

Wolfgang Gerlach

Wolfgang.Gerlach@CeBiTec.Uni-Bielefeld.DE  
 AG Genome Informatics, Technical Faculty, Bielefeld University, Germany

## Introduction

We have implemented a dynamic FM-index for a collection of texts, a self-index whose space usage is bound by the 0-order entropy of the text to be encoded. The index allows to insert or delete texts of the collection, which avoids reconstruction of the whole index as it is necessary for static indices even for small modifications. We have shown that the insertion of small texts into our dynamic FM-index can be faster than the construction of the static FM-index (v.2) by Ferragina and Manzini [2]. Furthermore, we have shown that an immediate result of the dynamic FM-index is the space-efficient construction of a compressed suffix array which consists of the Burrows-Wheeler Transformation, sampled suffix array, and its inverse.

## Burrows-Wheeler Transformation

The *Burrows-Wheeler Transformation* (BWT) [1] permutes a text in such a way that the permutation is reversible. To construct the BWT,  $bwt(t)$ , for some text  $t$  of length  $n$ , think of a conceptual matrix  $M(t)$  (Figure 1), where the rows consist of all  $n$  cyclic permutations of  $t$ . We sort the rows in lexicographical order. Then the symbols in the last column  $L$ , read top-down, denote the BWT of  $t$ .

F	L	
12:	\$	mississippi
11:	i	mississippi
8:	ippi	mississ
5:	issippi	miss
2:	ississippi	m
1:	mississippi	\$
...	...	...

⇒ "ipssm\$piissii"

Figure 1: Conceptual BWT matrix  $M(t)$ , where  $t = \text{"mississippi\$"}$

The  $\$$  denotes a special unique character that is lexicographically smaller than all other characters. An application of the BWT is its use in text compression. The BWT of a text usually contains a high number of repetitions and the compression can be improved by using techniques like the move-to-front encoding and the run-length encoding prior to the actual compression.

## Reverse Transformation

Rank is an operation which is defined as  $\text{rank}_c(t, i) := |\{j \in [1..i] \mid t[j] = c\}|$ .  $C[1, \sigma]$  is an array, where  $C[c_s]$  contains the number of occurrences of characters  $\{s, c_1, \dots, c_{s-1}\}$  in  $t$ . Furthermore, we define  $\text{map}_c(L, i) := C[c] + \text{rank}_c(L, i)$  and the so called *LF-mapping*,  $\text{LF}(L, i) := \text{map}_{L[i]}(L, i)$  [2].  $F$  denotes the first column and  $L$  denotes last column in the conceptual matrix. Starting with  $p$  and applying LF-mapping iteratively reconstructs  $t$  backwards.

