

---

# A GLIMPSE OF LEMPEL-ZIV ALGORITHMS

---

## Advanced algorithms

*Santi Enrico*  
*University of Udine*  
*Academic year 2022/23*

## Contents

1	Introduction . . . . .	3
1.1	The data compression problem . . . . .	3
2	The Lempel-Ziv family . . . . .	4
2.1	Static and dynamic dictionaries . . . . .	4
2.2	LZ77 . . . . .	4
2.2.1	Encoding . . . . .	5
2.2.2	Decoding . . . . .	5
2.2.3	Considerations . . . . .	5
2.3	LZ78 . . . . .	5
2.3.1	Encoding . . . . .	6
2.3.2	Decoding . . . . .	7
2.4	LZW . . . . .	7
2.4.1	Encoding . . . . .	7
2.4.2	Decoding . . . . .	8
2.5	Considerations . . . . .	8
3	Burrows–Wheeler transform and Wavelet trees . . . . .	9
3.1	How the Burrows–Wheeler transform it works . . . . .	9
3.2	Wavelet trees . . . . .	10
4	An efficient LZ77 implementation . . . . .	11
4.1	Suffix arrays . . . . .	11
4.2	LZ77 a high level implementation . . . . .	11

## 1 Introduction

We begin this section by briefly recalling the data compression problem, its relevance and some approaches developed to deal with it.

### 1.1 The data compression problem

Data compression can be seen as a procedure that given as input some data (without loss of generality we can think about text) tries to shrink its size by returning a representation which uses less amount of memory (we can think about bits or digits) while keeping all or the majority of the original information.

This problem isn't new and has been tackled in several ways throughout the years, a relevant and historical example is the Morse code which assigns shorter codes to letters which are more frequently found in English text. An analogous idea has been used again in 1952 by Huffman in the Huffman codes.

All compression methods can be subdivided in two classes [1]:

- **Lossless:** These algorithms don't lose any information of the original data allowing the existence of a procedure that given the compressed data are able to get the original input. An example of family of algorithms belonging to this class is the Lempel-Ziv one (e.g. LZ77, LZ78, LZW,...).
- **Lossy:** This class of methods on the other hand presents some information loss during the compression phase. This implies that the decompression algorithm could not always return exactly the original input. JPEG is an example of such algorithm for image files.

Algorithms belonging to these classes are used in different contexts, usually involving different kind of data (e.g. for audio and video transmission some information loss is allowed while for inherently textual data usually it's not).

Nowadays compression is crucial, it allows to have faster communications saving a lot of the required bandwidth <sup>1</sup>. Another advantage which would be interesting to investigate is the average amount of physical volume saved by compression with respect to a *world without compression* (e.g. think to how many more hard drives and storage devices we would need if all the movies, images, and documents were stored naively).

---

<sup>1</sup> The term bandwidth is used not to refer to the spectrum of frequencies used during a communication but rather informally as the amount of data transferred

## 2 The Lempel-Ziv family

We now focus on a particular set of compression methods used to compress textual data, the Lempel-Ziv algorithms. Before diving into the first algorithm let's recall the concept of dictionary. The idea is that in the source code (i.e. the code to compress) usually certain substrings are present much more frequently than others, and thus it makes sense to store them in a structure and just send them once alongside their occurrence in the text, by using a reference to the dictionary (e.g. their index). One of the ideas of the Lempel-Ziv algorithms is indeed exploiting the structure of the source code (not just the character probabilities like Huffman) to achieve a good compression rate.

### 2.1 Static and dynamic dictionaries

There are two different kind of dictionaries, Static and Dynamic[2]:

- **Static:** The dictionary in this case is created and subsequently used without being modified. Methods adopting this technique can be efficiently used when we have prior knowledge about the source code (e.g. we know the complete string or the words that are present in the string to encode).
- **Dynamic or adaptive:** This technique is the one used and introduced [1] by the Lempel-Ziv class of algorithms. The idea is not to maintain a static set of words where each word mapped to some shorter identifier but rather expand the dictionary as the source code is read (while also removing from it).

The latter approach provides some advantages, the main one consisting in the capacity to work on a stream of data (where we don't know necessary the whole sentence to transmit).

