Notes on Ferragina and Manzini work (2005): Text indexing and compression

Théo Roncalli
Student
Paris-Saclay University, Paris
theo.roncalli@universite-paris-saclay.fr

Donatien Legé
Student
Paris-Saclay University, Paris
donatien.lege@universite-paris-saclay.fr

February 2021

1 Introduction

As the amount of data grows exponentially, the matter of compression and indexing is important, especially in the field of bioinformatics where the pattern search and mapping are two fields that draw attention for some years. They are employed in many applications such as for manipulating ChiP-Seq or RNA-seq data. Numerous tools have been proposed to perform indexing such as trie or B-tree. Nevertheless, these data structures can grow very fast and require important memory space. The suffix array is an interesting alternative to overcome the issue of spatial complexity. Here, we are interested in the work of Ferragina and Manzini (2005) who focus on the *Burrows-Wheeler transformation* data structure (Burrows and Wheeler, 1994). First, it was used for data compression, but then Ferragina and Manzini (2005) uses it as a full-text index. In this work, we present the contribution of the work of Ferragina and Manzini (2005). Section 2 describe an important concept: the *Burrows-Wheeler transformation* (Burrows and Wheeler, 1994). This concept is important for understanding how indexing can be performed. Section 3 introduces the algorithms performed for indexing. Section 4 is a discussion about the performance of the algorithms. The reader can find some implementation of the indexing algorithms[1] at the following address: `https://github.com/Theo-Roncalli/Text-Indexing`.

2 Burrows-Wheeler transform

Among the many concepts used by Ferragina and Manzini (2005), the *Burrows-Wheeler transformation* (BWT) developed by Burrows and Wheeler (1994) is at the heart of their work. The BWT algorithm consists in rearranging a text T into a new string L of same length in such a way that identical characters bring together and that the transformation is reversible. With a more formal definition, the BWT string corresponds to the last column of a sorted circular shift of all text's suffices. Here, Ferragina and Manzini (2005) consider the output string L as an important part of their index data structure. For building it, an alphabetical order must be defined over the alphabet $\mathcal{A}$. Then, the text input T is transformed by following four steps: (1) concatenate the text T with a special character \$ at the end of the text where this character must be the first character in the alphabetical order and must not be present in the initial text T, (2) build a matrix $\mathcal{M}$ of dimension $n \times n$ whose rows are the cyclic shifts of the text and where n is the size of the new text, (3) sort the rows of the matrix $\mathcal{M}$ by lexicographic order and (4) construct the BWT string by setting L as the string corresponding to the last column of the matrix $\mathcal{M}$. From a combinatorial optimization perspective, the BWT string can be computed in a linear complexity $\mathcal{O}(n)$ from the suffix array data structure.

Example 1. Let us consider an input string $T = tester$. First of all, we append a smaller character \$ than any other character in the text T. In Table 1, the step (2) and (3) are performed. We obtain $F = \$eerstt$ and $L = rttees\$$ by taking the first and last column of the matrix $\mathcal{M}$ respectively. The output text L correspond to the BWT string. We observe that identical letters are contiguous in the permutated text L.

As index L contains numerous contiguous identical characters, some lossless data encoding allows an important compression on the output text L. The compression of the output string L is important in order to reduce the space complexity. The more intuitive is the *run-length encoding* which consists in replacing long runs of identical characters by its number following

[1] The proposed algorithm implementations are not competitive with those proposed by the authors because some data structures are not respected in order to make the code easier to write but taking more RAM memory.

Table 1: Construction of the BWT matrix $\mathcal{M}$ for the input string $T = \text{tester}$

	Step (2)			Step (3)	
				F	L
1	tester$		1	$tester	
2	ester$t		2	er$test	
3	ster$te	$\Longrightarrow$	3	ester$t	
4	ter$tes		4	r$teste	
5	er$test		5	ster$te	
6	r$teste		6	ter$tes	
7	$tester		7	tester$	

by its data value. For instance, if we consider the BWT string $L = rtteesr\$$ (example 1), we obtain the encoding text $L' = r2t2es\$$. In this example, the text has not been compressed with regard to its initial and new length. But if the text contains long runs of repeated characters – which is the case with longer input text T, the compression is important. Ferragina and Manzini (2005) proposes another compression: the BW_RLX algorithm which is an extension of the *move-to-front encoding* (Bentley *et al.*, 1986). By simplification, and because the codes are implemented in Python, we have only used the run-length encoding. Indeed, the BW_RLX algorithm requires manipulation of bits and ASCII or UTF encoding at the same time, which is not a simple task for us to use on Python. Hence, our compression is less competitive since the BW_RLX encoding allows to have a smaller empirical entropy.