## Backwards Search

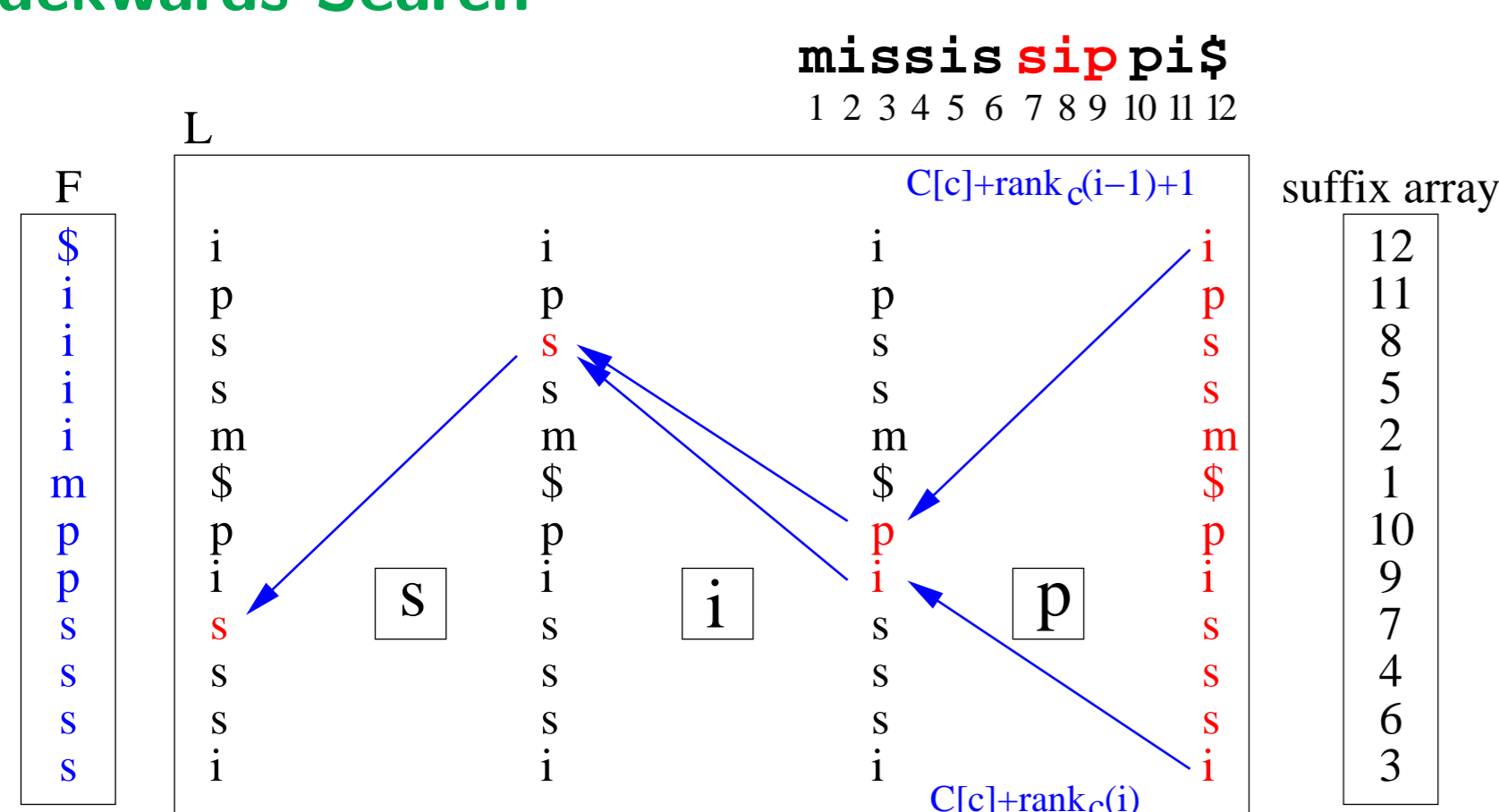


Figure 2: Example of the backwards search

A rather new application of the BWT is its usage as a self-index, which in contrast to the traditional sequence-based indices like the suffix tree and suffix array does not need to store the text in addition to the index, as it is already contained in the BWT implicitly. Similar to the reconstruction

of the text, it is possible to find the interval in the BWT that corresponds to the locations of some pattern in the text. While this approach only allows to count the number of occurrences, using a sampled suffix array in addition to this, allows to locate the occurrences.

## Direct Construction of BWT

Given a dynamic sequence that allows to insert new characters at arbitrary positions and that provides the operation rank, it is possible to construct the BWT while reading the text backwards. We call this *direct construction* as it is similar to the concept of online construction where the data is read in forward direction. Reading the next character and inserting it into the BWT corresponds to the insertion of a suffix into the suffix array. Since  $\$$  has not to be stored explicitly, each step in the direct construction consists of only one modification (see Figure 3) of the BWT which can be determined by computing  $\text{LF}(L, i) + 1$ , where  $i$  denotes the position of the previously inserted character and  $+1$  accounts for the special character. Repeating the procedure for the whole string  $t$  and finally inserting  $\$$  thus constructs  $bwt(t)$ .

