Universidade Nova de Lisboa

Faculdade de Ciências e Tecnologia

Departamento de Informática

Master's Thesis in Computer Engineering

2nd Semester, 2009/2010

Burrows-Wheeler Transform in Secondary Memory

28038 Sérgio Miguel Cachucho Pereira

Advisor

Professor Luís Manuel Silveira Russo

November 23, 2010

Student Number: 28038
Name: Sérgio Miguel Cachucho Pereira

Dissertation Title:
Burrows-Wheeler Transform in Secondary Memory

Keywords:

- Suffix arrays
- External sorting
- Heap
- Pattern matching
- Indexes

Palavras-Chave:

- Arrays de sufixos
- Ordenação em memória secundária
- Heap
- Identificação de padrões
- Índices

Abstract

A suffix array is an index, a data structure that allows searching for sequences of characters. Such structures are of key importance for a large set of problems related to sequences of characters. An especially important use of suffix arrays is to compute the Burrows-Wheeler Transform, which can be used for compressing text. This procedure is the base of the UNIX utility bzip2. The Burrows-Wheeler transform is a key step in the construction of more sophisticated indexes. For large sequences of characters, such as DNA sequences of about 10 GB, it is not possible to calculate the Burrows-Wheeler transform in an average computer without using secondary memory. In this dissertation we will study the state-of-the-art algorithms to construct the Burrows-Wheeler transform in secondary memory. Based on this research we propose an algorithm and compare it against the previous ones to determine its relative performance. Our algorithm is based on the classical external Heapsort. The novelty lies in a heap that is especially designed for suffix arrays, which we call String Heap. This algorithm aims to be space-conscious, while trying to handle the disk access dominance over main memory access. We divide our solution in two parts, splitting and merging suffix arrays, the latter is the main application of the String Heap. The merging part produces the BWT, as a side effect of merging a set of partial suffix arrays of a text. We also compare its performance against the other algorithms. We also study a second version of the algorithm that accesses secondary memory in blocks.

Resumo

Um array de sufixos é um índice, uma estrutura de dados que facilita a pesquisa de sequências de caracteres. Estas estruturas são centrais para um grande conjunto de problemas sobre sequências de caracteres. Uma aplicação particularmente importante dos vectores de sufixos é a construção da transformação de Burrows-Wheeler, que permite comprimir um texto. Este procedimento é a base da aplicação bzip2. Calcular a transformação de Burrows-Wheeler é também um passo crucial na construção de índices mais sofisticados. Para sequências de letras muito grandes, como sequências de ADN de cerca de 10 GB, não é eficiente calcular a transformação de Burrows-Wheeler num computador mediano sem usar memória secundária. Nesta dissertação irão ser estudados algoritmos de construção da transformação de Burrows-Wheeler em memória secundária ao nível do estado-de-arte. Com base nessa pesquisa iremos propor um algoritmo e compará-lo com os algoritmos estudados de forma a determinar o seu desempenho relativo. O nosso algoritmo baseia-se no clássico Heapsort com recurso a memória secundária. A inovação é o heap, que é especialmente vocacionado para arrays de sufixos, por isso damos-lhe o nome de String Heap. O algoritmo proposto visa poupar memória principal, assim como lidar com o facto de os acessos a disco serem dominantes em relação aos acessos a memória. Dividimos a solução em duas partes, divisão e junção de arrays de sufixos; a segunda parte é a principal aplicação do String Heap. A parte de junção produz a Transformação de Burrows-Wheeler como efeito secundário da junção de um conjunto de arrays de sufixos parciais de um texto. Também comparamos os resultados desta parte face aos outros algoritmos. Estudamos ainda uma segunda versão do algoritmo, que acede a memória secundária em blocos.

Table of Contents

1. Introduction 1
  1.1 Motivation 1
  1.2 Problem Description 2
2. Related Work 4
  2.1 Terminology 4
  2.2 Sorting Algorithms 8
  2.3 Quickheap 10
  2.4 Suffix Arrays in Secondary Memory 16
3. Our solution 24
  3.1 Text Generator 24
  3.2 Splitting Part 25
  3.3 Merging Part 27
4. Algorithm Implementation 33
  4.1 Initial Parameters 33
  4.2 Splitter 33
  4.3 Merger 34
  4.4 BWT Builder 43
5. Algorithm Execution 44
6. Experimental Validation 53
  6.1 Prototype Configuration 53
  6.2 Version with Blocks of Suffixes 60
  6.3 State of the art 63
7. Conclusions 78
8. Bibliography 80

Table of Figures

Figure 1. *Example of how to obtain the suffix array of a given text.* 5

Figure 2. *Example of the Burrow-Wheeler Transform.* 6

Figure 3. *Example of the isomorphism between Quicksort and binary search trees.* 8

Figure 4. *The ternary search tree of the example of Figure 3.* 9

Figure 5. *Example of the Quickheap.* 13

Figure 6. *Example of inserting a new element in a Quickheap.* 16

Figure 7. *Definition of three different terms related to the LCP.* 17

Figure 8. *The Doubling Algorithm and its pipeline representation.* 19

Figure 9. *The DC3-algorithm.* 21

Figure 10. *The data-flow graph for the DC3-algorithm.* 22

Figure 11. *Illustration of the splitting part of our algorithm.* 25

Figure 12. *Illustration of the merging part of our algorithm.* 27

Figure 13. *Suffixes and the respective indexes for the text "mississippi".* 44

Figure 14. *Representation of the heap data structures after inserting the first three suffixes.* 46

Figure 15. *Representation of the heap data structures before extracting the first suffix (12).* 47

Figure 16. *Representation of the heap data structures after inserting a new suffix (11).* 48

Figure 17. *Representation of the heap data structures before extracting a suffix (9).* 49

Figure 18. *Representation of the heap data structures before inserting a new suffix (6).* 49

Figure 19. *Representation of the heap data structures before extraction of a suffix (6).* 50

Figure 20. *Representation of the heap data structures before extraction of a suffix (10).* 51

Figure 21. *Representation of the heap data structures before extraction of a suffix (8).* 51

Figure 22. *Representation of the heap after all extractions.* 52

Figure 23. *Table with the experimental results for different values of partial suffix arrays.* 54

Figure 24. *Chart illustrating the data from Figure 23 that refers to memory usage.* 54

Figure 25. *Chart illustrating the data from Figure 23 that refers to running time.* 55

Figure 26. *Table with the experimental results for different buffer sizes.* 57

Figure 27. *Chart showing the data from Figure 26 that refers to memory usage.* 57
Figure 28. *Table with the experimental results for different numbers of suffixes in the heap.* 58
Figure 29. *Chart with the data from Figure 28 that refers to memory usage.* 59
Figure 30. *Chart with the data from Figure 28 that refers to running time.* 59
Figure 31. *Table with the experimental results for different LCP values.* 61
Figure 32. *Chart showing the data from Figure 31 that refers to the memory usage.* 61
Figure 33. *Chart showing the data from Figure 31 that refers to the running time.* 62
Figure 34. *Table with the experimental results for the three algorithms with the file dblp.xml.100MB.* 64
Figure 35. *Chart showing the data from Figure 34 that refers to the heap without blocks.* 65
Figure 36. *Chart showing the data from Figure 34 that refers to the heap with blocks.* 65
Figure 37. *Chart showing the data from Figure 34, comparing the running times of the three algorithms, and the merging part of both versions of our algorithm.* 66
Figure 38. *Table with the experimental results for the three algorithms with the file dna.100MB.* 67
Figure 39. *Chart showing the data from Figure 38 that refers to the heap without blocks.* 67
Figure 40. *Chart showing the data from Figure 38 that refers to the heap with blocks.* 68
Figure 41. *Chart showing the data from Figure 38, comparing the running times of the three algorithms, and the merging part of both versions of our algorithm.* 69
Figure 42. *Table with the experimental results for the three algorithms with the file proteins.100MB.* 70
Figure 43. *Chart showing the data from Figure 42 that refers to the heap without blocks.* 70
Figure 44. *Chart showing the data from Figure 42 that refers to the heap with blocks.* 71
Figure 45. *Chart showing the data from Figure 42, comparing the running times of the three algorithms, and the merging part of both versions of our algorithm.* 72
Figure 46. *Table with the experimental results for the three algorithms with the file dblp.xml.200MB.* 73
Figure 47. *Chart showing the data from Figure 46 that refers to the heap without blocks.* 73
Figure 48. *Chart showing the data from Figure 46 that refers to the heap with blocks.* 74
Figure 49. *Chart showing the data from Figure 46, comparing the running times of the three algorithms, and the merging part of both versions of our algorithm.* 74

Figure 50. *Table with the experimental results for the three algorithms with the file dblp.xml.300MB.* 75
Figure 51. *Chart showing the data from Figure 50 that refers to the heap without blocks.* 76
Figure 52. *Chart showing the data from Figure 50 that refers to the heap with blocks.* 76
Figure 53. *Chart showing the data from Figure 50, comparing the running times of the three algorithms, and the merging part of both versions of our algorithm.* 77

1. Introduction

1.1 Motivation

Nowadays text processing is a significant and fast expanding area within computer science. There are several applications, studies and research areas that depend on the existence of fast text processing algorithms. One such area is biology. Having machines that automatically search a DNA sequence and look for patterns in a short amount of time is essential for researchers, and has greatly contributed to important discoveries that would otherwise be impossible. These machines use sophisticated text processing algorithms, capable of handling long DNA sequences in a very short time. Another research area that requires algorithms is automated text translation. Like in the previous example, it also needs to process text in very limited amounts of time, as well as to look for patterns at the same time.

These algorithms are most of the times built on top of some state-of-the-art string processing data structures. One such structure is the suffix array, which is an array containing all the suffixes of a given text, in lexicographical order. This makes the process of looking for a pattern much more efficient, since the suffixes related to the query are stored in consecutive positions in the array. Furthermore being sorted makes it possible to use an $O(\log n)$ algorithm to look for a pattern. This procedure contrasts with the need of scanning all the text to search for the wanted patterns, which would be less time efficient for large texts. These characteristics make suffix arrays especially suitable to solve problems that require handling large amounts of data, such as the ones mentioned above. To solve this sort of problems searching all the text for patterns is not an option, because that would take too much time. Therefore having a data structure that can obtain the same results in considerably less time is critical. This makes researching better ways to implement it an important issue.

The main setback of suffix arrays is that most of the existing algorithms that use secondary memory are only appropriate for quite small input texts, which in many cases makes them less suitable to solve certain problems. This happens mainly because those algorithms need to store all the suffixes in main memory, alongside with the text, which obviously restricts them to be used only with texts that are much smaller than the memory of the computer where it will be processed. As mentioned before, the use of suffix arrays makes certain operations much faster, which is especially significant with problems that require handling large texts, such as processing a DNA sequence or a book. Although, most of the current implementations of algorithms to build suffix arrays using secondary memory can't themselves handle large input texts, reason why the use of suffix arrays are yet to reach its full potential.

This work aims to improve this situation. We will propose an algorithm that builds the suffix array for a given text, but that doesn't need all the suffixes to be stored in main memory at the same time, instead just a small part of them will, all others will remain in secondary memory. Achieving this goal will make the use of suffix arrays wider, especially for those problems that require handling large amounts of data.

1.2 Problem Description

Since suffix arrays are very useful indexes to solve problems that require handling text and looking for patterns in it, the motivation for building suffix arrays is clearly established, and this is the main focus of this work. A very important side effect that can be obtained from our approach is the computation of the Burrows-Wheeler transform of the text, which is a crucial piece for more sophisticated compressed indexes.

The main challenge of building the suffix array of a text is to obtain a solution that is suitable for large inputs. This issue is raised mainly due to memory constraints. The existing algorithms need to store in main memory both the suffixes and the text itself, however in this work we will present an algorithm whose working space is essentially the text. This means we need to use secondary memory to store the suffixes.

There are two issues to be taken into account when building suffix arrays: to be space-conscious and streaming. We define a space-conscious solution as one that doesn’t need to store the whole working data in main memory, using for that purpose the secondary memory. In the context of the suffix arrays, a solution that stores in main memory both the text and the suffix array is not an example of a space-conscious solution. Otherwise, a solution, like ours, that stores in main memory only the text and a small part of the suffixes, keeping the rest of the suffixes in secondary memory is a space-conscious solution.

A streaming solution is one that runs from the beginning of its execution up to the end without blocking. The use of secondary memory raises the issue of blocking I/Os, because the disk access is dominant over main memory access in the complexity of these algorithms, thus at some point in the execution it has to block waiting for new data from the disk. However we want our algorithm to reduce this problem by defining an efficient I/O policy to prevent as much as possible our algorithm from blocking due to disk accesses.

Most of the existing algorithms don’t have both of these characteristics. There are some that are streaming because everything is stored in main memory, but which, because of that fact, are not space-conscious. In the other hand, the ones that are space-conscious fail to be streaming, having poor policies to deal with I/O accesses.

2. Related Work

2.1 Terminology

This work is about text and operations on it, hence strings are mentioned very often. In this context, a string and a text are the same thing, both mean a sequence of characters. When talking about strings, it is common to mention some related concepts. The size of a string is the number of characters it contains. The alphabet is the set containing all characters used in a text. Sorting, or ordering, is the process of arranging items in some ordered sequence. In this work will be used lexicographical order to sort the strings (suffixes) contained in a given array.

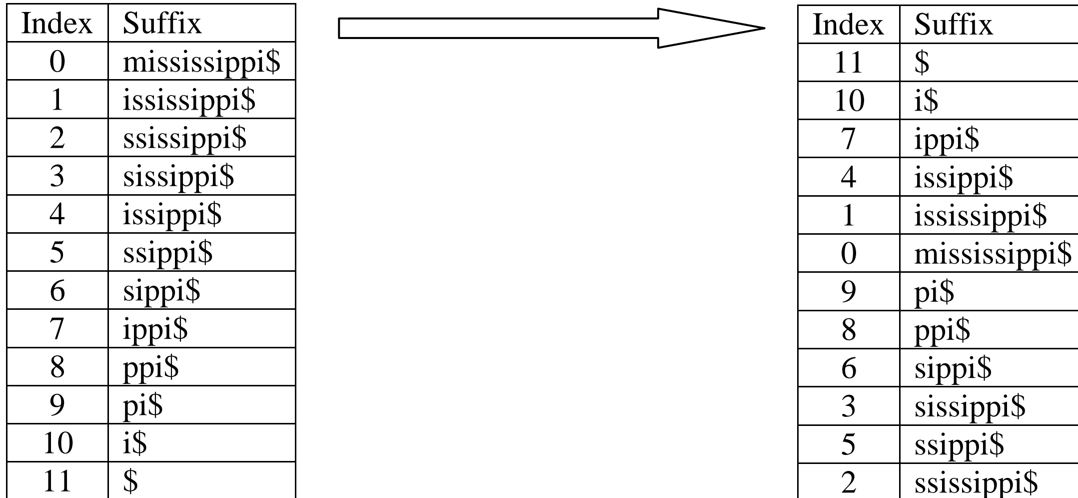
2.1.1 Suffix Arrays

One important focus of this work is to obtain the suffix array of a given text. A suffix of a text is a substring containing all the text characters from a given position up to the last one, in the exact same order as found in the given text. Moreover, a suffix array is a lexicographically sorted array, containing all the suffixes of a given text, thus a suffix array of a text with n characters is an array containing its n suffixes sorted. The suffix array of a text $T_{1,n}$ is an array $A[0,n[$ containing a permutation of the interval $[0,n[$, such that $T_{A[i],n} < T_{A[i+1],n}$ for all $0 \leq i < n-1$, where "<" between strings is the lexicographical order [1].

To obtain the suffix array of a text one needs to obtain every suffix from the original text and to store them in an array. Note that it is enough to store pointers to the string, not the suffixes themselves. After this step all the suffixes are in the array, but still unsorted. To sort them one needs to use a sorting algorithm. Of course the choice of the algorithm to use is a key decision, since it represents most of the complexity in the process of getting the suffix array from a text. I will discuss this later in this dissertation.

Figure 1 contains an example of how to obtain the suffix array from a given text, in this case mississippi$. The illustration shows the original order of the suffixes of the text (on the left), and the resulting suffix array (on the right).

Text = mississippi$



Index	Suffix
0	mississippi$
1	ississippi$
2	ssissippi$
3	sissippi$
4	issippi$
5	ssippi$
6	sippi$
7	ippi$
8	ppi$
9	pi$
10	i$
11	$

Index	Suffix
11	$
10	i$
7	ippi$
4	issippi$
1	ississippi$
0	mississippi$
9	pi$
8	ppi$
6	sippi$
3	sissippi$
5	ssippi$
2	ssissippi$

Suffix Array = {11, 10, 7, 4, 1, 0, 9, 8, 6, 3, 5, 2}

Figure 1. *Example of how to obtain the suffix array of a given text.*

Puglisi et al. [6] made an extended researched about suffix arrays, from the origins until nowadays. Suffix arrays were introduced in 1990 by Manber & Myers, along with algorithms for their construction and use. From then to now suffix arrays have become the data structure of choice for solving many text processing problems. The designers of algorithms to build suffix arrays want them to have minimal asymptotic complexity, to be fast “in practice”, even with large real-world data inputs, and to be lightweight, which means use a small amount of working storage beyond the space used by the text. Until the date of this paper there are no algorithms that achieve all the three goals proposed.

2.1.2 The Burrows-Wheeler Transform

The Burrows-Wheeler Transform (BWT) is a transformation from strings to strings that can be reversed. This operation makes the transformed text easier to compress, by local optimization methods, due to the much more frequent occurrence of repeated characters in the BWT than in the original text. The most usual way of getting the BWT of a text is by constructing its suffix array.

Text	All rotations	Sort the rows	BWT
mississippi$	mississippi$	$mississippi	ipssm$pissii
	ississippi$m	i$mississipp	
	ssissippi$mi	ippi$mississ	
	sissippi$mis	issippi$miss	
	issippi$miss	ississippi$m	
	ssippi$missi	mississippi$	
	sippi$missis	pi$mississip	
	ippi$mississ	ppi$mississi	
	ppi$mississi	sippi$missis	
	pi$mississip	sissippi$mis	
	i$mississipp	ssippi$missi	
	$mississippi	ssissippi$mi	

Figure 2. *Example of the Burrow-Wheeler Transform. The BWT of the text "mississippi$" is "ipssm$pissii".*

As Figure 2 shows, obtaining the Burrows-Wheeler Transform of a given text is a simple process. The algorithm creates an array with size equal to the length of the text given. In the first position of that array is placed the original text and in the remaining positions are placed all the rotations possibly obtained from that text. By rotation I mean each character $T[i]$ of the text go to the position $T[i-1]$, except the character $T[0]$, which goes to the position $T[n-1]$, where n is the length of the text and the size of the array as well. This procedure is performed n-1 times, i.e. all

the possible rotations are obtained, except the original text, which is already in the first position of the array and would be obtained again in the n-th rotation. The second step of the algorithm is to sort the rows that resulted from the rotations, i.e. sorting the resulting array by lexicographical order, which means that it is similar to a suffix array. The Burrows-Wheeler Transform of the text given is a string containing the last characters of each of the rotations, in the sorted order obtained in the second step. The string obtained in the end of the algorithm has the same size and contains the same characters as the original text, but in the BWT they are in a different order, according to the steps described.

If one compares Figure 2 with Figure 1, illustrating the BWT and the suffix array of a text, respectively, there are obvious similarities. In the first step, the rotations part, the BWT algorithm is equal to the suffix array construction, except that in the case of suffixes the first character of the text is dropped instead of moved to the end of the text. Yet the structure is exactly the same. The second step, the sorting part, is common to both the BWT and the suffix array construction. The last part, extracting the last characters to make the resulting string is exclusive to the BWT algorithm, yet all other steps are common to both of them.

Having such similarities, it is natural to use algorithms to construct suffix arrays also to obtain the Burrows-Wheeler Transform, and that is what we are going to do in this work. I will frequently mention algorithms to build the suffix array of a given text, but that discussion applies also to obtaining the BWT. Furthermore, the algorithm we will create to build suffix arrays can be easily adapted to obtain the BWT, since these are similar problems.

2.2 Sorting Algorithms

2.2.1 **Quicksort**

In any discussion on sorting algorithms, Quicksort is a mandatory reference, since it is very fast, and consequently widely used. Quicksort is a divide and conquer algorithm, it consists in choosing a partitioning element and then permuting the elements such that the lesser ones go to one group, and the greater ones to the other. Hence dividing the original problem in two sub problems which are solved recursively until the list is sorted.

Dividing each sub problem in two new sub problems resembles the structure of a binary tree, where each node references two subsequent nodes. This means there is an isomorphism between them. If we link each Quicksort partitioning element with the two partitioning elements of the subsequent subarrays, eventually we obtain a structure similar to a binary search tree. This isomorphism means that the results of analyzing binary search trees apply to the structure of Quicksort. This is particularly interesting if one observes that the process of building a binary search tree is analog to the process of sorting an array using Quicksort.

