The Miracle of Self-Indexing

Combining Text Compression and String Matching

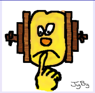
Gonzalo Navarro

`www.dcc.uchile.cl/gnavarro`

`gnavarro@dcc.uchile.cl`

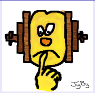
Department of Computer Science
University of Chile

Compact Data Structures



- They are data structures modified to use little space.
- Not just compression: they must retain their functionality and direct access.
- Motivation: Improve performance thanks to memory hierarchy.
- Especially if we can operate in RAM a structure that otherwise would need the disk.
- In this talk I will focus on compact data structures for text, where the success has been notorious.

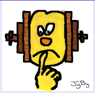
Compressed Full-Text Self-Indexes



- Consider a text $T[1,n]$ over an alphabet of size σ.
- It requires $n \log \sigma$ bits of space in plain form.

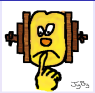
  (we will use base-2 logarithms by default).
- Searching for patterns $P[1,m]$ in T is a fundamental problem.
- It can be done in $O(n)$ time, but this can be unacceptable if T is large.
- We can use an index over T.
- If T is made of natural language, under some restrictions, a relatively small inverted index would do.
- But there are many sequences that cannot be handled this way: DNA, proteins, Oriental languages, music sequences, numeric signals, ...

Compressed Full-Text Self-Indexes



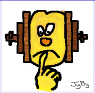
- An index that works for any string is the suffix array.
- But it requires $n \log n$ bits, way larger than $n \log \sigma$.
- For relatively large texts, the suffix array will not fit in main memory.
- And it will be rather slow to search on disk.
- This has been a dilemma for decades, with researchers trying different tricks to reduce the constant of the space.

Compressed Full-Text Self-Indexes



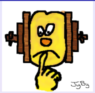
- Suddenly, a miracle happened in 2000.
- It was discovered that the suffix array was compressible...
- ... up to the space the compressed text would need.
- Within that space, not only it could offer indexed searching...
- ... but also extracting any text substring.
- Thus the text was unnecessary and could be discarded.
- Hence the compressed index replaced the text, and was called a self-index:
  - A compressor with indexing plus, or
  - An index able of reproducing the text.

Compressed Full-Text Self-Indexes



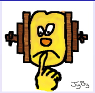
- There are many alternative self-indexes today.
- Despite they have opened a whole new area of research, and proved competitive in practice...
- ... they are still surrounded by a veil of mystery.
- They are deemed too complicated and impractical by many people, despite the open-source implementations being out there: http://pizzachili.dcc.uchile.cl.
- This may come from the first proposals, which were indeed complex.
- Not anymore the case.

Compressed Full-Text Self-Indexes



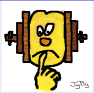
- In this talk, I’ll do my best to unveil the mystery.
- I will choose one of the most elegant self-indexes: the FM-index.
- I will start from the very basics, explaining only the minimum necessary to understand how a very competitive implementation works.
- I hope to expose the simplicity, elegance, and practicality of this data structure...
- ... and encourage you to go for more yourselves.

Bit Sequences



- Consider a bit sequence $B[1,n]$.
- We are interested in the following operation on B:
  - $rank_b(B,i)$: how many times does b occur in $B[1,i]$?
- Note $rank_0(B,i) = i - rank_1(B,i)$.
- By default let us assume $b = 1$.

Bit Sequences



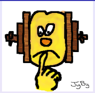
Result

- *rank* can be answered in constant time.
- This is easy by storing all the answers in $O(n \log n)$ bits, but one only needs

$$n + O\left(\frac{n \cdot \log\log n}{\log n}\right) \;=\; n + o(n)$$

bits of space (the n bits for $B[1, n]$ plus a sublinear extra).
- The solutions are practical.

Rank



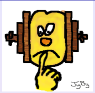
Rank in Constant Time

- We cut B into blocks of $b = (\log n)/2$ bits.
- We store and array $R[1, n/b]$ with the *rank* values at block beginnings.
- As we need $\log n$ bits to store a *rank* value, the total of bits used by R is

$$\frac{n}{b} \cdot \log n \;=\; \frac{n}{(\log n)/2} \cdot \log n \;=\; 2n$$

- Let us accept that space for now (it is more than promised).

Rank

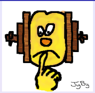


- ¿How to compute $rank(B,i)$?
  - Decompose $i = q \cdot b + r$, $0 \leq r < b$.
  - The number of 1's up to qb is $R[q]$.
  - We must count the 1's in $B[qb+1, qb+r]$.
- Assume we have a table $T[0, 2^b - 1][0, b-1]$, such that

$$T[x,r] \;=\; \text{total of 1's in } x[1,r]$$

where we see x as a stream of b bits.