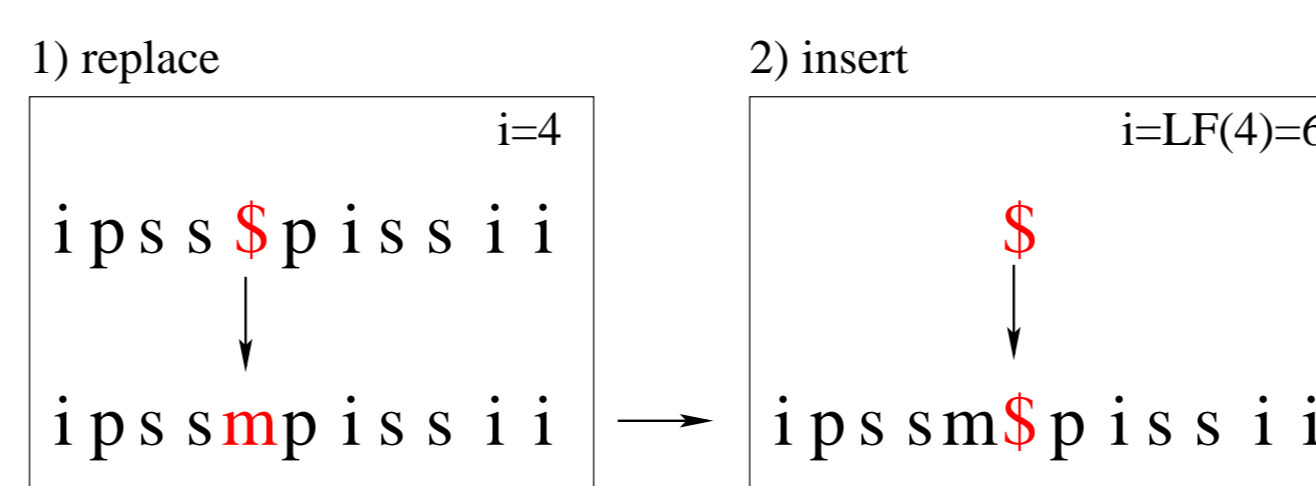


Figure 3: Transformation of  $bwt(\text{"ississippi\$"})$  into  $bwt(\text{"mississippi\$"})$

## Wavelet Tree

The *balanced wavelet tree* [3] is a balanced binary tree of height  $h = \lceil \log(\sigma) \rceil$ . Each leaf represents a single distinct alphabet symbol and each symbol is represented by one leaf. We associate a bit vector  $B_x$  with each node  $x$  except the leaves. If the corresponding leaf for symbol  $t[i]$  belongs to the left subtree of  $x$ ,  $B_x[i]$  is set to 0, 1 otherwise. To look-up the symbol at position  $i$  in the sequence, we start by performing a look-up of the bit at position  $i$  in the bit vector associated with the root. We get some bit  $b$  and compute  $i \leftarrow \text{rank}(b, i)$ . If  $b = 0$ , we go to the left child, otherwise to the right child. Given the new value for  $i$ , we perform look-up again like we did for the root and repeat the whole procedure until a leaf is reached. To compute  $\text{rank}_c(t, i)$ , we do the same as in look-up. Then,  $\text{rank}(b, i)$  applied to the last visited bit vector gives  $\text{rank}_c(t, i)$ .

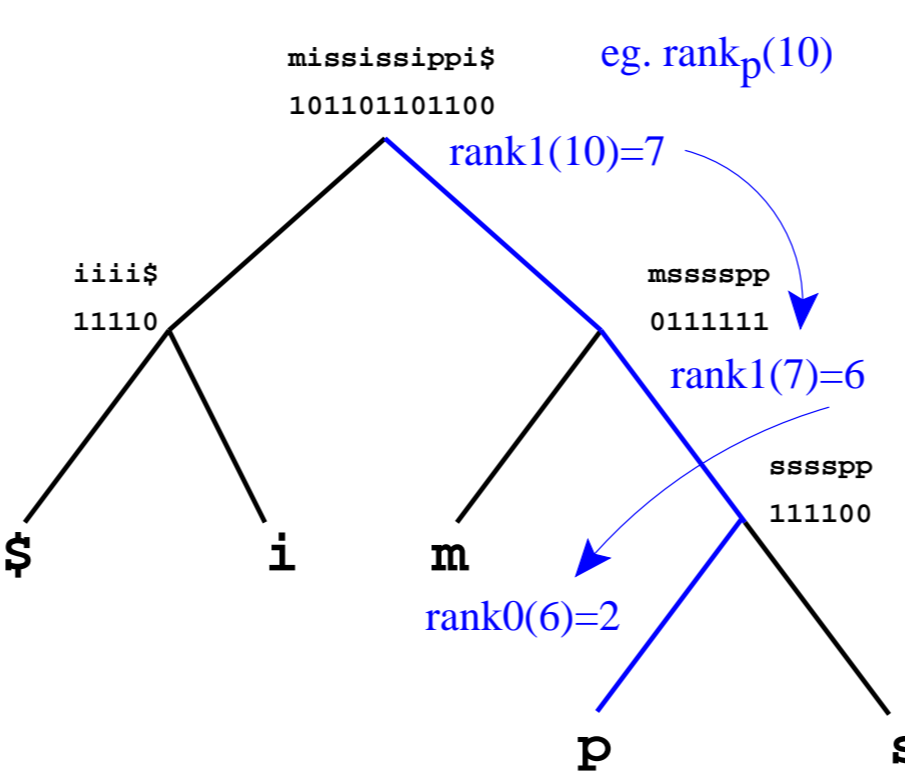


Figure 4: Computation of  $\text{rank}_p(10)$  in the Wavelet Tree for "mississippi"

Using a Huffman-shaped tree, its shape reflecting the character distribution, the average height of paths in the tree is  $h = \lceil H_0 \rceil$  and the space usage is bound by the 0-order entropy of the text.