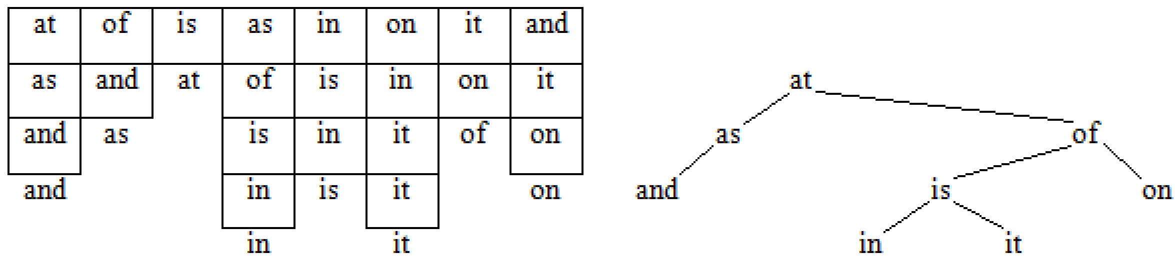


Figure 3. *Example of the isomorphism between Quicksort and binary search trees.*

2.2.2 Multikey Quicksort

Knowing the advantages of using a divide and conquer algorithm (one with a tree based structure), algorithm designers found out that a ternary partitioning method was more desirable, and eventually a Multikey Quicksort appeared [2]. This algorithm is isomorphic to ternary search trees, in the same way that Quicksort is to binary search trees. This Multikey Quicksort is based on the traditional Quicksort, as it also groups the lesser elements to one side, and the greater ones to the other. The key difference is the treatment given to the elements equal to the partitioning element. In the traditional Quicksort they are either put in one side or in the other, depending on how the programmer did it, however in the Multikey Quicksort those elements are handled in a different way. They form a third group, between the lesser and the greater. This group contains the partitioning element and every other equal to it. Given this fundamental difference, ternary search trees have the same features as binary search trees, but instead of two they contain three pointers, one to the left child (lesser elements), one to the middle child (equal elements) and one to the right child (greater elements).

There is another important difference between the traditional Quicksort and the Multikey Quicksort proposed by Bentley et al. In Quicksort, elements link to other elements, making a structure similar to a binary search tree like the one illustrated in Figure 3. However, in the Multikey Quicksort elements are compared character by character, resulting in a ternary search tree like the one illustrated in Figure 4, below.

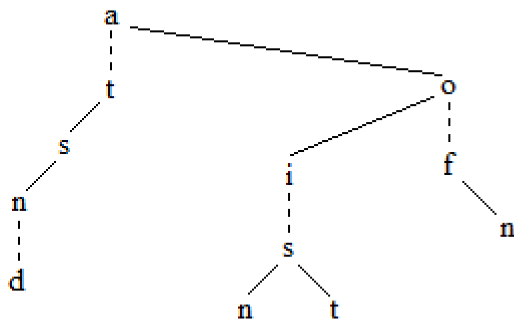


Figure 4. *The ternary search tree of the example of Figure 3.*

As seen, the character by character comparison results in a tree branched in each new character, contrasting with the example of Quicksort, in Figure 3, where the tree is branched by element. This aspect is especially important when comparing strings, because the character by character approach makes it unnecessary to compare all characters between two strings, since it is enough to compare until the first non common character, which most of the times is before the end of one of the strings. This number of positions strictly necessary to compare is called the Longest Common Prefix between two strings, and is explained later in this section.

Bentley et al. [2] first tested the Multikey Quicksort, to which they introduced some enhancements. One of those was partitioning around the median of three elements, resulting in a more balanced tree, which is a simple but effective method of reducing the risk of the worst case to happen. In a performance test they tested the original Multikey Quicksort, against their tuned algorithm and the fastest sorting algorithm at that time. Their tuned Multikey Quicksort showed to be a lot faster than the original algorithm, yet it wasn't as fast as a highly tuned Radix Sort, which was the fastest algorithm at that time. Even then, in certain contexts, their tuned Multikey Quicksort proved especially suitable, and in those contexts it showed to be even faster than the mentioned Radix Sort.

2.3 Quickheap

On this discussion on sorting algorithms, the mentioned ones are meant to sort entire arrays. Although, it is fruitful for the purpose of this work to also mention those cases in which we only need to sort a certain number of elements, not the whole array. The latter procedure is called Partial Sorting. To do it we have to obtain the smallest element of the array as many times as the number of elements we want to get. The process of getting the smallest element from an array is called Incremental Sorting, and this problem can be solved using a heap. In this work one such heap will be reviewed and used: the Quickheap (illustrated in Figure 5).

Quickheap is a practical and efficient heap for solving the Incremental Sorting problem. It solves the problem in $O(n + k \log k)$ time, where n is the size of the array from which we want to get the smallest k elements [3]. Quickheap is also very fast on the insertions of new elements to the array, which is a particularly useful feature to this work, and will be covered with more focus later in this dissertation.

Like Quicksort and all other divide and conquer algorithms, Quickheap consists in recursively partitioning the array. The process of dividing a partition into two is done by picking an element, which is called the partitioning pivot. Navarro et al. [3] used the first element of a partition to be its partitioning pivot, so in the example of Figure 3 that is also the procedure, but further in this work some enhancements to the pivot selection criteria will be introduced, such as picking the median of three random selected elements to be the pivot.

Quickheap is intended to sort the array incrementally, which means that it returns one element at a time. The operation of getting only the smallest element of the array is called *Quickselect*, and one can say that Quickheap is a sequence of *Quickselect* calls. To perform those *Quickselect* calls, it is required an auxiliary structure, which is used to store the partitioning pivots in such a way that the last added pivot is always on the top position. To perform this way, the best structure to use is a stack, and for the purpose of this dissertation it is frequently mentioned as S. In the beginning of the Quickheap execution, i.e. before the first *Quickselect* call is performed, the stack contains only one element, which is used as a sentinel.

Sentinels are used to maintain structures homogeneous. Using this special value avoids having to constantly make assertions to detect events such as null pointers, which if not properly detected and handled would most likely make the program crash. In this work, the sentinel used in the stack guarantees that the stack will never be empty, thus it makes erasable all the lines used to prevent problems occurred because of such an event. Though, this is an equally effective method to avoid such crashes. Navarro et al. [3] used as sentinel the value of the array's length, which is a pointer to a not valid position. This would avoid having null pointer occurrences when the stack contains no valid pivots, but would produce another tricky situation: pointing to a null position. For the same reasons mentioned before, in this work we decided to use a new sentinel to solve this issue. Thus, we added to the array's first null position a value, which in this context is suitable to be the maximum integer value, so that any value will always be smaller than it. This

is especially practical in this work, since such a comparison is done very often and this requires only one *if* statement to perform the task and avoiding crashes as well.

Creating the stack, adding the sentinel and adding the second sentinel to the array are the first steps when *Quickselect* is called for the first time. After those initializations, the next operation is to check the value stored on the top of the stack, and check it against the array. The array is stored in a circular way, so, if on the top of the stack is stored the index of the first element of the array, it is just popped and returned. If the top value points to any other element of the array, then the procedure is different. The array positions between the first and the one pointed by the top of the stack are the partition that will be repartitioned. One element is picked from this partition to be the partitioning pivot (by the median of three elements, as described above). At this point, every element of this partition is compared against the picked pivot, and is either put on its left, if lesser, or on its right, if greater. The pivot is put in the first position after the elements smaller than it, and that position's index is added to the top of the stack. This procedure is recursively repeated to each newly created partition, until the case in which the element pointed is the first, and immediately popped and returned.

In the end of each recursion of *Quickselect* the stack is changed: when the top value points to the first position, the top is popped; if, otherwise, the stack is repartitioned, the new found pivot is added to the top of the stack.

Figure 5, in the next page, shows the first three *Quickselect* executions of a Quickheap. It shows the state of both the array and the stack at the beginning of the first execution, and how they change as the array is partitioned and new pivots are found.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
51	81	74	12	58	92	86	25	67	33	18	41	49	63	29	37

$S = \{16\}$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
33	37	29	12	49	41	18	25	51	67	86	92	58	63	74	81

$S = \{8, 16\}$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
18	25	29	12	33	41	49	37	51	67	86	92	58	63	74	81

$S = \{4, 8, 16\}$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
12	18	29	25	33	41	49	37	51	67	86	92	58	63	74	81

$S = \{1, 4, 8, 16\}$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
12	18	29	25	33	41	49	37	51	67	86	92	58	63	74	81

$S = \{0, 1, 4, 8, 16\}$

Here ends the first *Quickselect*, returning the smallest element (12). The second *Quickselect* drops the last added index from the stack, and gets the next smallest element.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
18	29	25	33	41	49	37	51	67	86	92	58	63	74	81

$S = \{1, 4, 8, 16\}$

The second *Quickselect* ends returning the smallest element (18). In the third one the process is repeated, returning the next smallest value (25) in its end.

2	3	4	5	6	7	8	9	10	11	12	13	14	15
29	25	33	41	49	37	51	67	86	92	58	63	74	81

$S = \{4, 8, 16\}$

2	3	4	5	6	7	8	9	10	11	12	13	14	15
25	29	33	41	49	37	51	67	86	92	58	63	74	81

$S = \{3, 4, 8, 16\}$

2	3	4	5	6	7	8	9	10	11	12	13	14	15
25	29	33	41	49	37	51	67	86	92	58	63	74	81

$S = \{2, 3, 4, 8, 16\}$

Figure 5. *Example of the Quickheap getting recursively the smallest elements of the given array. The algorithm continues making recursive calls until all elements have been returned.*

2.3.1 **Quickheap Implementation**

To implement a Quickheap, four structures are required: an array *heap* to store the elements; a stack S to store the indexes of the pivots partitioning the *heap*; an integer *first* indicating the first cell of the heap; and an integer n indicating the size of the heap. These are the data structures needed to perform the basic operations with the lowest possible complexity.

When creating the Quickheap, the mentioned structures must be initialized. The value of n must be equal to or greater than the number of elements that will be added to the *heap*. The array *heap* is an array with size equal to n. The auxiliary stack S is a new stack, and at this time it remains empty. The value of *first* is initialized with 0. If one wants to create the Quickheap from an array, then the given array is copied to the array *heap* and the other attributes are updated. The value of n is the number of elements of the given array, *first* is the index of the first position of the array. And the stack S is a new stack, to which is added the index to the n-th position of the array, as a sentinel value. This operation can be done in time O(1), if the given array is directly used as *heap*.

Quickheaps are most used in contexts that require low complexity on getting the minimum element. That's because Quickheaps are structured in such a way in which this operation performs particularly fast. As described above, and illustrated in Figure 5, every time we obtain the minimum value of the *heap* it repartitions, taking the smaller elements recursively to the first positions of the *heap*, making it faster to get the minimum value. Thus, in cases when the first element of *heap* is already a pivot, it is popped and returned, making the complexity O(1). Otherwise, in the cases in which it is required to repartition the *heap*, the complexity is, in average, O(log n), because the problem's domain is recursively reduced to its half, until the first position is a pivot.

Following the above approach, if we create a Quickheap from an existing array, and then obtain minimum values from it: we first use the given array as *heap*, having a complexity O(1), and then obtain as many minimum values as we want, O(log n) for each one.

There is another approach to solve the same requirement: if we sequentially insert the elements of the given array to *heap*, one by one, and in the end obtain the wanted minimum elements from it. This procedure has a complexity O(log n) for each inserted element, totaling O(n log n) for the whole process of inserting the elements of the array to the *heap*. But after that is O(1) for obtaining each minimum value. Unsurprisingly, the first approach is more effective to solve this example. If we want all the n elements in a sorted order they have the same complexity, but if we want only some of them, the first approach only orders the *heap* partially, whereas the second one orders the whole *heap* when the insertions are performed. Even then, for the purpose of this work, the insertion operation is especially attractive, because there will be progressive insertions according to the solution presented in the introduction of this dissertation.

To insert a new element x into *heap* is required to find the partition where it fits, that means the pivot limiting that partition in the upside must be greater than x, and the pivot limiting it by the downside must be smaller than x. To begin with the process of finding that partition, we move the sentinel (element in the n-th position of *heap*) one position forward (to the (n+1)-th position), and put x to the n-th position, which is in the last partition. Then x is compared against the pivot limiting that partition by the downside, and if the pivot is greater than x they switch positions. This procedure is repeated until x is greater than a pivot, case in which the process is not repeated anymore, or x reaches the first position of *heap*, case in which there are no more pivot to compare it against. This procedure is illustrated in Figure 6, in the next page, where x is 35.

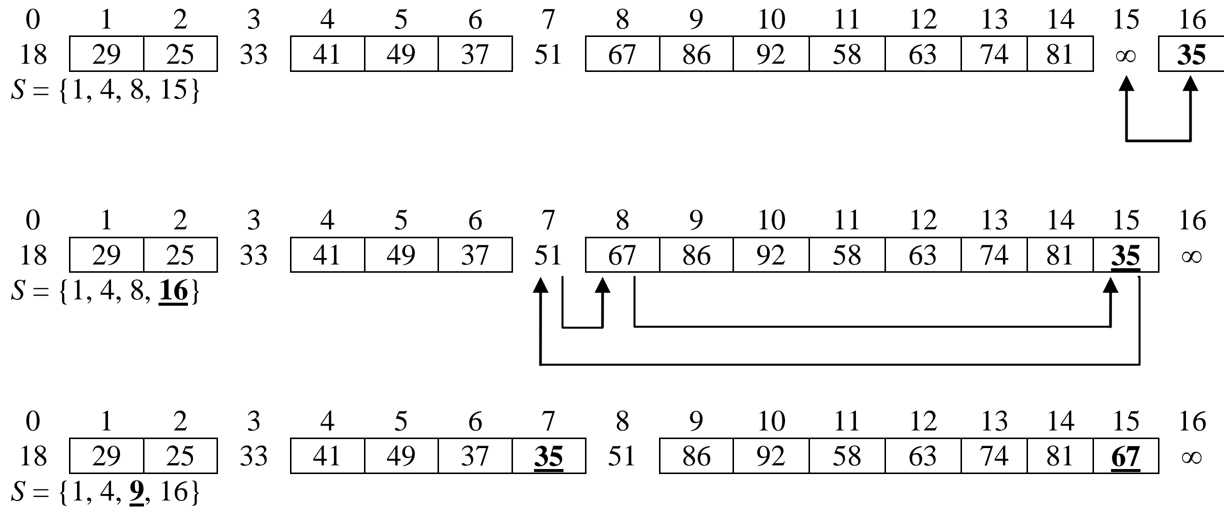


Figure 6. *Example of inserting a new element in a Quickheap. The new value (35) is compared to the pivots until one smaller than it is found.*

Navarro et al. [3] proved that operations upon Quickheaps, such as inserting an element or reorganizing the *heap* (when obtaining the minimum), cost O(log m) expected amortized time, where m is the maximum size of the Quickheap. This analysis is based on the observation that statistically Quickheaps exhibit an exponentially decreasing structure, which means that the pivots form, on average, an exponentially decreasing sequence.

The insertion of new elements in a heap is an especially important operation in the context of this work, because, as mentioned in the introduction of this dissertation, we will build a heap for suffix arrays. Those suffix arrays are going to pop elements to the heap, which are inserted in it.

2.4 Suffix Arrays in Secondary Memory

Suffix arrays are a simple data structure, which is very useful for text processing. For small inputs, it is faster to store suffix arrays in main memory, however, in this work we want the solution to be applicable to huge inputs, and for those, being stored in main memory is not an

option, since it is prohibitive because of the spatial constraints that would limit us to work with small texts. The small main memory size makes us think about a good alternative that uses secondary memory.

Up to 2007, the only secondary memory implementations of suffix array construction are so slow that measurements can only be done for small inputs [4]. For that reason, suffix arrays are rarely or never used for processing huge inputs. Instead, less powerful techniques are used, such as having an index of all words appearing in a text. However, using secondary memory to construct suffix arrays has significant potential for improvement, and eventually better techniques and algorithms will appear.

When presenting those algorithms some notation will be used. *T* always identifies the original text of which will be built the suffix array, thus $T[i]$ denotes the i-th position of the text *T*. I will also frequently mention the term LCP, which means Longest Common Prefix. The LCP is an integer value that denotes the number of characters that two string share from the initial position. For example, the LCP between “Portugal” and “Portuguese” is “Portug”, which has a length of 6 characters. It is frequent to express complexity calculations referring to the LCP value, and there are also some related terms, which we present in Figure 7, bellow. lcp(i, j) denotes the longest common prefix between SA[i] and SA[j], where SA is the suffix arrays of a text *T*.

$$maxlcp = \max_{0 \le i \le n} lcp(i, i+1)$$

$$\overline{lcp} = \frac{1}{n} \sum_{0 \le i < n} lcp(i, i+1)$$

$$\overleftrightarrow{lcp}(i) = max \begin{cases} lcp(i-1, i) \\ lcp(i, i+1) \end{cases}$$

Figure 7. *Definition of three different terms related to the LCP.*

2.4.1 A Secondary Memory Implementation of Suffix Arrays

Dementiev et al. reviewed [4], in 2008, a set of algorithms that built suffix arrays. They observed that algorithms that only used main memory would be faster than the ones using secondary memory. But since their paper is about secondary memory, they analyzed more deeply the latter ones, concluding that fast secondary memory algorithms that handle large inputs in small amounts of time are yet to appear. The existing ones have high I/O complexity, resulting in unpractical algorithms.

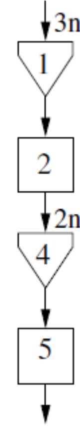
The authors considered that main memory is too expensive for big suffix arrays one may want to build, stating that disk space is two orders of magnitude cheaper than RAM. For that reason they, like us, concluded that it would be a lot worthy to come up with a fast but space-conscious algorithm to build suffix arrays using secondary memory.

In their paper Dementiev et al. presented some techniques that aim to reduce the gap between the main memory and the secondary memory algorithms, and for that purpose they presented some techniques. Those include using a double algorithm, pipelining, and a technique to discard suffixes.

One of the techniques proposed is using a doubling algorithm. The basic idea of this technique is to replace each character of the input by a lexicographic name, which respects the lexicographic order. What makes the technique proposed by Dementiev et al. better than previous secondary memory implementations is that their procedure never actually builds the resulting string of names, rather, it manipulates a sequence P of pairs, where $P[i] = (c, i)$. Each character c is tagged with its position i in the input. This sequence of pairs is sorted, what makes it easier for scanning and comparing adjacent tuples, and the process continues as the code chunk in Figure 8 shows. The doubling algorithm proposed computes a suffix array using $O(\text{sort}(n) \lceil \log maxlcp \rceil)$ I/Os and simplifies the pipelining optimization proposed.

```
Function doubling(T)
    S:= [((T[i], T[i+1]), i) : i ∈ [0, n)]                    (0)
    for k := 1 to ⌈log n⌉ do
        sort S                                                (1)
        P:= name(S)                                           (2)
        invariant ∀(c, i) ∈ P :
            c is a lexicographic name for T[i, i + 2^k)
        if the names in P are unique then
            return [i : (c, i) ∈ P]                           (3)
        sort P by (i mod 2^k, i div 2^k))                     (4)
        S:= ⟨((c, c′), i) : j ∈ [0, n),                       (5)
                (c, i) = P[j], (c′, i + 2^k) = P[j + 1]⟩

Function name(S : Sequence of Pair)
    q:= r:= 0;   (ℓ, ℓ′):= ($, $)
    result:= ⟨⟩
    foreach ((c, c′), i) ∈ S do
        q++
        if (c, c′) ≠ (ℓ, ℓ′) then r:= q;   (ℓ, ℓ′):= (c, c′)
        append (r, i) to result
    return result
```



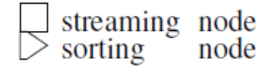


Figure 8. *The Doubling Algorithm and its pipeline representation (as presented by Dementiev et al. [4]).*

The pipeline approach proposes that computations in many secondary memory algorithms can be viewed as a data flow through a direct acyclic graph. The authors propose that this graph can be drawn with three types of nodes: file nodes, which represent data that has to be stored physically on disk; streaming nodes, which read a given sequence and output a new sequence using internal buffers; and sorting nodes, which read a sequence and output it in sorted order. The edges of this graph are labeled with the number of machine words flowing between two nodes. The pipeline representation of the Doubling Algorithm is shown in Figure 8, alongside its code.

Dementiev et al. proved that the computations of a data flow graph $G = (V = F \cup S \cup R, E)$ with edge flows $w: E \to \mathbb{R}_+$ and buffer requirements $b: V \to \mathbb{R}_+$ can be executed using $\left(\sum_{e \in E \cap (FxV \cup VxF)} scan(w(e)) + \sum_{e \in E \cap (VxR)} sort(w(e)) \right)$ I/Os. For that reason they conclude

that it is possible to implement the Doubling Algorithm using ($sort(5n)\lceil\log(1 + maxlcp)\rceil + O(scan(n))$) I/Os, which is illustrated in Figure 8.

The authors of this paper also presented a technique to discard suffixes, preventing unnecessary iterations of the algorithm. In the code presented in Figure 8, a lexicographic name is assigned to each suffix, which will not change in posterior iterations. The idea of discarding is to remove tuples considered finished from being analyzed again, thus reducing the work and I/Os in later iterations. It is possible to apply discarding to the presented Doubling Algorithm in two places. In the function *name* we can take the rank from the previous iteration and add to it the number of suffixes with the same prefix. The second place is on line (5), where some suffixes can be discarded. As a rule to discard suffixes we need a structure containing the not yet discarded suffixes, marking the ones used in a given iteration. The ones not marked at the end of some iteration can be discarded, since they will not be used in later iteration either.

Dementiev et al. observed that the Doubling Algorithm with discarding can run using $sort\left(5n\log\overleftrightarrow{lcp}\right) + O\left(sort(n)\right)$ I/Os.

The Doubling Algorithm can be generalized to obtain the a-tupling algorithm. For example, if we choose $a = 4$ we obtain the Quadrupling algorithm, which needs 30% less I/Os than the Doubling Algorithm. Even then, the authors consider that $a = 5$ would be the optimal value, but decided to use $a = 4$ because 4 is a power of two, making all calculations a lot easier. Besides it is only 1.5% worse, and needs less memory.

