Alphabet-Friendly FM-Index

Author: Rodrigo González

Santiago, November 8th, 2005

Outline

- Motivations
- Basics
- Burrows-Wheeler Transform
- FM-Index
- Compression boosting
- Wavelet tree
- Alphabet-Friendly FM-Index
- Experimental results
- Conclusions

Motivations

- In pattern matching, the main problem consists in determining the number of occurrences of a pattern P over a text T, where text and pattern are sequences of characters from an alphabet Σ of size σ.
- Many different indexing data structures have been proposed for this aim, such as inverted lists, suffix trees, suffix arrays and succinct indexes.
- Over time, the size of textual information has been growing, hence the importance of having succinct indexing data structures for pattern matching. This has motivated a search for less space-demanding indexes, in particular, using the same or less space than the original text.
- One of the most recent succinct indexes is the Alphabet Friendly FM-Index, from now on AF-FMI.
- http://pizzachili.dcc.uchile.cl/

Basics

- The **suffix array** of a text T is an array A that contains all the starting positions of the suffixes of the text T, such that T[A[1]…n]<T[A[2]…n]<…<T[A[n]…n], that is, array A gives the lexicographic order of all suffixes of the text T.
- The **empirical entropy H** resembles the entropy defined in the probabilistic setting. However, the empirical entropy is defined for any string and can be used to measure the performance of compression algorithms without any assumption on the input.
- So, when we create a succinct index structure it is natural to ask how close the index structure is to H_k and which kind of search the index structure supports.

Burrows-Wheeler Transform (1)

- Burrows and Wheeler introduced a new compression algorithm based on a reversible transformation now called the **Burrows-Wheeler Transform** (**BWT** from now on).
- The BWT consists of three basic steps:
  - append at the end of T a special character # smaller than any other text character
  - form a conceptual matrix M whose rows are the cyclic shifts of the string T# sorted in lexicographic order
  - construct the transformed text BWT by taking the last column of matrix M .

Burrows-Wheeler Transform (2)

Burrows-Wheeler Transform (3)

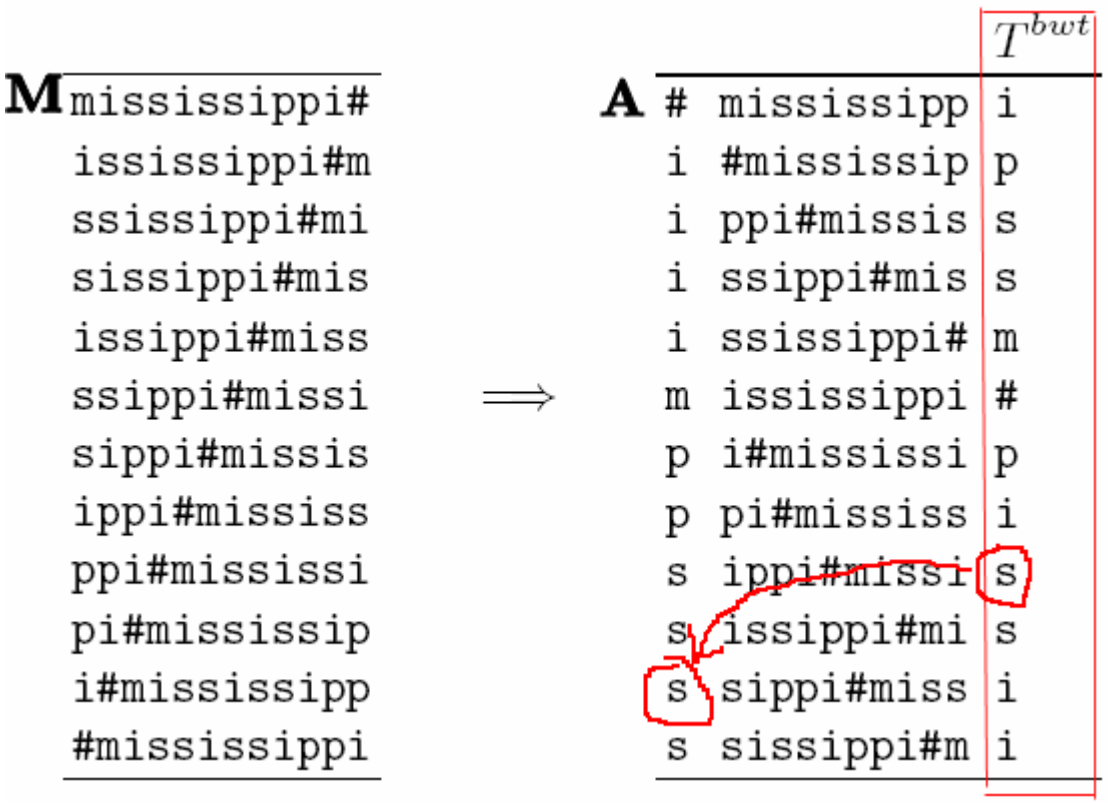
- The inverse of the BWT consists in calculating:

  LF(i)=C[BWT[i]]+Occ(BWT[i],i)
- For $c \in \Sigma$, let C[c] be the total number of text characters which are alphabetically smaller than c.
- For $c \in \Sigma$, let Occ(c,q) be the number of occurrences of character c in the prefix BWT[1,q].
- LF(·) comes from Last-to-First column mapping because character BWT[i] is located in the first column of M at position LF(i). In this way, we can navigate the text T backwards. That is, if T[k]= BWT[i], then T[k-1]= BWT [LF(i)]

Burrows-Wheeler Transform (4)

- For example if i=9 then

  LF(9)=C[BWT[9]]+Occ(BWT[9],9) = C[s]+Occ(s,9) = 8+3 = 11



FM-Index (1)

- The **FM-index** is a full-text self-index that allows to efficiently search for the occurrences of an arbitrary pattern P as a substring of the text T.
- The FM-index is composed of a compressed representation of BWT with auxiliary structures that permit us to compute in O(1) time the value of Occ(c,q) with $c \in \Sigma$ and for any $q \in [1...n]$.

FM-Index (2)

- The pseudo code of the counting operation that works in p phases is:

Algorithm get_rows($P[1,p]$)
$i \leftarrow p$;First$\leftarrow$ 1;Last$\leftarrow$ n;
while ((First$\leq$Last) **and** (i$\geq$1)) **do**
    $c \leftarrow P[i]$;
    First$\leftarrow$ $C[c] + Occ(c,\text{First}-1) + 1$;
    Last$\leftarrow$ $C[c] + Occ(c,\text{Last})$;
    $i \leftarrow i - 1$;
if (Last $<$ First) **then return** "no rows prefixed by $P[1,p]$"
else return (First, Last);

- To obtain the position of each row where P[1,p] is prefix, it is necessary to mark one text position every $O(\log^2 n / \log \log n)$ and then to use LF(.) until finding a marked position.