Rank



- Since each cell of T can take values in $[0, b-1]$, T needs

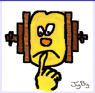
$$\begin{aligned} 2^b \cdot b \cdot \log b &= 2^{\frac{\log n}{2}} \cdot \frac{\log n}{2} \cdot (\log\log n - 1) \\ &\leq \frac{1}{2}\sqrt{n}\log n \log\log n = o(n) \quad \text{bits.} \end{aligned}$$

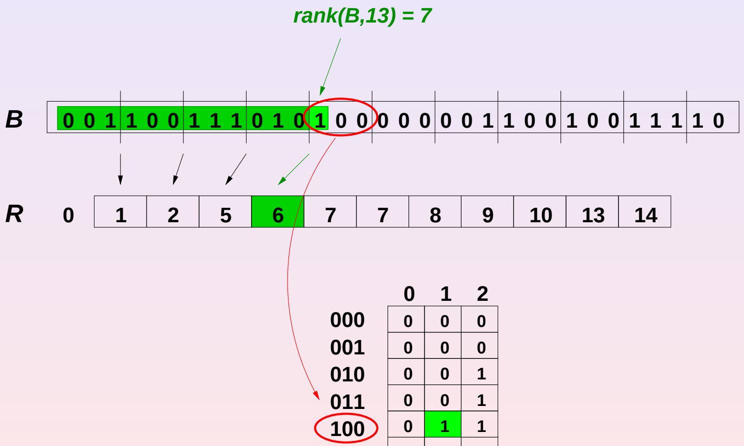
  For example, if $n = 2^{32}$, T just needs 512 KB.
- Then, the final answer is obtained in constant time as:

$$R[q] + T[B[qb+1..qb+b], r]$$

- Good, but we spent $3n + o(n)$ bits...

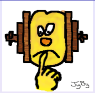
Rank





	0	1	2
000	0	0	0
001	0	0	0
010	0	0	1
011	0	0	1
100	0	1	1
101	0	1	1
110	0	1	2
111	0	1	2

Rank



Achieving $o(n)$ extra bits

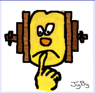
- We also cut B into superblocks of

$$s = \frac{(\log n)^2}{2} = b \cdot \log n \quad \text{bits.}$$

- We store an array $S[1, n/s]$ with the *rank* at the beginning of superblocks.
- Now the values of R are stored adding just from the beginning of the corresponding superblock:

$$\begin{aligned} R[q] &= rank(B, qb) - S[\lfloor q/\log n \rfloor] \\ &= rank(B, qb) - rank(B, \lfloor q/\log n \rfloor \cdot \log n) \end{aligned}$$

Rank



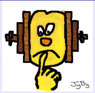
- As we need $\log n$ bits to store a *rank* value in S, the total number of bits used by S is

$$\frac{n}{s} \cdot \log n \;=\; \frac{n}{(\log n)^2/2} \cdot \log n \;=\; \frac{2n}{\log n} \;=\; o(n)$$

- As the R values are stored relative to the superblock, they can be only as large as $s = O(\log n)^2$, and thus only need $2 \log\log n$ bits. Thus R now occupies

$$\begin{aligned} \frac{n}{b} \cdot 2\log\log n \;&=\; \frac{n}{(\log n)/2} \cdot 2\log\log n \\ &=\; \frac{4n \cdot \log\log n}{\log n} \;=\; o(n) \quad \text{bits.} \end{aligned}$$

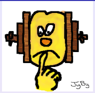
Rank

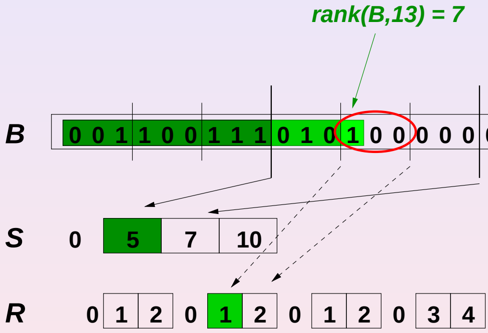


- ¿How we compute $rank(B, i)$?
  - Decompose $i = q \cdot b + r$, $0 \leq r < b$.
  - Decompose $i = q' \cdot s + r'$, $0 \leq r' < s$.
  - Add the contents of three tables:

$$rank(B, i) \ = \ S[q'] + R[q] + T[B[qb + 1..qb + b], r]$$

Rank

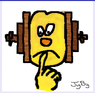




T

	0	1	2
000	0	0	0
001	0	0	0
010	0	0	1
011	0	0	1
100	0	1	1
101	0	1	1
110	0	1	2
111	0	1	2

Rank on Compressed Sequences

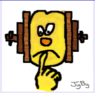


- In many applications, the sequence $B[1,n]$ contains few or many 1's.
- Let us focus on the case where there are $m << n$ 1's.
- Binary entropy:

$$H_0(B) \;=\; \frac{m}{n}\log\frac{n}{m} + \frac{n-m}{n}\log\frac{n}{n-m}$$

- We can have constant-time *rank* using just $nH_0(B)+o(n)$ bits.