### 2.2 LZ77

This first algorithm uses a technique called *sliding window*, which consists in keeping in memory a (fixed) limited portion of the input stream. This window is further subdivided into two sections, the *search buffer* and the *lookahead buffer* (both of fixed size). The former is to the left of the latter and contains the last part of the input stream already encoded (compressed), meanwhile the search buffer contains a portion of the stream which has to be compressed, the first character of this buffer contains the first character not yet encoded. The resulting output of the algorithm will be a series of triples. We will see now more in detail how the encoding process works.

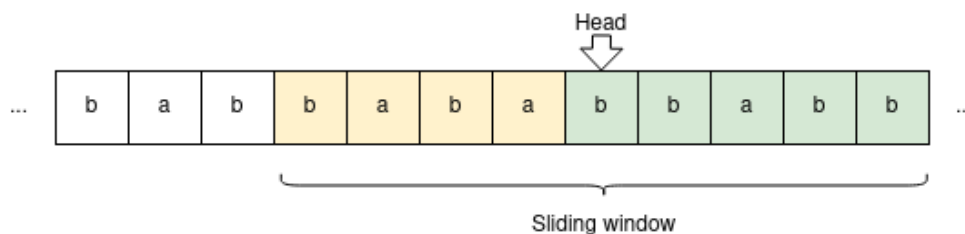


Fig. 1: A sliding window of size 9 (the search buffer is yellow, the lookahead green).

### 2.2.1 Encoding

Let's call the size of the search buffer  $m$  and the size of the lookahead  $n$  (in general  $m > n$ ), the encoder begins by setting the search buffer empty and the lookahead buffer to the first  $n$  characters of the input stream. The procedure then works by repeating until the end of the input these steps:

- Scan the search buffer backwards choosing the substring contained into it matching the longest prefix of the lookahead buffer, let's call this substring *match*. If more matches of equal length exists a simple rule can be established to break the symmetry (e.g. take the rightmost match).
- Construct the following triple  $(i, l, a)$  as output where  $i$  is the distance<sup>2</sup> between the index of the head (first cell of the lookahead buffer) and the index of the first character of *match*,  $l$  is the length of *match* and  $a$  is the first character in the lookahead buffer not matching the *match* found.
- Shift both buffers of  $l + 1$  characters to the right.

As shown in [3] the complexity of the encoding process is  $O(n \log(|\Sigma|))^3$  for what concerns bit space and  $O(n \log(\log(|\Sigma|)))$  for the time complexity, in Section 4 some of the tools used to achieve such performances will be discussed while a different implementation running in  $O(n)$  time and requiring  $O(2n \log(n))$  [4] will be presented.

### 2.2.2 Decoding

The decoding procedure is rather simple, it starts with an empty search buffer, then works by repeating the steps:

- Consider a triple  $(i, l, a)$ , copy as output the  $l$  characters from the buffer starting at position  $m - i$ .
- Append the character  $a$  to the output and shift the search buffer to consider the last  $m$  characters.

### 2.2.3 Considerations

It's quite immediate to see that this algorithm works well when there are many close repetitions, while it can be also observed that the size of the buffers influences its performance. This last consideration must be analyzed a bit, it's true that larger buffers (especially the search one) can lead to longer matches and thus less triples as output but these fewer triples would present bigger numbers which would increase the size of the encoded sequence. Several variants and improvements of this algorithm were proposed [1].

## 2.3 LZ78

A different approach with respect to LZ77 is taken by LZ78 or LZ2 which doesn't use the sliding window. This approach thus gets rid of the parameters (the sizes of the buffers) which impacted LZ77 performances.

---

<sup>2</sup> i.e. the number of characters

<sup>3</sup>  $\Sigma$  is the alphabet used for the input

### 2.3.1 Encoding

The dictionary used is still dynamic and consists of a list of triples/entries  $(i, s, o)$  where  $i$  is the number of the entry in the dictionary,  $s$  is a string and  $o$  is a pair  $(p, c)$  returned as output when  $s$  is found in the input stream. This dictionary will define a (prefix) trie.

At the beginning of the algorithm the dictionary starts just with the empty entry  $(0, \text{null}, \epsilon)$  and continues repeating the following steps (which will increase the dictionary size):

- Read an input symbol  $x_i$  and look for the entry with only such character in  $s$ .
- If such entry is found read the next symbol  $x_{i+1}$  and look for the entry with  $s = x_i x_{i+1}$ , and so on...
- At the first iteration the condition above isn't satisfied (i.e. we don't find an entry containing the string  $x = x_i \dots x_j$ ) add a new entry  $(i', x, o')$  where  $i'$  is the first position available in the dictionary and the output  $o'$  consists of the index in the dictionary where the entry with the string  $x_i \dots x_{j-1}$  is found (i.e. the prefix of  $x$  excluding the last character)<sup>4</sup> and last character of  $x$  (i.e.  $x_j$ ).

