pasco in the 1970s [35]. Arithmetic coding is a more general technique than Huffman coding, as it can achieve a better compression ratio (it can code symbols arbitrary close to 0-th order entropy) by encoding a sequence of symbols as a single number in the interval $[0, 1)$. This number is then converted into a binary representation.

Consider any prefix coder, like Huffman coding. If we apply Huffman coding to a sequence of symbols, it must use *at least one bit* (or more generally, an integer number of bits) per symbol, that is more then ten times the entropy of the source. This makes any prefix coder far from the 0-th order entropy of the source. From the definition of Huffman Coding we can easily derive that it can be optimal only if $-\log p$ is a natural number, thus if and only if $p = 2^{-k}$ for some $k \in \mathbb{N}$. Arithmetic coding relaxes the request to define a prefix-free code for each symbol, and instead defines a strategy in which every bit of the output can be used to represent more than one symbol.

Encoding and Decoding Process

Let $S[1, n]$ be a sequence of symbols drawn from an alphabet Σ and $P(\sigma)$ be the probability of symbol $\sigma \in \Sigma$. The encoding process of arithmetic coding is described in algorithm 1. The algorithm is an iterative process: it starts by initializing the interval $[0, 1)$ and then iterates over the symbols of the input sequence, splitting for each symbol the interval into a sub-interval with length proportional to the probability of the symbol. We then replace the current interval with the sub-interval corresponding to the next symbol and continue until all symbols are processed.

Algorithm 1 Arithmetic Coding

Require: $S[1, n]$, $P(\sigma)$ for each $\sigma \in \Sigma$
Ensure: A sub-interval $[l, l + s)$ of $[0, 1)$
  Compute the cumulative probabilities $C(\sigma) = \sum_{\sigma' \in \Sigma : \sigma' \leqslant \sigma} P(\sigma')$
  $s_0 = 1$, $l_0 = 0$, $i = 1$
  while $i \leqslant n$ **do**
    $s_i = s_{i-1} \cdot P(S[i])$
    $l_i = l_{i-1} + s_{i-1} \cdot C(S[i])$
    $i = i + 1$
  end while
  return $x \in [l_n, l_n + s_n), n$

At the end, the algorithm doesn't output a sub-interval, but a single number x in the interval $[l_n, l_n + s_n)$ and the length of the input sequence; this number has to be a dyadic fraction, that is a fraction

with a denominator that is a power of 2. The encoding sequence is then given by the numerator of x in its binary representation, using k bits, where k is the power of 2 that is the denominator of x. The details on how to choose a dyadic number in the interval $[l_n, l_n + s_n)$ that can be encoded in a few bits can be found in [11, 22, 40].
The decoding process shown at 2. Since the same statistical model is used by the encoder and decoder, the decoder can reconstruct the original sequence by following the same steps as the encoder.

Algorithm 2 Arithmetic Decoding

Require: The binary sequence $b[1, k]$ representing the compressed output, $P(\sigma)$ for each $\sigma \in \Sigma$, n
Ensure: The sequence $S[1, n]$
  Compute the cumulative probabilities $C(\sigma) = \sum_{\sigma' \in \Sigma : \sigma' \leqslant \sigma} P(\sigma')$
  $s_0 = 1$, $l_0 = 0$, $i = 1$
  while $i \leqslant n$ **do**
    Split the interval $[l_{i-1}, l_{i-1} + s_{i-1})$ into $|\Sigma|$ sub-intervals
    Find the σ corresponding to the sub-interval containing x
    $S = S.append(\sigma)$
    $s_i = s_{i-1} \cdot P(\sigma)$
    $l_i = l_{i-1} + s_{i-1} \cdot C(\sigma)$
    $i = i + 1$
  end while
  return S

Efficiency of Arithmetic Coding

It's easy to derive that if we choose the probabilities of the symbols to be the empirical frequencies of the symbols in the input sequence, denoting with $p_s = \frac{n_s}{n}$ the probability of symbol s in the sequence, then the size of the final interval will be

$$s_n = \prod_{i=1}^{n} p_{S[i]} = \frac{n_{S[1]} \cdot n_{S[2]} \cdots n_{S[n]}}{n^n} = \prod_{s \in \Sigma} \left(\frac{n_s}{n}\right)^{n_s} \tag{1}$$

What makes this formula intriguing is its independence from the sequence's symbol order. Instead, it's solely determined by the frequency of each symbol in the sequence. Consequently, the output remains unchanged regardless of any permutations in the input sequence. We can now observe that any interval of size γ accommodates a dyadic number where the denominator's power equals $-\log\gamma + 1$. Hence, the encoder requires no more than this number of bits.

$$1 - \log \prod_{s \in \Sigma} \left(\frac{n_s}{n}\right)^{n_s} = 1 + \sum_{s \in \Sigma} n_s \log \frac{n}{n_s} \tag{2}$$

That is just one bit far from the empirical entropy of the sequence. In can also be proved [11, 22, 40] that the number of bits emitted by arithmetic coding for a sequence S of n symbols is at most $2 + n\mathcal{H}$, where $\mathcal{H}$ is the empirical entropy of the sequence S.

Theorem 2.39. *The number of bits emitted by arithmetic coding for a sequence S of n symbols is at most $2 + n\mathcal{H}$, where $\mathcal{H}$ is the empirical entropy of the sequence S.*

TBD: Do I add the proof or can I just reference the books?