Compressed Representation



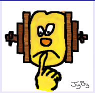
- Divide the sequence into blocks of $b = (\log n)/2$ bits.
- Let c_i be the number of 1's in a block B_i.
- The number of distinct blocks of length b with c_i 1's is

$$\binom{b}{c_i}$$

and hence we will try to represent B_i using

$$\left\lceil \log \binom{b}{c_i} \right\rceil \text{ bits.}$$

Compressed Representation

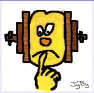


- The representation of B_i will be (c_i, o_i), where
  - The class, c_i, needs $\lceil \log(b+1) \rceil$ bits.
  - The offset, o_i, needs $\lceil \log \binom{b}{c_i} \rceil$ bits.
- The c_i's are of fixed width, and take in total

$$\frac{n}{b} \cdot \lceil \log(b+1) \rceil \;=\; O\left(\frac{n \cdot \log\log n}{\log n}\right)$$

- The o_i's are of variable width. We will concatenate them all.
- Later we see how to decode them.

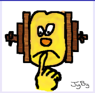
Compressed Representation



- The concatenation of the o_i's takes

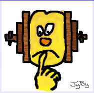
$$\begin{aligned}
\sum_{i=1}^{n/b} \left\lceil \log \binom{b}{c_i} \right\rceil &\le \sum_{i=1}^{n/b} \log \binom{b}{c_i} \quad + \; n/b \\
&= \log \prod_{i=1}^{n/b} \binom{b}{c_i} \quad + \; O(n/\log n) \\
&\le \log \binom{(n/b)\cdot b}{\sum_{i=1}^{n/b} c_i} \quad + \; O(n/\log n) \\
&= \log \binom{n}{m} \; + \; O(n/\log n) \\
&\le nH_0(B) \; + \; O(n/\log n) \text{ bits.}
\end{aligned}$$

Compressed Representation



- We have represented B using $nH_0(B) + o(n)$ bits!
- But how can we extract a block B_i in constant time?
- First problem: where is o_i?
  - We can store the positions in $P[1, n/b]$...
  - ... but this requires $(n/b) \log n = 2n$ extra bits.
  - We define superblocks of $s = b \cdot \log n$ bits...
  - ... and store $P[1, n/s]$ only for superblocks.
  - We will have $P'[1, n/b]$ with pointers relative to the superblock (like in *rank*).
  - Since $|o_i| = O(\log n)$, each $P'[i]$ needs $O(\log \log n)$ bits.
  - Total: $O(n \log \log n / \log n)$ bits.

Compressed Representation

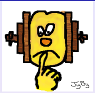


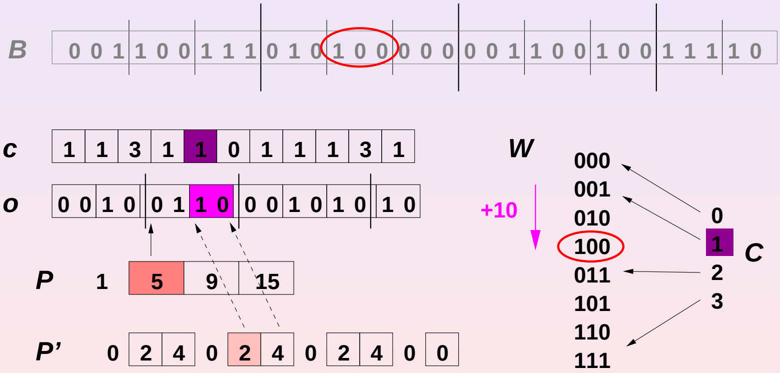
- Second problem: which block does (c_i, o_i) represent?
  - We will have a table $W[0, 2^b - 1]$ where all bitmaps of b bits are stored grouped by class.
  - The positions where each class begins in W will be precomputed in an array

$$C[c] \;=\; 1 + \sum_{i=0}^{c-1} \binom{b}{i}$$

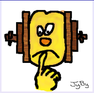
  - W is sorted so that o_i is the corresponding index within the zone of class c_i.
  - Hence (c_i, o_i) represents $W[C[c_i] + o_i]$.
  - These tables occupy $O(\sqrt{n} \log n)$ bits.

Compressed Representation



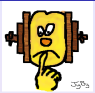


Rank



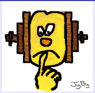
- The structures for *rank* occupy $o(n)$ bits on top of B.
- They need constant-time access to blocks of B.
- This is what we have obtained with the compressed representation!
- Hence we have obtained *rank* in constant time over the compressed representation (to H_0).
- More precisely, we use $nH_0(B) + O(n \log\log n / \log n)$ bits.

Symbol Sequences



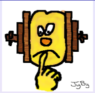
- Assume now we handle a sequence $S = s_1 s_2 \ldots s_n$ over an alphabet Σ of size σ.
- A plain representation of S needs $n \log \sigma$ bits.
- We wish to solve *rank* over this sequence:
  - $rank_c(S, i) =$ number of occurrences of c in $S[1, i]$.
- How can we extend our results on bits?