Function $DC3(T)$

$S{:=}\ [((T[i,i+2]),i) : i \in [0,n), i \ \mathbf{mod}\ 3 \neq 0]$ (1)

sort S by the first component (2)

$P{:=}\ name(S)$ (3)

if the names in P are not unique **then**

  sort the $(i,r) \in P$ by $(i\ \mathbf{mod}\ 3, i\ \mathbf{div}\ 3)$ (4)

  $\mathrm{SA}^{12}{:=}\ DC3([c : (c,i) \in P])$ (5)

  $P{:=}\ [(j+1, \mathrm{SA}^{12}[j]) : j \in [0, 2n/3)]$ (6)

sort P by the second component (7)

$S_0{:=}\ \langle (T[i], T[i+1], c', c'', i) :$ (8)

  $i\ \mathbf{mod}\ 3 = 0, (c', i+1), (c'', i+2) \in P\rangle$

$S_1{:=}\ \langle (c, T[i], c', i) :$ (9)

  $i\ \mathbf{mod}\ 3 = 1, (c,i), (c', i+1) \in P\rangle$

$S_2{:=}\ \langle (c, T[i], T[i+1], c'', i) :$ (10)

  $i\ \mathbf{mod}\ 3 = 2, (c,i), (c'', i+2) \in P\rangle$

sort S_0 by components 1,3 (11)

sort S_1 and S_2 by component 1 (12)

$S{:=}\ merge(S_0, S_1, S_2)$ comparison function: (13)

  $(t, t', c', c'', i) \in S_0 \leq (d, u, d', j) \in S_1$

    $\Leftrightarrow (t, c') \leq (u, d')$

  $(t, t', c', c'', i) \in S_0 \leq (d, u, u', d'', j) \in S_2$

    $\Leftrightarrow (t, t', c'') \leq (u, u', d'')$

  $(c, t, c', i) \in S_1 \leq (d, u, u', d'', j) \in S_2$

    $\Leftrightarrow c \leq d$

return [last component of $s : s \in S$] (14)

Figure 9. *The DC3-algorithm (the 3-tupling algorithm presented by Dementiev et al. [4]).*

Given this fact and the techniques presented before, the authors defined a three step algorithm that outlines a linear time algorithm for suffix array construction. The three steps to be followed are: construct the suffix array of the suffixes starting at positions i mod $3 \neq 0$; construct the suffix array of the remaining suffixes using the result of the first step; and merge the two suffix arrays into one. A pseudocode for an external implementation of this algorithm is shown in Figure 9. Figure 10, in the next page, illustrates its pipeline representation.

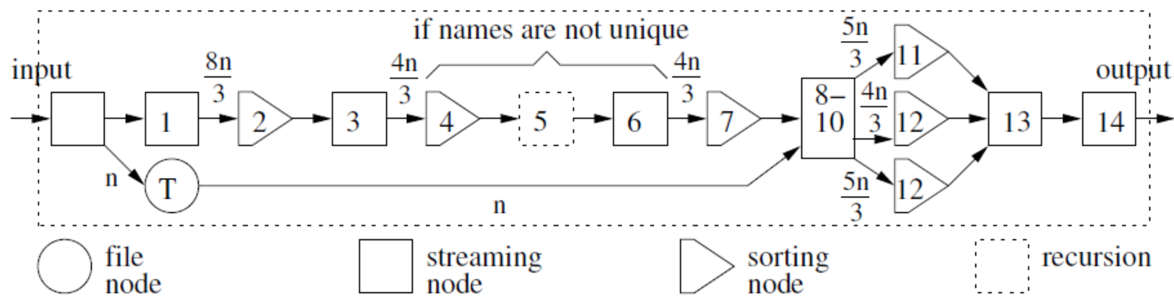


Figure 1. *The data-flow graph for the DC3-algorithm (as presented by Dementiev et al. [4]).*

Dementiev et al. proved that their DC3-algorithm outperforms all previous secondary memory algorithms, since they consider it theoretically optimal and not dependent on the inputs, which are big improvements from previous algorithms. The algorithm proposed is especially useful in this context, since it can be seen as the first practical algorithm able to build suffix arrays. In their tests, this algorithm processed 4GB characters overnight on a low cost machine.

For the purpose of this work, the research done by Dementiev et al. is especially useful, since our goal is also to build suffix arrays using secondary memory. In this context, the DC3-algorithm is a state-of-the-art algorithm to build suffix in secondary memory, providing a good basis for us to compare against our work solution.

2.4.2 Burrow-Wheeler Tranform in Secondary Memory

Karkkainen [5] proposes an algorithm to obtain the BWT of a text in small space, by constructing the suffix array in blocks of lexicographically consecutive suffixes. With such blocks it is easy to compute the corresponding blocks of the BWT. The proposed algorithm

obtains the BWT of a text of length n in $O(n \, log \, n + n\sqrt{v} + Dv)$ time and $O(n/\sqrt{v} + v)$ space (in addition to the text and the BWT itself), for any $v \in [1, n]$. Here $D_v = \sum_{i \in [0,n[} \min(d_i, v) = O(nv)$, where d_i is the length of the shortest unique substring starting at i.

Ferragina et al. [7] recently proposed an algorithm (bwt-disk) that computes the BWT of a text in secondary memory. Their solution uses $O(n)$ disk space and can compute the BWT with $O(n^2(MB \log n))$ I/Os and $O(n^2/M)$ CPU time, where $M = O(n / (\log n \, \log_{M/B} \frac{n}{B}))$ and B is the number of consecutive words in each disk page. Their algorithm makes sequential scans, what lets them take full advantage of modern disk features that make sequential disk access much faster than random accesses, because sequential I/Os are much faster than random I/Os.

The authors compared their bwt-disk algorithm against the best current algorithm for computing suffix arrays in external memory, the DC3 algorithm, already covered in this dissertation. Their tests showed that the memory usage reported to both algorithms was similar, although their algorithm showed to run faster than the DC3 algorithm. Furthermore, the bwt-disk algorithm computes the BWT of the text while the DC3 algorithm only computes the suffix array, the additional cost of computing the BWT was ignored.

These are the state-of-the-art algorithms for building the suffix array and obtaining the BWT of a text, thus these are the algorithms we will test our solution against. Furthermore they contain interesting details, such as reading data by disk sequentially instead of randomly, proposed by Ferragina et al. [7], or reading data in blocks of a given size, instead of one by one or any other way, proposed by Karkkainen [5].

3. Our solution

We propose a solution that uses secondary memory to store a significant part of the working data during the construction of the suffix array (and the BWT), so that the main memory usage is much smaller. This makes our algorithm suitable to compute the suffix array of a bigger text in a computer with the same amount of main memory.

For this purpose we divide our algorithm in two parts: the splitting part and the merging part. The splitting part consists in receiving the text, splitting it in several chunks, and outputting a set of suffix arrays, one of each chunk. The merging part is the opposite, it gathers the suffixes from these suffix arrays and merges them into the suffix array of the original text, using a data structure created by us, which is the string heap.

3.1 Text Generator

To run our algorithm, we first need a text, which, through this document, we call original text. We created a text generator that generates a random text of size n, from a specific alphabet. This kind of generation is suitable for debugging and testing specific aspects, such as examples with a lot of branching, which produce a lot of pivots, or others with less dispersion, which produce bigger common prefixes and less pivots.

We will also use real texts, such as DNA or protein sequences, English books, C programs' source code, etc. We used this kind of examples to perform the final tests of our program, because they give a much more accurate vision of how our algorithm will perform in a real world situation.

3.2 Splitting Part

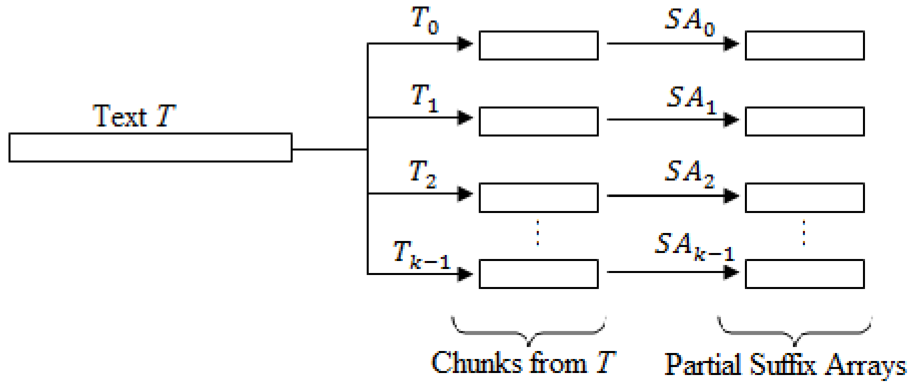


Figure 12. *Illustration of the splitting part of our algorithm.*

As shown in Figure 11, in the splitting part of our algorithm a text T is divided into k chunks. From these chunks we obtain k suffix arrays, which are stored in secondary memory.

3.2.1 Partial Suffix Arrays

Given a text T of size n, we divide it into k chunks of size c, i.e. $T_0 = T[0 \ldots c]$, $T_1 = T[c \ldots 2c-1]$, $T_2 = T[2c \ldots 3c-1]$, $T_{k-1} = T[(k-1)c \ldots kc-1]$.

For each of these text chunks we build the respective suffix array: $SA_0, SA_1, SA_2, SA_{k-1}$. We refer to these arrays as partial suffix arrays, since each of them is the suffix array of a chunk of the text T. These arrays are not built in the canonical way, where we assume that there is a $ symbol after each T_i, i.e. the partial suffix arrays are not the suffix arrays of $T_i\$$, instead they sort only the first c suffixes of $T_i T_{i+1} \ldots T_{k-1}$.

3.2.2 **Algorithm to Build the Partial Suffix Arrays**

To build the partial suffix arrays we use a suffix tree algorithm for each of the k chunks. This algorithm runs in O(c) time for each chunk and can be processed in parallel.

This algorithm builds the suffix tree for each set of suffixes, and gives the respective suffix array. Since this algorithm builds the suffix tree of the text it is also possible to obtain the LCP between blocks of suffixes inside each partial suffix array, which reduces the time complexity of the merging part of our algorithm.

This procedure uses main memory, which is not a problem since the chunks of the text that originate the partial suffix arrays are relatively small and at this point our program will not use main memory but to build the partial suffix arrays one by one. Thus, the only data stored in main memory in the splitting part of our algorithm is the text and its suffix tree of one of the chunks.

Each newly created partial suffix array is stored in secondary memory before the next one is created, each one in a specific file. This way we never fill the main memory with suffixes, in practice only a small part of the suffixes will be stored in main memory alongside the text at each moment.

As we mention through this document, our algorithm has two versions. The first one is a simpler approach to the partial suffix arrays, where each one only contains the indexes of the suffixes (integer values). After building the suffix tree of a given chunk of the text, the algorithm follows its branches and prints its leafs (suffix indexes).

Since the suffix tree also enables to obtain the LCP between branches (groups of suffixes), with the same complexity, we used those to produce a second version of our algorithm. In this one each partial suffix array stores blocks, instead of simple indexes.

3.2.3 Blocks of Suffix Indexes

To build the partial suffix arrays in the version with blocks, the suffix tree algorithm doesn't need to reach the leafs of the suffix tree in order to print the index of a given suffix. Given an LCP value, the algorithm follows the branches up to that value, and then prints all the suffix indexes of that group of branches.

Each block contains 3 data structures:

- An integer number (*bkn*), representing the number of suffix indexes stored in the block.
- An integer number (*bklcp*), representing the Longest Common Prefix shared by the elements stored in the block.
- An array (*bkar*), of size *bkn*, storing the suffix indexes in this block.

Each block has size (*bkn*+2), where *bkn* > 0, i.e. each block contains at least one suffix index.

3.3 Merging Part

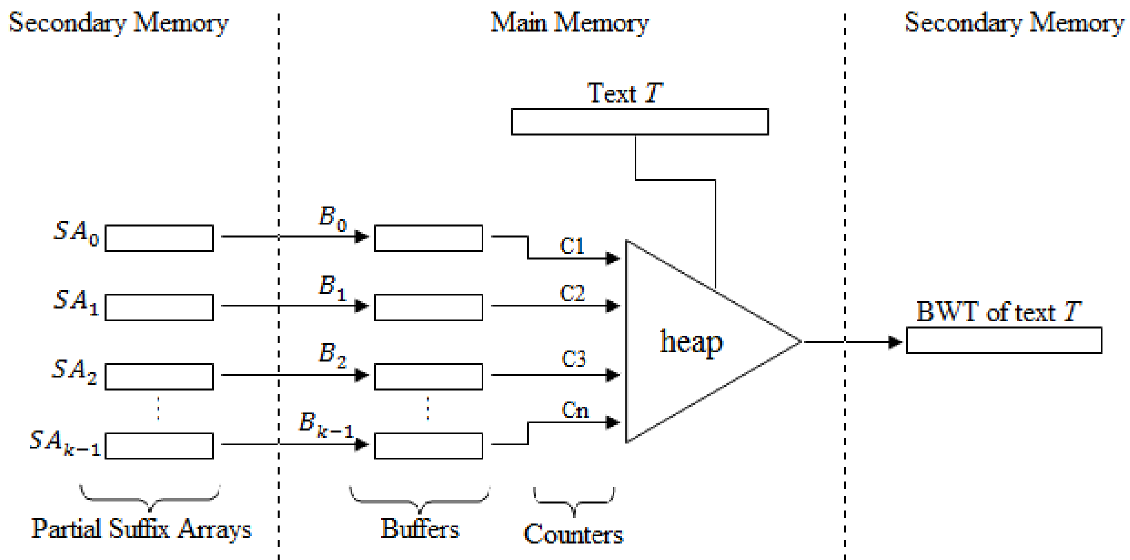


Figure 12. *Illustration of the data that will be stored in main memory and in secondary memory in the merging part of our algorithm.*

As shown in Figure 12, the merging part of our algorithm consists in receiving values from the partial suffix arrays, gathering them into the heap and successively returning the minimum of the values contained in the heap at each moment. As the heap pops suffixes they are printed to secondary memory, obtaining the suffix array of the text T, or, as Figure 12 illustrates, its Burrows-Wheeler Transform. The data printed is indifferent in terms of complexity, since the computation of a position of the BWT has time complexity O(1) if we know the respective position of the suffix array, which is the case in our program. As we mentioned in the related work section of this document: BWT $[i] = T\,[SA[i]-1]$.

Note that all the partial suffix arrays are stored in secondary memory in the beginning of the merging part of our algorithm. Each suffix index stored in there is brought to main memory only when it is pushed into the respective buffer. Once in main memory a suffix remains in the buffer until it is popped to the heap, what happens only at the heap's request for new elements from that specific buffer. It remains in the heap as long as it is not the smallest suffix in there. In that moment it is printed to the BWT file, thus again to secondary memory.

3.3.1 Buffers

Each of the described partial suffix arrays has a buffer that stores the suffix indexes in main memory. Despite being in main memory, those indexes are still out of the heap, and are popped into it only when needed.

In this context, the main intent of having buffers is to help prevent the heap from blocking. Without buffers the heap would need to ask for new indexes directly to secondary memory, which would make it block more frequently. Having buffers we can ask for them asynchronously, what we consider a valuable enhancement, since the disk accesses are dominant in terms of time consumption.

As mentioned above, the buffers can receive suffix indexes alone or in blocks. In the first approach the buffers get the suffixes one by one, and pop them to the heap in the same order. In the second approach, despite storing blocks the procedure is similar, the buffers receive the blocks from the partial suffix arrays, and pop them into the heap in the same order they received them.

Each buffer has a counter that tells how many suffix indexes from the respective partial suffix array are yet to be passed to the heap. When a counter reaches zero all the suffix indexes for which that buffer was responsible for were already popped to the heap, meaning that the buffer has no more work to do in that execution.

3.3.2 **String Heap**

The particular heap structure that we use is based on the Quickheap proposed by Navarro, et al. [3], which is a hierarchical algorithm, inspired on the Quicksort algorithm. We also use a technique proposed by Bentley, et al. [2], who pointed out that using ternary Quicksort was better than using the original binary Quicksort, for building suffix arrays. We use both studies in our work, particularly to make our heap. The main idea is to have a heap similar to the one presented by Navarro, et al., but introducing the enhancement based on the observation of Bentley, et al., which is: ternary searches perform better than binary searches to compare suffixes.

Being ternary means that each partitioning produces three partitions. These are: the group of elements whose first character is lesser than the first character of the pivot; the group of elements whose first character is equal to the one of the pivot; and the group of the ones with a greater first character comparing with the one of the pivot. All the comparison operations are always performed character by character, meaning that the comparison between two suffixes end as soon a different character is found.

We also store the LCP between suffixes, what reduces the complexity of any comparison operation, since there is no need to compare from the beginning of the suffixes when we already know that they share a number of characters in the beginning. Therefore, this enhancement

reduces the average time complexity of the insertion operation from $O(\log n . \overline{lcp})$ to $O(\log n + \overline{lcp})$, where n is the size of the text and $\overline{lcp}$ is the average LCP between the text suffixes. This enhancement avoids the need to compare all the characters up to the LCP value, $\overline{lcp}$ is accumulative, thus each compared character is valid not only for one suffix but for all that share that character with it.

Using this approach on a character by character partitioning system is especially relevant because we are working with strings, reason why we call this structure a String Heap.

The heap is internally divided in partitions, as in the Quickheap presented in the related work section of this document. Each partition is a group of contiguous positions of the heap, whose elements share a common LCP (*x*). One of those elements is the pivot of the partition, and its first *x* characters represent the first *x* characters of any other suffix in the partition. The pivots define frontiers between partitions, since a partition's pivot is its last position

3.3.3 **Input/Output Operations**

The way the suffixes are popped from secondary memory to main memory is a crucial issue raised in the implementation of the algorithm. To better answer this question we need to take three factors into account: the number of partial suffix arrays, the size of each buffer, and the number of elements popped from secondary memory to the buffers on each read.

For each partial suffix array the heap has a counter that tells the number of elements popped from each respective partial suffix array, and which still remain in the heap. These counters are initialized with zero, since the heap begins empty, and every time a suffix is popped from a partial suffix array into the heap, the respective counter is incremented by one unit. Conversely each time a suffix is popped from the heap to the BWT, the counter respecting to the partial suffix array from where that suffix came from is decremented by one unit. In the version with blocks of suffix indexes, when a block enters the heap, the counter respecting the buffer where it came from is incremented the number of elements of the block.

For the correctness of our algorithm's results the heap must always contain at least one element from each partial suffix array, meaning that no counter can reach zero, except either in the beginning of the execution or when its buffer is empty, i.e. passed all the elements from its partial suffix array to the heap. Whenever the heap runs out of elements from any of the partial suffix arrays the execution stops.

In the beginning of the execution the heap asks for suffixes from all the partial suffix arrays, until it contains at least one from each of them. Yet as the heap pops values it redefines that policy, and will tend to ask for values rather to the suffix arrays where those values belonged, aiming to maintain a balanced number of elements of each suffix array. Note that the I/Os take longer to bring new suffixes to the heap than the heap takes to return results, since a main memory read is much faster than a disk read. Because of this fact the heap tends to pop results at a faster pace than it receives, thus, for the reason mentioned above, it tends to block waiting for new suffixes to be pushed. To mitigate the effects of this issue we need to decrease as much as we can the number of times it blocks.

Being popped from suffix arrays of parts of the text, we know that each element obtained by the heap is smaller than any other that will ever come from the same partial suffix array, still it can be greater than the values in the other partial suffix arrays. This is why the heap always needs to contain at least one element from each partial suffix array, otherwise there would be no guarantee that the suffixes from the missing buffers weren't smaller than the suffixes in the heap. For this reason, if at any time the heap contains no elements from a given buffer, and for some reason the heap is unable to obtain elements from it, the execution has to stop, because there is no guarantee that the heap is able to pop the minimum value from all the elements that are yet to be returned. If such a situation occurs, it is mandatory that the execution blocks, otherwise we can't guarantee the correctness.

As the execution continues, some partial suffix arrays are left with no more elements to pop. At that time the heap discards those suffix arrays, and will not ask them for new suffixes any more. The algorithm ends when the heap is empty, which means that all partial suffix arrays are also empty and that all elements have been returned.

When deciding the best policy to adopt to prevent the heap from blocking we need to consider some tradeoffs. The heap must contain elements from all the non empty partial suffix arrays, otherwise the execution will block. To avoid this problem we could bring a new element to the heap every time one is popped from it, yet such a policy would represent too many I/Os. We could also flood the heap with many values from each buffer, but too many values in the heap means exaggerated main memory consumption. The policy we choose must reflect all these concerns, trying to manage these problems.

The factors we need to pay attention when making such a decision are the number of partial suffix arrays in which we divide the text, the size of each buffer, and the number of elements in the heap. The more partial suffix arrays we have the less suffixes each one contains; conversely, the less partial suffix arrays we have, the more elements each partial suffix array will contain. Each of these situations has an outcome. If we have more partial suffix arrays their counters in the heap tend to have smaller values, which means that the heap will ask for new suffixes more often.

If each partial suffix array contains less elements means, in the case of blocks of suffixes, that the blocks are smaller, not taking full advantage from having the suffixes grouped in blocks by their LCP. On the other hand, if we have less partial suffix arrays we get much more from that fact, since the blocks will be bigger, requiring less processing in the heap. However, popping those bigger blocks to the heap means more main memory consumption.

We also need to decide the number of suffix indexes that are pushed into main memory on each read to the buffers. The more elements read on each operation, the less impact I/Os have on its execution, however, if the buffers store too many suffixes that can represent too much main memory consumption.

The size of the heap is also a concern. Since the operations running time depends on the number of elements it contains, we don't want it to get too big, what would increase those operations running time. However, if the heap contains few elements it will block too many times, what will bring an extra I/O cost.

4. Algorithm Implementation

4.1 Initial Parameters

Before running our algorithm we need to set the following parameters:

- Size of the input text (*n*). The number of characters of the text given as input.
- Size of the heap (*m*). The maximum number of suffixes the heap can contain at a time.
- Size of each partial suffix array (*c*). The size of each chunk of the text from which the partial suffix arrays are obtained.
- Size of each buffer (*d*). The maximum number of suffixes a buffer can contain at a time.
- LCP to build the blocks (*bklcp*). The size of the prefix that the suffixes inside the partial suffix array must share to belong to a common block. This parameter is needed only if we are running the version that uses blocks of suffix indexes.