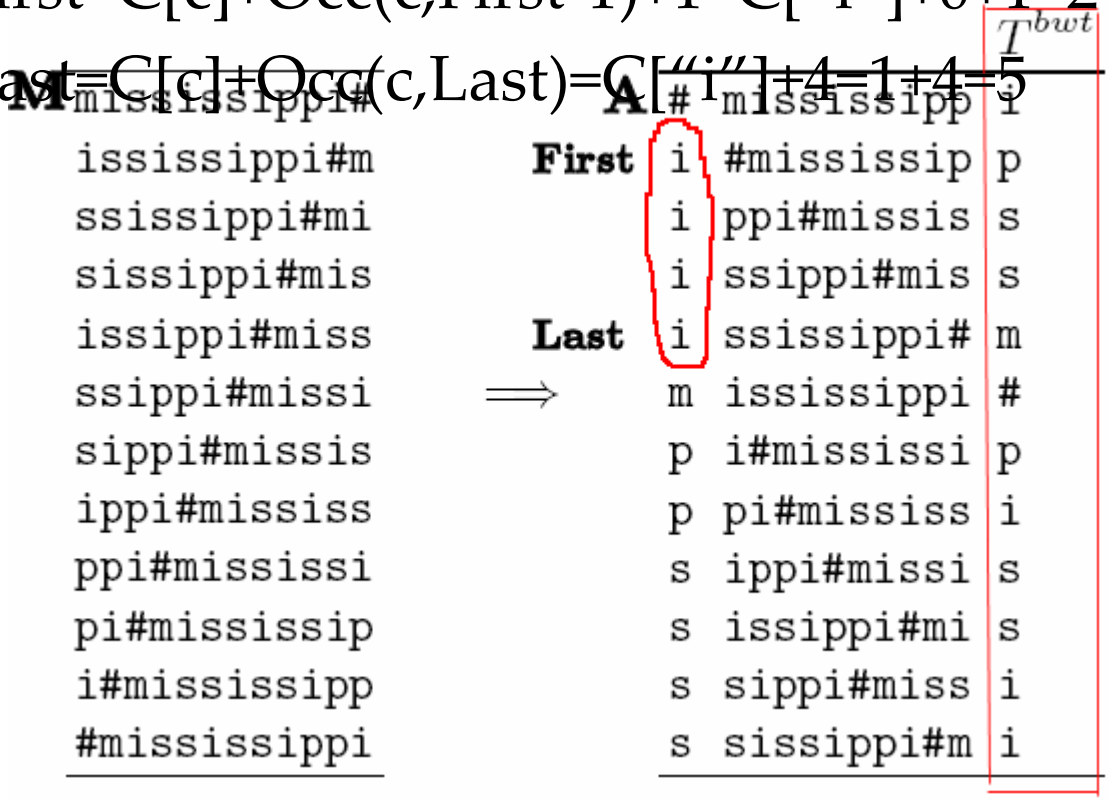
FM-Index (3)

- For example if P="si"

  c=P[i]=P[2]="i"

  First=C[c]+Occ(c,First-1)+1=C["i"]+0+1=2

  Last=C[c]+Occ(c,Last)=C["i"]+4=1+4=5

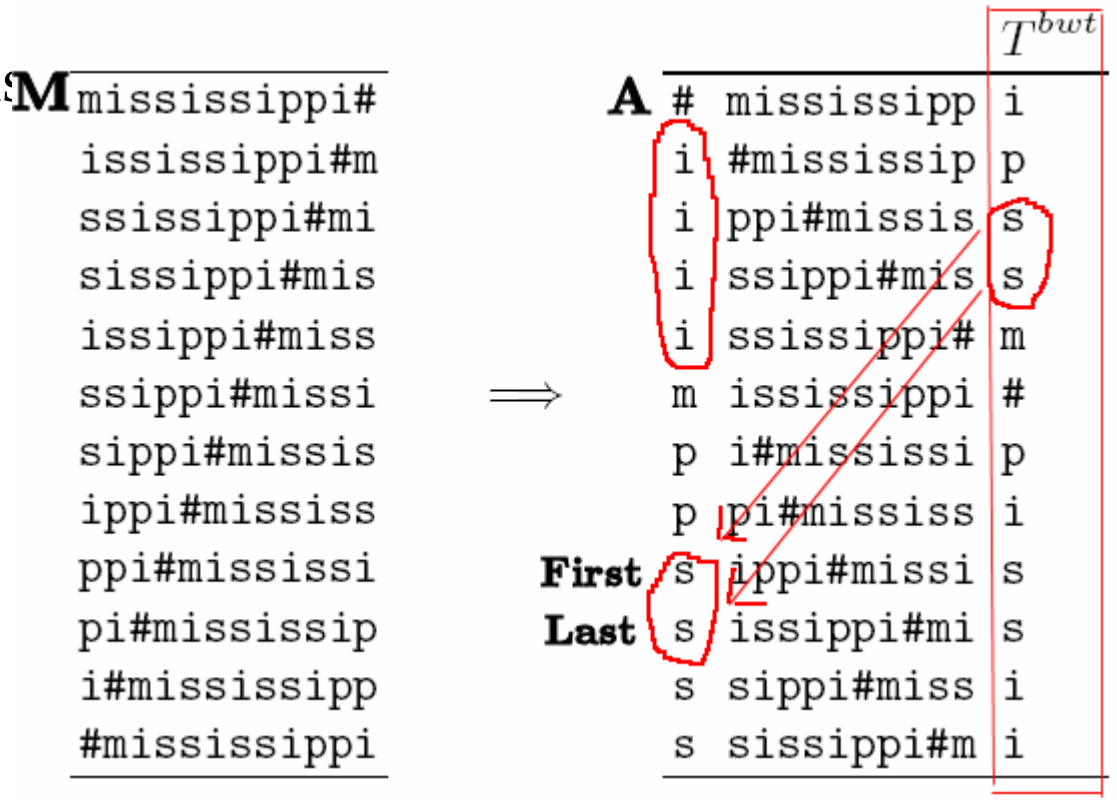


FM-Index (4)

c=P[1]="s"

First=C[c]+Occ(c,First-1)+1=C("s")+Occ("s",1-1)+1=8+0+1=9

Last=C[c]+Occ(c,Last)=C("s")+Occ("s",12)=8+2=10



Compression boosting (1)

- The key idea is that one can take an algorithm whose performance can be bounded in terms of the zero-order entropy and obtain, via the booster, a new compressor whose performance can be bounded in terms of the k-th order entropy, simultaneously for all k.
- The boosting technique computes an optimal partition $S=\{s_1,s_2\ldots s_z\}$ of BWT, such that, if we compress each s_i with a zero-th order entropy, we have a k^{th} order entropy bound for the compressed BWT.