The Wavelet Tree



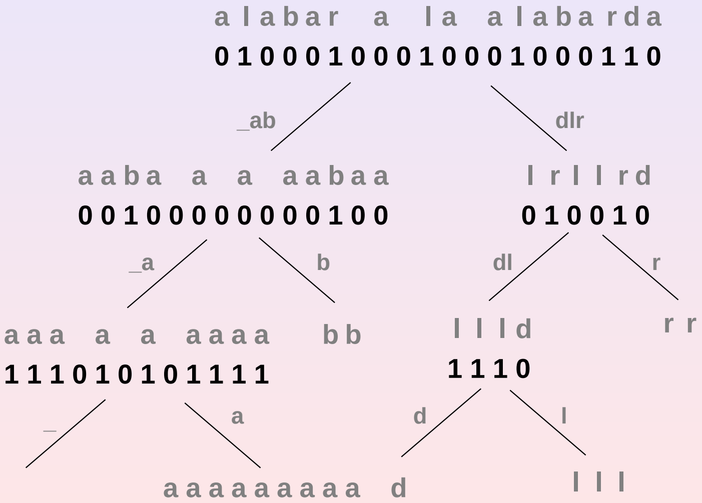
- Is an elegant solution that lets us store $S[1, n]$:
  - Using $n \log \sigma + o(n \log \sigma)$ bits.
  - Solving *rank* in time $O(\log \sigma)$.
  - Obtaining $S[i]$ in time $O(\log \sigma)$.
- Has many other applications (geometry, binary relations, etc.)

The Wavelet Tree

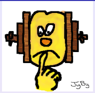


- Assume we split the alphabet into two subsets of the same size (or almost).
- We create a bitmap indicating to which subset each symbol of S belongs to.
- We store that bitmap in the root of the wavelet tree.
- For the left/right subtree, we choose the symbols of S from each subset.
- We continue recursively until each subset has just one symbol.
- It is easy to see that all the bitmaps add up to $n \log \sigma$ bits.

The Wavelet Tree

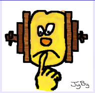


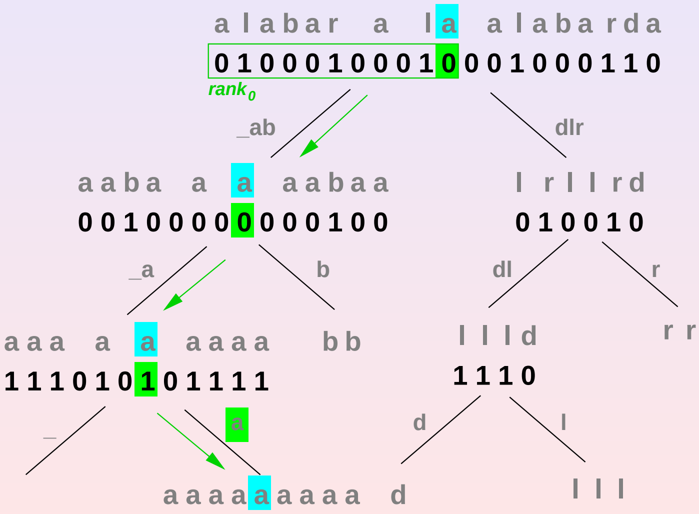
The Wavelet Tree



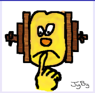
- How we retrieve symbol $S[i]$?
  - Look at $B[i]$ in the root bitmap.
  - If $B[i] = 0$,
    - Go to the left subtree.
    - The new position is $i' = rank_0(B, i)$.
  - If $B[i] = 1$,
    - Go to the right subtree.
    - The new position is $i' = rank_1(B, i)$.
  - When we arrive at a leaf, the corresponding symbol is $S[i]$.
- Time: $\log \sigma$ evaluations of *rank*.
- Need to preprocess the bitmaps for *rank* queries.

The Wavelet Tree



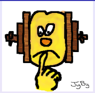


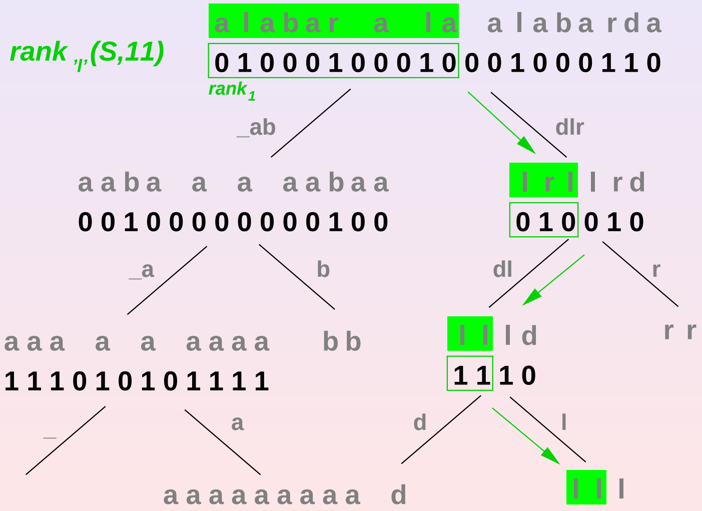
The Wavelet Tree



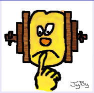
- How we compute $rank_c(S, i)$?
  - Let B be the root bitmap.
  - If c belongs to the left subtree,
    - Move to the left subtree.
    - The new position is $i' = rank_0(B, i)$.
  - If c belongs to the right subtree,
    - Move to the right subtree.
    - The new position is $i' = rank_1(B, i)$.
  - When we arrive at a leaf, the answer is i.
- Time: $\log \sigma$ evaluations of $rank$.