Note that the number of partial suffix arrays (*k*) is also a fundamental parameter for the execution of our algorithm, however we don't need to set it manually. Having the size of the text (*n*) and the size of each partial suffix array (*c*) our program calculates it in the beginning of its execution.

4.2 Splitter

The splitter divides the text into *k* chunks, each of them with size *c*. For each chunk it builds the respective suffix tree, and follows its branches to obtain the *c* suffixes one by one. For each partial suffix array the splitter creates a new file, and each suffix is written to it once it is obtained. If we want to store blocks instead of single suffixes the branches of the suffix tree are followed until an LCP of at least *bklcp* is reached, moment when a new block is created with the number of suffixes contained in those branches, the LCP between them, and the suffix indexes themselves. In both cases the splitter creates *k* files, one for each partial suffix array.

Note that the suffixes obtained with the suffix tree have indexes in the context of that chunk, however, before storing them in the partial suffix arrays they are added the number of suffixes stored in the previous suffix arrays. Thus, the indexes stored in the files are global indexes, instead of partial, even though they are stored in partial suffix arrays.

4.3 Merger

4.3.1 Data Structures

The merger consists in two types of structures: heap and buffers. We have one heap and *k* buffers, one for each partial suffix array. The buffers feed the heap with elements, and the heap sorts the elements and pops them into the final suffix array, which we use to create the Burrows-Wheeler Transform of the text.

Therefore the merging part of our algorithm begins with the creation of the heap and initialization of its data structures. The heap contains two structures that maintain the data related to the buffers, we also created a type *Buffer* to better distribute the tasks involved in this part of the algorithm.

- *Buffer* – A data structure created by us that contains:
  - *file* – A permanent reference to the file where the respective partial suffix array is stored.
  - *ar* – An array storing the indexes of the suffixes it contains at each moment.
  - *elemsleft* – An integer counting down the elements left to return from its respective partial suffix array. Note that each buffer will never contain all the elements in its respective partial suffix array (*c*), thus it has a smaller capacity (*d*) and is used as a circular array.

- *bufs* – An array of elements of type *Buffer*, to store pointers to the buffers where the suffixes come from. This array has *k* positions, one for each buffer.

- *bufcount* – An array of integers to store the number of elements from each buffer that are in the heap at each moment. Note that each position of this array counts the elements of the buffer pointed by the same position in the array *bufs*.

The heap also has data structures whose purpose is to maintain its internal operations:

- *ar* – An array of integers to store the indexes of the suffixes in the heap at each moment. This array has size m+2, i.e. the maximum number of elements that the heap can contain at each time plus the two sentinels. Note that this is a circular array, thus its size (m) can be much smaller than the total number of suffixes (n).

- *pvts* – An array of integers that stores the pivots. Each of the integer numbers stored is the position of a pivot, i.e. if a value i is in the array *pvts* the element *ar*[i] is a pivot.

- *lcps* – An array of integers that stores the value of the LCP in each partition of *ar*. Each position of this array corresponds to the same position in the array *pvts*, i.e. *lcps*[i] contains the value of the Longest Common Prefix between *ar*[*pvts*[i]] and any other element of the partition in which it is the upper limit and *ar*[*pvts*[i+1]] is the lower limit. Note that we define that even though *ar*[*pvts*[i]] belongs to this partition *ar*[*pvts*[i+1]] does not, therefore the LCP value *lcps*[i] is not valid between the lower limit and any element of the partition.

- *bplcps* – An array of integers that stores the value of the LCP between each pair of pivots. Each position of this array also corresponds to the same position in the array *pvts*, i.e. *bplcps*[i] contains the value of the LCP between *ar*[*pvts*[i]] and *ar*[*pvts*[i+1]]. Note that this LCP value has no relation with the LCP value of the partition between these pivots, furthermore these arrays are used for different purposes, which we explain later in this document.

- *i* – An integer value with the position of the last pivot in the array *ar*. This pivot is the smallest element in the heap, thus the next to return.

Note that the order of the pivots in *pvts* is inverse to the order of the suffixes in *ar*. The array *pvts* has decreasing order, i.e. *pvts*[*i*] contains a bigger value that *pvts*[*i*+1], apart from sentinels. In the cases that the heap rotates this is not absolutely true, since *pvts*[i+1] can be bigger than *pvts*[*i*], however the order is the same and *pvts*[i+1] is always closer to the smaller sentinel than *pvts*[*i*].

As we mentioned in the related work section, the pivots array (*pvts*) is a stack, thus we implemented it this way because it is more time efficient than implementing it in the same order than the heap. This fact is explained in the next section, mainly in the insertion and extraction operations, in the heap.

In both arrays *ar* and *pvts* we use sentinels to avoid making conditions to catch the case when the index gets out of the valid positions, and yet preventing the program to crash. In both cases the sentinels are located in the position before the first and after the last valid position, the small and large sentinels, respectively.

4.3.2 Operations

The type *Buffer* is used to store elements and, when the heap asks, pop them into there. Its operations are:

- Initialization – For the array *ar* it allocates *d* positions in memory. The integer *elemsleft* is initialized with the value *c*, since in the beginning of the execution all elements from the partial suffix array are yet to be popped to the heap. It also opens the file where its partial suffix array is stored, and keeps that reference.

- *next* – This function is responsible for returning the next suffix index to the heap. In the case we are using blocks of suffixes it returns the next block of suffix indexes.

In the heap these are the basic operations. The remaining are presented in the next sub-section.

- Initialization
    - *ar* – This array uses O(m) space, which is allocated in the beginning of the execution of the algorithm.

    - *pvts* – This array uses O(log m) space on average, which is allocated in the initialization.

    - *lcps* and *bplcps* – Being analog to *pvts* each of these arrays also use O(log m), and both are allocated in the beginning as well.

- *check_buffer* – This function receives the buffer index (b), and checks if *bufcount*[b] > 0. If not, that means the heap has run out of suffixes from the buffer b, therefore at least one new suffix needs to be inserted from that buffer.

- Comparison function – This function is used in every comparison between suffixes inside the heap. It receives the indexes of the two suffixes to compare (A and B) and compares only the character in the position given. This function returns a value: 0 if T[A] is equal to T[B]; negative if T[B] is greater than T[A]; and positive if T[A] is greater than T[B]. If the index A exhausts the text size this function returns -1; if B exhausts the text size 1 is returned. This function shows a representative case of the benefits of using sentinels: if A is compared with a sentinel the function returns -1 (small sentinel) or 1 (large sentinel); conversely if B is compared with a sentinel the function returns 1 or -1, respectively if it is the small sentinel or the large sentinel. Without sentinels we would have to set conditions to catch the cases in which the index given was not a suffix, which would require unnecessarily complicated code.

4.3.3 **Heap Insertion**

This function is used to insert new suffixes in the heap, thus it receives the index (*b*) of the buffer where to obtain those. The insertion begins with evoking the function *next* of the buffer stored in *bufs*[*b*], obtaining the suffix *suf*. This suffix is compared with all the pivots, one by one. For each pivot *pvts*[*i*] the maximum number of characters compared is *lcps*[*i*], i.e. comparing the characters until the LCP of the partition is enough to decide if *suf* needs to be compared with the next pivot.

Since we know the LCP in each partition and the LCP between each pair of pivots, we use those values in the insertion of new elements:

- *bplcps* – The LCP between pivots are very important to reduce the execution time of our algorithm, by reducing the number of characters that need to be compared on each insertion. Once the new suffix (*suf*) is compared with one pivot, *pvts*[*i*], the LCP between them (*lcp*) is stored, and in the cases when *lcp* > 0 and *bplcps*[*i*] > 0 there is a number of characters that does not need to be compared again, i.e. min(*lcp*, *bplcps*[*i*]) is that number, where min is a function that returns the minimum of two numbers.

- *lcps* – The LCP of a partition is used when we need to decide if the new suffix is inserted in an existing partition or if it needs to be set as a new pivot. It can be inserted in a partition if, and only if, the LCP between *suf* and the pivot that is the upper limit of the partition, *lcp*, is at least equal to the partition LCP, *lcps*[*i*], i.e. *lcp* >= *lcps*[*i*]. This is essential to maintain the correctness of the heap.

The insertion function runs two different loops, one inside the other. Each iteration of the outer loop means a new pivot compared against the new suffix. Each iteration of the inner loop means a new character compared between the current pivot and *suf*.

As mentioned, we don't need to compare more characters than the LCP of the current partition, thus one of the conditions to break the inner loop is that, if *lcp* = *lcps*[*i*]. The other condition is when we can decide in which partition to insert *suf* without comparing every character up to the LCP value. This happens if, before exhausting the partition LCP, the comparison between the

current characters of *suf* and the pivot is not zero, i.e. if the character is different then suf is different than the pivot, before the LCP.

Once the inner loop is broken the outer one is also broken if, and only if, *suf* is greater than the compared pivot, i.e. if the inner loop didn't exhaust the partition LCP, and at some character the comparison returned that the pivot's character is smaller than the new suffix's one. According to other conditions, the new suffix can be inserted into the heap in two different ways:

- The new suffix is inserted in the partition between *ar*[*pvts*[i]] and *ar*[*pvts*[i-1]] – This case happens if, and only if, the LCP between the previously compared pivot, *pvts*[i-1], and *suf* is at least equal to the partition's, i.e. LCP(*pvts*[i-1], *suf*) >= *lcps*[i-1].

- The new element becomes a new pivot between *ar[pvts*[i]] and *ar[pvts*[i-1]] – This case happens if the LCP between previously compared pivot, *pvts*[i-1], and *suf* is smaller than the partition's, i.e. LCP(*pvts*[i-1], *suf*) < *lcps*[i-1]. In this case the pivots are put one position ahead, i.e. *pvts*[*j*] becomes *pvts*[*j*+1] for every *j* >= i, and *pvts*[i] = *suf*, to avoid unnecessary character comparisons in the next insertions, *bplcps[*i] is set with the value LCP(*pvts*[i], *suf*), i.e. the LCP between the newly created pivot and the next one.

Whenever a new suffix is inserted into the heap, the counter of elements from the buffer it came from is incremented by one unit, i.e. *bufcount*[b] = *bufcount*[b]+1. If instead of one single suffix a block of suffixes is inserted, then the counter is incremented by as many units as the size of the block.

The insertion of a new suffix into the heap runs in $O(p + \overline{lcp})$ time on average, where *p* is the number of pivots in the heap at the moment of the insertion, and $\overline{lcp}$ is the average LCP between the new suffix and those pivots.

In the version that stores blocks of suffix indexes in the partial suffix arrays, the insertion has slight differences, yet the principles remain the same.

One of the suffixes from the block is used to compare against the pivots. Any suffix of the block could be used since they all have the same *bklcp* characters, in our case we pick the first suffix. This suffix performs exactly the same way as if it was a single suffix to be inserted, i.e. like *suf* in the previous explanation.

When it is compared against a suffix there is a new condition to brake the inner loop: when the comparison LCP (*lcp*) exceeds the block LCP (*bklcp*). If we have a clearly established block with a relevant common prefix between its suffixes we don't want to lose that information, so if *bklcp* is exhausted there is no point in continuing the comparison.

In the outer loop there are more noticeable changes: the new block will be gathered with an existing partition if, and only if, its LCP is equal to the partition's LCP, i.e. *bklcp* = *lcps*[i]. Otherwise it creates a new partition with the elements of the block and the LCP of the block.

In the end of this operation the counter of elements from the partial suffix array where the block came from is incremented by the number of elements in the block (*bkar*[0]). All the rest of the insertion operation is equal to the one performed in the insertion of single suffixes.

The insertion of a block of suffixes into the heap runs in $O(p + max(\overline{lcp}, bkar[0]))$ time on average, where *p* is the number of pivots in the heap at the moment of the insertion, $\overline{lcp}$ is the average LCP between the new block and the pivots, and *bkar*[0] is the first position of the block, i.e. the number of suffixes in it, and *max* is a function that returns the greater of two integer numbers. Note that $\overline{lcp}$ has at most the same value as the LCP of the block, *bkar*[1].

The only difference between the complexity of inserting a block and a single suffix is the addition of the block size. This value appears because of the need to copy the *bkar*[0] positions of the block to the heap. In the insertion of a single suffix this value is 1.

4.3.4 Heap Repartitioning

To be able to return the minimum value correctly and in minimal time we need to make sure that the first position of *ar* contains the minimum of the heap. This happens only when the first position of *ar* (after the small sentinel) is pointed by the last position (before the large sentinel) in *pvts*, however we don't guarantee this in the insertion (which is intended). When the minimum is in the first position and is a pivot we say that the heap is consistent. When it is inconsistent and we want to extract the minimum there is a need for repartitioning until it is consistent.

While the heap is inconsistent, its first position is repartitioned. This procedure implies finding new pivots and grouping the suffixes accordingly. To find a new pivot we use the median of three method, which consists in randomly picking three positions from the partition and selecting their median, which is the pivot. This method helps balancing the new partitions, since this way the probability of having a pivot too large or too small is considerably lesser.

Since for each partition we have the LCP between its suffixes (*lcps*[i]), we only need to compare from there on, i.e. from the character *T* [*lcps*[i]+1]. For, in each execution of the repartitioning function we select a pivot, and divide the remaining suffixes in three groups: the ones with the character *T* [*lcps*[i]+1] greater than the pivot's (*lcps*[i]+1)-th character, the ones with equals, and the ones with the (*lcps*[i]+1)-th character lesser than the pivot's. This procedure is repeated recursively until the heap is consistent.

This operation runs in $O(p_0\ (logp_0 + \ \overline{lcp}))$ time on average, where p_0 is the number of suffixes in the first partition of the *ar* array, containing the positions between *pvts*[0] and *pvts*[1].

4.3.5 **Heap Extraction**

Before extracting the minimum of the heap we need to make sure it is consistent, therefore we repartition it.

Extracting a suffix from the heap implies returning it in the end of the operation, therefore the second thing to do is to read it before the removal. The minimum (*res*) is the last pivot in the *pvts* array (the first when looking to the element in the array *ar*), thus the one pointed by *i* (the variable of the heap), i.e. *ar*[*pvts*[*i*]] is the value returned in the end of the operation.

This operation also includes updating the references to the suffix that will be removed. These include the arrays *ar*, *pvts* and the LCP arrays. Hence, the suffix is erased from the array and the small sentinel is set to its position, i.e. goes one position forward. In the array *pvts* the same happens, the pointer to the pivot is erased and the small sentinel goes to its position. The arrays *lcps* and *bplcps* are set to zero in that position, since the partition and the pivot disappeared.

For each element extracted from the heap, the counter of elements of the buffer it came from decreases one unit, i.e. *bufcount*[*b*] = *bufcount*[*b*]-1. The value *b* is obtained by knowing the number of suffixes in each partial suffix array and the global index of this suffix, i.e. *b* = *res* / *d*.

Once the counter is decreased a new issue is raised: we need to check whether that was the only suffix from its buffer in the heap. For this purpose the method *check_buffer* is called with *b* as argument.

Note that this operation is the best explanation why we used the *pvts* array inverse to the *ar*, i.e. like a stack. This way the element to be popped is always on the top, making the extraction operation run in O(1) time, apart from the repartitioning time, which is described in the previous sub-section.

4.4 BWT Builder

The output of our program is the Burrow-Wheeler Tranform of the text T (or its suffix array if we want), which we store in secondary memory.

To achieve this we use every character extracted from the heap to obtain the respective element in the BWT, and write it to the file. Each element i of the BWT is given by the expression $BWT[i] = T[SA[i]-1]$, where SA is the suffix array of the text T. If we want to obtain the suffix array in the end, we just need to print SA[i] instead of BWT[i].

5. Algorithm Execution

In this section we show and explain an illustrated example of the heap running.

We use the same input text as in previous examples in this report, such as in the related work section, i.e. T = “mississippi”. In this specific example we use a heap with the same size as the number of suffixes (11), however in the normal tests the heap is much smaller than the input text since the array *ar*, in the heap, is circular. However the representation is much clearer with a non-circular heap, because of the LCP “boxes” around the suffixes’ characters, thus for visual explanation purposes we use a non-circular heap.

First of all we run the splitter, giving the text and the number of partial suffix arrays, which we defined as 3.

Note that our internal representation of suffix indexes begins in the index 2, instead of 0. This is because in the array *ar* in the heap the first two positions are sentinels and an index smaller than 2 would point to one of them, which is not the intention here. With this change the suffixes of the text “mississippi” are as illustrated by Figure 13.

Index	Suffix
2	mississippi
3	ississippi
4	ssissippi
5	sissippi
6	issippi
7	ssippi
8	sippi
9	ippi
10	ppi
11	pi
12	i

Figure 13. *Suffixes and the respective indexes for the text “mississippi”.*

The splitter divides the text T into 3 chunks, 2 of them with 4 suffixes each, and the third with 3 suffixes. After the splitting part each file contains the following indexes:

- Partial suffix array 0
  - 3 – ississippi
  - 2 – mississippi
  - 5 – sissippi
  - 4 – ssissippi
- Partial suffix array 1
  - 9 – ippi
  - 6 – issippi
  - 8 – sippi
  - 7 – ssippi
- Partial suffix array 2
  - 12 – i
  - 11 – pi
  - 10 – ppi

Note that in the files only the indexes are written, not the suffixes themselves, we wrote them here merely to better illustrate where the numbers come from.

After this the merger is launched, creating the heap and the buffers. On the creation, each buffer gets one suffix from the respective suffix array, in the simple approach of one suffix at a time. If we are using a block approach, they get a block of suffixes instead.

After its creation the heap is immediately asked to pop the minimum, which is impossible since it doesn't contain any elements. For this reason, the extraction operation, in the heap, checks all the buffers for elements prior to the first extraction, therefore it gets one element from each buffer. Figure 14 illustrates the heap data structures after these insertions. In these figures we represent the pivots array with the same orientation as the suffix indexes array to avoid arrow crossing.

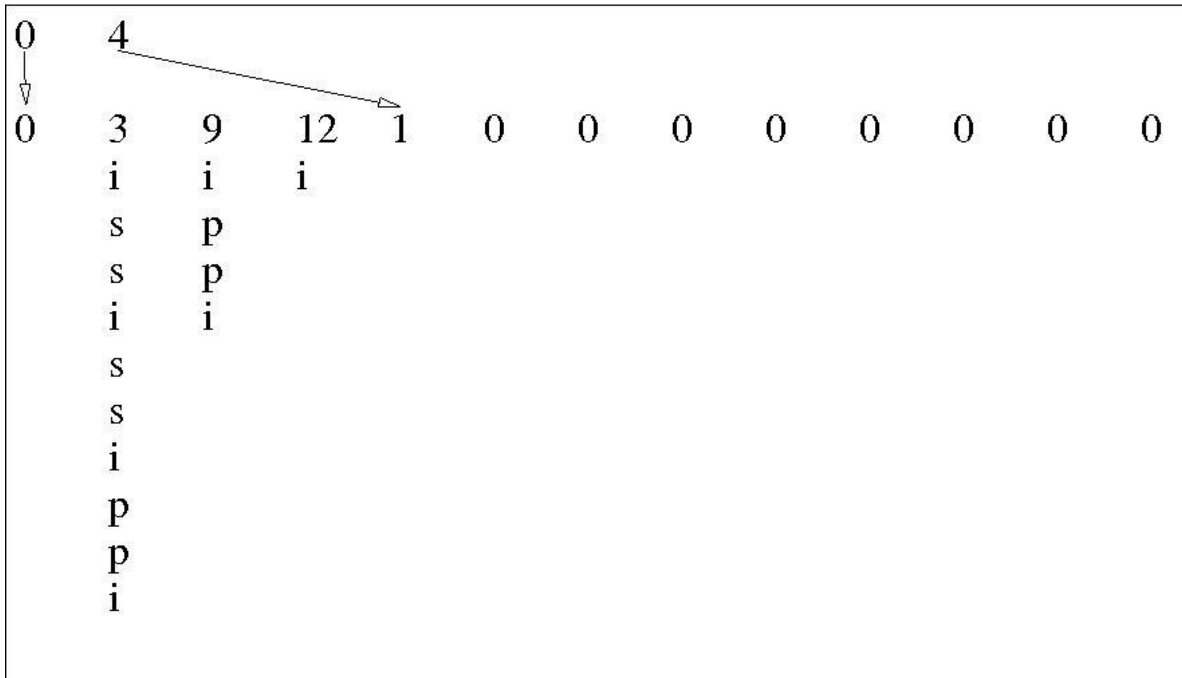


Figure 14. *Representation of the heap data structures after inserting the first three suffixes, and before the first extraction.*

In this representation we can see the *pvts* array on the top and the *ar* array in the middle. From each pivot an arrow points to its respective position in *ar*. Bellow each suffix index is the suffix string itself. Note that in this representation the *pvts* array has the same orientation than *ar*, i.e. inverse to its representation in the code. In this example *pvts*[0] = 4 and *pvts*[1] = 0.

Figure 14 illustrates the next state of the heap: there is only one partition, limited by the sentinels, thus in the *lcps* and *bplcps* arrays there is nothing but zeros.

Not needing to insert suffixes from any more buffer, the heap can now extract the minimum, however it first needs to repartition until a pivot is in the first position. This situation can be seen in Figure 15, where the index 12 is ready to be popped.