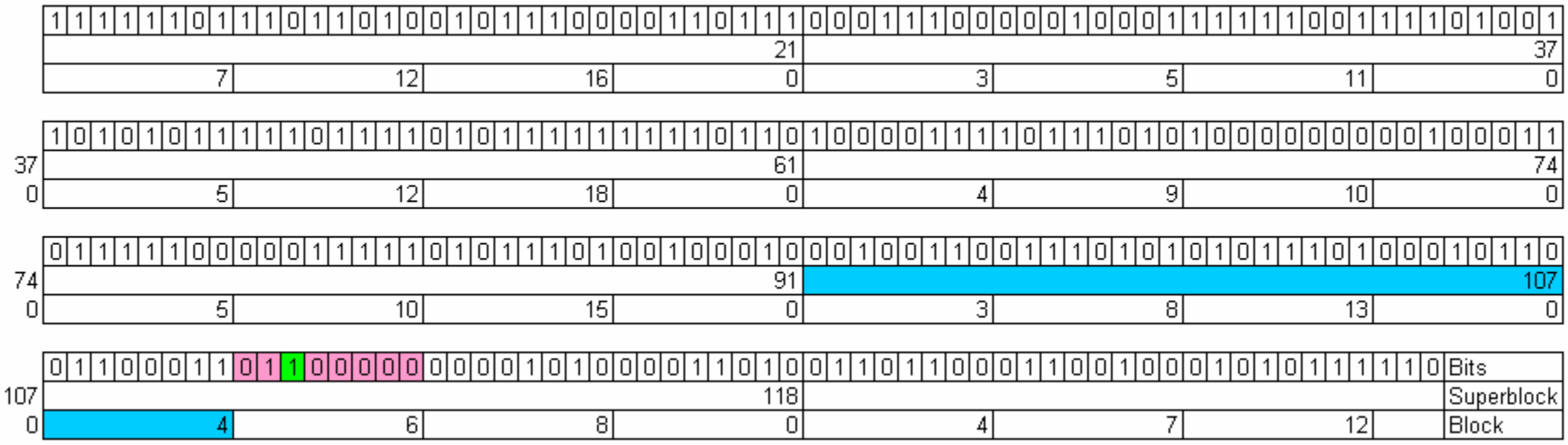
Rank

- First we show a binary rank:
- $Rank_1(Bits,i)$=numbers of ones in Bits[1,i]
- $Rank_1(Bits,203)$=

  =Superblock(203/32)+Block(203/8)+count(01100000,203%8)

  =Superblock(6)+Block(25)+popcount(01100000)

  =107+4+2=113

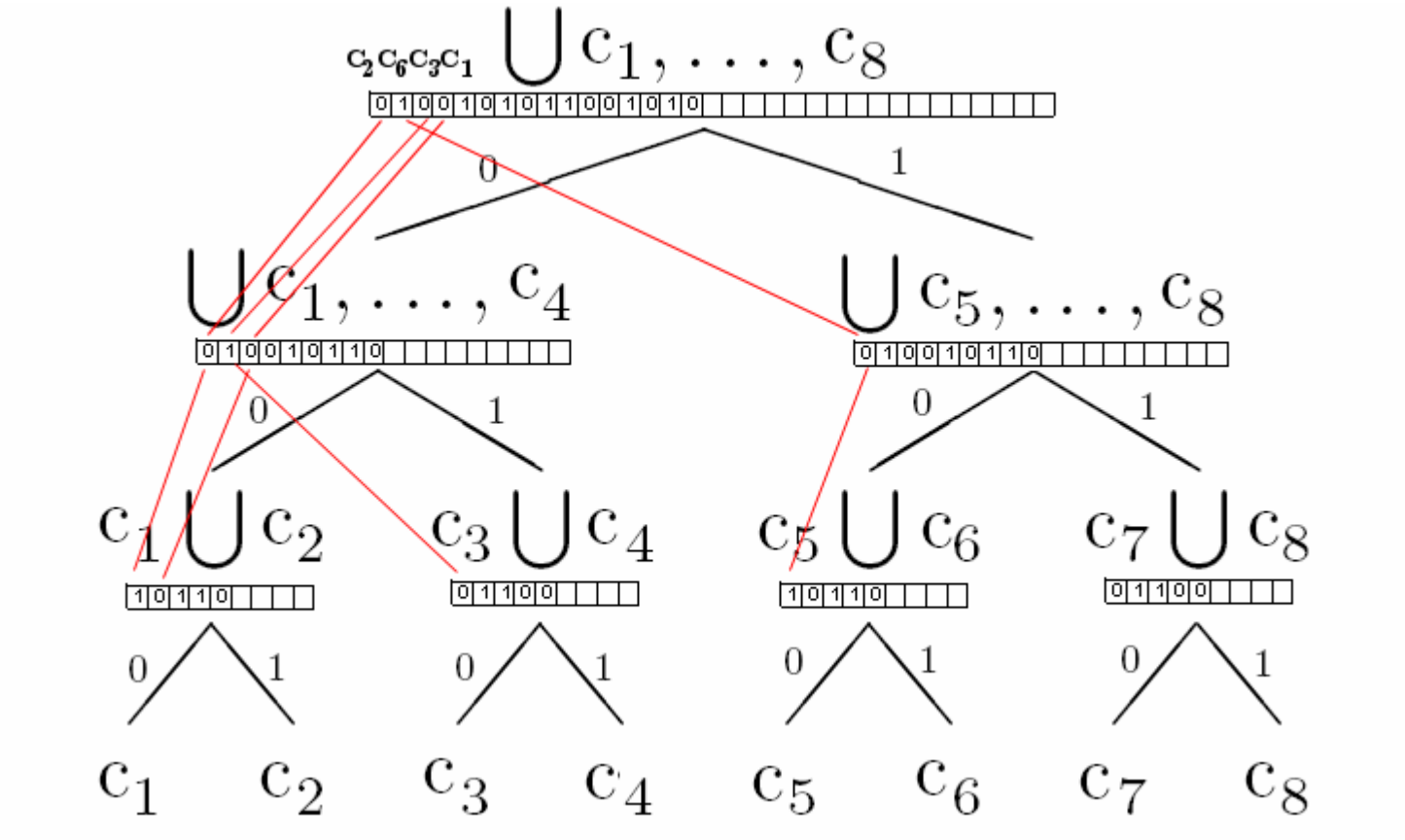


Wavelet tree (1)

- Theorem. Let W[1..w] denote a string over an arbitrary alphabet $\sum$. The wavelet tree built on W uses $wH_0(W)$ + $O(\log|\sum|\ (w \log\log w)/\log w)$ bits of storage and supports in $O(\log|\sum|)$ time the following operations:
  - given q, $1 \le q \le w$, the retrieval of the character W[q];
  - given $c \in \sum$ and q, $1 \le q \le w$, the computation of the number of occurrences $Occ_W(c, q)$ of c in W[1..q].

Wavelet tree (2)

- C_1=0000,C_3=0010



Alphabet-Friendly FM-Index (1)

- To build the FM-index we need to solve two problems:
  a) to compress BWT up to $H_k(T)$
  b) to compute Occ(c, q) and BWT[q] in time independent of n.
- We use the boosting technique to transform problem a) into the problem of compressing the strings $s_1, s_2 \ldots s_z$ up to their zero-th order entropy, and we use the wavelet tree to create a compressed (up to H_0) and indexable representation of each s_i thus solving simultaneously problems a) and b).