3 Indexing algorithms

The search pattern consists in finding the number and positions of pattern's occurrences. Ferragina and Manzini (2005) propose to count the number of occurrences before to get their positions. We denote $P_{1:p}$ a pattern to search in text T, c a character in the alphabet $\mathcal{A}$, C a dictionary of length $\mathrm{Card}\,(\mathcal{A})$ such as $\mathrm{C}\,[c]$ corresponds to the occurrence number of character c, $\mathrm{occs}\,(c, i)$ a function or data structure who corresponds to the number of occurrences of c in the prefix $L_{1:i}$ and $\mathrm{next\_letter}\,(c)$ a function returning the next letter of c in the lexicographic order. Algorithm 1 computes the values of $\mathcal{F}^{(i)}$ and $\mathcal{L}^{(i)}$, which corresponds respectively to the first and last row in $\mathcal{M}$ prefixed by suffix pattern $P_{i:p}$. More generally, it returns only the values of $\mathcal{F}^{(1)}$ and $\mathcal{L}^{(1)}$ if and only if the first is inferior or equal to the second. In this algorithm, we start with the last letter of P. The first row prefixed by the letter P_p is $\mathrm{C}\,[c]+1$ since the rows are lexicographically ordered and hence it is the number of characters smaller than P_p plus one. The reasoning is same for computing the last row. Then, at each iteration, we enlarge the word of one letter, and we update the first and last row index prefixed by the new word. The new terms added for update the indices $\mathcal{F}^{(i)}$ and $\mathcal{L}^{(i)}$ is the term in the function $\mathrm{occs}\,(c, i)$. This procedure is repeated until the enlarged word corresponds to the pattern P or $\mathcal{F}^{(i)} > \mathcal{L}^{(i)}$. In this last case, there is no pattern P in text T.

Example 2. As the previous example 1, let us consider the input text $T = tester$ and the index $L = rttees\$$. Here we have $C[\$] = 0, C[e] = 1, C[r] = 3, C[s] = 4, C[t] = 5$. Let us search the occurrence number of pattern $P = te$ in T. First, we obtain:

$$\begin{aligned} \mathcal{F}^{(2)} &= \mathrm{C}\,[e] + 1 = 2 \\ \mathcal{L}^{(2)} &= \mathrm{C}\left[\mathrm{next\_letter}\,(e)\right] = 3 \end{aligned}$$

Algorithm 1 backward search

Set L the BWT index of the input string T, P the pattern to search
Initialize $c \leftarrow P_p$, $\mathcal{F}^{(p)} \leftarrow \mathrm{C}\left[c\right] + 1$, $\mathcal{L}^{(p)} \leftarrow \mathrm{C}\left[\mathrm{next\_letter}\left(c\right)\right]$
for $i = p - 1 : 1$ **do**
  $c \leftarrow P_i$
  $\mathcal{F}^{(i)} \leftarrow \mathrm{C}\left[c\right] + \mathrm{occs}(c, \mathcal{F}^{(i+1)} - 1) + 1$
  $\mathcal{L}^{(i)} \leftarrow \mathrm{C}\left[c\right] + \mathrm{occs}(c, \mathcal{L}^{(i+1)})$
  if $\mathcal{F}^{(i)} > \mathcal{L}^{(i)}$ **then**
    return$\{\emptyset\}$
  end if
end for
return $\left(\mathcal{F}^{(1)}, \mathcal{L}^{(1)}\right)$

Indeed, the matrix $\mathcal{M}$ (see Figure 1) is prefixed by $P_2 = e$ at the second and third line. Then we compute the first and last line prefixed by $P_{1:2} = te$ as follows:

$$\begin{aligned} \mathcal{F}^{(1)} &= \mathrm{C}\left[t\right] + \mathrm{occs}\left(t, 1\right) + 1 = 6 \\ \mathcal{L}^{(1)} &= \mathrm{C}\left[t\right] + \mathrm{occs}\left(t, 3\right) = 7 \end{aligned}$$

because we have $\mathrm{occs}\left(t, 1\right) = 0$ and $\mathrm{occs}\left(t, 3\right) = 2$. We observed that effectively, the sixth and seventh rows in $\mathcal{M}$ are prefixed by te. The number of pattern occurrences is $\mathcal{L}^{(1)} - \mathcal{F}^{(1)} + 1 = 2$.