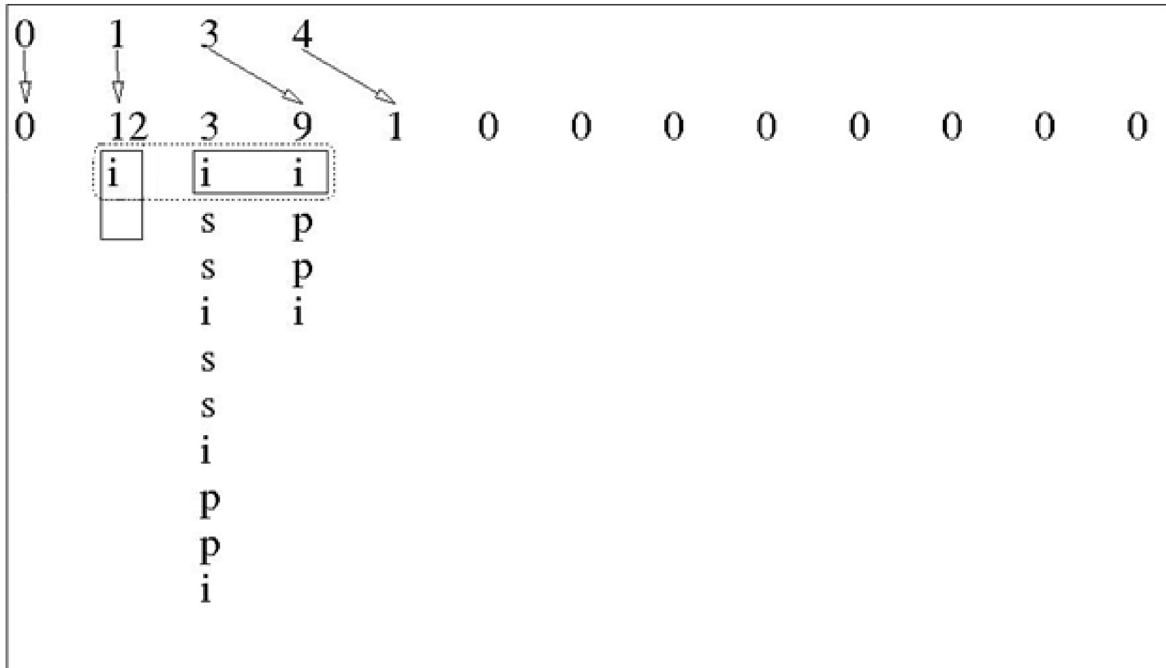


Figure 15. *Representation of the heap data structures immediately before extracting the first suffix (12).*

In this figure we can see the representation of the LCP of the partition between *ar*[2] and *ar*[3] (*lcps*[2]), which has value 1 and is represented by an LCP-sized box around the common prefix between the suffixes of the partition. Also the LCP between the pivots *ar*[2] and *ar*[1] (*bplcps*[2]) is represented in a similar way. The type of trace used is different to avoid any confusion between the *lcps* and the *bplcps* arrays' representation. The partition that only contains the suffix index 12 has LCP = 2, which is a result from the repartition, however this LCP is meaningless, since the partition only contains one element.

After repartitioning the partition shown in Figure 15, the suffix with index 12 was defined as the new pivot. Being a pivot in the first position of the heap, after the small sentinel, this suffix will be returned next.

After being returned the heap will have no suffixes from partial suffix array 2, therefore it will ask for a new one. Such situation is illustrated in Figure 16.

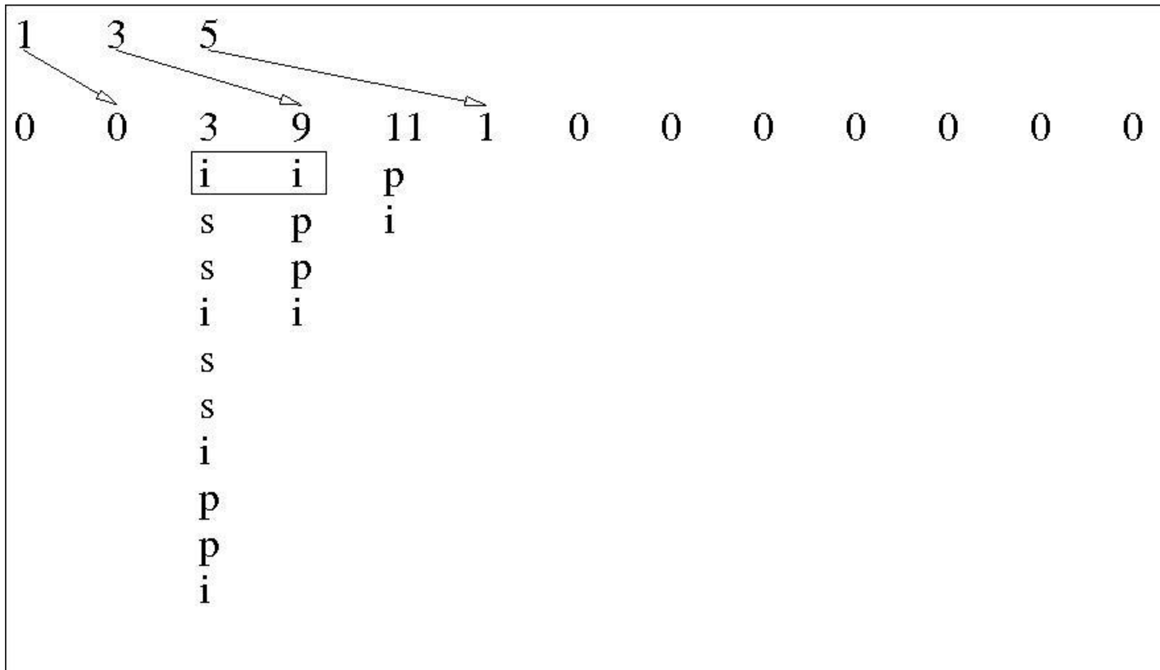


Figure 16. *Representation of the heap data structures after inserting a new suffix (11).*

The suffix 11 was inserted from the partial suffix array 2 and was compared against the pivot 9. The LCP of the partition between 0 and 9 (*lcps*[1]) was 1, and LCP(*suf*, *pvts*[1]) = 0, thus the new suffix proved larger than the pivot before reaching the partition LCP, which is a condition to break the outer loop in the insertion operation. In this case 11 is inserted in the previous partition, not being set as a new pivot, this happens because that partition's LCP is 0, and as such any suffix can be inserted there.

After this situation the heap is inconsistent and needs to repartition its first partition (between *ar*[0] and *ar*[1]) to find a new minimum, which will be 9. After the extraction the heap will need a new suffix from the partial suffix array 1. These situations are illustrated by Figures 17 and 18, respectively.

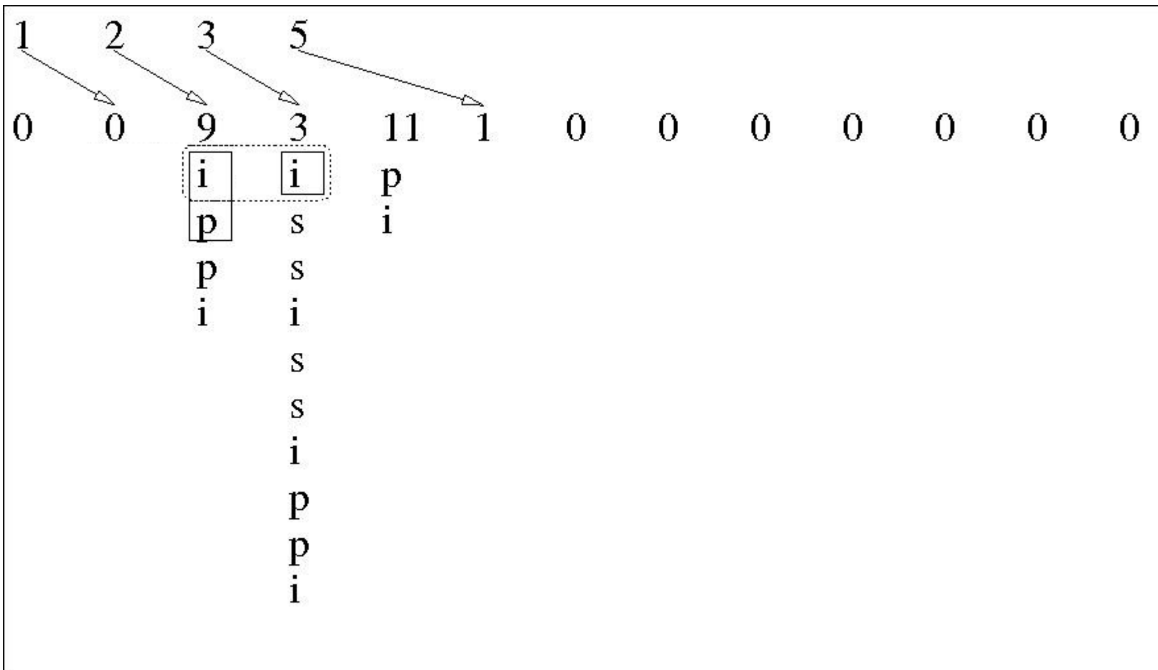


Figure 17. *Representation of the heap data structures before extracting a suffix (9).*

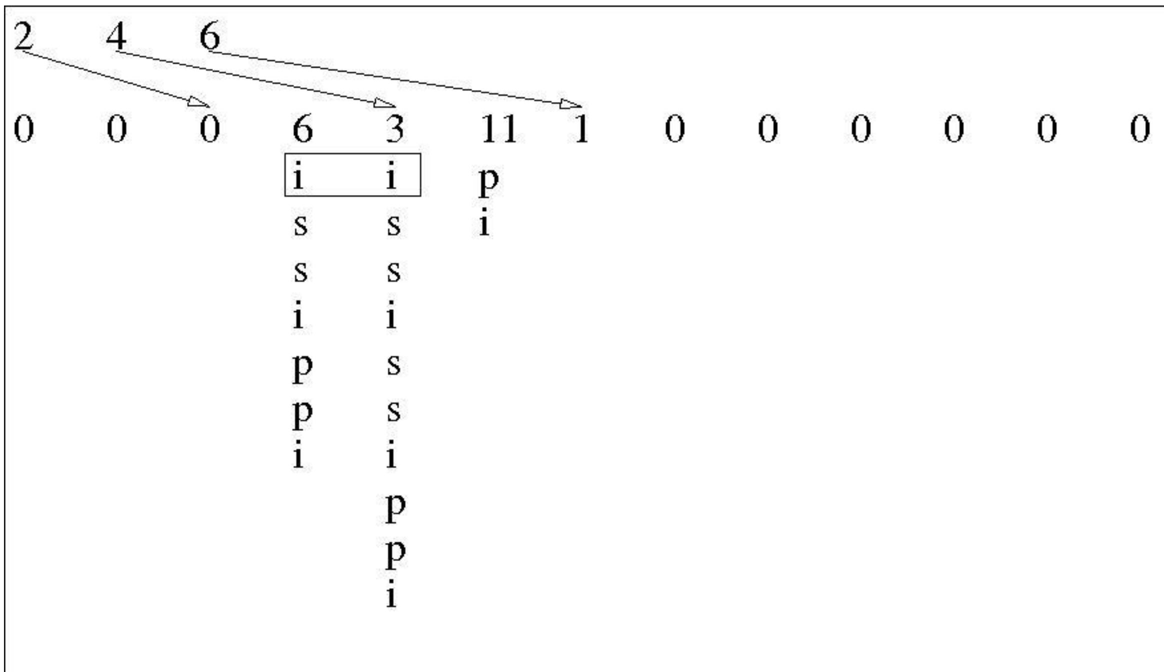


Figure 18. *Representation of the heap data structures before inserting a new suffix (6).*

Note, in Figure 18, that the new suffix (*suf* = 6) was inserted in an existing partition, sharing its existing LCP (*lcps*[1] = 1). Both were compared until the common LCP exhausted the partition LCP, then the new suffix passed to be compared with the sentinel, and proved greater. In this case it was inserted in the existing partition because of the previously obtained LCP, that matched the partition's.

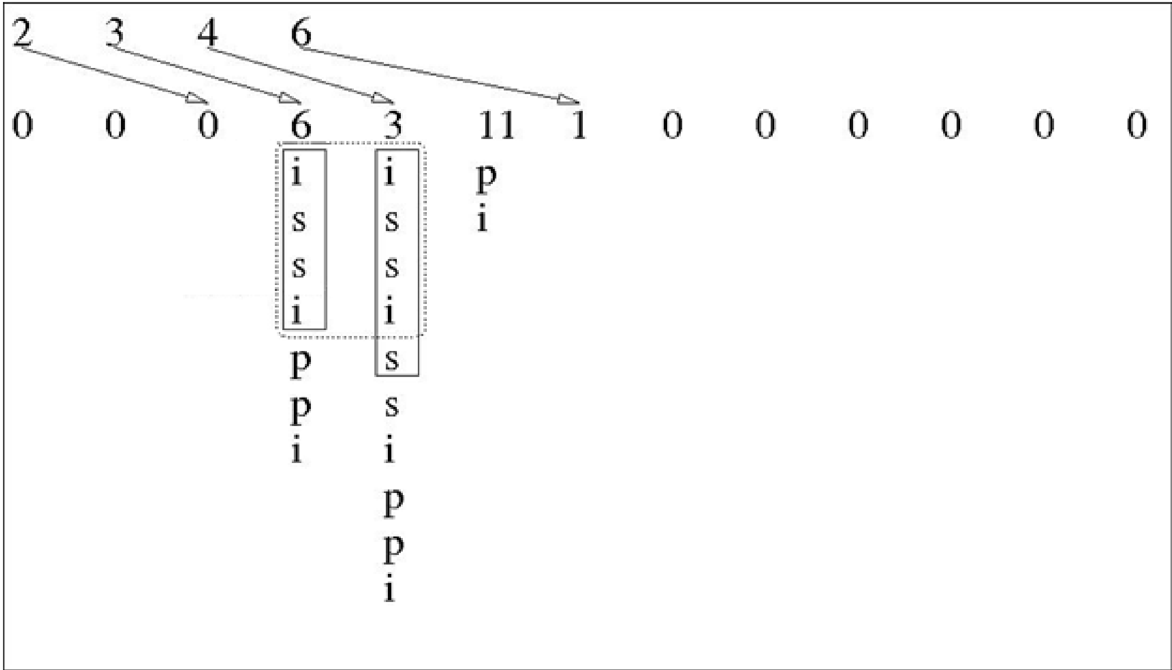


Figure 19. *Representation of the heap data structures before extraction of a suffix (6).*

In Figure 19 note the fact that the repartitioning function needed 4 iterations to divide the suffixes 3 and 6 in different partitions. Now that one pivot (*ar*[3] = 6) is in the first position of *ar* it will be popped and a new suffix index will be pushed into the heap, again from the partial suffix array 1.

Figure 20. *Representation of the heap data structures before extraction of a suffix (10).*

Figure 21. *Representation of the heap data structures before extraction of a suffix (8).*

Figure 20 shows the heap state after some extractions. We can see that the suffix *ar*[1] (10) is ready to be popped. Note, in Figure 21, that after that extraction no element came from the partial suffix array 2, since the suffix with index 10 was the last one from there. Identifying this fact the heap will not ask for new suffixes from that partial suffix array any more.

The heap execution continues until it reaches its final state, when all counters reach zero and there are no more elements in the buffers. Such state is illustrated by Figure 22, where only the sentinels remain.

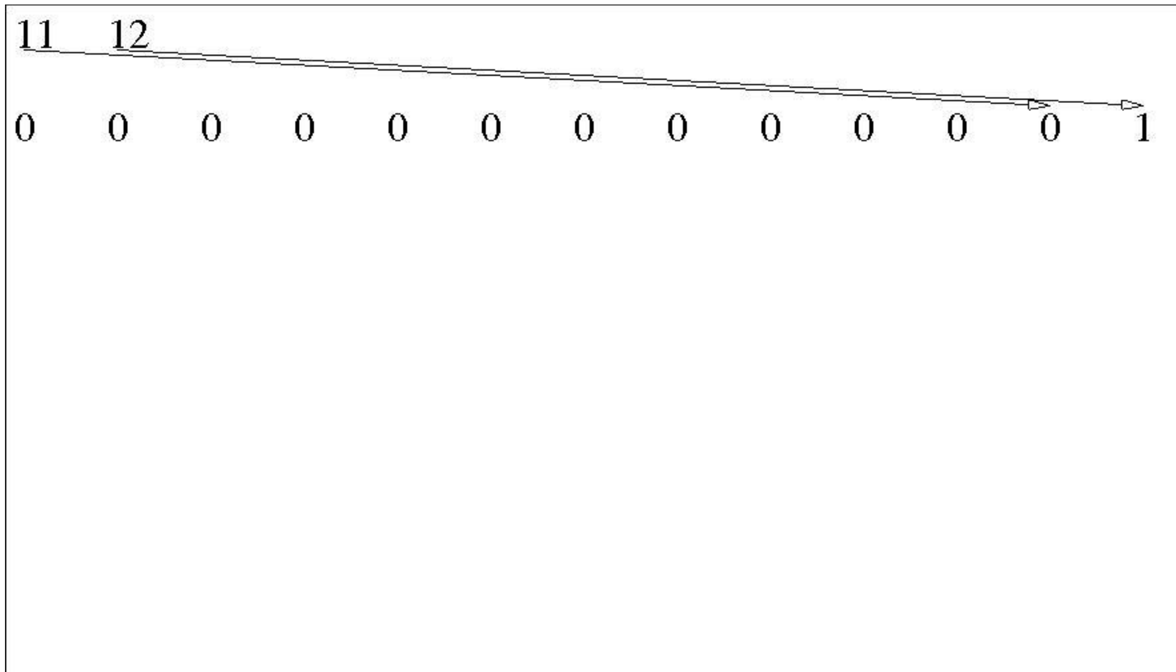


Figure 22. *Representation of the heap after all extractions.*

6. Experimental Validation

In this section we show the experimental tests we made to our prototype. It is divided in two parts: comparison between different configurations of our prototype, and comparison between our prototype and the state of the art algorithm in obtaining the Burrows-Wheeler Transform in secondary memory. All the files used in the tests shown were downloaded from the pizza and chili website [8].

6.1 Prototype Configuration

All tests in this sub-section were performed for the same input text, *dblp.xml* trunked at 100 MB. In this sub-section we tested the best number of partial suffix arrays, and the best buffer and heap sizes.

6.1.1 **Number of Partial Suffix Arrays**

This is a relevant parameter to the running time of our algorithm, and is inversely proportional to the partial SA size. Figures 23- 25 show the experimental results for changes on this parameter.

Number of PSA	PSA Size	Memory Peak (Splitting)	Memory Peak (Merging)	Splitting Time	Merging Time	Total Time
5	20	810	100	193	183	376
10	10	458	100	185	203	388
20	5	282	100	178	212	390
25	4	245	100	175	228	403
40	2,5	192	100	171	235	406
50	2	173	100	170	281	451
100	1	137	100	161	326	487
1000	0,1	103	100	100	388	488

Figure 23. *Table with the experimental results for different values of partial suffix arrays (PSA). Column 1 shows the number of PSA, column 2 shows the size of each of them (in MB). Columns 3 and show the memory usage of this configuration in the splitting part and merging part (in MB). Columns 5, 6 and 7 show the running time in the splitting part, merging part and total (in seconds).*

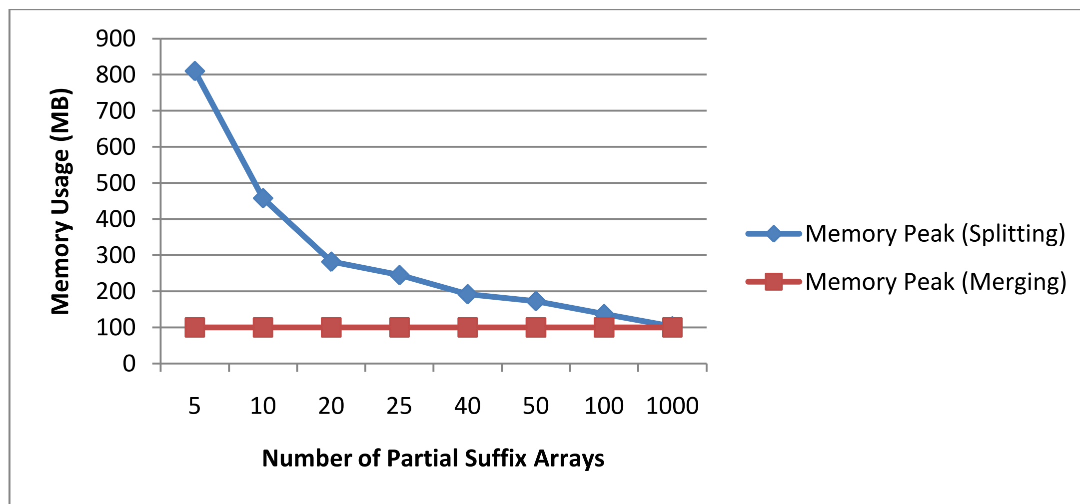


Figure 24. *Chart illustrating the data from Figure 23 that refers to memory usage. Both lines shown are the memory usage (MB) as a function of the number of partial suffix arrays.*

Results illustrated in Figure 24 show that the memory usage in the merging part isn't affected by changes on the partial suffix arrays sizes and numbers. This is a predictable result since neither the heap nor the buffers will contain a significantly different number of elements due to changes in the number of partial suffix arrays.

However, the memory usage in the splitting part changes significantly: the less partial suffix arrays there are, the more memory the suffix tree algorithm uses. This is a predictable result as well. For the same text, the fewer chunks it is divided in, the bigger each one of them is; thus the suffix tree of each individual chunk is also larger what takes more memory to build, leading to a higher memory peak. If we define a large number of partial suffix arrays, e.g. 1000, the memory consumption is about the size of the text, meaning that the working data for building the suffix tree is almost only the text.

For any partial suffix array number the splitting part memory peak is also the overall memory peak, what means that it is also dominant over the merging part memory peak. In this configuration the merging part only used memory to store the text, since both the heap and the buffer have residual values.