Alphabet-Friendly FM-Index (2)

- Construction
  - First we determine the optimal partition $s_1,s_2\ldots s_z$ using the booster.
  - Build a binary string B that keeps track of the starting positions in BWT of the s_i's.
  - For each string s_i, i = 1…z build:
    - the array $C_i[1, |\sum|]$ such that $C_i[c]$ stores the occurrences of character c within $s_1,s_2\ldots s_{i-1}$
    - the wavelet tree T_i.

Alphabet-Friendly FM-Index (2)

- To compute BWT[q], we first determine the substring s_y containing the q-th character of BWT by computing $y=Rank_1(B,q)$. Then we exploit the wavelet tree T_y to determine BWT[q].
- To compute Occ(c, q), we initially determine the substring s_y where the row q occurs, $y=Rank_1(B,q)$. Then we exploit the wavelet tree T_y and the array $C_y[c]$ to compute $Occ(c,q)=Occ\_s_y(c,q')+C_y[c]$, where $q' = q - sum(j=1..y-1;|s_j|)$

Alphabet-Friendly FM-Index (3)

- The size of the new index built on a string T[1,n] is bounded by $nH_k(T)+O(\log_{|\sum|}(n \log\log n)/ \log n)$ bits, where $H_k(T)$ is the k-th order empirical entropy of T.
- The above bound holds simultaneously for all $k \leq \alpha \log_{|\sum|} n$ and any constant $0<\alpha< 1$. Moreover, the index design does not depend on the parameter k, which plays a role only in analysis of the space occupancy.
- Using the AF-index, the counting of the occurrences of an arbitrary pattern P[1,p] as a substring of T takes $O(p \log |\sum|)$ time. Locating each pattern occurrence takes $O(\log_{|\sum|}(\log^2 n/ \log\log n))$ time. Reporting a text substring of length l takes $O((l + \log^2 n/ \log\log n) \log |\sum|)$ time.

Alphabet-Friendly FM-Index (4)

- Using O(n) additional bits of storage, we can design a variant of the AF-FMI that counts the pattern occurrences in $O(p/\varepsilon)$ time (independent on $\sum$), and $O((\log^2 n/\log\log n)\ |\sum|^{\varepsilon}/\varepsilon)$ time to locate each occurrence.
- To achive this we show first an approach with flat bit arrays
  - Let W[1,w] be a substring of the BWT computed for the indexed text T. Instead of the wavelet tree, we process W by building $|\sum|$ bit arrays S_c such that $S_c[i] = 1$ if and only if W[i] = c. We then construct a FID data structure over each S_c to support in constant time $Rank_1()$ and $Select_1()$ queries.
- Now a more general tradeoff with r-ary trees
  - Let us consider an r-ary tree built over a string W[1,w]. Much like a wavelet tree, each node u of the r-ary tree is responsible for a subset of the alphabet $\sum$, say $\sum_u$, so that u represents the subsequence W_u of W formed by the characters in $\sum_u$. The root node stands for the whole $\sum$, and thus stores the whole string W. Each node u with children $u_1 \ldots u_r$ splits its alphabet $\sum_u$ into r equally-sized subsets $\sum_{u1} \ldots \sum_{ur}$.

Alphabet-Friendly FM-Index (5)

- In each node u_j , child of u, we store a bit array S_{uj} of length $|W_u|$ such that $S_{uj}[q] = 1$ whenever $W_u[q] \in \sum_{uj}$. S_{uj} is indeed the characteristic vector of the characters of W_u copied into W_{uj}. A FID data structure is then built over S_{uj} to answer in constant time $Rank_1()$ and $Select_1()$ queries.

- Finally, if we also have $|\sum|=O(\text{polylog } n)$, the counting of the occurrences of an arbitrary pattern P[1,p] as a substring of T takes O(p) time. Locating each pattern occurrence takes $O(\log^{1+\varepsilon} n)$ time. Reporting a text substring of length l takes $O(l + \log^{1+\varepsilon} n)$ time.

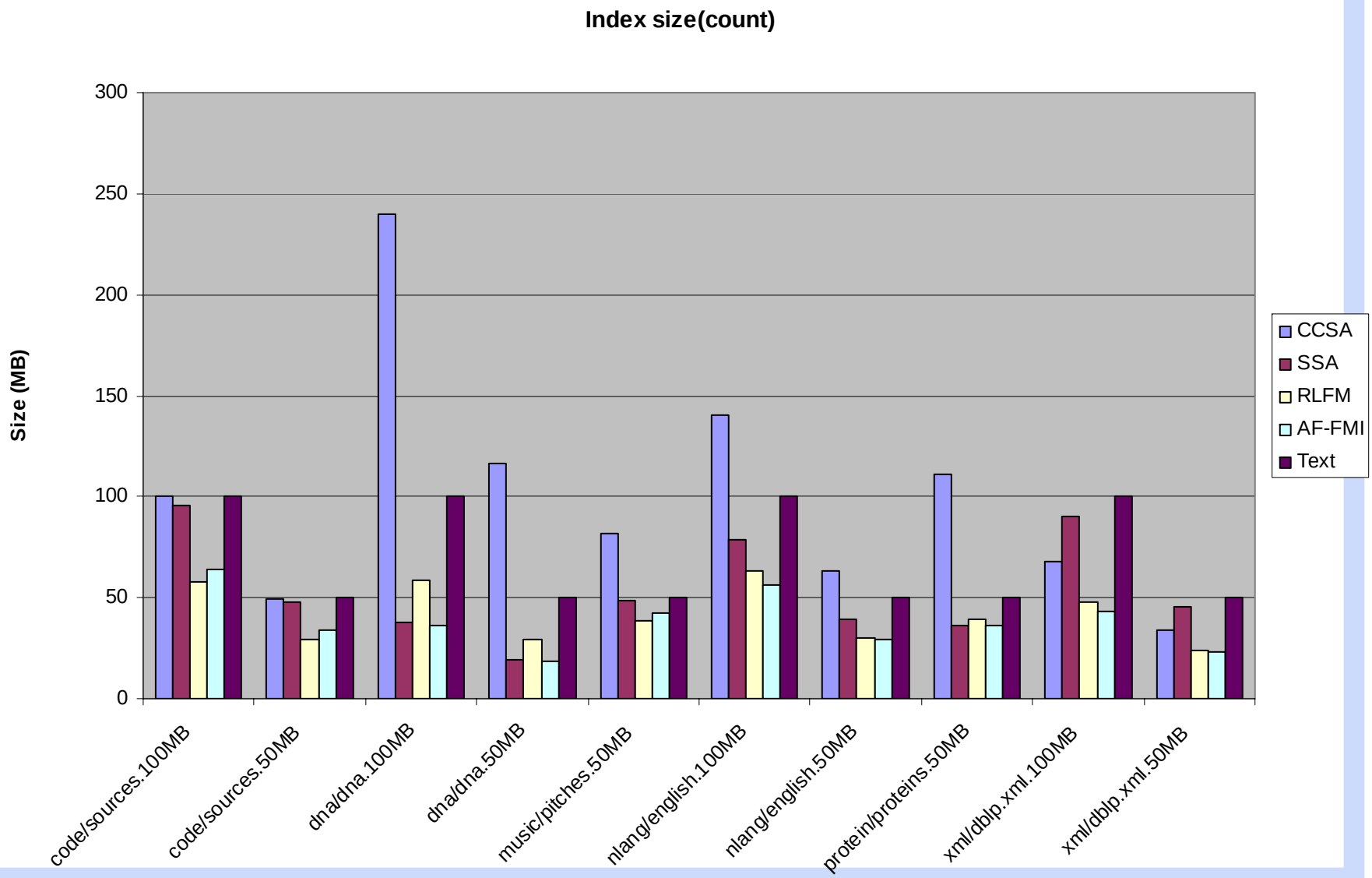
Experimental Results (1)

