Compressed Indexes for Text Searching

Gonzalo Navarro
Center for Web Research
Department of Computer Science
University of Chile

Road Map

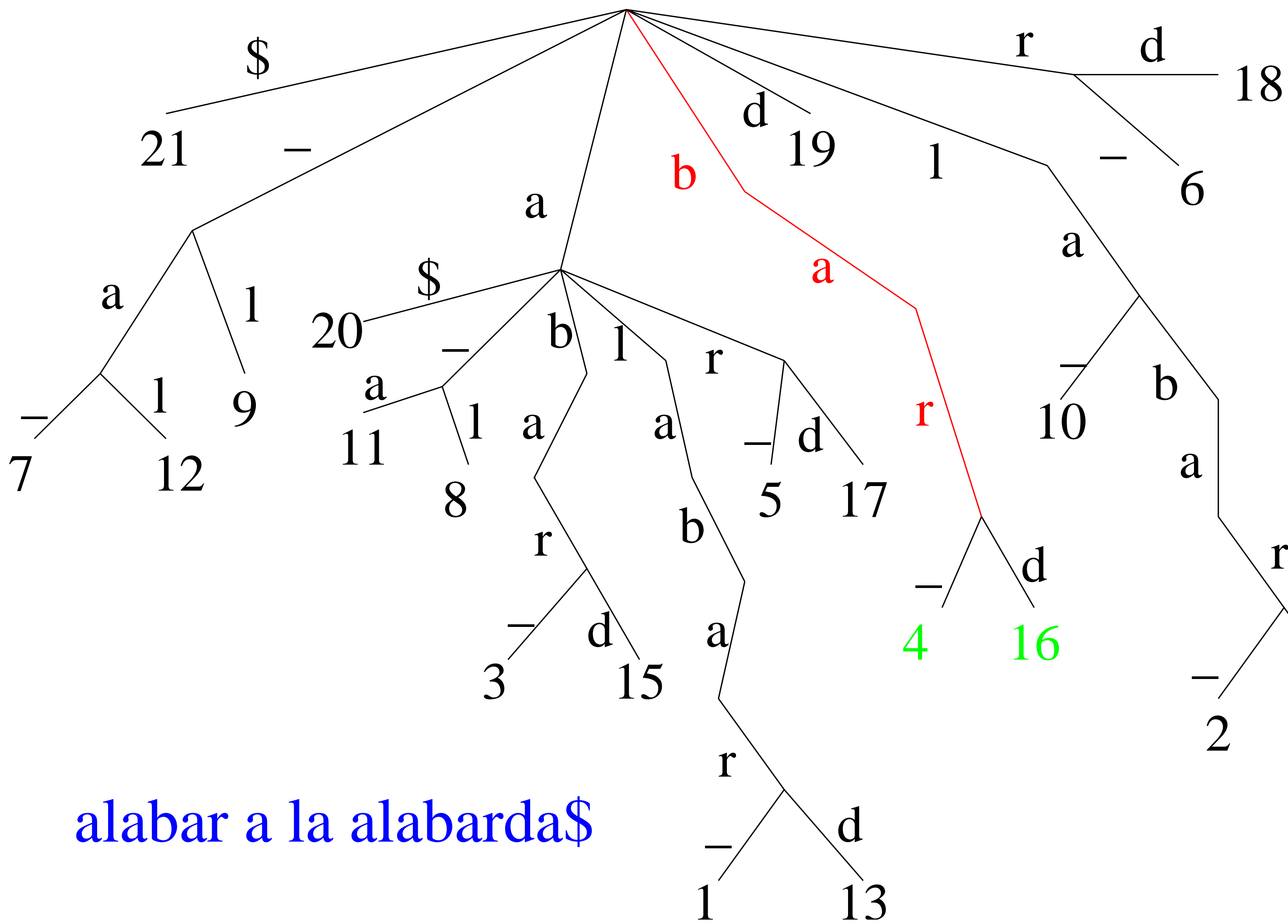
- Text databases are more than natural language
- Classical data structures
  - Suffix tries
  - Suffix trees
  - Suffix arrays
- Succint data structures
  - Compact PAT trees
  - Compact Suffix Arrays
- Self-indexing data structures
  - Compressed Suffix Arrays
  - FM-Index
  - LZ-Index: stay for Diego's talk
- State of the art and open challenges

Text Databases are More than Natural Language

- Natural language → inverted indexes.
- Other applications
  - Computational biology
  - Oriental (and even agglutinating) languages
  - Multimedia databases
- Current challenges
  - Huge sizes
  - Complex search patterns
  - Need fast search
  - Dynamic
- In this talk we focus on
  - Reducing their space (so they fit in main memory)
  - Exact substring search procedures
  - This is the kernel of the problem...
  - ... but just the beginning of a complete solution.