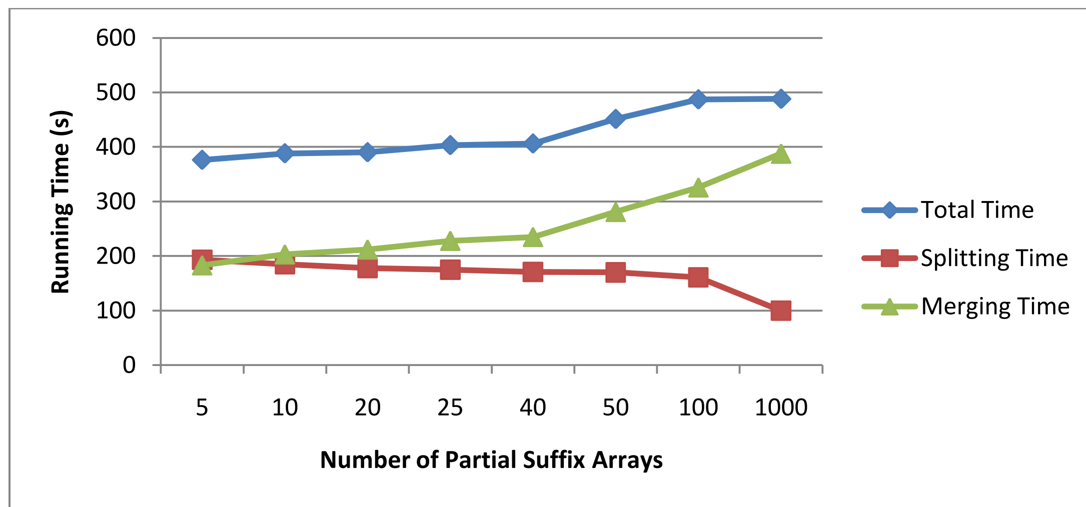


Figure 25. *Chart illustrating the data from Figure 23 that refers to running time. All the three lines shown are running time (seconds) as a function of the number of partial suffix arrays.*

In terms of running time, while the splitting part takes less time as the number of buffers increases, the merging part does the opposite. Yet, the overall running time increases slightly as the number of partial suffix arrays increases.

The suffix tree algorithm used in the splitting part takes the most time building the suffix tree and following its branches. As we see in Figure 25, the running time of this part is faster if we have more chunks of smaller size, rather than fewer chunks of bigger size. This happens because with small chunks the suffix tree construction is a simple process, while building and following the branches of a complex suffix tree has a bigger cost.

In each partial suffix array the suffixes are sorted; also, the less partial suffix arrays we have the more suffixes each one contains. Thus, we conclude that in a context with a small number of partial suffix arrays, a significant part of the sorting process is performed in the splitting part, rather than in the merging part. For, the merging part has lower running times with fewer partial suffix arrays.

Overall the additional cost in the merging part overwhelms the smaller running time of the splitting part for bigger numbers of partial suffix arrays, thus the faster running times of our algorithm are obtained with few partial suffix arrays.

6.1.2 Buffer Size

We also tested the size of the buffers. Figures 26 and 27 show the experimental results we obtained:

Buffer Size	Memory Usage	Running Time
0,0001	100	198
0,001	100	195
0,01	100	198
0,1	104	196
1	140	197
10	500	198

Figure 26. *Table with the experimental results for different buffer sizes. Column 1 shows the buffer size, column 2 shows the memory usage in the merging part (in MB). Column 3 shows the running time of the merging part (in seconds).*

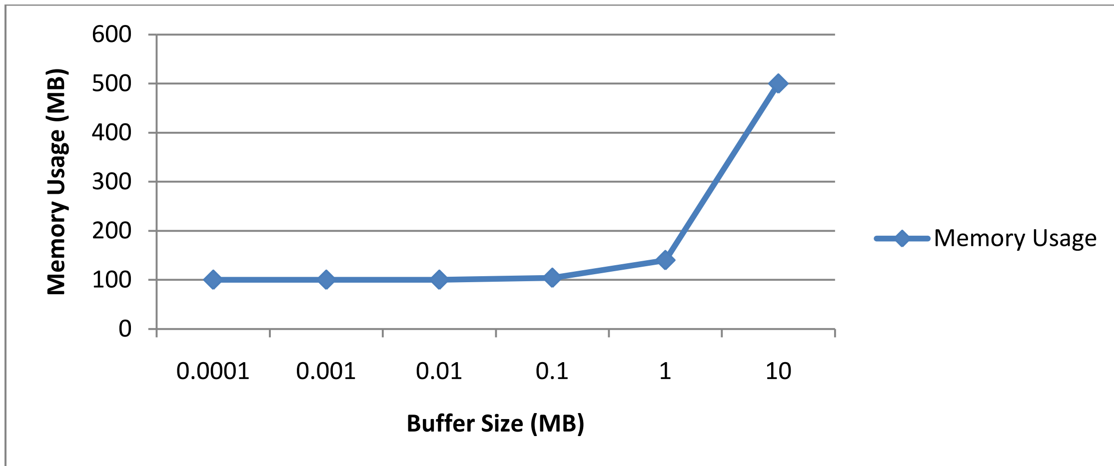


Figure 27. *Chart showing the data from Figure 26 that refers to memory usage. The line traced is the memory usage (in MB) as a function of the buffer size (in MB).*

Experimental results show that the buffer size has no major influence on the algorithm's running time; it remains about the same despite the changes in the buffer size, as we can see in Figure 26. Yet, the memory usage increases very much from a certain buffer size on, what is a natural consequence of having to allocate more main memory for the same number of buffers.

These results mean that no matter how many suffixes we bring to main memory on each read, I/Os still have great influence over the running time.

6.1.3 **Heap Size**

Increasing the number of suffixes in the heap also increases the main memory usage. Figures 28 and 29 show the experimental results for the memory usage and running time changes according to the change of the number of suffixes in the heap.

Note that we show the suffixes of each partial suffix array that can be in the heap at a given moment, rather than the total number of suffixes in the heap. The total number is, therefore, dependent on the number of partial suffix arrays.

Suffixes per Partial SA	Memory Usage (Merging)	Running Time (Merging)
1	100	196
10	100	412
100	100	383
1000	100	372
10000	100	392
100000	108	424
1000000	180	498
10000000	900	945

Figure 28. *Table with the experimental results for different numbers of suffixes in the heap. Column 1 shows the elements of each partial suffix array in the heap, column 2 shows the memory usage in the merging part (in MB). Column 3 shows the running time of the merging part (in seconds).*

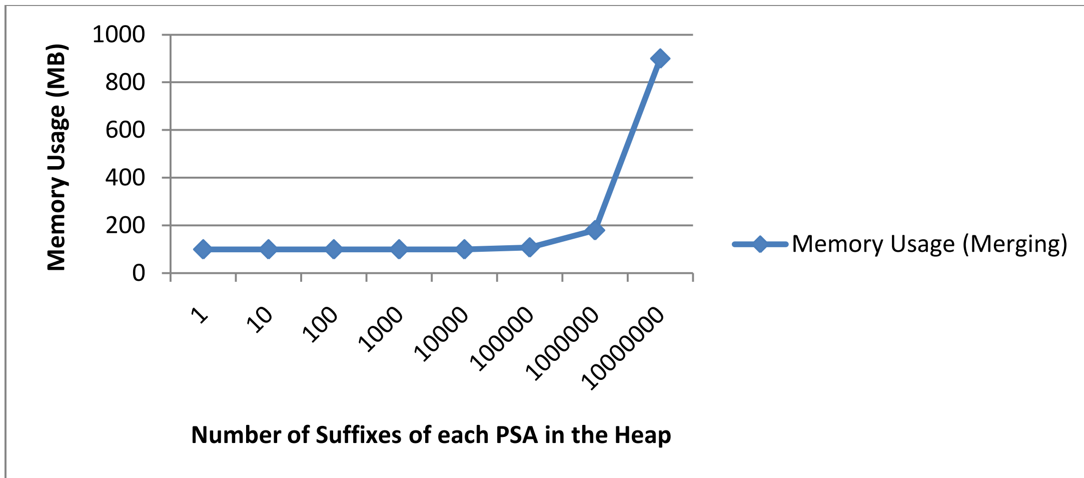


Figure 29. *Chart with the data from Figure 28 that refers to memory usage. The line traced is the memory usage as a function of the number of suffixes of each partial suffix arrays in the heap.*

Each suffix (1 byte) added to the heap means marginal increase in the memory usage, however if we increase it in 1 MB from each partial suffix array, the memory usage bursts. This is very much predictable, what we wanted to analyze with this test was whether the increase in memory was compensated by a decrease in running time, which as Figure 30 documents, didn't happen.

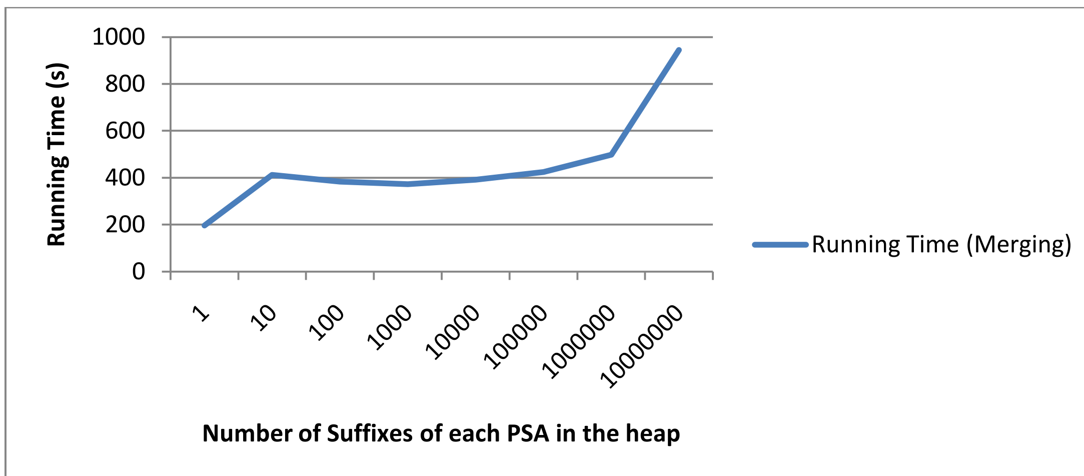


Figure 30. *Chart with the data from Figure 28 that refers to running time. The line traced is the running time as a function of the number of suffixes of each partial suffix arrays in the heap.*

As we can see, flooding the heap with suffixes isn't efficient. Despite having more elements in main memory, the running time is greater.

The first value shown on the chart corresponds to the same configuration used in the previous sub-section, where there was only one element of each partial suffix array in the heap at each moment. As the chart documents, this is the best configuration to the heap, since the running time easily doubles with the insertion of more suffixes of each partial SA. This fact is due to the great increase of the cost of inserting new suffixes in the heap. With more suffixes there are more pivots to compare new suffixes against on insertions, and with a text of 100 MB the insertion operation runs 100000000, thus a slight increase in the complexity of each single operation means a great overall time increase, as the slope in Figure 30 illustrates.

6.2 Version with Blocks of Suffixes

The version that builds the partial suffix arrays using blocks of suffixes, instead of individual suffixes, shares the core functions with the base version, so all the tests performed above are also valid for this one, since the parameters discussed also exist in this version.

There is only one parameter that we need to test especially for this version: the size of the LCP from which the blocks were built. Figures 31, 32 and 33 show the experimental results we obtained testing this parameter.

LCP	Splitting Memory	Merging Memory	Splitting Time	Merging Time	Total Time
2	192	112	185	461	646
6	192	112	180	425	605
10	192	112	183	415	598
15	192	112	182	382	564
20	192	112	185	373	558
25	192	112	183	365	548
30	192	112	180	355	535
40	192	112	184	328	512
50	192	112	182	336	518
100	192	112	184	298	482
200	192	112	184	283	467
1000	192	112	183	282	465

Figure 31. *Table with the experimental results for different LCP values. Column 1 shows the LCP value, columns 2 and 3 show the memory usage in the splitting part and in the merging part (in MB). Columns 4, 5 and 6 show the running times of the splitting part, merging part and total (in seconds).*

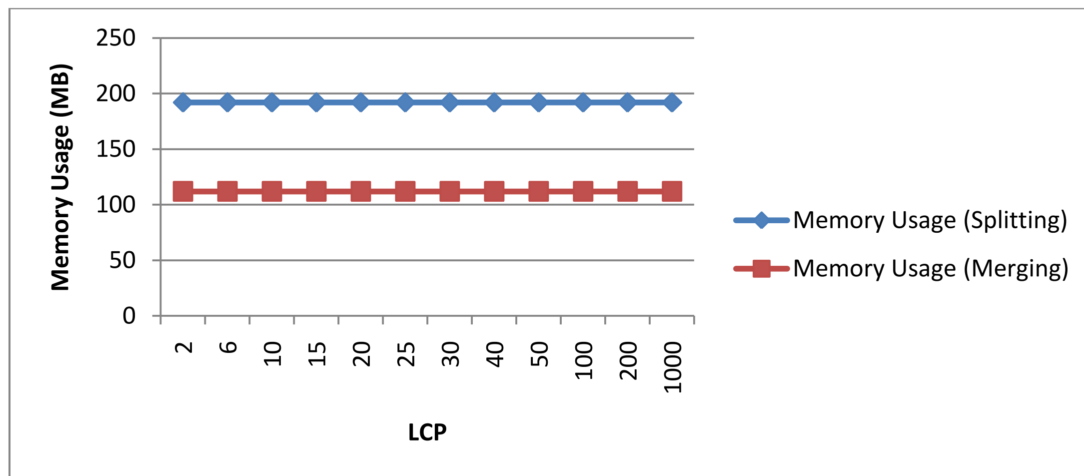


Figure 32. *Chart showing the data from Figure 31 that refers to the memory usage. The line traced is the memory usage in the merging part as a function of the LCP size set for the blocks of suffixes.*

As Figure 32 documents, memory usage both in the splitting part and in the merging part remains constant when the LCP value changes.

In the splitting part this happens because even with the LCP changes the partial suffix arrays still contain the same number of suffixes, and as we discussed in the previous sub-sections, that's the parameter that influences memory usage the most.

In the merging part, as we mentioned, the lesser the LCP the bigger the blocks, thus bigger blocks mean more suffixes in the heap at each time. However, these numbers are still not significant, and even the shortest LCP values would produce unnoticeable changes to the chart trace.

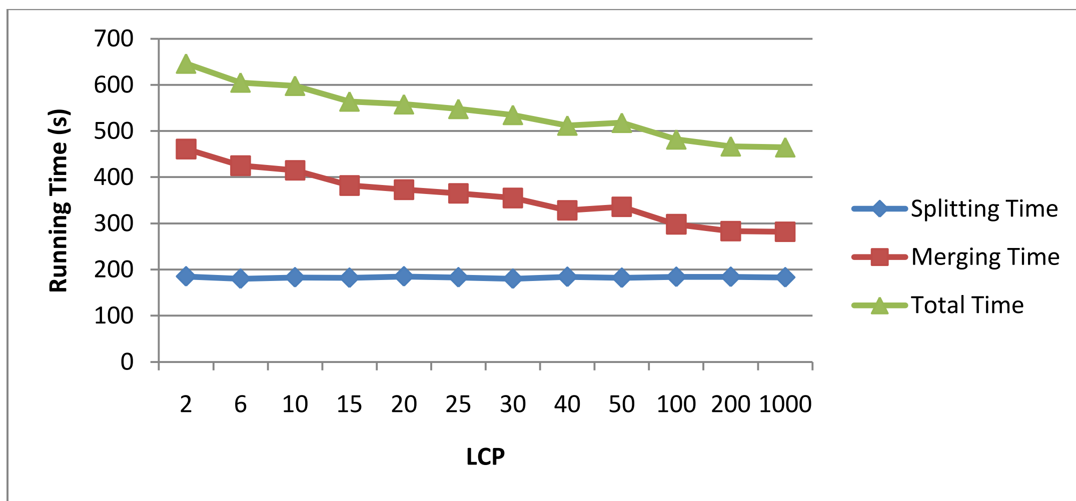


Figure 33. *Chart showing the data from Figure 31 that refers to the running time. The line traced is the running time of the merging part as a function of the LCP size set for the blocks of suffixes.*

The splitting running time remains constant, since the size of the chunks and the number of partial suffix arrays are the same. The experimental results illustrated both in Figure 32 and 33 confirm what we mentioned in the introduction of the splitting part of the algorithm: the suffix tree algorithm we use takes the same amount of memory and the same running time to build the suffix tree and pop results, either popping the LCPs alongside the suffixes or not.

The merging part running time is the one that is affected the most by changes upon the LCP value. As we see in Figure 33, the smaller the blocks the best the algorithm performs. This means that having blocks with significant LCP sizes in the heap is not as efficient as we expected. This is probably due to the number of verifications we need to perform when inserting a block that we don't need to do when inserting single suffixes.

When inserting a block the program needs to check if it will be inserted near the bound of the heap; in such a case the block has to be divided in two. One of those parts is inserted in the last positions of the heap's *ar* array, while the remaining are inserted in its first positions. This happens because we use a circular array, however this is a dispensable verification when inserting a single suffix. Despite producing a marginal increase in the running time of a single block insertion, these extra verifications get perfectly obvious in the overall picture, as Figure 33 documents.

6.3 State of the art

In this section we compare our algorithm against the state of the art in this research area. We covered the two faster algorithms in the related work of this document

The algorithm proposed by Ferragina et al. [7] is the most efficient at the moment to produce the Burrows-Wheeler Transform of a text, and that is the algorithm we use to compares ours against.

We also planned to run the second most efficient algorithm, proposed by Dementiev et al. [4], however we could not properly use the prototype from the link in the author's paper.

6.3.1 Different types of text

As we mentioned, text processing algorithms must be able to handle texts with different natures. By text nature we mean having or not string patterns, enabling the creation of large blocks with considerable LCPs, or otherwise almost random texts, hard to handle for a pattern driven approach.

In this section we compare the two versions of our algorithm between them and against the one proposed by Ferragina et al. [7] (bwtdisk), trying to indentify each one's capacity of handling different types of text.

In each test we use fixed amounts of memory and measure the running times of each algorithm for each of them.

First we compare the three algorithms using the 100 MB XML file we used in the previous subsections: *dblp.xml* trunked at 100 MB. Figures 34- 37 document our experimental results on this file.

dblp.xml.100MB

	Heap Without Blocks			Heap With Blocks			bwtdisk
Memory Usage	Running time (Splitting)	Running Time (Merging)	Total Running Time	Running time (Splitting)	Running Time (Merging)	Total Running Time	Running Time
100	100	389	489	146	503	649	503
190	169	230	399	184	283	467	420
270	178	212	390	196	258	454	349
450	185	195	380	201	269	470	196
800	193	183	376	209	227	436	25

Figure 34. *Table with the experimental results for the three algorithms with the file dblp.xml.100MB. Column 1 shows the memory usage (in MB), columns 2, 4 and 4 show the running times for the heap without blocks: in the splitting part, merging part and total (in seconds). Columns 5, 6 and 7 show the running times for the heap with blocks: in the splitting part, merging part and total (in seconds).*

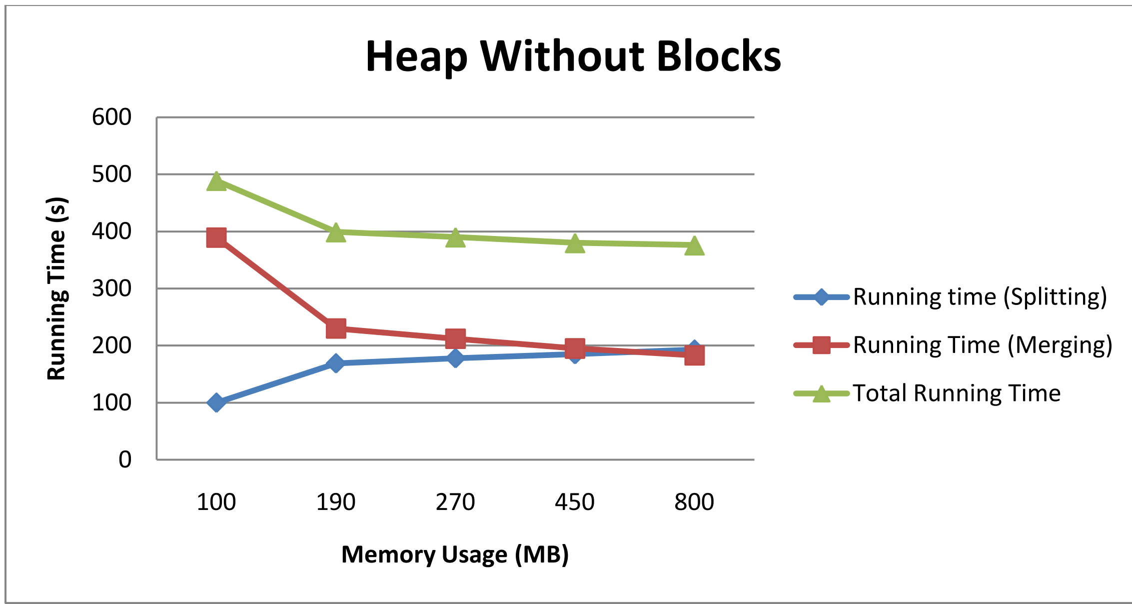


Figure 35. *Chart showing the data from Figure 34 that refers to the heap without blocks. All the three lines traced are the running time of that version, in the splitting part, merging part and total (in seconds).*

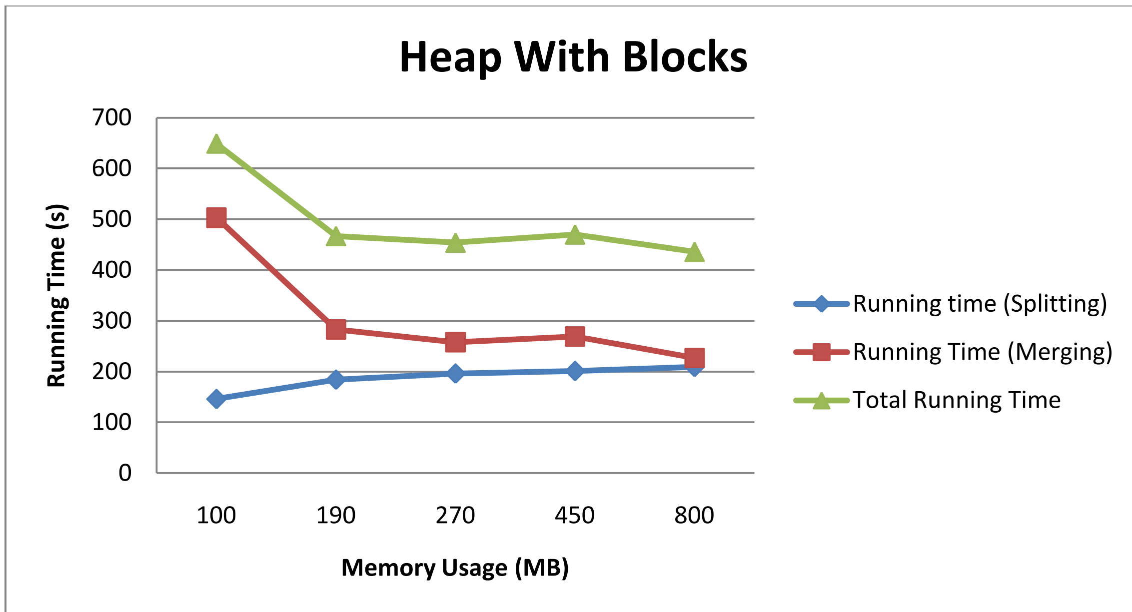


Figure 36. *Chart showing the data from Figure 34 that refers to the heap with blocks. All the three lines traced are the running time of that version, in the splitting part, merging part and total (in seconds).*

The slopes of both charts are not very different, however both the splitting part and the merging part take slightly longer in the version with blocks.