- We have the following site that supports the test bed to all our experimental results:

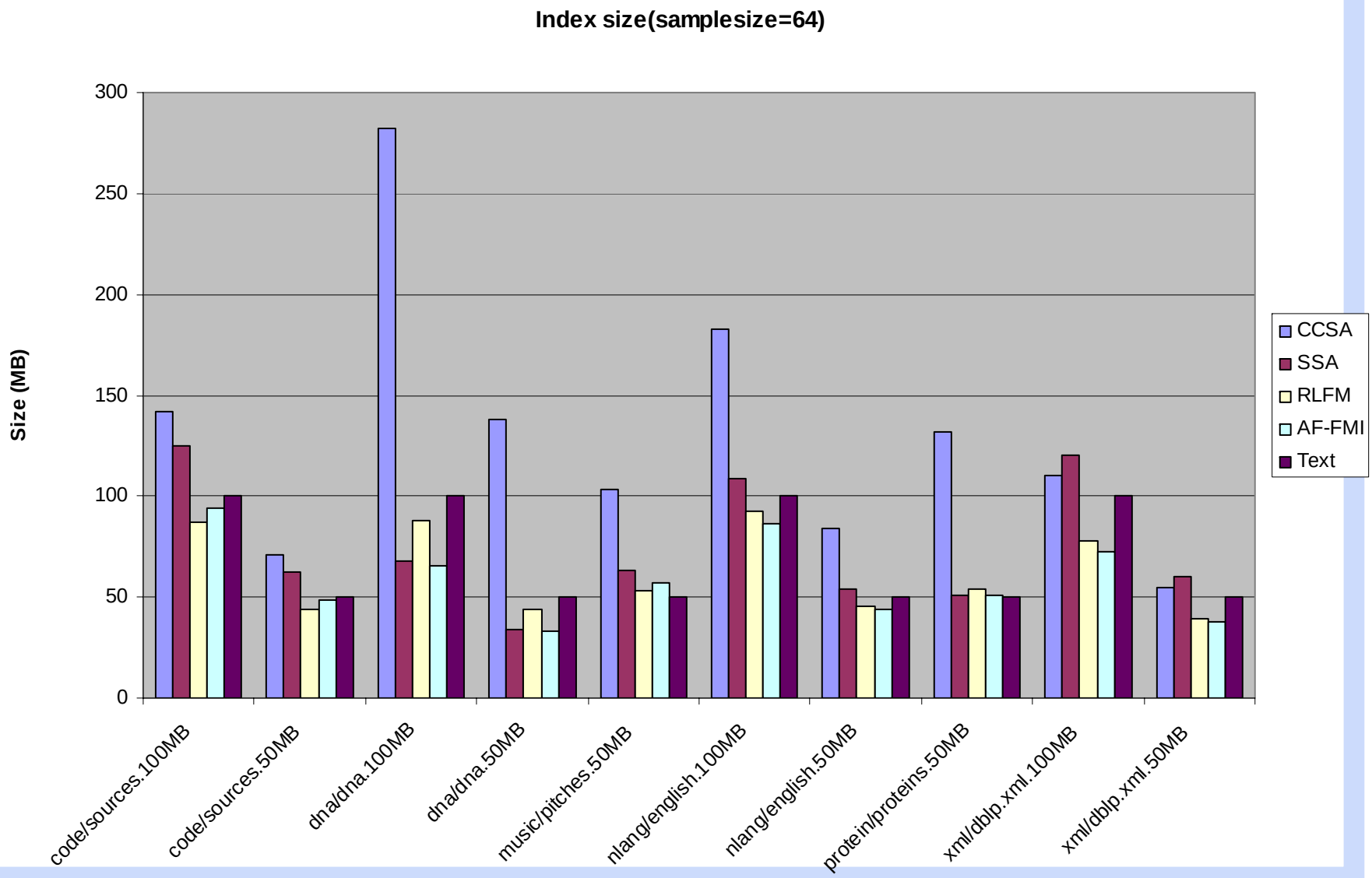
  http://pizzachili.dcc.uchile.cl/

- We compare several indexes over a collection of files.
  - code/sources.100MB
  - code/sources.50MB
  - dna/dna.100MB
  - dna/dna.50MB
  - music/pitches.50MB
  - nlang/english.100MB
  - nlang/english.50MB
  - protein/proteins.50MB
  - xml/dblp.xml.100MB
  - xml/dblp.xml.50MB

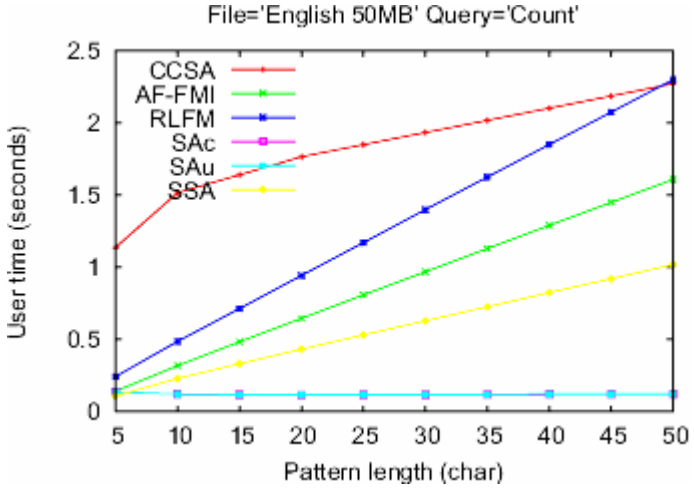
Experimental Results (2)