## Dynamic Bit Vector

Our implementation of a bit vector is a variant of the dynamic bit vector that is described in [5] with minor differences. The structure is based on a red-black tree. For a bit sequence of length  $n$  it is guaranteed that the height of the tree remains  $\mathcal{O}(\log n)$ . Each internal node of the tree contains  $\mathcal{O}(\log n)$  bits of the sequence and has stored the number of bits that are stored within this subtree and the

number of bits (partial rank answers) that are set. Rank is computed by a top-down walk, where the partial rank answers are summed up to the complete rank answer. Inserts and look-ups take  $\mathcal{O}(\log n)$  time and the space usage is  $\mathcal{O}(n)$  bits.

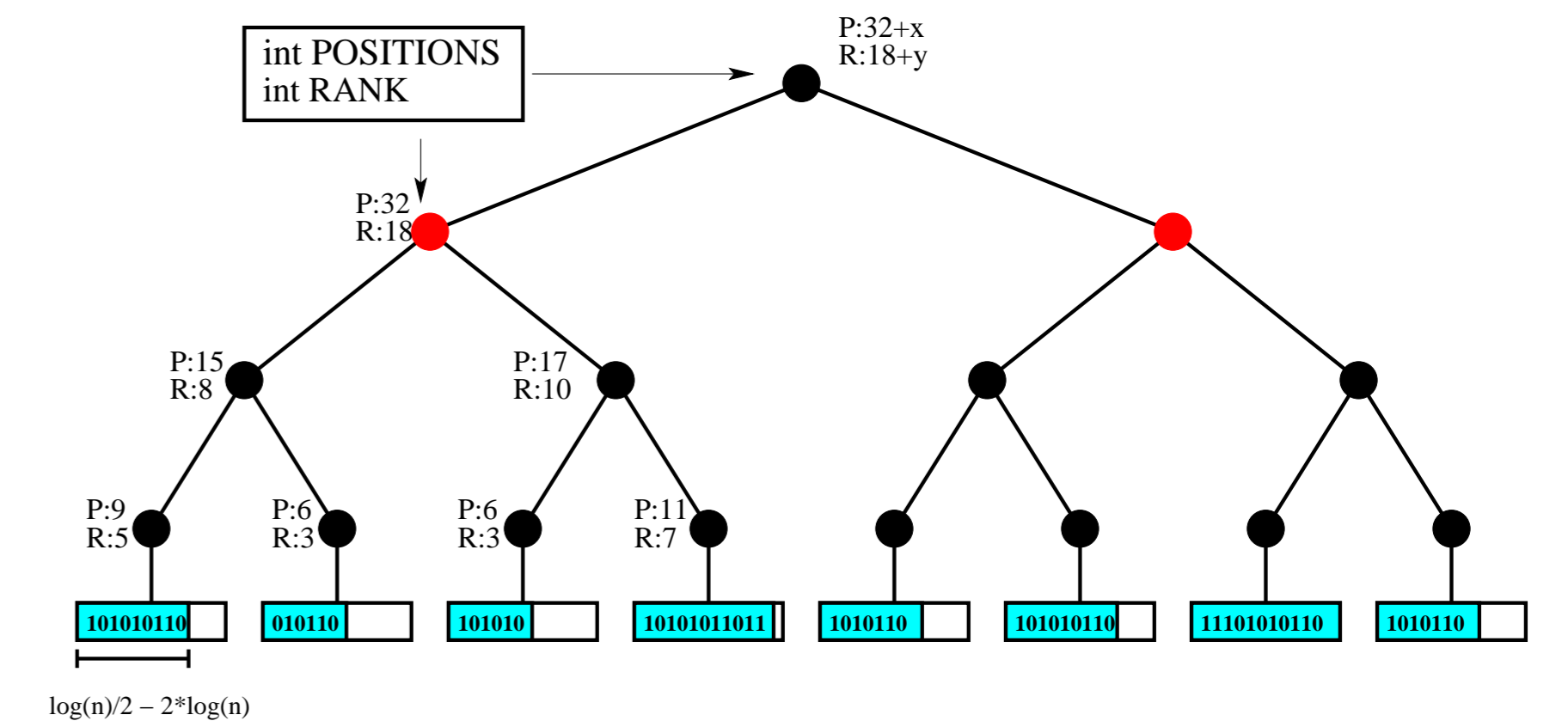


Figure 5: Example of a Dynamic Bit Vector