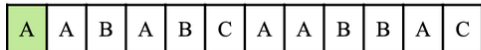
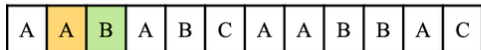




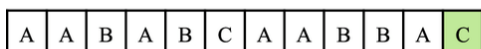


	Tuple $(i, c)$	Dictionary $D[i]$
	$(0, A)$	$D[1] = A$
	$(1, B)$	$D[2] = AB$
	$(2, C)$	$D[3] = ABC$
	$(1, A)$	$D[4] = AA$
	$(0, B)$	$D[5] = B$
	$(5, A)$	$D[6] = BA$
	$(0, C)$	$D[7] = C$
 Next Character  Matched Letter		

Fig. 2: An example of how LZ78 expands the dictionary [5].

Note that while in LZ77 the amount of memory needed for the encoding process was fixed at the beginning (by fixing the parameters) here the dictionary can grow indefinitely (indeed we don't remove any entry during the process). This memory trade-off on the other hand allows LZ78 can capture patterns and hold them indefinitely (this works ideally, in practical implementations the dictionary at some point has to stop growing).

<sup>4</sup> We know that such entry exist since otherwise the if condition would have failed before

### 2.3.2 Decoding

The decoding process works similar to the encoder, creating the same dictionary while the pairs in input are read. Once a pair is read, the *chain of indexes*<sup>5</sup> is followed until we reach the root (i.e. the empty entry), printing as output in reverse order the characters found in the path.

## 2.4 LZW

As mentioned, throughout the years many variants of LZ77 and LZ78 have been proposed. Lempel-Ziv-Welch is a variant of LZ78, which allows the algorithm to output not a pair of elements but a single value  $p$  (the reference to a dictionary entry of the pair of LZ78).

### 2.4.1 Encoding

To achieve what stated above LZW uses a simple but clever trick which consists in recognising substrings in the input stream only when these are matched exactly by an entry in the dictionary, leaving the first mismatching character as the first character for the next iteration (to the next substring). Clearly, although only matching strings are recognised a new entry will be added in the dictionary containing the concatenation of the matching string and the new 'mismatching' character.

While the new main idea used in the encoding process has been introduced in the paragraph above, it requires an additional consideration, since the dictionary differently of LZ78, can't be initialized with just the empty entry, but must contain every single character of the alphabet used for the input stream. The reason for this is quite simple to understand, since now we deal with just what we have already have in the dictionary we must be able to deal with all the most basic substrings we can find in the stream (these entries can be seen as the base cases of the recursive procedure beneath the algorithm, such base case before was just the empty string).

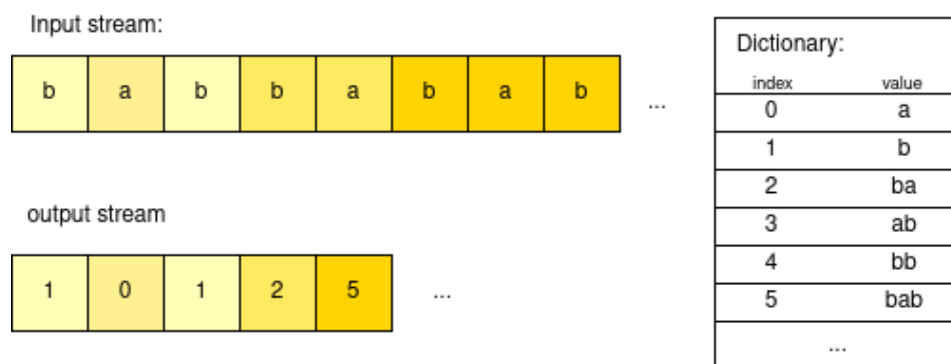


Fig. 3: An example of the LZW encoding.

<sup>5</sup> Take the first element of the pair, following it retrieve the next pair in from the dictionary and so on.