Remark 2.40. *In practical implementations, probabilities are often rounded to a small value to avoid dealing with arbitrary precision floating-point numbers. The arithmetic coder is periodically reset, writing bits to the output stream when the interval becomes sufficiently small and rescaling it by the corresponding power of two. This enables the use of arithmetic coding with integer arithmetic. However, even with a small interval, fully determining the next bits to be written isn't always possible. For further details on practical implementation, refer to [31]*

3 RANK AND SELECT

Change this introduction, I don't like it

In the previous chapter, we introduced various methods for compressing data. Now we introduce the concept of compressed data structures: data structures that store data in a compressed form, allowing for efficient access to the data. This will lead to the introduction of the so called *pointer-less programming*, where we ditch the traditional pointers and instead use compressed data structures built upon binary arrays that allow for efficient access to the data. In this chapter, we will introduce the concept of bitvectors, and show how they can be compressed to support rank and select queries in constant time. We will then introduce wavelet trees, a data structure that generalizes the concept of bitvectors to support rank and select queries on arbitrary alphabets.

3.1 BITVECTORS

Maybe call this section "RRR: A succint data structure for rank and select" or something like that

Consider the following problem [11]: 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 [38] 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 $\texttt{rank}_1(i)$ 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 4 shows a visual representation of the RRR data structure.

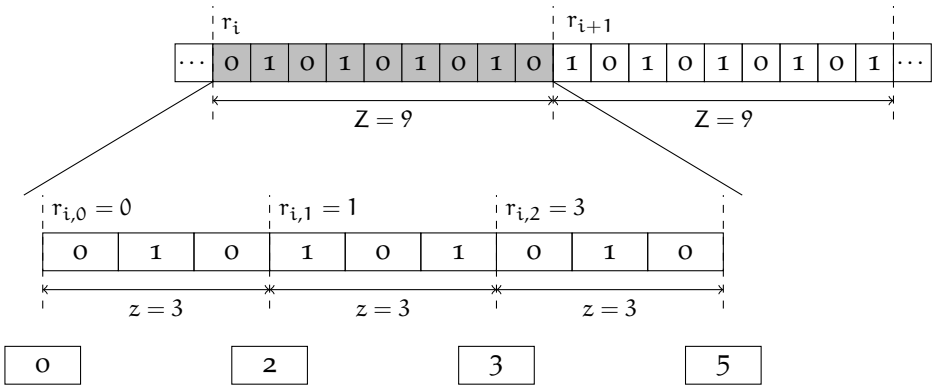

Figure 4: The RRR Rank data structure. The first level is composed of blocks of size Z, the second level of blocks of size z, and the third level is an entry of the lookup table.

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.

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

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.

We can now state the following theorem [11]:

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 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.

Remark 3.5. *If z (the size of the small blocks) can be stored in a single memory word, the* `popcount` *operation can be executed efficiently using bit manipulation operations like* `std::popcount` *in C++[5]. This approach ensures constant time execution, especially when z occupies only a few memory words, allowing for the utilization of SIMD (single instruction, multiple data) operations for faster performance. [11]*

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 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[6], 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

TBD: Do I talk about [21]?

Remark 3.6 (Pratical Considerations). *Optimizing memory access can*

Improve this, I think there are errors

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

5 https://en.cppreference.com/w/cpp/numeric/popcount

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

significantly boost performance: thus aligning block and superblock lengths to word or byte boundaries is crucial. For instance, setting the superblock length (Z) to 256 allows storing the second level as an array of $\lceil n/z \rceil$ bytes, resulting in a total space usage of $1.375 \cdot n$ for both $z = 32$ and $z = 64$. To further minimize space overhead, introducing another level with super-superblocks of length 2^{16} can bring down the total space usage to below $1.313 \cdot n$ for $z = 32$ and $1.189 \cdot n$ for $z = 64$. To reduce cache misses, interleaving the secondo and the third level ensures that the accessed entries from both arrays fit in the same cache line, enhancing performance. For example, with $z = 64$, choosing $n/Z = 8$ allows storing two w-bit words per superblock, minimizing space usage to $1.250 \cdot n$. [32]

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:

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

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 3.7 (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, 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[7] 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[8].

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[9]. 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[10].

Remark 3.8 (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*.*

In 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.

7 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

8 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.

9 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.

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

Algorithm 3 $\mathtt{Select}_1$ 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 3.9. *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. □

TBD: Do I talk also about how to compress via Elias Fano? When the numbers of 1s in B is much smaller than n, where I can achieve $nH_0(B) + O(m)$ bits of space occupancy. This approach poses a much lower overhead over the entropy. It supports Select in constant time, while access and Rank in $O(\log \frac{n}{m})$ [32, 11]. On a further note, in [14] Ferragina and Venturini achieved higher order compression for general strings, supporting access in constant time (so adding rank and select for bitvectors are natural extensions), do I talk about this as well?

3.2 WAVELET TREES

Improve this introduction, just a draft

Wavelet trees, introduced in 2003 by Grossi, Gupta, and Vitter [17] are a self indexing data structure: meaning they can answer rank and select queries, while still allowing to access the text. This combination makes them particularly useful for compressed full-text indexes like the FM-index [13]. In such indexes, wavelet trees are employed to efficiently answer rank queries during the search process.

Upon closer examination, one can recognize that the wavelet tree is a slight extension of an older (1988) data structure by Chazelle [7], commonly used in Computational Geometry. This structure represents points on a two-dimensional grid, undergoing a reshuffling process to sort them by one coordinate and then by the other. Kärkkäinen (1999) [26] was the first to apply this structure to text indexing, although the concept and usage differed from Grossi et al.'s proposal four years later

Wavelet Trees can be seen in different ways: (i) as sequence representation, (ii) as a permutation of elements, and (iii) as grid point representation. Since 2003, these perspectives and their interconnections have proven valuable across diverse problem domains, extending beyond text indexing and computational geometry, where the structure originated [33, 20, 12].

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[11]. 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.

However, dealing with sequences is much more complex than dealing with bitvectors (as we have seen in Section 3.1). In [32] 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

11 The size of the alphabet varies depending on the application. For example, in DNA sequences, the alphabet is $\Sigma = \{A, C, G, T\}$ (in **??** we will focus more on this specific case), while in other case it could be of millions of characters, such as in natural language processing.