Now that we know how to count the occurrence number of a pattern P in a text T, we may ask where it appears in this text. Ferragina and Manzini (2005) proposes to mark some rows of matrix $\mathcal{M}$ such as we know their position in text T. For that, we can mark the rows whose the index is a multiple of a parameter η – in addition to the first one. Furthermore, the unknown position of row i is not a problem. Indeed, we can use a property of the BWT which is the *last-to-first column mapping*. It allows to move from the last column L to the first column F in matrix $\mathcal{M}$:

$$\mathrm{LTF}\left(i\right) = \mathrm{C}\left[L_i\right] + \mathrm{occs}\left(L_i, i\right)$$

This function is verified since the i-th character c in text F is also the i-th character c in text L. Hence, we can use this function to move back in L to its previous letter. Algorithm 2 performs the mapping between L and T. For localize the positions of a pattern P, we must (1) run Algorithm 1 for getting $\mathcal{F}$ and $\mathcal{L}$ values and then (2) run Algorithm 2 on each position in the range $\mathcal{F}$ to $\mathcal{L}$. The time complexity is linear with respect to the length of pattern P and the occurrence number $\mathcal{L}^{(1)} - \mathcal{F}^{(1)} + 1$.

Algorithm 2 L-to-T mapping

Set L the BWT index of the input string T, $i^{(0)}$ an integer corresponding to a position in L, $\mathcal{D}_m$ a data structure of marked rows r_i
Initialize iter $\leftarrow 0$
while $r_i \notin \mathcal{D}_m$ **do**
  iter $\leftarrow$ iter $+ 1$
  $i^{(\mathrm{iter})} \leftarrow \mathrm{LTF}\left(i\right)$
end while
return $r_{i^{(\mathrm{iter})}} + \mathrm{iter}$

Example 3. Let us consider the input text $T = tester$. As seen in Table 1, we have $F = \$eerstt$. Let us suppose that $\eta = 3$. We have the following data structure $\mathcal{D}_m$ mapping between L and T: $r_1 : 6, r_3 : 1, r_6 : 3$. Let us suppose that we search the position in text T of the fourth row in text L. We have: $\text{LTF}(4) = \text{C}[e] + \text{occs}(e, 4) = 1 + 1 = 2$. So we move to the second row and we compute the row containing the character preceding it: $\text{LTF}(2) = \text{C}[t] + \text{occs}(t, 2) = 5 + 1 = 6$. As we know that $r_6 : 3$, we can break. We have repeated the procedure two times. Hence the fourth character in text L corresponds to the fifth character in text T since we have: $r_6 + 2$.

4 Conclusion

In their article, Ferragina and Manzini (2005) propose to use two index data structures: The first one is only based on the *Burrows-Wheeler transformation* (Burrows and Wheeler, 1994) while the second one combine it with the LZ78 algorithm (Ziv and Lempel, 1978). Here we have only describe the first one which is an interesting choice in terms of tradeoff between time and spatial complexity. Indeed, it does not take lot of memory space since it is compressed. Moreover, the full-text index have a time complexity linear with respect to the size of the pattern and the number of occurrences. Nevertheless, the spatial complexity grows exponentially with respect to the size of the alphabet[2]. In the case of bioinformatics, this is not a real problem since the alphabet is small, even if we are considering amino acids instead of nucleotides. The index proposed here have some limits: it cannot search for similar but different patterns and a modification in the text implies a full updating of the index.

About our implementation of algorithms[3], some simplifications have been made. For instance, Ferragina and Manzini (2005) proposes to use a B packed-tree for the data structure D_m. In our case, we have store all mappings between the indices of L and T with an array by considering a step $\eta = 1$. Furthermore, it takes more memory space in our case. Also, we have implemented the compression with the *run-length encoding* instead of the *BW_RLX* algorithm[4]. Hence, the realised implementation is not competitive in terms of spatial complexity but they give a preliminary code for indexing.

[2]This exponential spatial complexity is due to the computation of $\text{occs}(c, i)$ since it requires to store some values of occurrence number for each character c.

[3]We recall that this implementation is available here.

[4]It still gives good results. We have performed this compression on a text of 150000 characters which has been compressed by 40%.

References

Bentley, J. L., Sleator, D. D., Tarjan, R. E., & Wei, V. K. (1986), A Locally Adaptive Data Compression Scheme, *Communications of the ACM*, 29(4), pp. 320-330.

Burrows, M., & Wheeler, D. (1994), A Block-Sorting Lossless Data Compression Algorithm, *In Digital SRC Research Report*.

Ferragina, P., & Manzini, G. (2005), Indexing Compressed Text, *Journal of the ACM (JACM)*, 52(4), pp. 552-581.

Kosaraju, S. R., & Manzini, G. (2000), Compression of Low Entropy Strings with Lempel–Ziv Algorithms, *SIAM Journal on Computing*, 29(3), pp. 893-911.

Ziv, J., & Lempel, A. (1978), Compression of Individual Sequences via Variable-Rate Coding, *IEEE transactions on Information Theory*, 24(5), pp. 530-536.