In Figure 37 we introduce the bwtdisk algorithm.

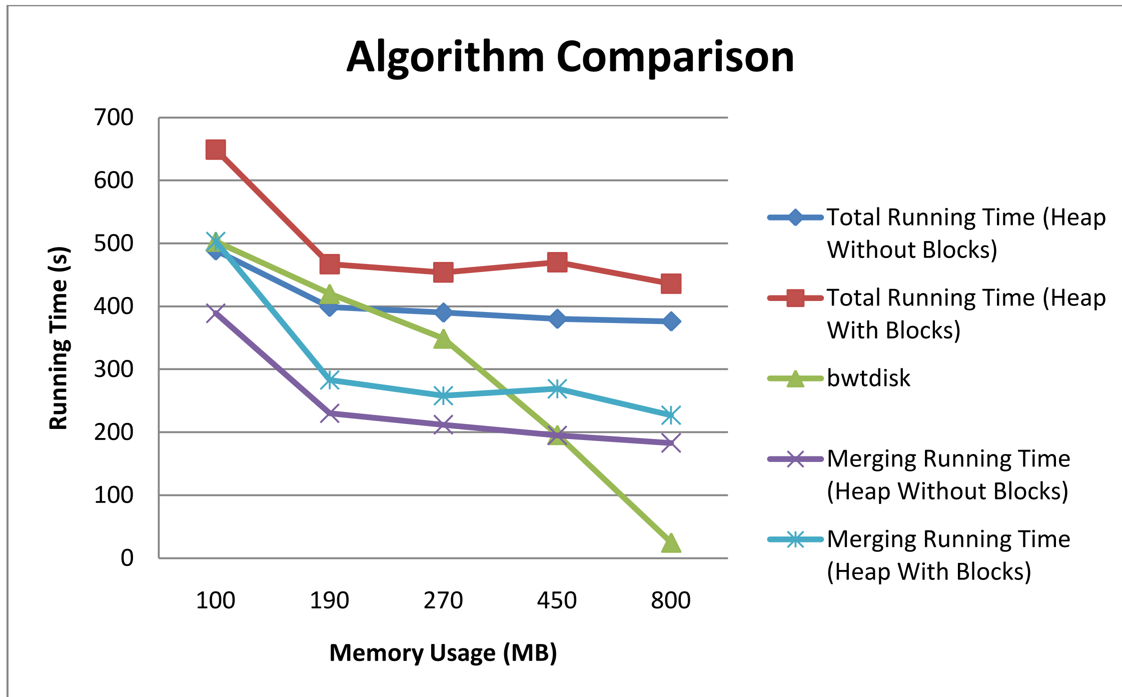


Figure 37. *Chart showing the data from Figure 34, comparing the running times of the three algorithms, and the merging part of both versions of our algorithm. All the five lines traced are running times (in seconds).*

We also show the traces for the merging parts in the charts comparing the three algorithms, since that is the main part of this work. The heap is the data structure we propose and whose results we want to compare against the other algorithms, despite building the BWT not from the text but from a set of suffix arrays.

The bwtdisk algorithm shows great advantage for more than 200 MB of memory, however our algorithm has similar running times with less memory, mainly for the version without blocks.

Here we compare the three algorithms using the 100 MB file that contains gene DNA sequences: *dna* trunked at 100 MB. Figures 38-41 document our experimental results on this file.

dna.100MB

	Heap Without Blocks			Heap With Blocks			bwtdisk
Memory Usage	Running time (Splitting)	Running Time (Merging)	Total Running Time	Running time (Splitting)	Running Time (Merging)	Total Running Time	Running Time
100	125	600	725	149	656	805	556
190	200	426	626	202	704	906	423
290	210	359	569	212	685	897	319
460	232	369	601	235	679	914	208
800	233	215	448	246	701	947	30

Figure 38. *Table with the experimental results for the three algorithms with the file dna.100MB. Column 1 shows the memory usage (in MB), columns 2, 4 and 4 show the running times for the heap without blocks: in the splitting, merging and total (in seconds). Columns 5, 6 and 7 show the running times for the heap with blocks: in the splitting, merging and total (in seconds).*

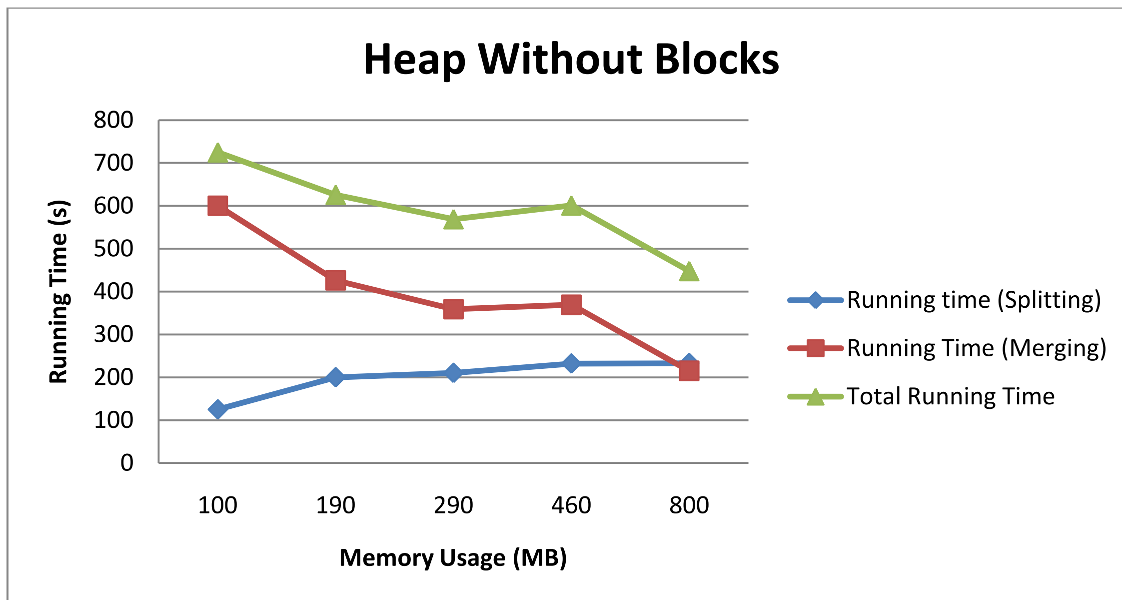


Figure 39. *Chart showing the data from Figure 38 that refers to the heap without blocks. All the three lines traced are the running time of that version, in the splitting part, merging part and total (in seconds).*

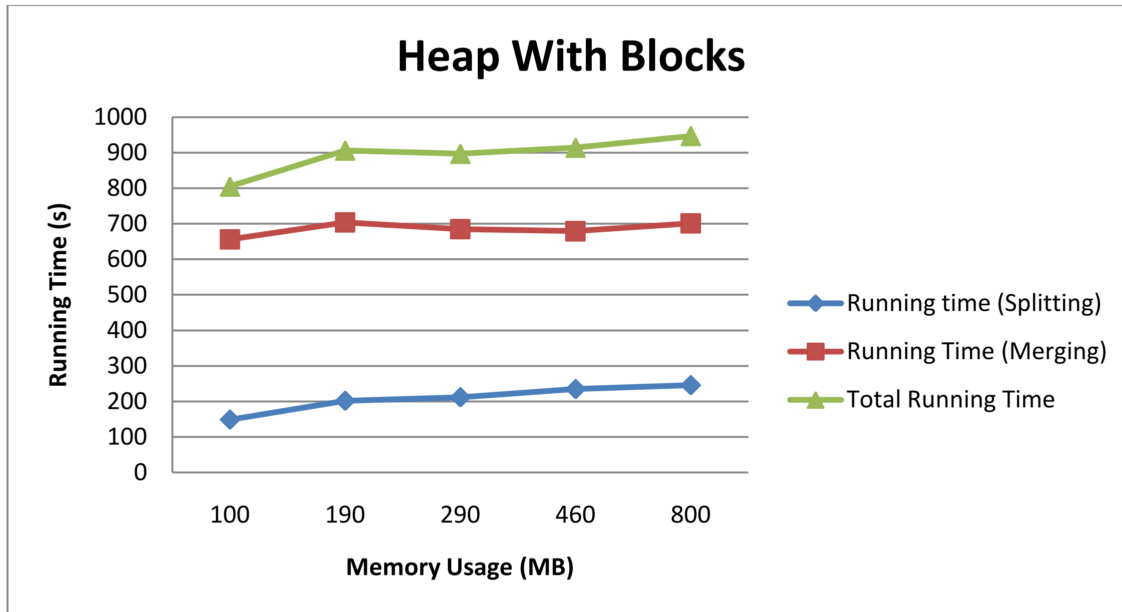


Figure 40. *Chart showing the data from Figure 38 that refers to the heap with blocks. All the three lines traced are the running time of that version, in the splitting part, merging part and total (in seconds).*

While the version without blocks showed the same behavior for this file than for the XML one, the version with blocks didn't. None of the parts performed faster when given more memory, instead the merging part kept about constant and the splitting part increased its running time.

We didn't expect this behavior, however we recognize that the version with blocks is more sensitive to differences between file types, whether they contain more or less patterns.

We will expand this discussion on the conclusions section of this document.

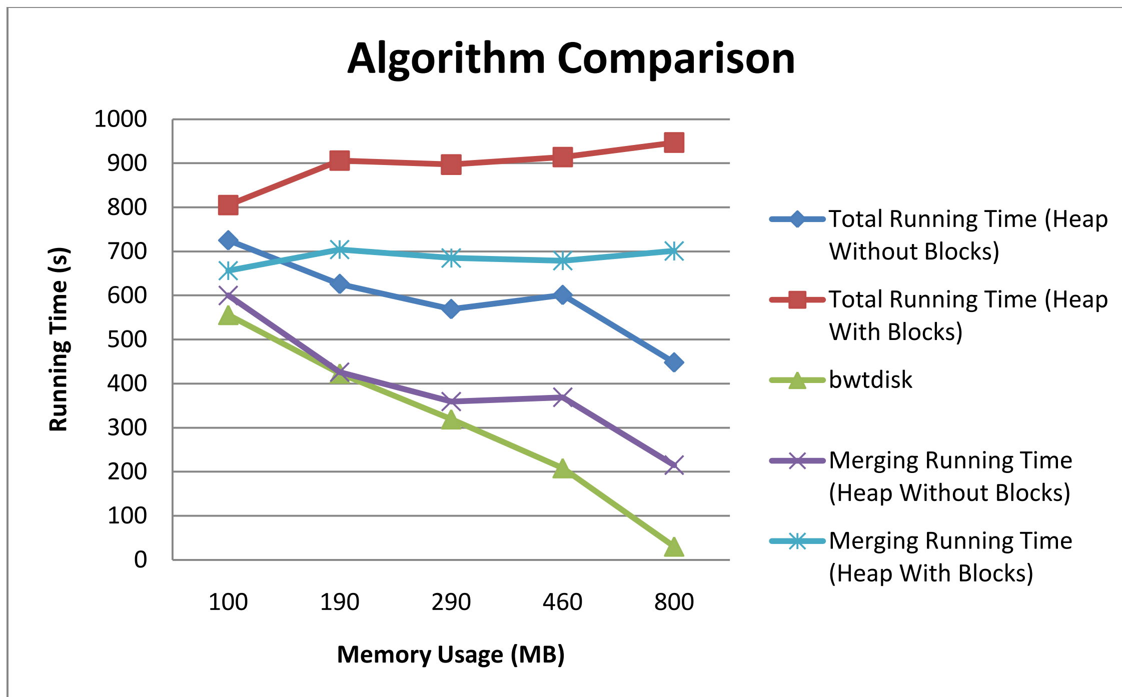


Figure 41. *Chart showing the data from Figure 38, comparing the running times of the three algorithms, and the merging part of both versions of our algorithm. All the five lines traced are running times (in seconds).*

The bwtdisk algorithm outperformed the total times of both versions of our algorithm with this file, however the merging part alone is very competitive when given less than 300 MB of memory, especially the merging part of the version without blocks.

Here we compare the three algorithms using the 100 MB file that contains protein sequences: *proteins* trunked at 100 MB

Figures 42- 45 document our experimental results on this file.

proteins.100MB

	Heap Without Blocks			Heap With Blocks			bwtdisk
Memory Usage	Running time (Splitting)	Running Time (Merging)	Total Running Time	Running time (Splitting)	Running Time (Merging)	Total Running Time	Running Time
100	136	1561	1697	159	1739	1898	503
190	271	972	1243	280	1321	1601	444
280	303	880	1183	299	1243	1542	326
460	328	727	1055	330	1234	1564	224
800	342	641	983	352	1201	1553	38

Figure 42. *Table with the experimental results for the three algorithms with the file proteins.100MB. Column 1 shows the memory usage (in MB), columns 2, 4 and 4 show the running times for the heap without blocks: in the splitting part, merging part and total (in seconds). Columns 5, 6 and 7 show the running times for the heap with blocks: in the splitting part, merging part and total (in seconds).*

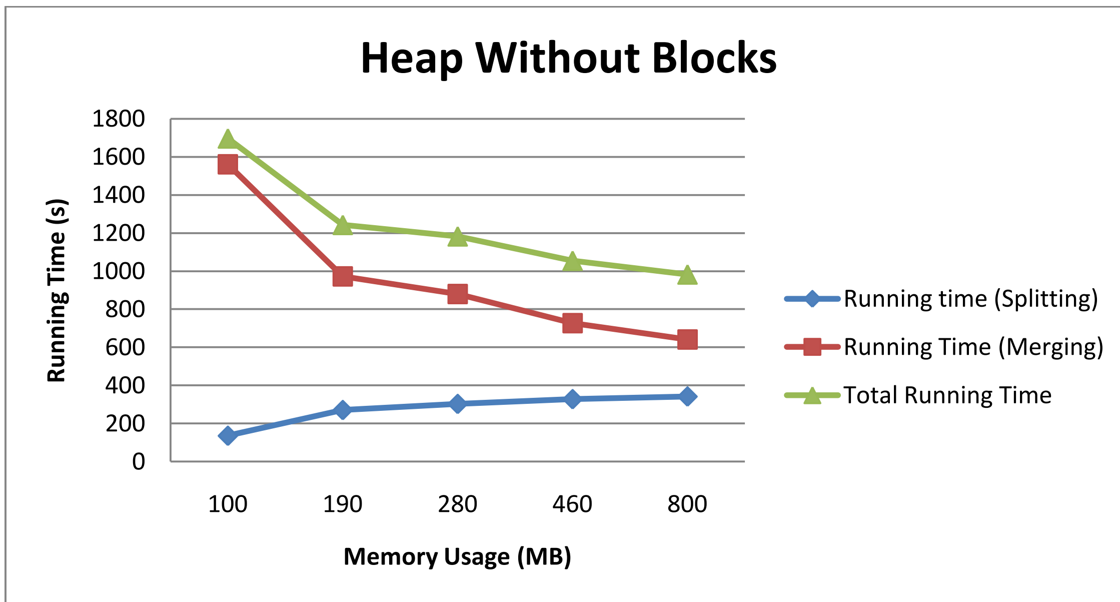


Figure 43. *Chart showing the data from Figure 42 that refers to the heap without blocks. All the three lines traced are the running time of that version, in the splitting part, merging part and total (in seconds).*

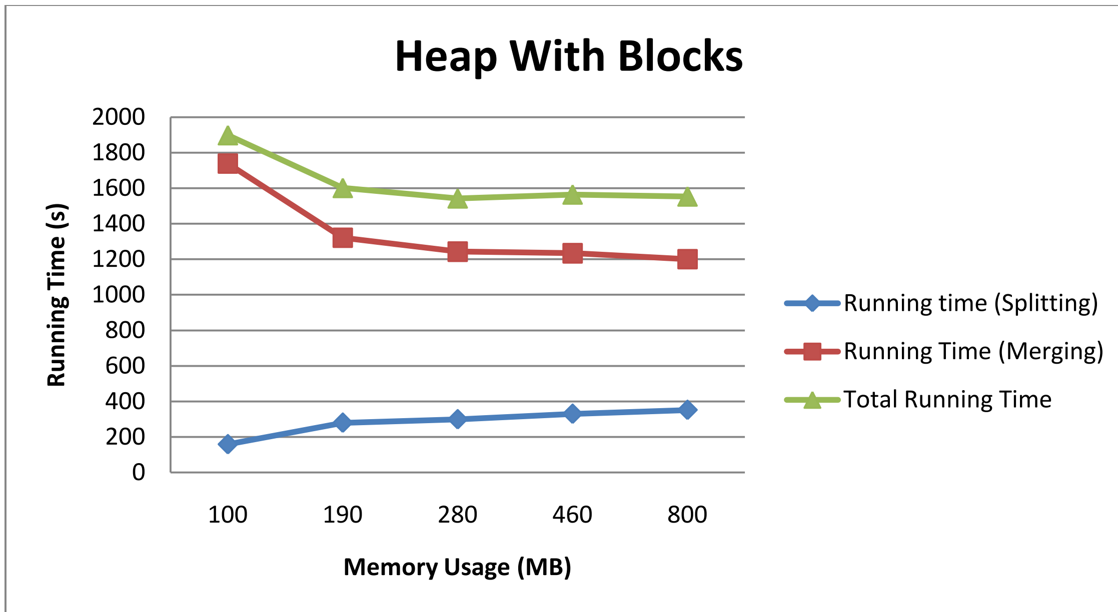


Figure 44. *Chart showing the data from Figure 42 that refers to the heap with blocks. All the three lines traced are the running time of that version, in the splitting part, merging part and total (in seconds).*

With this file, as with the previous one, the version without block shows the expected slope, i.e. decreasing the running time as the memory usage increases. However, with this file the running time is about the double, which was not expected, since the files have the same size.

The version with blocks also doubles the previous running time, thus the reason for this event is not a matter of versions of the algorithm but an issue about the algorithm itself.

The version with blocks also maintains the behavior shown in the previous file, i.e. the running time doesn't decrease when the memory usage increases.

This discussion is expanded in the conclusions section of this document.

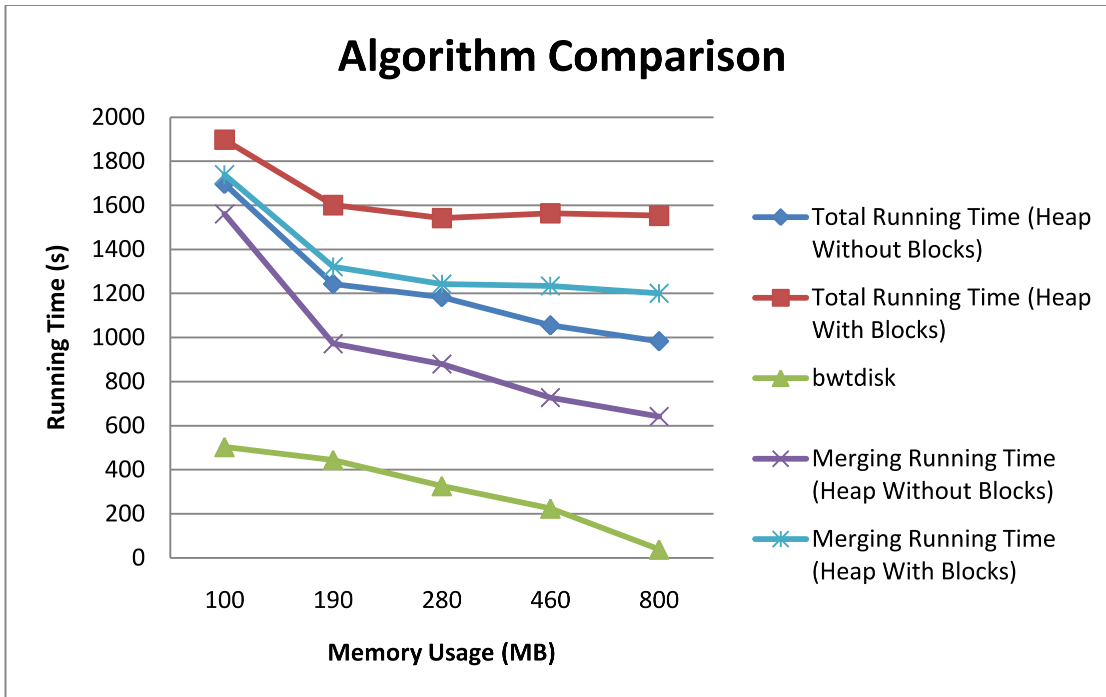


Figure 45. *Chart showing the data from Figure 42, comparing the running times of the three algorithms, and the merging part of both versions of our algorithm. All the five lines traced are running times (in seconds).*

With this specific file, our algorithm has a poor performance, no matter the version. The bwtdisk algorithm outperforms even the merging part alone.