## Experiments

Figure 6 shows the space usage of our implementation (DFMI),  $bwt$  and  $dnabwt$  [4] and the theoretical space usage of the suffix array that is needed to construct the BWT of different types of texts.

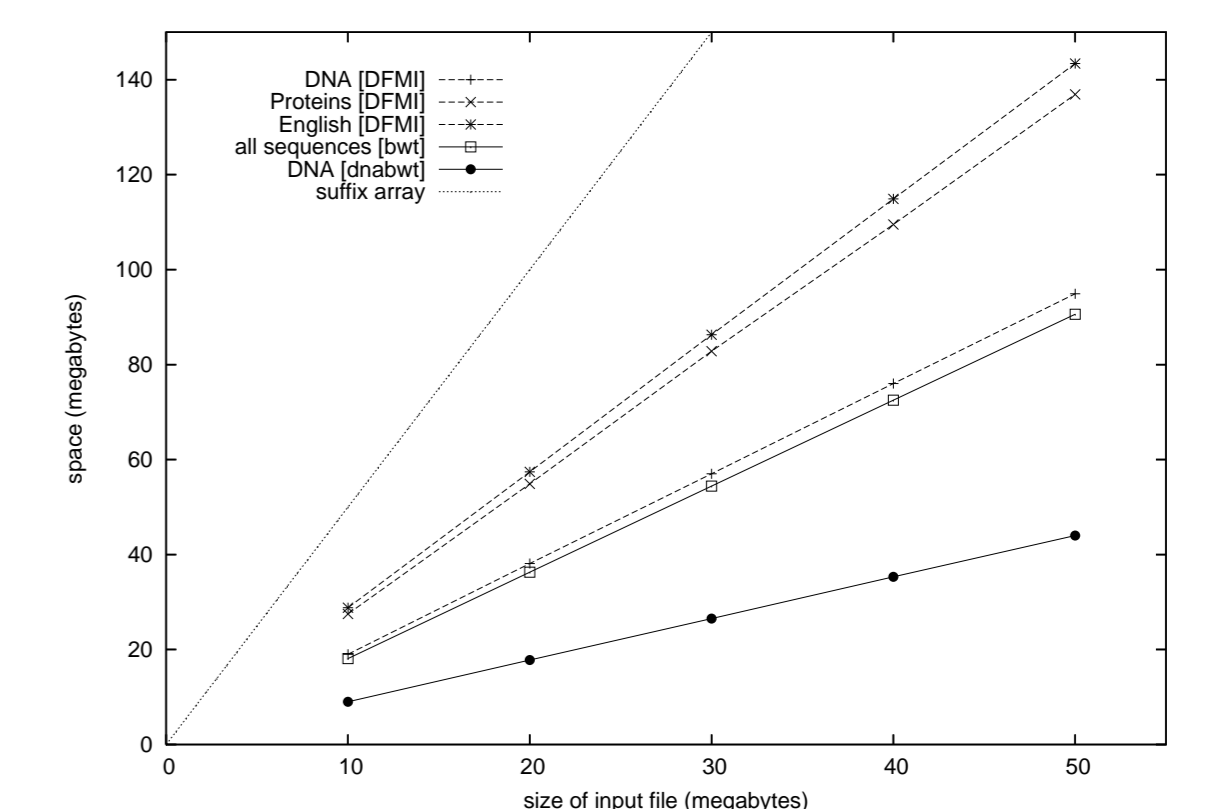


Figure 6: Space comparison of DFMI,  $bwt$ , and  $dnabwt$

Since our dynamic index does not need to be reconstructed when new texts are added, it can be faster than a static index. We have compared our index with the static FM-index of [2]. For the experiments, we built the indexes for a 100 MB file containing DNA and then added 1 MB upto 8 MB DNA sequences. For DNA sequences (Figure 7) smaller than 7 MB we were able to show that our index can be faster than the static index, which needs to reconstruct the whole index. For protein and English text sequences (not shown) we have been faster for sequences smaller than 4 MB.