### 2.4.2 Decoding

The decoding process is rather straight forward, start with the initialized dictionary (one entry per alphabet symbol), then repeat:

- Read value  $x_i$  from the encoded sequence, look for the corresponding entry and concatenate the value to the output.
- Peek the next  $x_{i+1}$  value from the sequence, look for the entry, take the first character, concatenate  $x_i$ , if such new string doesn't exist in the dictionary add it.

## 2.5 Considerations

Many other variants of Lempel-Ziv (LZ1) and LZW have been proposed and are actually utilized, each having its advantages and limitations, leaving most of the times to the developer the choice of which one is more suited for a specific application.

Research especially on LZ77 is still being carried out, for example the complexities presented in 2.2.1 have been achieved quite recently by using among other tools the Burrows-Wheeler transform. In addition a lot of work and research is focused on improving LZ77's complexity in specific cases (e.g. when the input text presents a high number of repetitions).

As a practical example Lempel-Ziv-Markov (LZMA) is one of the algorithms used in 7-Zip and LZMA can be seen as a fancy extension of LZ77.



### 3 Burrows–Wheeler transform and Wavelet trees

Let’s now give a glimpse of some of the tools mentioned and used in some recent works (e.g. [6]), such as the Burrows–Wheeler transform and Wavelet trees.

To get an intuition of what we are going to briefly describe, we can think to the BTW as a reversible function (more precisely a permutation) that given an input string returns a string (and an integer index) which can be compressed more than the original one [7]. In the next subparagraph describing the idea of the BTW we are going to use a slightly different approach than [7] since some steps in such paper quite difficult to visualize and are explained in an easier way in [8], thus we are gonna follow an in-between approach.

#### 3.1 How the Burrows–Wheeler transform it works

Given a string  $s$ , we first concatenate to it a character  $\bullet$  not present in it, so we get  $s\bullet$ . We assign to this character the smallest lexicographic value, we then generate all rotations of  $s\bullet$  and sort them lexicographically (the  $\bullet$  symbol allows to the rotation to be all distinct). We can think to the partial result of the described procedure as a matrix. Let’s visualize what described by recalling the example made in [7]:

mississippi•	•ippississi	m
ississippi•m	im•ippissis	s
ssissippi•mi	ippississim	•
sissippi•mis	issim•ippis	s
issippi•miss	ississim•ip	p
ssippi•missi	m•ippississ	i
sippi•missis	pississim•i	p
ippi•mississ	ppississim•	i
ppi•mississi	sim•ippissi	s
pi•mississip	sissim•ippi	s
i•mississipp	ssim•ippiss	i
•mississippi	ssissim•ipp	i

The procedure applied on the string *mississippi*, note that every column and every row is a permutation of  $s\bullet$ .

The result of BTW consists in the last column (without the  $\bullet$ ) of such sorted matrix alongside the index of the position in which  $\bullet$  was removed, in the example above the result is  $\langle msspipissii, 3 \rangle$ . There are obviously clever ways to encode the input string without explicitly creating the matrix, and the same can be done for the decoding process, thus such procedures can actually be implemented in space  $O(|s|)$ . For the decoding process it’s sufficient to reconstruct the first and last column of the sorted matrix, this will enable us to reconstruct the first row (which corresponds to  $s\bullet$ ). We won’t go in detail on the decoding procedure but the last column comes free from the output of the decoding process, and we can get the first one simply by sorting the elements of the last column we just retrieved. To get the elements of the first row we exploit the fact that the first and last column are adjacent characters in  $s$  since all rows are just rotations.

Why this transformation is remarkably useful in data compression it’s clearly explained in

the original paper [9]. The high level idea is that the encoded string will consist mostly of regions with the same character repeated many times, interleaved by just some other characters, in some sense this transform condenses the majority of the occurrences of a character in  $s$  close together.

To visualize this consider the example presented in [9], suppose  $s$  is long English sentence and assume that many instances of ‘the’ are present in it. Consider now the sorted matrix of rotations, and think to the rows starting with ‘he’, it’s likely that they end with ‘t’ (obviously some of them could end with other characters). Now, since the rows are sorted, the ones starting with ‘th’ will be one after the other resulting the ‘t’ characters in the last row to form a rather well defined region. An analogous reasoning can be done for the other characters. This technique can be used to increase the performances of many lossless compression algorithms (such as LZ77).