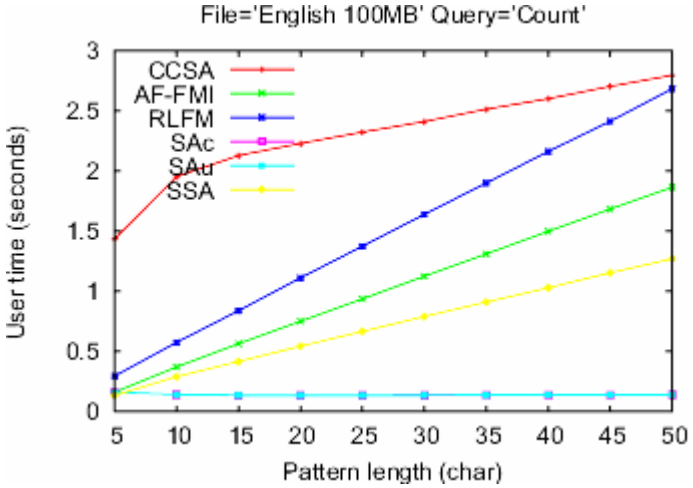


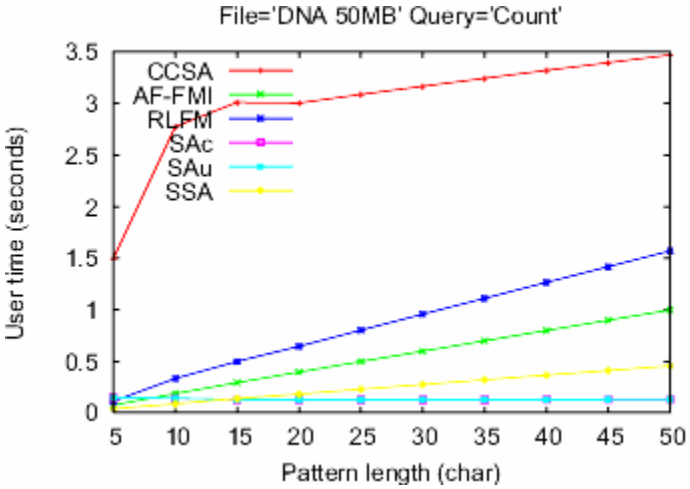
Experimental Results (3)

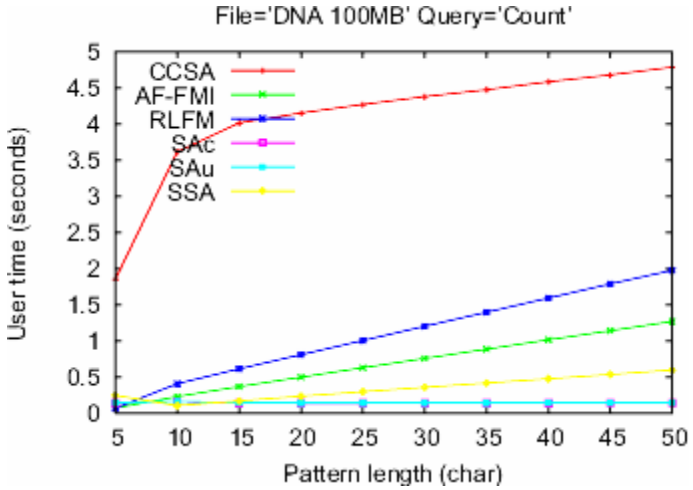


Experimental Results (4)

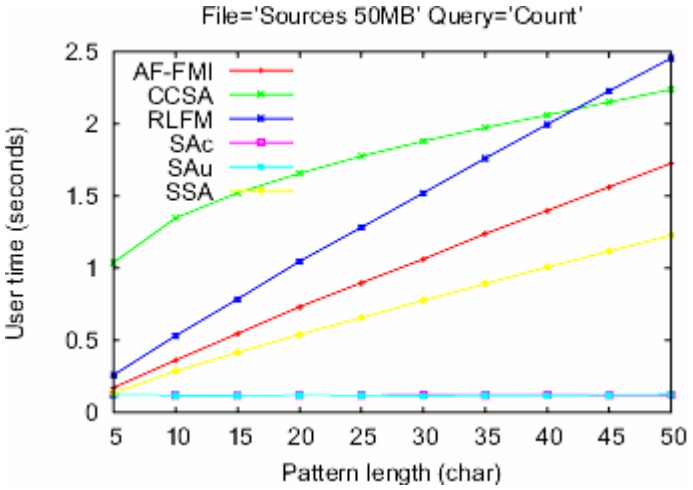


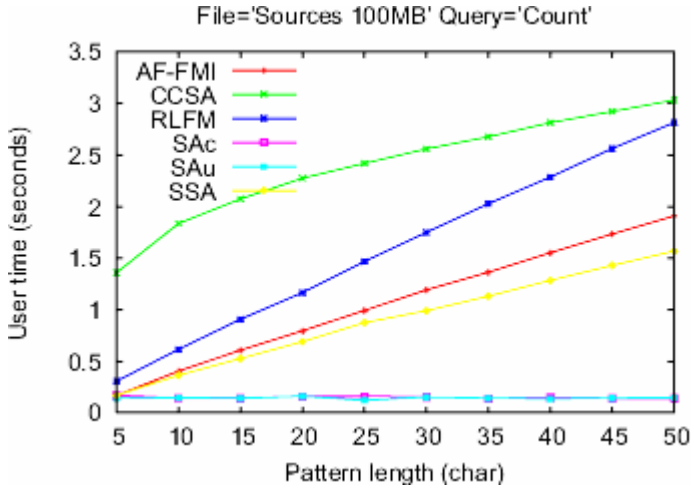


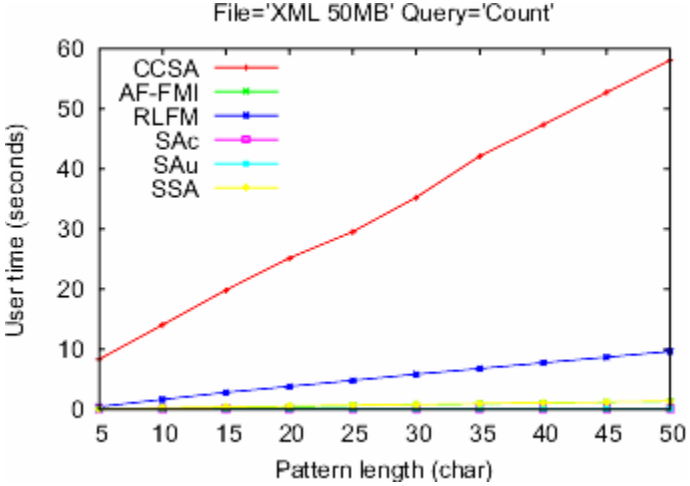


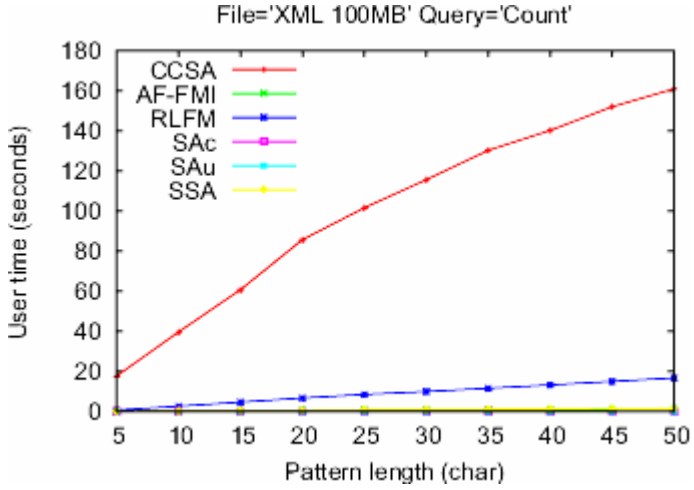


Experimental Results (5)