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

$$\mathrm{rank}_c(S, i) = \mathrm{rank}_1(B_c, i)$$
$$\mathrm{select}_c(S, j) = \mathrm{select}_1(B_c, j)$$

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.[12]

Remark 3.10 (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[13]. 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 $[\lfloor (a + b)/2 \rfloor + 1, \dots, b]$. [33]

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 are outlined in Algorithm 4. Excluding the sequence S and the final wavelet tree T, the algorithm uses $n \log \sigma$ bits of space [14].

12 Even if we use a compressed representation of the bitvectors, the space occupancy would still have the dominant term $n\sigma$, that is at least $\Omega(n\sigma \log \log n / \log n)$ bits if we still want to support constant time rank and select queries.

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

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

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 3.11. *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 in then upper bounded by $n\lceil \log \sigma \rceil$ bits. [33]. However, if we also interested in storing the topology of the wavelet tree, then another $O(\sigma \log n)$, that can be critical for large alphabets. In [8, 42] are presented some techniques to build wavelet tree in a space-efficient way.*

Example 3.12 (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 5

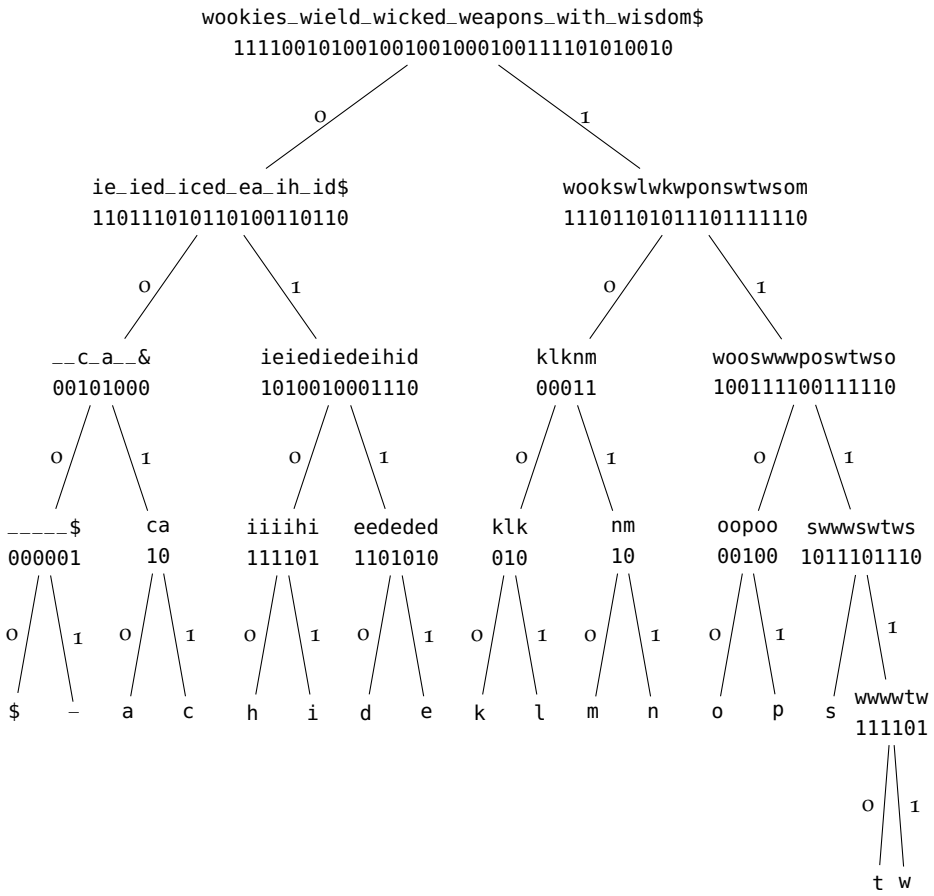

Figure 5: Wavelet tree for the sequence `wookies_wield_...`

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 $\texttt{rank}_0$ and $\texttt{rank}_1$ 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 o 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 [38] 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 7.

TBD if to add: In 3.11 we mentioned that storing the topology of the wavelet tree requires $O(\sigma \log n)$ bits. This may be critical for large alphabets, and in this section we will show that this term can be removed by slightly altering the balanced wavelet tree shape. [30, 29].

Look for section 2.3 of [33]

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 [17] 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 [32] 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 [38] 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 [36] the authors showed[15] 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 [18], 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

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

Tree in 2.6.1, 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

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

A counter effect noted by the authors in [18] 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[16] 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 3.13. *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 [28], 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 [19] Grossi and Ottaviano gave the wavelet the shape of a trie, making it possible to handle a sequence of strings.

16 And also for $\mathrm{rank}_c(S, j)$ or $\mathrm{select}_c(S, j)$ with $c = S = [1]$

3.2.2.3 *Higher Order Entropy Coding*

TODO and TBD: Ask Grossi how in depth to go with this section, Ferragina and Manzini in [12] give a very technical explanation of the method. While in [33] Navarro gives a more high-level explanation. Furthermore, all this methods relies on the BTW transform, which is not covered in this thesis, do I add a section on it?

3.3 DEGENERATE STRINGS

Given a finite non-empty alphabet Σ, a *string* X of length N over Σ is a sequence of N symbols from Σ. We denote with Σ^* the sets of all the strings in Σ, including the trivial one ϵ on length 0. We can now introduce the concept of a *degenerate string*, presented for the first time by Fischer and Paterson in 1974 [16] and has been used in various contexts since then [4].