The Simplest Solution: Suffix Tries

- A trie that stores all the suffixes of the texts.
- Each leaf represents a unique substring (and extensions to suffix).
- Each internal node represents a repeating substring.
- Every text substring is the prefix of some suffix.
- So any substring of length m can be found in $O(m)$ time.
- Many other search problems can be solved with suffix tries:
  - Longest text repetitions.
  - Approximate searching in $O(n^\lambda)$ time ($n =$ text size).
  - Regular expression searching in $O(n^\lambda)$ time.
  - ... and many others.
- Let n be the text size, then the trie
  - on average is $O(n)$ space and construction time...
  - but in the worst case it is $O(n^2)$ space and construction time.



alabar a la alabarda$
21
19
18
6
20
7
12
9
11
8
10
5
17
4
16
3
15
2
1
13

Guaranteeing Linear Space: Suffix Trees

Weiner, McCreight, Ukkonen, K

- Works essentially as a suffix trie.
- Unary paths are compressed.
- It has $O(n)$ size and construction time, in the worst case.
- Construction can be done *online*.
- After the $O(m)$ search, the R occurences can be collected in $O(R)$ time.
- So it's simply perfect!
- Where is the trick?
  - The suffix tree needs at least 20 times the text size.
  - It is hard to build and search in secondary memory.
  - It is difficult to update upon changes in the text.
- Very compact implementations achieve $10n$ bytes.
- Other variants: Patricia trees, level-compressed tries... not better in practice.

alabar a la alabarda$

$ 21 _ a l a _ l 7 12 9

a $ 20 _ a 11 l 8 bar _ 3 d 15 labar _ 1 d 13 r _ 5 d 17

bar _ 4 d 16

d 19

la _ 10 bar _ 2

r d 18 _ 6

Just the Tree Leaves: Suffix Arrays

Manber & Myers, Gonnet et al., Kurtz et al., Kärkkäinen, Baeza-Yates e

- An array of all the text positions in lexicographical suffix order.
- Much simpler to implement and smaller than the suffix tree.
- Simple searches result in a range of positions.
- It can simulate all the searches with an $O(\log n)$ penalty factor.
- It can be built in $O(n)$ time, but construction is not online.
- In practice it takes about 4 times the text size.
- Linear-time construction needs $8n$ bytes, otherwise construction is $O(n \log n)$
- Paying $6n$ extra space, we can get rid of the $O(\log n)$ penalty.
- Builds in secondary memory in $O(n^2 \log(M)/M)$ time ($M =$ RAM size).
- Searching in secondary memory is painful.
- But it can be improved a lot with sampling supraindexes.
- Still dynamism is a problem.

alabar a la alabarda$

bar

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

Minimizing the Impact of the Tree: Compact PAT Trees

Jacobson, Munro, Raman, Clark &

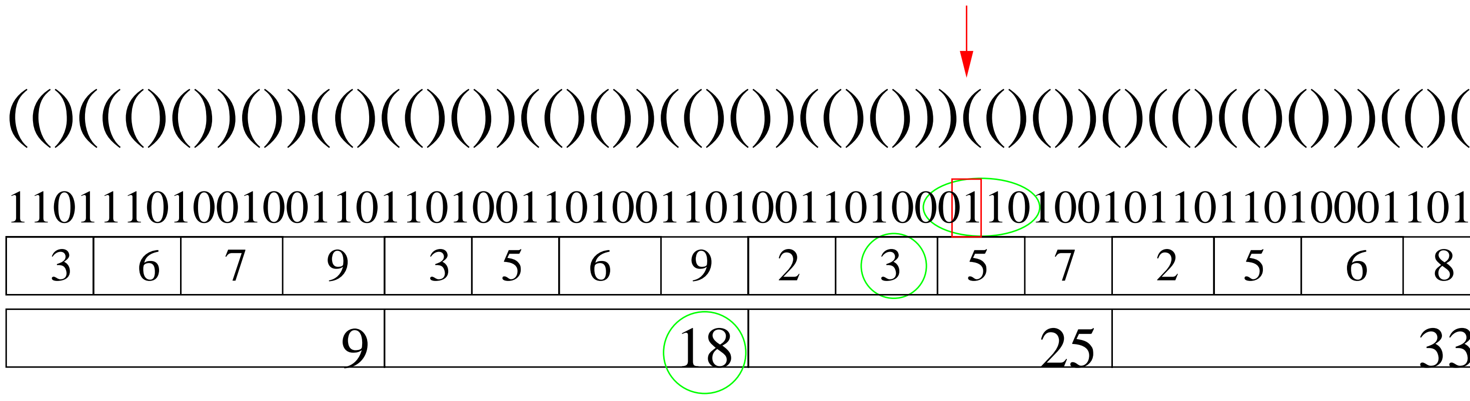
- Separate a suffix tree into the leaves (suffix array) plus the structure.
- The tree structure itself can be represented as a sequence of parentheses.
- This sequence can be traversed with the usual tree operations.
- Basic tools are: ranking of bits and finding closing parentheses.
- The result is functionally similar to a suffix tree.
- Experimental results report $5n$ to $6n$ bytes.
- So it is still too large.
- Worse than that, it is built from the plain suffix tree.
- Search in secondary storage is reasonable (2–4 disk accesses), by storing subtr
  single disk pages.
- Dynamism is painful.
- The $rank$ mechanism is very valuable by itself.