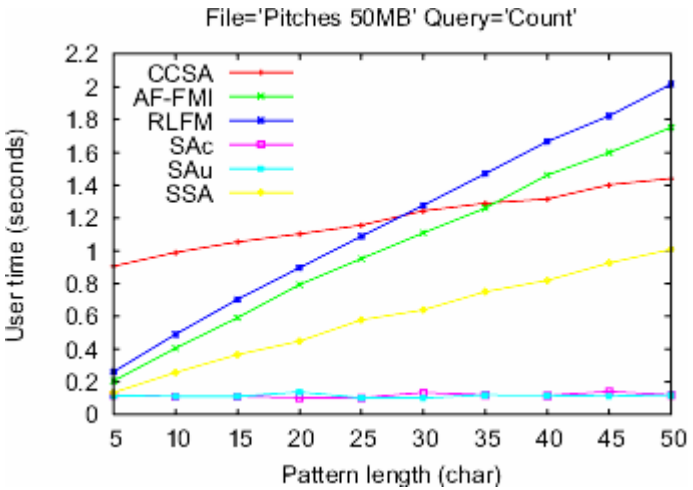


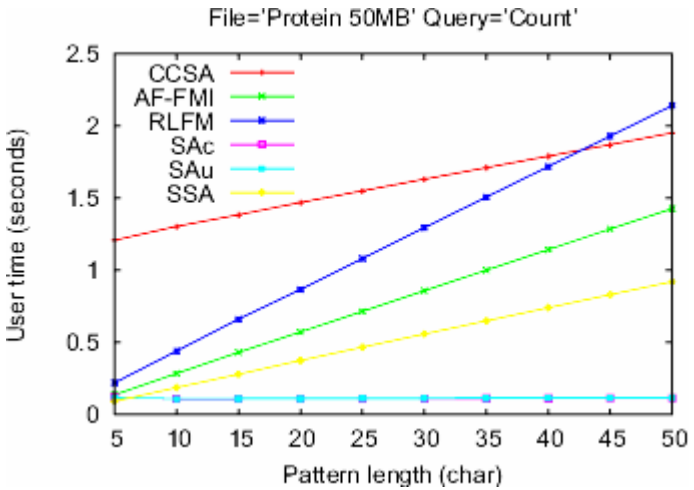




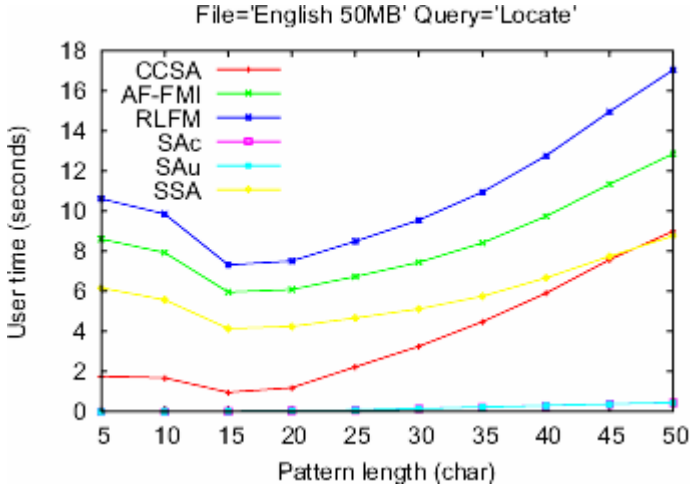


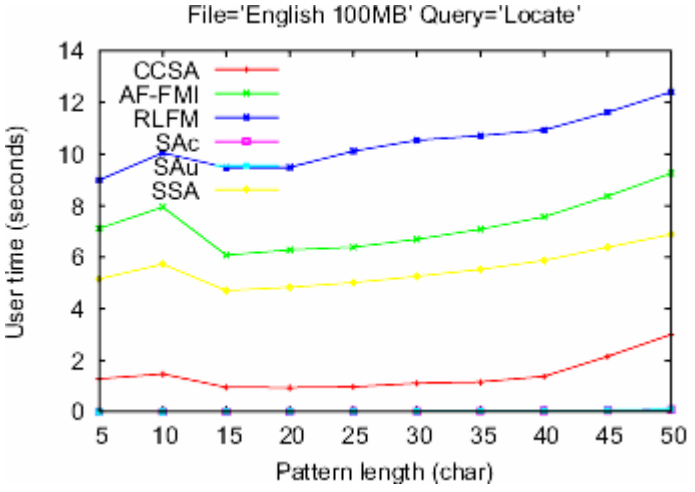
Experimental Results (6)

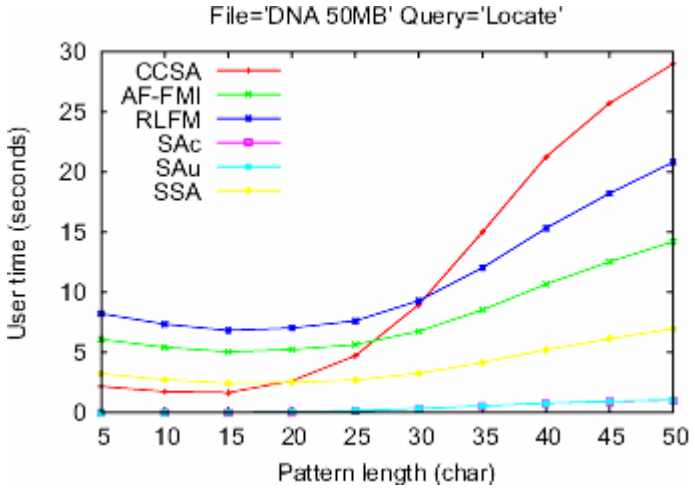


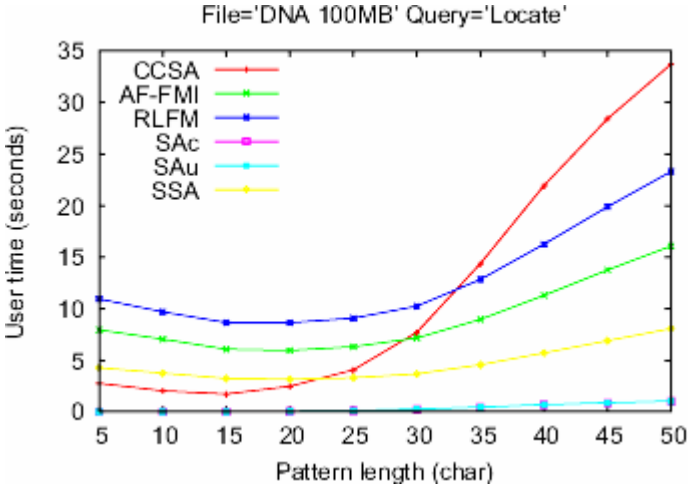


Experimental Results (7)