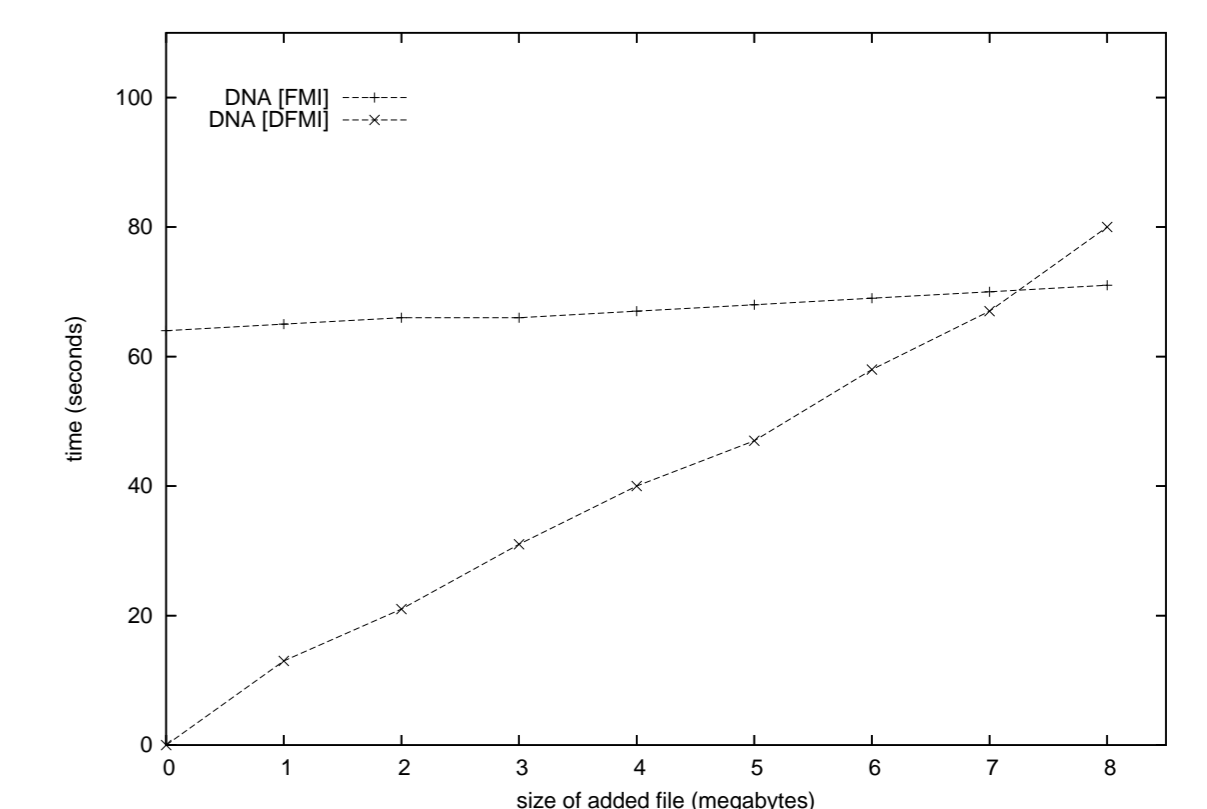


Figure 7: Dynamic versus static FM-Index using DNA

## Acknowledgments

I thank my supervisors Veli Mäkinen, Klaus-Bernd Schürmann, and Jens Stoye for making this diploma thesis possible and for their advice during my work on this thesis.

## References

- [1] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital System Research Center, May 1994.
- [2] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *FOCS '00: Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, page 390, Washington, DC, USA, 2000. IEEE Computer Society.
- [3] R. Grossi and J. S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM J. Comput.*, 35(2):378–407, 2005.
- [4] J. Kärkkäinen. Fast  $bwt$  in small space by blockwise suffix sorting. *Theoretical Computer Science*, 2006. to appear.
- [5] V. Mäkinen and G. Navarro. Dynamic entropy-compressed sequences and full-text indexes. In *Proceedings of the 17th Annual Symposium on Combinatorial Pattern Matching (CPM 2006)*, LNCS 4009, pages 307–318, 2006.