Definition 3.14 (Degenerate String)**.** *A* degenerate string *is a sequence* $X = X_1 X_2 \dots X_n$*, where each* X_i *is an element of* Σ^**. We call* n *the* length *of* X *and* $N = \sum_{i=1}^{n} |X_i|$ *the* size *of* X*.*

Definition 3.15 (Balanced Degenerate String)**.** *TODO*

TODO: Add an example of a degenerate string and add a few more lines to clarify the concept. Talk about their application in bioinformatics and why the literature is interested in them.

3.3.1 *Subset-Rank and Subset-Select*

We have seen in depth the rank and select operations in Section 3.1. Where given a string S from an alphabet $[1, \sigma]$, we showed how to answer the following queries efficiently:

- $\texttt{rank}_S(c, i)$: the number of occurrences of the symbol c in $S[1..i]$.
- $\texttt{select}_S(c, i)$: the position of the i-th occurrence of the symbol c in S.

We can now extend these operations to degenerate strings. This problem was recently studied for the first time by Alanko, Biagi, Puglisi and Vuohtoniemi in [3], where their goal was to support the following queries on degenerate strings:

- $\texttt{subset-rank}_X(c, i)$: the number of sets in $X[1..i] = X_1 \dots X_i$ that contain the symbol c.
- $\texttt{subset-select}_X(c, i)$: the position of the i-th set that contains the symbol c

Example 3.16. *Let's consider the following degenerate string over the alphabet* $\Sigma = \{A, B, C, D\}$*:*

$$X = \{AAB\}\{CD\}\{\}\{A\}\{BCD\}\{\}\{C\}\{AB\}\{\}\{D\}$$

Then for example we would have

$$\texttt{subset-rank}_X(C, 6) = 2 \qquad \texttt{subset-select}_X(A, 2) = 3$$

since the symbol C *appears in order in the sets* $\{CD\}$ *and* $\{BCD\}$ *and the second set containing the symbol* A *is* $\{A\}$ *ad index* 3*.*

This type of queries are crucial for solving various problems encountered in pangenomics, the study of entire genomes across a species. In the context of de Bruijn graphs, which can be used to represent relationships between overlapping substrings in biological sequences, researchers in [3] aimed to enable efficient membership queries.

Building upon this work, in [2] they introduced the *Spectral Burrows-Wheeler Transform* (SBWT). This transform represents a string's k spectrum (the collection of all k-length substrings) as a sequence of alphabet subsets, i.e a degenerate string. The authors demonstrated that the SBWT allows for efficient de Bruijn graph representation of all k-length substrings in a string S. Membership queries within this graph can be answered using just $2k$ `subset-rank` queries on S's SBWT.

Their experiments revealed significant performance improvements compared to previously existing methods. Their approach achieves two orders of magnitude faster query times while maintaining the same space usage. Additionally, with improved space efficiency, it offers a one order of magnitude speedup.

Not anymore since in [6] they improved everything

There is the big problem that in [6] they revisit the rank-select problem on degenerate strings, introducing a new, natural parameter and reanalyzing existing reductions to rank-select on regular strings. Plugging in standard data structures, the time bounds for queries are improved exponentially while essentially matching, or improving, the space bounds. Furthermore, they provide a lower bound on space that shows that the reductions lead to succinct data structures in a wide range of cases. Their most compact structure matches the space of the most compact structure of Alanko et al. [3] while answering queries twice as fast. They also provide an implementation using modern vector processing features; it uses less than one percent more space than the most compact structure of Alanko et al. [3] while supporting queries four to seven times faster, and has competitive query time with all the remaining structures.

There is of course a naive and straightforward way to support these queries in $O(1)$. Given a degenerate string X of length n over an alphabet $\Sigma = [1, \sigma]$, for each $c \in \Sigma$ store a bitmap B_c of length n where

the i-th bit is set to 1 if and only if c is in X_i. This way we can answer the queries in $O(1)$ time. In fact,

$$\texttt{subset-rank}_X(c, i) = \texttt{rank}_{B_c}(1, i)$$

However, this approach requires $O(\sigma n)$ bits of space, which is not efficient if the alphabet is large or the degenerate string is long.

3.3.1.1 *Subset Wavelet Trees*

In order to support the subset-rank and subset-select queries on degenerate strings, Alanko et al. [3] introduced the *Subset Wavelet Tree* (SWT). This data structure is built on top of the Wavelet Tree (WT) [17] (already covered in Section 3.2) and extends it to handle degenerate strings. In this section, we will first see how the SWT is constructed and then how it can be used to answer subset-rank and subset-select queries efficiently.

Structure

Imagine we have an alphabet with $\sigma = 2^n$ symbols, where n is a natural number. We can recursively construct a tree with σ levels to represent all possible subsets of this alphabet. Each node in the tree corresponds to a unique subset. The root node represents the entire alphabet. Each child node of a node v represents a subset, A_v, derived from its parent's alphabet. Let's delve deeper into this recursive process. For each node (except the root), we define its child nodes as follows: the left child represents the first half of the parent's alphabet, while the right child represents the second half.

We also introduce Q_v, a subsequence containing all subsets that include at least one element from A_v. Notably, when A_v represents the entire alphabet, Q_v also includes the empty set. In addition to representing subsets, each node v in our tree holds two bit vectors, L_v and R_v, with a length equal to the size of the corresponding subsequence Q_v.

- $L_v[i] = 1$: This indicates that the i-th subset in Qv contains at least one character from the *first half* of the alphabet associated with node v (Av).
- $R_v[i] = 1$: This indicates that the i-th subset in Q_v contains at least one character from the *second half* of the alphabet associated with node v (A_v).

We can leverage these bit vectors L_v and R_v to create a single string using the alphabet $\{0, 1, 2, 3\}$. The i-th character in this string is formed by a simple calculation:

$$S_v[i] = 2 \cdot R_v[i] + L_v[i] \tag{1}$$

This formula essentially packs the information from both bit vectors into a single digit. A value of 0 indicates the subset doesn't contain elements from either half, 1 signifies the first half only, 2 signifies the second half only, and 3 represents elements from both halves.