The Wavelet Tree





The Wavelet Tree

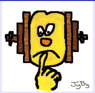


- If the bitmaps are uncompressed, the space is $n \log \sigma + o(n \log \sigma)$.
- If we compress the bitmaps using the technique shown before...
- Zero order entropy of a sequence: let there be n_c occurrences of symbol c in $S[1, n]$, then

$$H_0(S) \;=\; \sum_{c \in \Sigma} \frac{n_c}{n} \log \frac{n}{n_c}$$

- Lower bound to any encoding of S that always assigns the same code to the same symbol (e.g. Huffman).
- We will see that, by compressing the bitmaps of the wavelet tree to zero-order, we compress the sequence to zero-order as well.

The Wavelet Tree

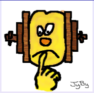


- We will consider the successive levels of the wavelet tree.
- Let $B[1,n]$ be the highest bits of the symbols.
- Say B has n_0 0s and n_1 1s.
- It is compressed to

$$nH_0(B) = n_0 \log \frac{n}{n_0} + n_1 \log \frac{n}{n_1}$$

.

The Wavelet Tree



- Consider now the subsequence $S_0[1, n_0]$ of the left child and B_0 the bitmap of its next highest bit.
- Let n_{00} and n_{01} be its number of 0s and 1s, respectively.
- B_0 is compressed to

$$n_0 H_0(B_0) \quad = \quad n_{00} \log \frac{n_0}{n_{00}} + n_{01} \log \frac{n_0}{n_{01}}$$

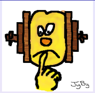
.

- Symmetrically with S_1. Adding up the space for the first two levels we get

$$n_{00} \log \frac{n}{n_{00}} + n_{01} \log \frac{n}{n_{01}} + n_{10} \log \frac{n}{n_{10}} + n_{11} \log \frac{n}{n_{11}}$$

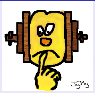
- If continuing up to level $\log \sigma$, we get $nH_0(S)$.

Texts



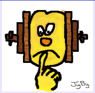
- Given an alphabet Σ of size σ...
- ... a text $T[1, n]$ is a sequence over Σ.
- This is just a sequence, but we are interested in other operations.
- Given a pattern $P[1, m]$:
  - Count the number of occurrences of P in T (*occ*).
  - Locate those *occ* occurrences in T.

Suffix Arrays

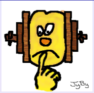


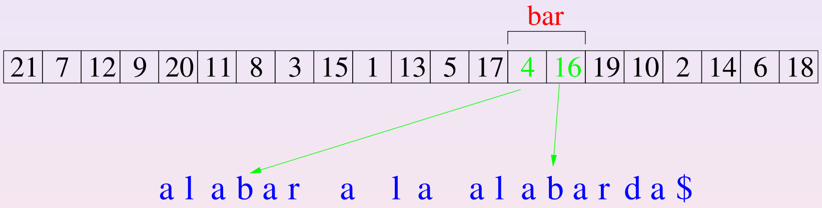
- Unlike inverted indexes, make no assumption on the text.
- Can retrieve any text substring.
- General model: consider the n suffixes $T[i, n]$ of T.
- Every substring of T is the prefix of a suffix of T.
- This structures indexes the set of suffixes of T and allows one to find all those sharing a given prefix.

Suffix Arrays



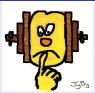
- It is an array with the n text suffixes in lexicographic order.
- Finding the substrings equal to P is the same as finding the suffixes that start with P.
- These form a lexicographical interval in the suffix array.
- This interval can be binary searched in $O(m \log n)$ time.





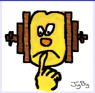
21	7	12	9	20	11	8	3	15	1	13	5	17	4	16	19	10	2	14	6	18

Suffix Arrays



- Yet, the suffix array yields space problems.
- Uses about $4n$ bytes (12GB for the human genome).
- Compressing it is of interest.
- But... ¿can we compress a permutation?.

The Burrows-Wheeler Transform (BWT)

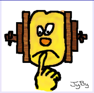


- Is a reversible permutation of T.
- It is used as a previous step to compression algorithms like bzip2.
- Puts together symbols having the same context.
- It is enough to compress those symbols “put together” to zero-order to achieve high-order entropy.
- Entropy of order k: if S_A is the sequence of the symbols that precede the occurrences of A in S, then

$$H_k(S) = \frac{1}{n} \sum_{A \in \Sigma^k} |S_A| H_0(S_A)$$

- Lower bound to encodings that consider the k symbols preceding the one to encode (e.g. PPM).