### 3.2 Wavelet trees

Wavelet trees are a succinct<sup>6</sup> data structure (a binary tree) which allows to (space) efficiently represent a string and answer some queries<sup>7</sup> on it [10].

Specifically given a string  $S$  of length  $n$  over  $\Sigma$  we can represent it with  $n \lceil \log_2(|\Sigma|) \rceil$  bits. The wavelet tree has height  $\lceil \log_2(|\Sigma|) \rceil$  and has one leaf for each symbol in the alphabet. Let’s now briefly see the construction of the tree, given  $S$  the root of the tree will contain a bit array of length  $n$ , we then start subdividing the alphabet into two halves, let’s call them  $\Sigma_0$  and  $\Sigma_1$ , the two children of the root will contain a bit vector of size  $n_1$  and  $n_2$  where these lengths are the number of characters of  $\Sigma_0$  and  $\Sigma_1$ . The bit array of the parent node will have a 0 in every position  $i$  in which  $S[i] \in \Sigma_0$  and a 1 where  $S[i] \in \Sigma_1$ . By iterating this process until we obtain nodes (leaves) containing only one character gives us the desired tree.

---

<sup>6</sup> Using asymptotically the same space used by the plain representation of the data.

<sup>7</sup> Such as accessing the  $i$  – th element in  $O(\log_2(|\Sigma|))$ , rank or select.

## 4 An efficient LZ77 implementation

We now put aside some of the structures introduced above used in different implementations to focus and sum up the algorithm in [4] which implements LZ77 with the complexities mentioned in 2.2.1. This implementation is easier to understand compared to the one in [3] since it doesn't use many tools, it just requires suffix arrays that we will briefly describe in the next subsections.

### 4.1 Suffix arrays

These are an alternative to suffix trees (we can build these arrays from them in linear time<sup>8</sup>), and they are just a permutation of the numbers  $[1, n]$  where  $n$  is the length of the string for which we want the suffixes [11]. This permutation indicates the suffixes in lexicographic order.

1	mississippi		11	i
2	ississippi		8	ippi
3	ssissippi		5	issippi
4	sissippi		2	ississippi
5	issippi		1	mississippi
6	ssippi	$\xrightarrow{\text{lex.sort}}$	10	pi
7	sippi		9	ppi
8	ippi		7	sippi
9	ppi		4	sissippi
10	pi		6	ssippi
11	i		3	ssissippi

The suffix array of the string mississippi is the yellow array on the right table.

The generic element of a suffix array  $SA$  can be described as:

$SA[i] = j \iff$  the suffix beginning in position  $j$  is the  $i$ -th in lexicographic order.

We also introduce the inverse of a suffix array  $SA$ , namely  $ISA$  which is defined as following:

$ISA[i] = j \iff SA[j] = i$ ,  $ISA$  tells us the lexicographic position of suffix  $i$ .

### 4.2 LZ77 a high level implementation

Let's first recall that the following algorithm aims to reduce the time complexity to perform LZ77, not space. The algorithm in [4] uses a slightly different LZ77 idea (i.e. it outputs pairs instead of triples) than the one presented in section 2.2.1 so we are going to present it with some modifications to make it consistent with our description. The output pairs/triples are called *fators* (*LPFs*, *longest previous factors*).

The high level idea of the implementation is the following:

<sup>8</sup> Given the suffix tree for a string, we could do a DFS on it where at each node we first expand the child denoted by the smallest lexicographical string. Keep in mind that we could also naively apply a sorting algorithm to sort the array of the suffixes of a string, but this in the best case would result in a time complexity of  $O(n^2 \log_2(n))$  where  $n$  is the length of the string.

The LPFs can be computed by calculating first two arrays, *next smaller values*, and *previous smaller values* defined as:

$$NSV[i] = \min\{j \in [i+1..n] | SA[j] < SA[i]\} \text{ or } 0 \text{ if set is empty}$$

Meaning that for each position we want the index of *SA* containing the starting index of the longest suffix which is lexicographically bigger than the *i*-th (lexicographically) suffix and such that its starting index is left of the *i*-th suffix in the string.

$$PSV[i] = \max\{j \in [1..i-1] | SA[j] < SA[i]\} \text{ or } 0 \text{ if set is empty}$$

Namely the starting index of the shortest suffix lexicographically smaller than the *i*-th (lexicographically) one such that its starting index is left of the *i*-th suffix in the string.