3.3.1.2 *Subset-Rank Queries*

The bit vectors L_v and R_v enable efficient rank queries. To find the rank of a character c at position i in the original alphabet, we perform the following steps:

1. **Traverse the Tree**: We navigate from the root node down to the leaf node where the associated alphabet A_v is the single-element set containing only character c.
2. **Prefix Length Calculation**: During this traversal, for each visited node v, we calculate the length of the prefix within the current subsequence Qv. This prefix encompasses all subsets in the original data $(X_1, \ldots, X_i)$ that include at least one character from A_v. Similar to traditional wavelet tree queries, we leverage rank queries on the L_v and R_v bit vectors to determine this prefix length.

Algorithm 8 offers pseudocode for this approach.

3.3.1.3 *Subset-Select Queries*

To answer subset-select queries, we follow a similar process to subset-rank queries.

1. **Traversal from Leaf to Root:** We begin at the leaf node where the associated alphabet A_v is the single-element set containing only character c. We then traverse back towards the root node.
2. **Updating Position** (i): As we move through each node v during traversal, we adjust the position i.
3. **Prefix Length with** c **Characters**: We calculate the length of the prefix within the current subsequence Qv. This prefix encompasses all subsets in the original data $(X_1, \ldots, X_i)$ that collectively contain exactly i occurrences of character c. As done before, we use select queries on the L_v and R_v bit vectors to determine this prefix length.

Algorithm 9 provides pseudocode for this approach. This method leverages the bit vectors to efficiently locate specific character occurrences without iterating through the entire dataset.

Algorithm 8 Subset-Rank Query

Require: c: character from $[1, \sigma]$, i: index
Ensure: The number of subsets X_j such that $j \leqslant i$ and $c \in X_j$

```
function SUBSET-RANK(c, i)
    v ← root
    [l, r] ← [0, σ]                          ▷ Range of alphabet indices
    while l ≠ r do
        if c ⩽ (l+r)/2 then
            r ← ⌊(l+r)/2⌋
            i ← rank_1(L_v, i)
            v ← left child of v
        else
            l ← ⌈(l+r)/2⌉
            i ← rank_1(R_v, i)
            v ← right child of v
        end if
    end while
    return i
end function
```

Space Complexity

The subset wavelet answer subset rank and select queries in logarithmic time $(O(\log \sigma))$, since at each level (there are $\log \sigma$ levels) we perform constant time operations. However, the actual number of bits used vary based on the data we are working with. For a general set sequence, it's typically $2n(\sigma - 1) + o(n\sigma)$ bits.

Visualizing the tree as a complete binary tree with σ leaves (not explicitly stored) helps. If the sets are full, each internal node stores $2n$ bits[17], leading to a total space complexity of $(\sigma - 1)2n$ due to the number of internal nodes and their size. However, for balanced degenerate strings 3.15 (where most sets have one element), space usage improves. Since each set element corresponds to at most one symbol per level, the total sequence length is bounded by set sizes. This translates to a space complexity of $2n \log \sigma$ bits across all $\log \sigma$ tree levels.

We can sum this up in the following theorem

Theorem 3.17 (Space Complexity of Subset Wavelet Trees)**.** *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 in the space complexity is due to...? The space required to store the bitvectors?

17 Since each set goes both to the left and to the right child at each level

Algorithm 9 Subset-Select Query

Require: c: character from $[1, \sigma]$, i: index
Ensure: The position of the j-th subset such that the i-th $c \in X_j$

```
function SUBSET-SELECT(c, i)
    v ← leaf node with alphabet A_v = {c}
    while v ≠ root do                          ▷ Traverse from leaf to root
        u ← parent of v
        if v = left child of u then
            i ← select_1(L_v, i)
        else
            i ← select_1(R_v, i)
        end if
        v ← u
    end while
    return i
end function
```

Rank for Base-3 and Base-4 Alphabets

The subset wavelet tree (SWT) relies on efficiently answering regular rank queries on small alphabet sequences stored within its nodes. At the root node, the sequence uses a base-4 alphabet ($\Sigma = \{0, 1, 2, 3\}$), while all other nodes use a base-3 alphabet ($\Sigma = \{0, 1, 2\}$).

However, the SWT requires a more specific operation than a standard rank query. It needs the sum of two rank queries: $\mathtt{rank}(i, \Sigma - 1)$ and either $\mathtt{rank}(i, \Sigma[0])$ or $\mathtt{rank}(i, \Sigma[1])$. We call these combined operations *rank-pair queries*. Here's how we can express rank-pair queries for base-4

$$\mathtt{rankpair}(i, 1) = \mathtt{rank}(i, 1) + \mathtt{rank}(i, 3) \tag{2}$$
$$\mathtt{rankpair}(i, 2) = \mathtt{rank}(i, 2) + \mathtt{rank}(i, 3) \tag{3}$$

For base-3 alphabets, we can express rank-pair queries as follows:

$$\mathtt{rankpair}(i, 0) = \mathtt{rank}(i, 0) + \mathtt{rank}(i, 2) \tag{4}$$
$$\mathtt{rankpair}(i, 1) = \mathtt{rank}(i, 1) + \mathtt{rank}(i, 2) \tag{5}$$

In the following section we will see different methods that Alanko et al. [3] proposed to answer rank and rank-pair queries on base-3 and base-4 alphabets. They developed these structures with the a clear and very specific goal in mind: answer efficiently membership queries on Spectral Burrows Wheeler Transform (SBWT) sequences. Moreover, they focused on 3 particular genomics dataset and exploited their specific proprieties to develop the most efficient data structures for their needs. This datasets are commonly used for k-mer indexing in bioinformatics and are the following:

1. **E. coli Pangenome:** This dataset consists of 3,682 E. coli genomes downloaded in 2020. It represents a subset of assemblies from NCBI's GenBank database[18] filtered for "Escherichia coli" and downloaded before March 22nd, 2016. The complete dataset is available at[19].