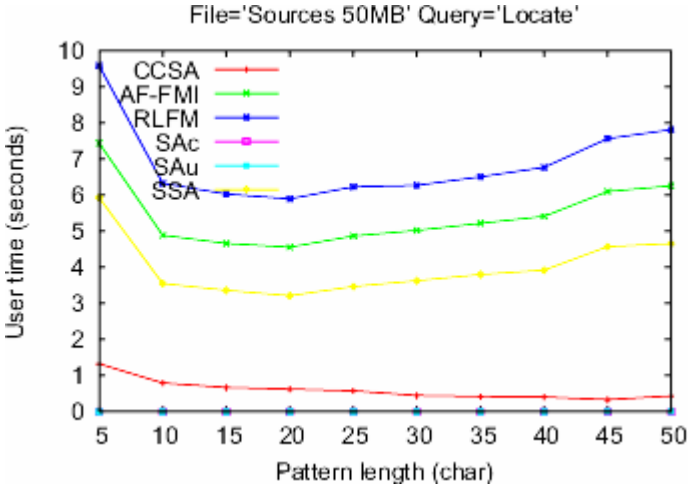


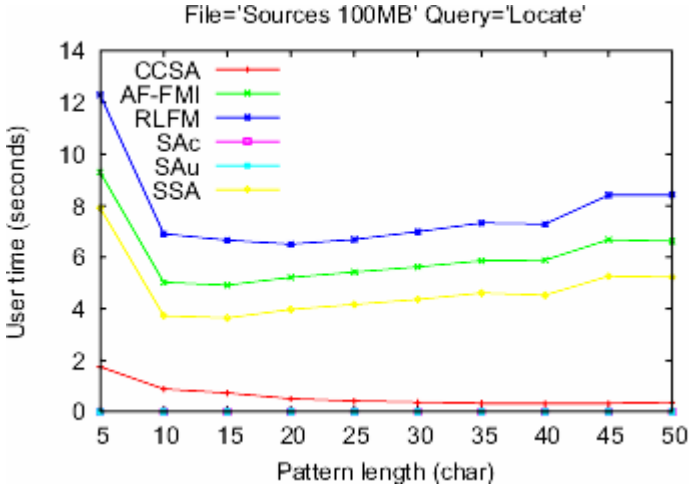


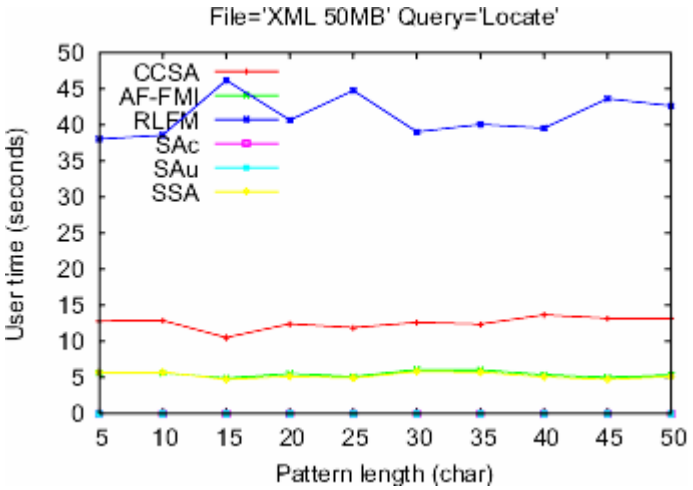


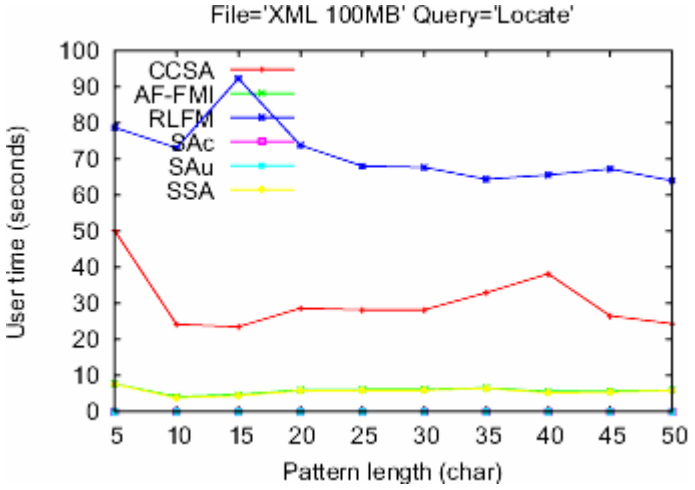


Experimental Results (8)

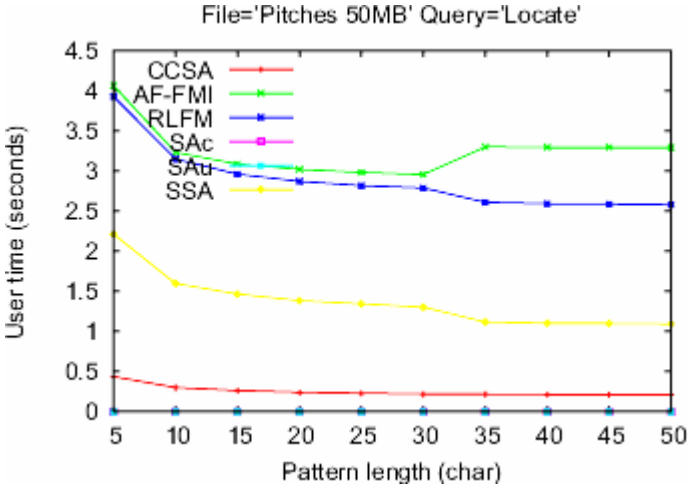


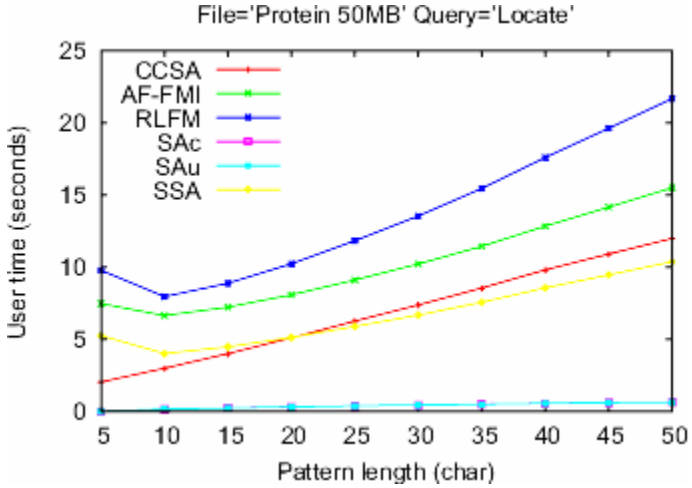






Experimental Results (9)





Conclusions

- We have shown that the combination of the FM-index, compression boosting techniques, wavelet trees, and fully indexable dictionaries, yields a simple data structure that uses asymptotically optimal space.
- The AF-Index is very competitive with the existing full-text self-indexes.

Questions?