The Burrows-Wheeler Transform (BWT)



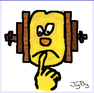
- Take all cyclic shifts of T.
- That is, $t_i t_{i+1} \dots t_{n-1} \$ t_1 t_2 \dots t_{i-1}$.
- The result is a matrix M of $n \times n$ symbols.
- Sort the rows lexicographically.
- M is essentially the list of suffixes of T in this order:

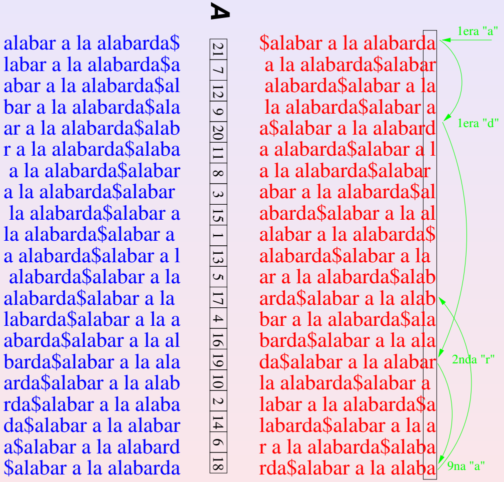
$$M[i] = T[A[i] \dots n] \cdot T[1 \dots A[i] - 1]$$

- The first column, F, holds the first characters of the suffixes: $F[i] = S[rank(D, i)]$.
- The last column, L, is the BWT of T, $T^{bwt} = L$.
- It is the sequence of symbols that precede the suffixes $T[A[i]..]$.

$$L[i] \quad = \quad T^{bwt}[i] \quad = \quad T[A[i] - 1]$$

(except if $A[i] = 1$, where $T^{bwt} = T[n] = \$$).



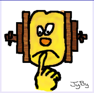


	A	
alabar a la alabarda$	21	$alabar a la alabarda
labar a la alabarda$a	7	a la alabarda$alabar
abar a la alabarda$al	12	alabarda$alabar a la
bar a la alabarda$ala	9	la alabarda$alabar a
ar a la alabarda$alab	20	a$alabar a la alabard
r a la alabarda$alaba	11	a alabarda$alabar a l
a la alabarda$alabar	8	a la alabarda$alabar
a la alabarda$alabar	3	abar a la alabarda$al
la alabarda$alabar a	15	abarda$alabar a la al
la alabarda$alabar a	1	alabar a la alabarda$
a alabarda$alabar a l	13	alabarda$alabar a la
alabarda$alabar a la	5	ar a la alabarda$alab
alabarda$alabar a la	17	arda$alabar a la alab
labarda$alabar a la a	4	bar a la alabarda$ala
abarda$alabar a la al	16	barda$alabar a la ala
barda$alabar a la ala	19	da$alabar a la alabar
arda$alabar a la alab	10	la alabarda$alabar a
rda$alabar a la alaba	2	labar a la alabarda$a
da$alabar a la alabar	14	labarda$alabar a la a
a$alabar a la alabard	6	r a la alabarda$alaba
$alabar a la alabarda	18	rda$alabar a la alaba

T alabar a la alabarda$

T^{bwt} araadl ll$ bbaar aaaa

The Burrows-Wheeler Transform (BWT)



- How to revert the BWT?
- For all i, $T = \ldots L[i]F[i]\ldots$.
- We know $L[1] = T[n-1]$, since $F[1] = \$ = T[n]$.
- Where is $c = L[1]$ in F?
- All occurrences of c appear in the same order in F and L:
  - Is the order given by the suffix that follows c in T.
- That c is at $F[C[c] + rank_c(L,i)]$, where

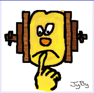
$$C[c] \;=\; \text{number of occurrences of symbols } < c \text{ in } T$$

- This is called LF-mapping, since it maps from L to F:

$$LF(i) = C(L[i]) + rank_{L[i]}(L,i)$$

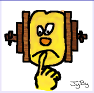
- Then $T[n-2] = L[LF(1)]$, $T[n-3] = L[LF(LF(1))]$, etc.

The FM-Index



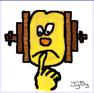
- Uses the connection between the suffix array and the BWT.
- Replaces the binary search by the more efficient backward search.
- To search for $P[1, m]$ we start with p_m.
- The segment of A corresponding to it is $A[C(p_m) + 1, C(p_m + 1)]$.
- In general, we will know that the occurrences of $P[i + 1, m]$ start at $A[sp_{i+1}, ep_{i+1}]$...
- ... and will use something similar to the LF-mapping to obtain the zone $A[sp_i, ep_i]$ corresponding to $P[i, m]$.
- We finish with sp_1 and ep_1.