2. **Human Gut Illumina Reads:** This dataset contains 17,336,887 short DNA sequences (reads) of length 502 base pairs obtained from a study on human gut microbiota[20] investigating irritable bowel syndrome and bile acid malabsorption [25].

3. **SARS-CoV-2 Genomes:** This dataset comprises 1,234,695 complete genomes of the SARS-CoV-2 virus, downloaded from NCBI datasets.

The authors shows that when the Spectral Burrows Wheeler Transform (SBWT) is built on these datasets, the resulting degenerate strings present a very skewed distribution of the alphabet symbols, with the vast majority of the sets containing only one element.

3.3.2 *Rank Methods for Subset Wavelet Trees*

In [3] they compare 5 methods that supports `rank` and `rank-pairs` queries on small alphabet sequences (they need it to answer those queries on the sequences stored at the nodes of the subset wavelet tree).

3.3.2.1 *Wavelet Trees*

In their paper, Alanko et. al acknowledge the wavelet tree (Section 3.2) as the current go-to method for performing rank queries on sequences with non-binary alphabets. Wavelet trees will serve as a baseline for comparison with the other data structures they will evaluate.

We have seen in Section 3.2 that we can implement the wavelet trees in different ways, different choices can influence the trade-off between space usage and query speed. The authors experimented with two options: standard bitvectors (faster but larger, Section 3.2) and RRR bitvectors (smaller but slower, as explained in Section 3.2.2.1).

18 ftp://ftp.ncbi.nlm.nih.gov/genomes/genbank/bacteria/assembly_summary.txt
19 zenodo.org/record/6577997
20 SRA identifier ERR5035349

They specifically used implementations from the Succinct Data Structures Library (SDSL)[21] because they are known to be the fastest available WT implementations[22].

I have a lot of doubts about this: why don't use an huffman shaped wavelet tree? Why didn't they try to adapt the wavelet tree to this type of queries?

There are multiple possible implementations of the WT in the SDSL (all immutable): Balanced wavelet (wt_blcd), Balanced wavelet tree for integer alphabets (wt_int), Wavelet matrix for integer alphabets (wm_int), Huffman-shaped wavelet tree (wt_huff), Hu-Tucker-shaped wavelet tree (wt_hutu), Run-length compressed wavelet tree (wt_rlmn), Fast select wavelet tree for integer alphabets (wt_gmr).

TODO: Check their code to see which one they used both for the standard and the RRR version.

3.3.2.2 *Scanning Rank*

The data structure comprises three layers to efficiently store and query the sequence X.

- **Highest Layer:** This layer divides X into superblocks of size s and maintains a table storing $\mathrm{rank}(i, c)$ for each symbol $c \in \Sigma$ and each superblock starting at position i. We can store these counts in a table of size $\sigma n/s$ words, so that we can access the count for any superblock j at the column j/s in constant time.
- **Middle Layer:** This layer further divides X into blocks of size b, a divisor of s. For each block starting at i, precomputed counts of each symbol c are stored, specifically the occurrences of c in the range between the block's start and its enclosing superblock's start. With $s = 2^{32}$, these counts require 32 bits each.
- **Lowest Layer:** This layer directly stores the sequence, packing 32 base-4 symbols per 64-bit word. This layer occupies approximately $64 \cdot \lceil n/32 \rceil$ bits of space, where n is the length of X.

A key optimization interleaves these counts with the actual sequence data within each block. In memory, a block consists of a 2-word header storing four (precomputed) counts, followed by $b/64$ words containing the packed symbols. This interleaving let us store the lower and middle layers in a single array A of $(2n/b + n/32)$ words (i.e., $64 \cdot (2n/b + n/32)$ bits) in memory. This ensures that the data

21 https://github.com/simongog/sdsl-lite

22 The SDSL library does not provide an implementation for rank-pairs queries, but they are just the sum of two rank queries, so they can be easily implemented on top of the rank queries.

structure is cache-friendly, as the entire structure can be loaded into memory in a single read operation.

Rank Queries

To answer $\texttt{rank}(i, c)$ queries with this structure, we follow these steps:

1. Locate the block containing position i.
2. Retrieve the count for c from its header.
3. Add this count to the corresponding count from the superblock table.
4. Scan the block's data section to count occurrences of c up to position i.

TODO: add pseudocode for the rank query.

This scanning typically involves examining whole words and possibly one partial word, which collectively contain the relevant part of the input sequence. Counting occurrences of bit patterns within these words is accelerated through bitwise operations. Notably, $\texttt{rank}-\texttt{pair}$ achieves a particularly fast implementation by using a single bitwise `AND`[23] and a single `popcount`[24] operation to count relevant symbol occurrences within a word.

3.3.2.3 *Sequence Splitting*

The authors propose a novel data structure designed to efficiently represent sequences with skewed distributions of symbols, such as those observed in subset sequences (TODO). The key idea is to exploit the dominance of certain symbols (e.g., 1 and 2 in base-4 sequences) by splitting the sequence into multiple components.

add a picture with the distributions

Given a base-4 sequence X of length n, we use the following components:

- **Bitvector L** representing the subsequence $X_{1,2}$ (only symbols 1 and 2), where each bit indicates whether the corresponding symbol is 1 (0) or 2 (1)
- **Bitvector R** representing the subsequence $X_{0,3}$ (only symbols 0 and 3), with each bit indicating whether the corresponding symbol is 0 (0) or 3 (1)
- **Predecessor data structure P** storing the positions i where $X[i] \in \{0, 3\}$. Both L and R are equipped with rank support structures, enabling efficient rank queries.