6.3.2 **Diferent Text Sizes**

In this section we show how the three algorithms perform for different text sizes. We use the XML of the first comparison, but now trunked at 200 MB and 300 MB.

Figures 46- 49 document our experimental results on this file.

dblp.xml.200MB

	Heap Without Blocks			Heap With Blocks			bwtdisk
Memory Usage	Running time (Splitting)	Running Time (Merging)	Total Running Time	Running time (Splitting)	Running Time (Merging)	Total Running Time	Running Time
235	327	785	1112	358	785	1143	888
270	349	561	910	368	681	1049	883
345	365	526	891	400	602	1002	832
550	386	473	859	433	563	996	577
920	409	389	798	438	638	1076	414

Figure 46. *Table with the experimental results for the three algorithms with the file dblp.xml.200MB. Column 1 shows the memory usage (in MB), columns 2, 4 and 4 show the running times for the heap without blocks: in the splitting part, merging part and total (in seconds). Columns 5, 6 and 7 show the running times for the heap with blocks: in the splitting part, merging part and total (in seconds).*

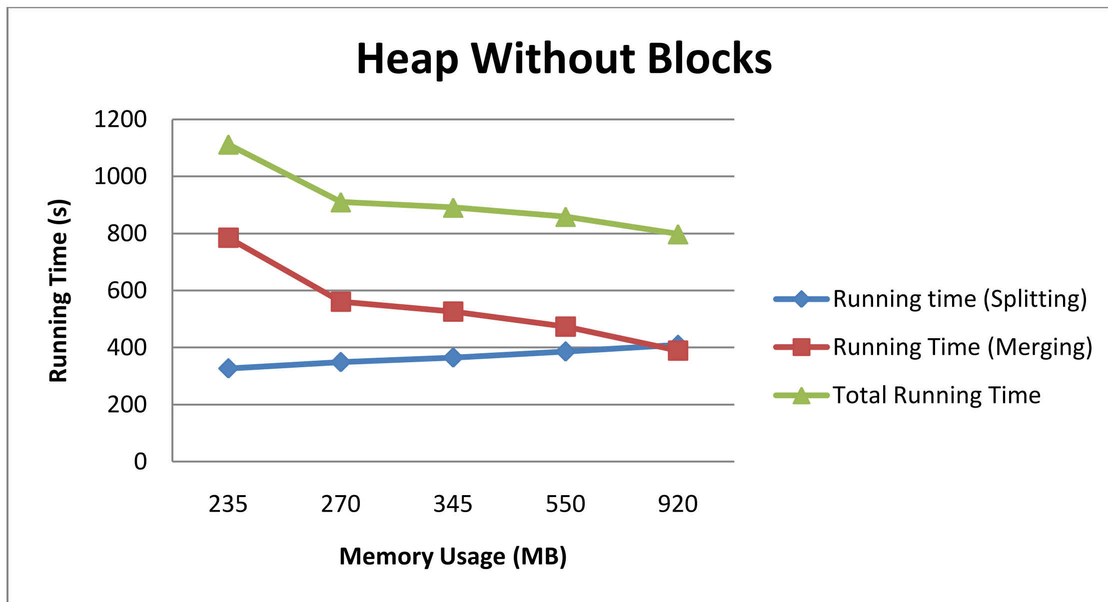


Figure 47. *Chart showing the data from Figure 46 that refers to the heap without blocks. All the three lines traced are the running time of that version, in the splitting part, merging part and total (in seconds).*

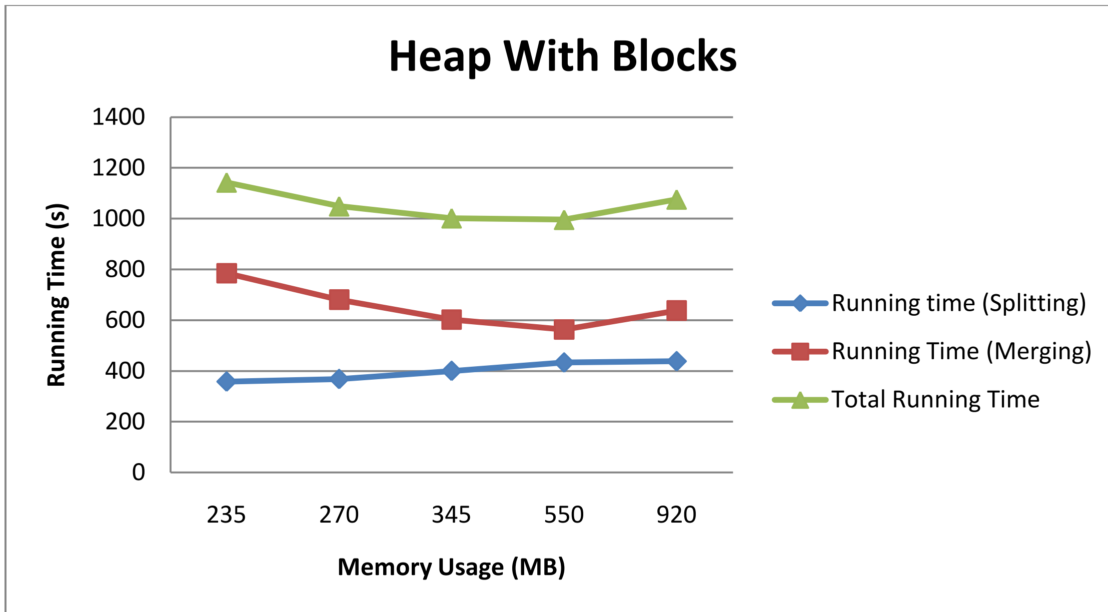


Figure 48. *Chart showing the data from Figure 46 that refers to the heap with blocks. All the three lines traced are the running time of that version, in the splitting part, merging part and total (in seconds).*

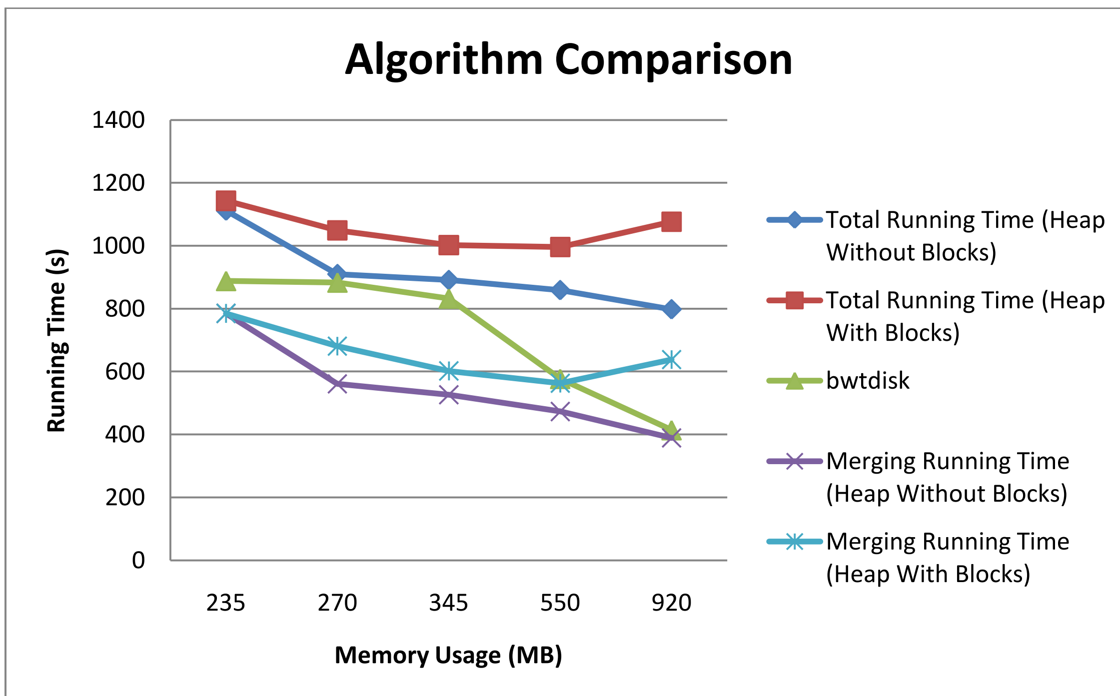


Figure 49. *Chart showing the data from Figure 46, comparing the running times of the three algorithms, and the merging part of both versions of our algorithm. All the five lines traced are running times (in seconds).*

The similarity between the slopes of each algorithm in Figures 35-37 and in Figures 47-49 documents that the increase of the text size doesn't affect very much the performance of our algorithm nor its running time relatively to the bwtdisk running time.

Finally, we also compare all three algorithms with a 300 MB XML file. Figures 50- 53 document our experimental results on this file.

dblp.xml.300MB

	Heap Without Blocks			Heap With Blocks			bwtdisk
Memory Usage	Running time (Splitting)	Running Time (Merging)	Total Running Time	Running time (Splitting)	Running Time (Merging)	Total Running Time	Running Time
300	486	1148	1634	557	1249	1806	1464
500	545	713	1258	587	925	1512	1113
700	573	636	1209	639	689	1328	988
1200	611	620	1231	675	845	1520	599

Figure 50. *Table with the experimental results for the three algorithms with the file dblp.xml.300MB. Column 1 shows the memory usage (in MB), columns 2, 4 and 4 show the running times for the heap without blocks: in the splitting part, merging part and total (in seconds). Columns 5, 6 and 7 show the running times for the heap with blocks: in the splitting part, merging part and total (in seconds).*

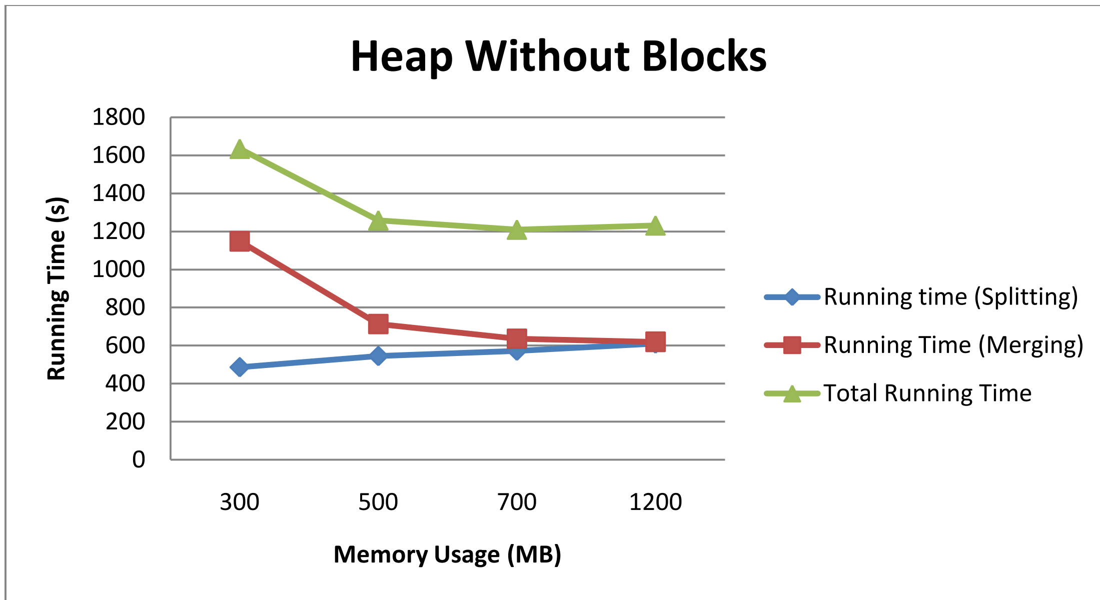


Figure 51. *Chart showing the data from Figure 50 that refers to the heap without blocks. All the three lines traced are the running time of that version, in the splitting part, merging part and total (in seconds).*

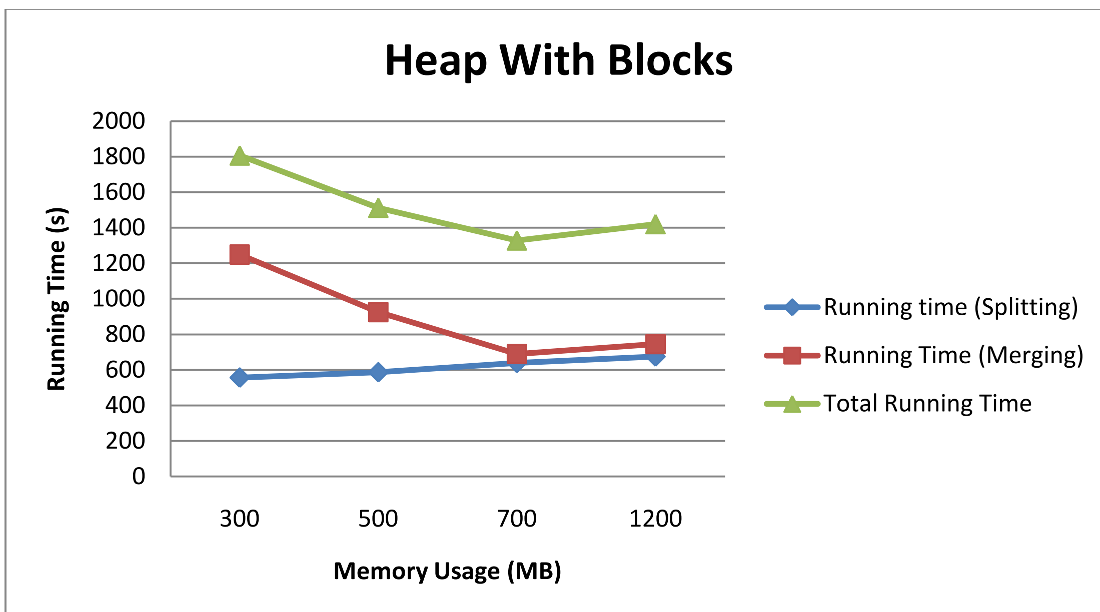


Figure 52. *Chart showing the data from Figure 50 that refers to the heap with blocks. All the three lines traced are the running time of that version, in the splitting part, merging part and total (in seconds).*

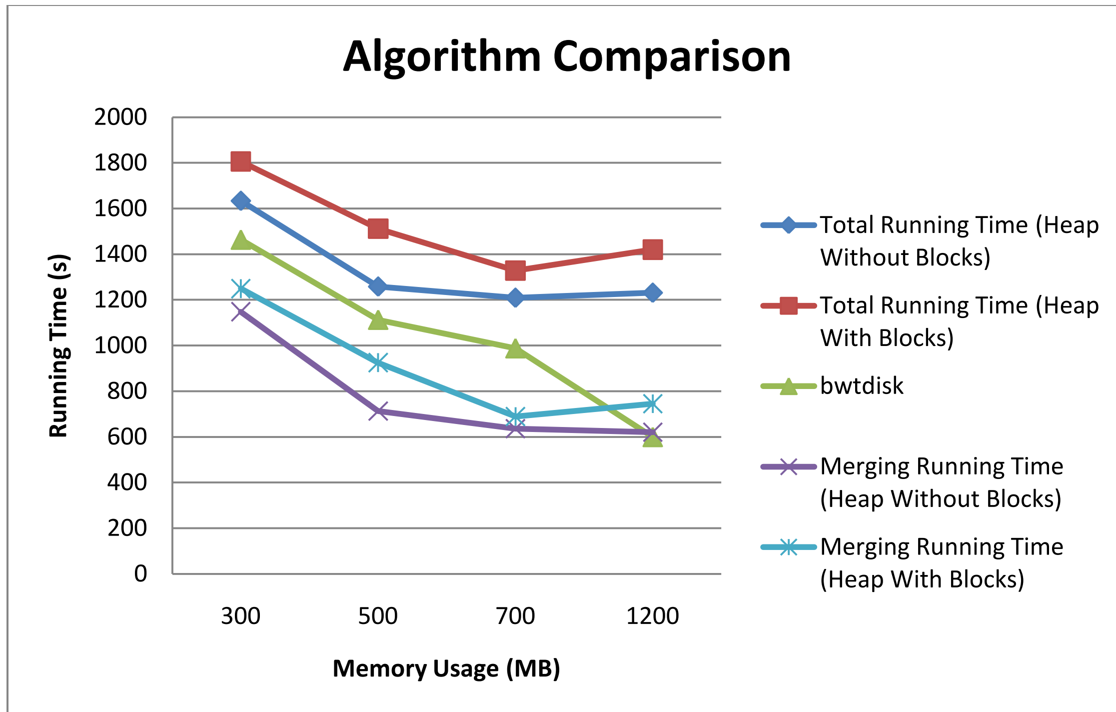


Figure 53. *Chart showing the data from Figure 50, comparing the running times of the three algorithms, and the merging part of both versions of our algorithm. All the five lines traced are running times (in seconds).*

From the test we showed in this section we conclude that our algorithm maintains its competitiveness against the bwtdisk algorithm. Thus, the size of the input text is not a concern for our algorithm, since its running time increases linearly, as well as the bwtdisk.

7. Conclusions

This work aimed to propose a new algorithm to build suffix arrays and the Burrows-Wheeler Transform of a text using secondary memory. We proposed an innovative solution, creating a heap data structure especially designed to handle strings.

We divided our solution in two parts altough the main focus of this work was the merging part. The creation of the String Heap is the greatest contribution of this work. For the splitting part we needed a linear algorithm to build suffix arrays using main memory, thus we used a suffix tree algorithm that performed acceptably. However in this section we discuss the merging rather than the splitting part, since it is our main focus and the one where we had the most unexpected results.

Our algorithm has to deal with the dominance of disk access over main memory access, which in practice is hard to overcome. Our tests, however, showed that even pushing more suffixes into main memory makes little difference. This confirms that the design of a streaming algorithm depends considerably on the time to access secondary memory.

We also created a second version of our algorithm, which aimed to make up some text types where potentially the base version would perform poorly. The use of blocks of suffixes grouped by their LCP intended to perform faster than the version without blocks for texts with lots of patterns. However this version didn't produce the expected results, showing lower running times when we approach it to the base version, i.e. with smaller blocks.

The tests proved that our solution is scalable, maintaining the results proportionally to the increase in the input text, which was one of our main goals. Yet, the type of file the algorithm receives has a significant impact in the running times of our algorithm. Although we didn't expect such an outcome, these results are probably due to the $\overline{lcp}$ factor. Each insertion operation runs in $O(p + \overline{lcp})$ time on average, which makes it sensitive to changes in the $\overline{lcp}$, leading it to

produce very different overall times for files with different pattern incidences. The version with blocks is even more sensitive to changes in the type of file, because files with more patterns produce more blocks, while fewer patterns mean less blocks. This partially explains the differences in the results of both versions of the algorithm, and also the fact that for some file types the running time of the version with blocks doesn't decrease when the memory usage increases.

Hence in this thesis we designed and implemented a secondary memory suffix array algorithm that is competitive against state of the art alternatives. We obtain better performance than bwtdisk in a few restricted configurations, and slower performance for more general cases. Moreover, using the relation between the performance of bwtdisk and the DC3 algorithm studied by Ferragina et al. [7], our algorithm always achieves in between performance.

8. Bibliography

[1] Navarro, Gonzalo and Makinen, Veli. *Compressed full-text indexes*. ACM Comput. Surv. 39. 1. New York, USA. 2007.

[2] Bentley, Jon L. and Sedgewick, Robert. *Fast Algorithms for Sorting and Searching Strings*. SODA ’97: Proceedings of the eighth annual ACM-SIAM symposium on Discrete Algorithms. Pages 360-369. Society for Industrial and Applied Mathematics, Philadelphia, USA. 1997.

[3] Navarro, Gonzalo and Paredes, Rodrigo. *On Sorting, Heaps, and Minimum Spanning Trees*. Algorithmica, Springer. 2009.

[4] Dementiev, Roman and Karkkainen, Juha and Mehnert, Jens and Sanders, Peter. *Better External Memory Suffix Array Construction*. J. Exp. Algorithmics. 12. Pages 1-24. ACM. New York, USA. 2008.

[5] Karkkainen, Juha. *Fast BWT in Small Space by Blockwise Suffix Sorting*. Theor. Comput. Sci. 387. 3. Pages 249-257. Elsevier Science Publishers Ltd. Essex, UK. 2007.

[6] Puglisi, Simon J. and Smyth, W. F. and Turpin, Andrew H. *A Taxonomy of Suffix Array Construction Algorithms*. ACM Comput. Surv. 39. 2. Page 4. ACM. New York, USA. 2007.

[7] Ferragina, Paolo and Gagie, Travis and Manzini, Giovanni. *Lightweight Data Indexing and Compression in External Memory*. LATIN. 2010.

[8] http://pizzachili.dcc.uchile.cl/texts.html