The FM-Index



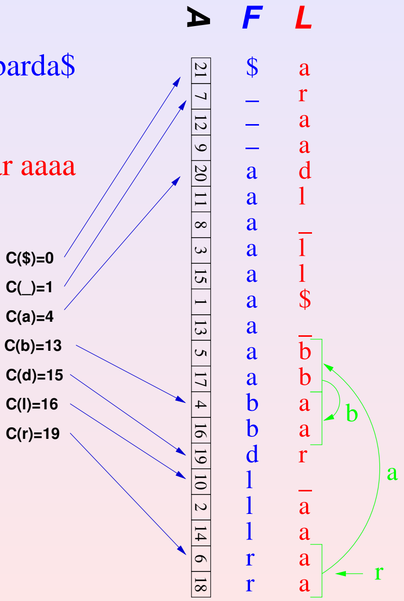
- Assume we start with the segment $A[sp_{i+1}, ep_{i+1}]$ of the occurrences of $P[i+1, m]$.
- How we update it to the occurrences of $P[i, m]$?
  - For a $sp_{i+1} \leq j \leq ep_{i+1}$, $M[j][1..m-i] = P[i+1, m]$.
  - The occurrences of $P[i, m]$ in T appear as $L[j] = p_i$ in this area, since $T = \ldots L[j] \cdot M[j][1..m-i] \ldots$
  - Hence the answer is the range of the $LF(j)$ where $L[j] = p_i$.
- This is computed simply as

$$\begin{aligned} sp_i &= C(p_i) + rank_{p_i}(L, sp_{i+1} - 1) + 1 \\ ep_i &= C(p_i) + rank_{p_i}(L, ep_{i+1}) \end{aligned}$$



T alabar a la alabarda$

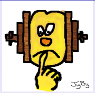
$\boldsymbol{T}^{bwt}$ araadl ll$ bbaar aaaa



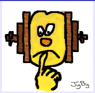
A	F	L
21	$	a
7	_	r
12	_	a
9	_	a
20	a	d
11	a	l
8	a	_
3	a	l
15	a	l
1	a	$
13	a	_
5	a	b
17	a	b
4	b	a
16	b	a
19	d	r
10	l	_
2	l	a
14	l	a
6	r	a
18	r	a

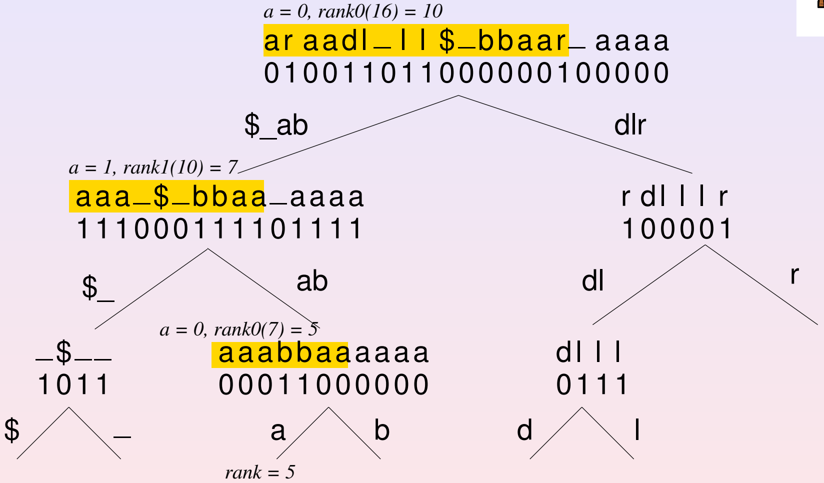
C($)=0
C(_)=1
C(a)=4
C(b)=13
C(d)=15
C(l)=16
C(r)=19

The FM-Index

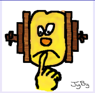


- After m iterations, we get the interval for $P[1, m]$.
- The cost is $2m$ calls to $rank_c$.
- Using a wavelet tree on $T^{bwt} = L$,
  - We get space $nH_0(T) + o(n \log \sigma)$.
  - We count in time $O(m \log \sigma)$.



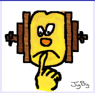

a = 0, rank0(16) = 10
ar aadl_ l l $_bbaar_ aaaa
01001101100000000100000
$_ab
dlr
a = 1, rank1(10) = 7
aaa_$_bbaa_aaaa
111000111101111
r dl l l r
100001
$_
ab
dl
r
a = 0, rank0(7) = 5
aaabbaaaaaa
00011000000
_$__
1011
dl l l
0111
$
_
a
b
d
l
rank = 5

The FM-Index



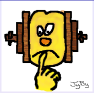
- But, how to obtain $A[i]$ and $T[l, r]$?
- We will sample and store some cells of A.
- We take those $A[i]$ pointing to positions of T which are multiples of some b.
- We store them in increasing order of i in contiguous form, in an array $A'[1, n/b]$.
- ... and we have a bitmap $B[1, n]$ marking the sampled is.
- The positions and the bitmap will occupy $O(\frac{n}{b} \log n)$ bits.