Using these two definitions we now define:

$$NSV_{text}[i] = SA[NSV[ISA[i]]]$$

$$PSV_{text}[i] = SA[PSV[ISA[i]]]$$

The key now is to notice that the position from the beginning of *S* to the longest match in a given position *i* is contained in  $NSV_{text}[i]$  or  $PSV_{text}[i]$ . To determine which one consider we need to see which is the one matching the longest prefix starting at position *i* (in the code below this is done via the *lcp* function). The algorithm from [4] works by first computing the two vectors above and the *SA*. For each position *i* in the string it then calls a subprocedure producing the factor and returning the next index in *S* from which start the new encoding from.

```

1 Compute the SA of S[1..n] (which we know can be done in linear time)
2 SA[0] ← 0
3 SA[n + 1] ← 0
4 top ← 0
5 for i ← 1 to n + 1 do
6     while SA[top] > SA[i] do
7         NSV_text[SA[top]] ← SA[i]
8         PSV_text[SA[top]] ← SA[top - 1]
9         top ← top - 1
10    top ← top + 1
11    SA[top] ← SA[i]
12 i ← 1
13 //for each index we get from the procedure we call the encoding function and
14 //update our next index
15 while i ≤ n do
16     i ← LZ-Factor(i, PSV_text[i], NSV_text[i])
17
18 //The function below, which for each position computes the factor has been
19 //modified from the original one in the paper
20 LZ-Factor(i, psv, nsv):
21     //we want to consider the longest prefix to compress a bigger string.
22     //in every case we store the index (from the current position), the
23     //length of the pattern and the first mismatching character.
24     if lcp(i, psv) ≥ lcp(i, nsv) then
25         len ← lcp(i, nsv)
26         (index, l, a) ← (i-psv, len, S[len+i])
27     else
28         len ← lcp(i, psv)

```

```
29         (index, l, a) ← (i-nsv, len, S[len+i])
30         //here was an additional check no more needed
31         output factor (index, l, a)
32         return i + l + 1
```

## References

- [1] K. Sayood, *Introduction to Data Compression, Fourth Edition*, 4th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2012.
- [2] D. Salomon, *Data Compression: The Complete Reference*. Berlin, Heidelberg: Springer-Verlag, 2006.
- [3] D. Belazzougui and S. J. Puglisi, “Range predecessor and lempel-ziv parsing,” 2015.
- [4] J. Kärkkäinen, D. Kempa, and S. J. Puglisi, “Linear time lempel-ziv factorization: Simple, fast, small,” in *Combinatorial Pattern Matching*. Springer Berlin Heidelberg, 2013, pp. 189–200. [Online]. Available: [https://doi.org/10.1007/978-3-642-38905-4\\_19](https://doi.org/10.1007/978-3-642-38905-4_19)
- [5] ResearchGate. [Online]. Available: [https://www.researchgate.net/figure/Example-of-compression-process-of-LZ78-for-the-text-data-AABABCAABBAC\\_fig1\\_337580417](https://www.researchgate.net/figure/Example-of-compression-process-of-LZ78-for-the-text-data-AABABCAABBAC_fig1_337580417)
- [6] A. Policriti and N. Prezza, “Lz77 computation based on the run-length encoded bwt,” *Algorithmica*, vol. 80, no. 7, p. 1986–2011, jul 2018. [Online]. Available: <https://doi.org/10.1007/s00453-017-0327-z>
- [7] G. Manzini, “The burrows-wheeler transform: Theory and practice,” in *Mathematical Foundations of Computer Science 1999*, M. Kutylowski, L. Pacholski, and T. Wierzbicki, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 34–47.
- [8] M. Crochemore, J. Désarménien, and D. Perrin, “A note on the burrows-wheeler transformation,” *arXiv preprint cs/0502073*, 2005.
- [9] D. W. M. Burrows, “A block-sorting lossless data compression algorithm,” Digital Systems Research Center, Palo Alto, Tech. Rep., 1994. [Online]. Available: <https://www.hpl.hp.com/techreports/Compaq-DEC/SRC-RR-124.pdf>
- [10] G. Navarro, “Wavelet trees for all,” *Journal of Discrete Algorithms*, vol. 25, pp. 2–20, 2014, 23rd Annual Symposium on Combinatorial Pattern Matching. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S1570866713000610>
- [11] D. Gusfield, *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.