23 https://en.cppreference.com/w/cpp/language/operator_logical
24 https://en.cppreference.com/w/cpp/numeric/popcount

In skewed distributions, most sets are singletons. This means $X_{1,2}$ will be long, while $X_{0,3}$, P, and R will be relatively small. This compression is especially beneficial for memory-constrained scenarios. For base-3 sequences, the bitvector L is omitted, as the predecessor structure P already stores the indices of the non-singleton sets (where $X[i] = 2$).

Rank Queries

Answering Rank queries on base-4 sequences, denoted as $\text{rank}_X(i, c)$, involves two steps:

- **Predecessor Query**: A predecessor query on P is performed for position i, returning p, the number of elements in P smaller than i (i.e., the rank of the predecessor of i in P).
- **Binary Rank Query:** The result of the predecessor query, p, is subtracted from i to obtain the appropriate index for a binary rank query. If $c \in \{1, 2\}$, the query is performed on bitvector L:

$$\text{rank}_X(i, 1) = \text{rank}_L(i - p, 0) \tag{6}$$
$$\text{rank}_X(i, 2) = \text{rank}_L(i - p, 1) \tag{7}$$

  If $c \in \{0, 3\}$, the query is performed on bitvector R:

$$\text{rank}_X(i, 0) = \text{rank}_R(p, 0) \tag{8}$$
$$\text{rank}_X(i, 3) = \text{rank}_R(p, 1) \tag{9}$$

Rank queries on base-3 sequences follow the same pattern as for base-4 sequences when dealing with singletons ($x \in \{0, 1\}$). More precisely

$$\text{rank}_X(i, 0) = \text{rank}_R(i - p, 0) \tag{10}$$
$$\text{rank}_X(i, 1) = \text{rank}_L(i - p, 1) \tag{11}$$

Since there's no second binary vector, the result of the predecessor query directly gives the rank for $c = 2$.

On the other hand, with this data structure, *rank-pair* queries can be answered more efficiently than two separate rank queries. This is because the predecessor query, which computes p, needs to be performed only once for both symbols in the query.

3.3.2.4 *Generalized RRR*

For this final method, the authors present a generalization of the RRR entropy compressed bitvector (see Section 3.1.1 and [38]) to accommodate base-3 and base-4 sequences. While generalizations of RRR exist in the literature [15], this specific adaptation draws inspiration from the work of Navarro and Providel [34].

Let X be a sequence of length n from a constant-sized alphabet σ. To efficiently answer rank queries, we employ a three-level indexing structure similar to the basic binary RRR structure. We divide X into blocks of size $b = O(\log n)$ and group them into superblocks of size $B = O(\log n)$, where B is a multiple of b. For each superblock, we precompute the counts of all symbols σ up to its starting position, storing these counts using $O(\log n)$ bits each. For each block, we precompute the counts of all symbols σ within the block, storing them using $O(\log \log n)$ bits each. The total space required for these precomputed counts is $O(n\sigma \log \log n / \log n) = o(n)$ since σ is constant.

A rank query $\text{rank}(i, c)$ is then answered by: (i) Looking up the precomputed count of symbol c up to the start of the superblock containing position i. (ii) Summing the precomputed counts of symbol c in the blocks preceding index i within the superblock. (iii) Counting the occurrences of symbol c in the prefix of length $p = i \mod b$ within the block containing index i.

To determine the symbol count within a block's prefix, additional meta-information is encoded for decoding the block's symbol sequence. Then, we loop to count the occurrences of symbol c within the prefix of length $i \mod b$. The authors introduce a equivalence relation within all the possibile σ^b blocks, such that two blocks are equivalent if and only if they contain tha same multiset of symbols[25]. We can use the equivalence classes to efficiently choose the meta-information to store: each block will store the rank r of the block within the lexicographically sorted list of all the blocks in the same equivalence class.

The `unrank` Function

The class and the lexicographically rank of the block within the class completely determine the block's content (i.e the sequence of symbols it contains). This means that we can define a function

$$\text{unrank}(r, d_0, \ldots, d_{\sigma-1})$$

that takes as input the lexicographic rank r and the precomputed counts of the symbols in the block and returns the block's content. A naive implementation of this function would require to precompute and store all the possibile answers to the function, similarly to the third layer (the lookup table) of the RRR rank data structure. However, for large values of b this is not feasible.

25 A multiset is a generalization of the concept of a set that, unlike a set, allows multiple instances of the same element.

Given the *multinomial coefficient*

$$\binom{n}{d_0 d_1 \dots d_{\sigma-1}} = \frac{n!}{d_0! d_1! \dots d_{\sigma-1}!} \tag{12}$$

defined so that the value is 0 if the sum of the d_i is greater than n or if any of the d_i is negative. We can introduce the lexicographic rank of a block $c_0, c_1, \dots, c_{b-1}$ in the equivalence class as

$$\text{lexrank}(c_0, c_1, \dots, c_{b-1}) = \sum_{i=0}^{b-1} \sum_{j=0}^{c_i - 1} \binom{b-1-i}{D_0(i) \ \dots \ D_j(i) - 1 \ \dots \ D_{\sigma-1}(i)} \tag{13}$$

where $D_k(i)$ is the number of occurrences of the symbol k in the suffix of length i of the block[26]. This function can be computed in $O(b\sigma)$ time. The term -1 that appears in the choices of the multinomial is there to avoid counting the block itself.

Formula 13 is a process that iterates through the symbols within a block from left to right, computing different ways to choose the remaining symbols in the block using the remaining counts, with the constraint that the complete block must be lexicographically smaller than the current block. The `unrank` function essentially reverses the `lexrank` function. This can be done by incrementally adding the multinomial terms in the inner sum until the total surpasses the target rank (r). At this point, the current symbol (j) is appended to the block's sequence, and the process moves on to the next iteration of the outer sum.