The FM-Index



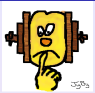
- Assume we want to find out $A[i]$ (but have not A).
  - If $B[i] = 1$, then $A[i] = A'[rank(B, i)]$.
  - Else, if $B[LF(i)] = 1$, then $A[LF(i)] = A'[rank(B, LF(i))]$, $A[i] = A[LF(i)] + 1$.
  - Else, ...
  - If $B[LF^t(i)] = 1$, then $A[i] = A'[rank(B, LF^t(i))] + t$.
- This ends in at most b steps.
- Hence, we can know $A[i]$ in time $O(b \log \sigma)$.
- For example, in time $O(\log^{1+\epsilon} n)$, spending $o(n \log \sigma)$ extra bits.
- Now we have replaced A!

The FM-Index



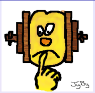
- Now we store the same sampling in another way.
- We store the i values by increasing position in T, in $E[1, n/b]$.
- Say we want to extract $T[l, r]$ (but have not T).
  - The first sampled position after it is $r' = 1 + \lfloor r/b \rfloor \cdot b$.
  - And is pointed from $A[i] = A[E[1 + \lfloor r/b \rfloor]]$.
  - We know $T[r' - 1] = L[i]$.
  - We know $T[r' - 2] = L[LF(i)]$.
  - ...
  - We know $T[l] = L[LF^{r'-l-1}(i))]$.
- The total cost is $O((b + r - l) \log \sigma)$.
- For example, in time $O((r - l) \log \sigma + \log^{1+\epsilon} n)$, spending $o(n \log \sigma)$ extra bits.
- Now we have replaced the text!

The FM-Index



- The $O(\log n)$ time can be improved to $O(\frac{\log \sigma}{\log \log n})$ in theory.
- The space is that of the wavelet tree.
  - $n \log \sigma + o(n \log \sigma)$ (like T), without compression.
  - $nH_0(T) + o(n \log \sigma)$ by compressing the bits.
  - Many efforts have been carried out to reduce this to $nH_k(T) + o(n \log \sigma)$, finally succeeding.
- Yet, it turned out that no effort was necessary!

The FM-Index



- Consider the $t = \sigma^k$ partitions of the BWT by context of length k

$$T^{bwt} \; = \; S \; = \; S^1 S^2 \ldots S^t$$

- By the H_k formula,

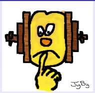
$$nH_k(T) \; = \; \sum_{i=1}^{t} |S^i| H_0(S^i)$$

- Now, recall that the zero-order compression we achieved for the wavelet tree was the sum of the zero-order entropies of small blocks:

$$nH_0(S) \; \leq \; bH_0(S_1) + bH_0(S_2) + \ldots + bH_0(S_{n/b})$$

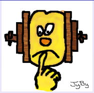
where the S_j's are the blocks of length b.

The FM-Index



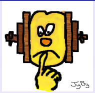
- Thus the wavelet tree compresses $S^1 S^2 \ldots S^t$ to $H_0(S^1) + H_0(S^2) + \ldots + H_0(S^t)$...
- ... plus some worst-case measure for the limits among strings.
- The total size of the compressed wavelet tree is $nH_k(T) + O(\sigma^k \log n)$.
- The second term is $o(n \log \sigma)$ for moderate $k \leq \alpha \log_\sigma n$, for any constant $0 < \alpha < 1$.

Epilogue



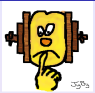
- This FM-index, and many other self-indexes, are implemented at mirrors http://pizzachili.dcc.uchile.cl http://pizzachili.di.unipi.it.
- The implementations are freely accessible for any purpose, and their practicality can be tested.
- We have published a practical survey considering most of the currently available indexes:
  Compressed Text Indexes: From Theory to Practice. *ACM Journal of Experimental Algorithmics (JEA)* 13:article 12, 2009.
- And also a more conceptual survey:
  Compressed Full-Text Indexes. *ACM Computing Surveys* 39(1), article 2, 61 pages, 2007.

Epilogue



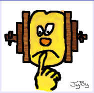
- This is, of course, not the end of the story.
- There are several problems, some totally, some partially, and some not at all solved.
- Some examples of almost totally solved extensions:
  - How to build the index in little space?
  - How to handle a dynamic text collection? (slow)
  - How to support more functionality, as in suffix trees?

Epilogue



- Some examples of only partially solved extensions:
  - How to support document retrieval? (theoretical)
  - How to compress highly repetitive collections? (some progress)
  - How to handle the index in secondary memory? (little progress).
- And of course, we have all the other data structures!

Epilogue



- A brave new world of algorithmic challenges is ahead:
  - For those who love classical algorithmics, it is plenty of opportunities to redesign classical data structures.
  - For those who like compression and information theory, this gives the new challenge of designing schemes that not only can recover the data but also retain functionality and direct access to it.
  - For those trying to walk the thin path between sterile theory and shallow practice, this is a marvellous opportunity to make them work together and achieve miracles!