Algorithm 10 provides a pseudocode implementation of this unranking process for a sequence based on a base-4 number system. The function prints the sequence of symbols in the block with rank r among the class of blocks with symbol counts d_0, d_1, d_2, and d_3

26 $c_i, \dots, c_{b-1}$

Algorithm 10 Base-4 block *unranking* algorithm

```
function Base4BlockUnrank(r, d_0, d_1, d_2, d_3)
    b ← d_0 + d_1 + d_2 + d_3                          ▷ Block size
    s ← 0                                              ▷ Blocks counted so far
    for i = 0 to b − 1 do
        for j = 0 to 3 do                              ▷ 0 to σ − 1
            d_j ← d_j − 1
            count ← (b−1−i choose d_0, d_1, d_2, d_3)
            d_j ← d_j + 1
            if s + count > r then
                print j
                d_j ← d_j − 1
                break
            else
                s ← s + count
            end if
        end for
    end for
end function
```

3.4 IMPROVEMENTS OVER PREVIOUS METHODS

In the previous section 3.3.1.1 we have seen how in [3] the authors introduced the Subset Wavelet Tree (SWT) to solve the subset rank-select problem. Their data-structure supports both `subset-rank` and `subset-select` queries in $O(\log \sigma)$ time and uses $2(\sigma - 1)n + o(n\sigma)$ bits of space in the general case.

In this section, we will detail the significant improvements made by Bille et. al in [6] over the previous methods. The authors introduce a series of novel reductions and data structures that not only enhance the theoretical bounds but also demonstrate substantial empirical improvements.

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 3.18. 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 3.19). Lastly, they implemented and compared their reductions to prior implementations, achieving twice the query speed of the most compact structure from [3] 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 3.18 (General Upper Bound)**.** *Let X be a degenerate string of length n, size N, and containing n_0 empty sets over an alphabet $[1, \sigma]$. Let $\mathcal{D}$ denote a $\mathcal{D}_b(\ell, \sigma)$-bit data structure for a length-ℓ string over $[1, \sigma]$, supporting:*

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

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

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

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

and supports:

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

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

(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 3.18, (i) and (ii) extend prior reductions from [2], 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 3.18 (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. [3], while exponentially reducing query times.

For $n_0 > 0$, Theorem 3.18 (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 3.18(iii) allows for tailored bitvector structures sensitive to n_0.

Theorem 3.19 (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 3.19 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. □

3.4.1 *Reductions*

Let $X, \mathcal{D}, \mathcal{B}$ be as in Theorem 3.18 and consider $\mathcal{V}$ a data structure (for example the one described by Jacobson in [24]), 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 3.18 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 6 from [6] illustrates this reduction for $n_0 = 0$.

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 6: to compute $\mathsf{subset\text{-}rank}(2, \mathtt{A})$, we first compute $\mathsf{select}_R(3, \mathtt{1}) = 6$. 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 $\mathsf{rank}_R(8, \mathtt{1}) = 4$ to determine that position 8 corresponds to X_4.

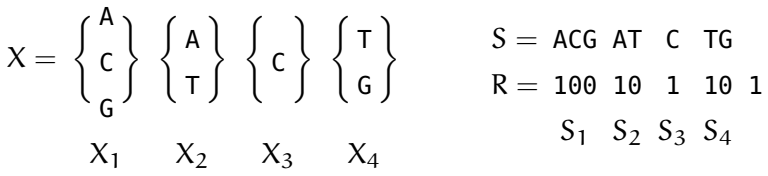

Figure 6: *Left:* A degenerate string X over the alphabet $\{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$ where $n = 4$ and $N = 8$. *Right:* The reduction from Theorem 3.18 (i) on X. White space is for illustration purposes only.

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 3.18 (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 3.18 (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 $\text{subset-rank}_X(i, c)$, calculate $k = i - \text{rank}_E(i, 1)$, which maps X_i to its corresponding set X''_k. Then, return $\text{subset-rank}_{X''}(k, c)$. This operation runs in $\mathcal{B}_r(n, n_0) + \mathcal{D}_r(N, \sigma) + O(1)$ time.

To support $\text{subset-select}_X(i, c)$, first determine $k = \text{subset-select}_{X''}(i, c)$, and then return $\text{select}_E(k, 0)$, 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.

3.4.2 *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 [3] 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 DAGS FOR EFFICIENT PREFIX QUERIES

BIBLIOGRAPHY

[1] N. Vereshchagin A. Shen V. A. Uspensky. *Kolmogorov Complexity and Algorithmic Randomness*. Mathematical Surveys and Monographs. Amer Mathematical Society, 2017.

[2] 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.

[3] 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.

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

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

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

[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, 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.

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

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

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

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

[13] 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.

[14] Paolo Ferragina and Rossano Venturini. "A simple storage scheme for strings achieving entropy bounds." In: *Theoretical Computer Science* 372.1 (2007), pp. 115–121. ISSN: 0304-3975.

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

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

[17] 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).

[18] 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.

[19] 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.

[20] 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.

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

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

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

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

[25] Ian B Jeffery et al. "Differences in fecal microbiomes and metabolomes of people with vs without irritable bowel syndrome and bile acid malabsorption." In: *Gastroenterology* 158.4 (2020), pp. 1016–1028.

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

[27] 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.

[28] 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).

[29] 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.

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

[31] 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.

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

[33] 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.

[34] Gonzalo Navarro and Eliana Providel. "Fast, small, simple rank/select on bitmaps." In: *International Symposium on Experimental Algorithms*. Springer. 2012, pp. 295–306.

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

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

[37] 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.

[38] 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.

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

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

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

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

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