(()((()())())(()(()())(()())(()())(()()))(()())()(()(()()))(()()))

alabar a la alabarda$

bar

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


3 6 7 9 3 5 6 9 2 3 5 7 2 5 6 8
9 18 25

Smaller than Suffix Arrays: Compact Suffix Arrays

Mä

- Exploits self-repetitions in the suffix array.
- An area in the array can be equal to another area, provided all the values are s
- Those repetitions are factored out.
- Search time is $O(m \log n + R)$ and fast in practice .
- Extra space is around $1.6n$.
- Construction needs $O(n)$ time given the suffix array.
- This is much better than anything else seen so far.
- Still no provisions for updating nor secondary memory.

alabar a la alabarda$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
21	7	12	9	20	11	8	3	15	1	13	5	17	4	16	19	10	2	14	6

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
21	7	12	9	20	11	8	3	15	1	13	5	17	4	16	19	10	*(9,4,0)*

0	1	2	3	4	5	6	7	8	9	10	11	12
21	7	*(3,2,1)*	*(10,2,0)*	8	*(12,2,0)*	1	13	*(9,2,0)*	*(5,2,0)*	19	10	*(6,4,0)*

Towards Self-Indexing

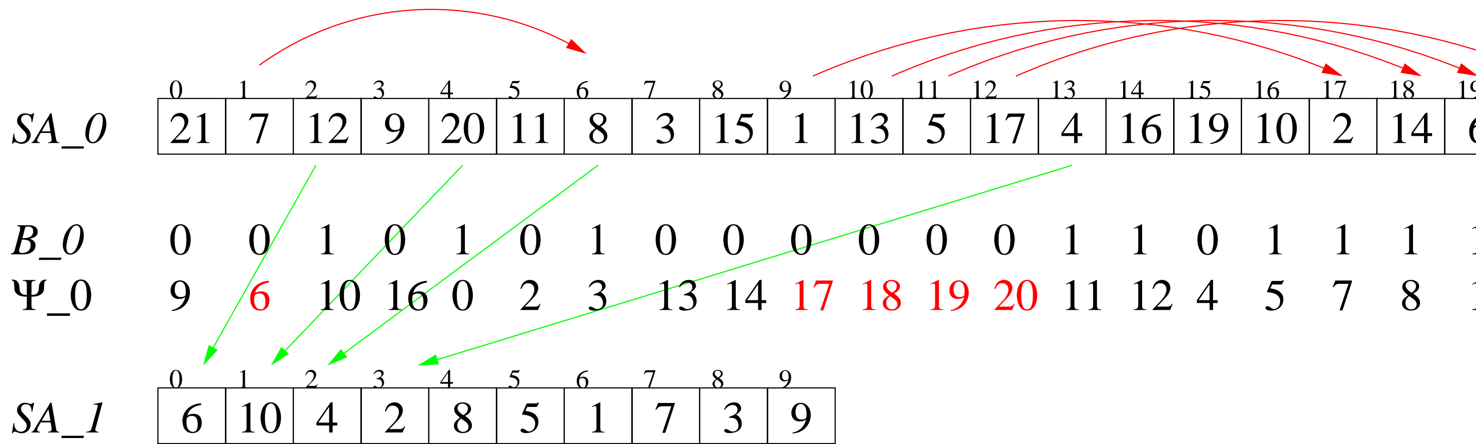
- Up to now, we have focused on compressing the index.
- We have obtained decent extra space, 1.6 times the text size.
- However, we still need the text separately available in plain form.
- Could the text be compressed?
- Moreover, could the compressed text act as an index by itself?
- Self-index: a data structure that acts as an index and comprises the text
- After the index is built, the text can be deleted.
- The retrieval of text passages is done via a request to the index.
- Hence, retrieving text becomes an essential operation of the index.
- Exciting possibility: the whole thing takes less space than the original text!

Exploiting Repetitions Again: Compressed Suffix Arrays

Grossi &

- We replace the suffix array by a level-wise data structure.
- Level 0 is the suffix array itself.
- Level $k+1$ stores only the even pointers of level k, divided by 2.
- Bit array $B_k(i)$ tells whether the i-th suffix array entry is even.
- Array $\Psi_k(i)$ tells where is the pointer to position $i+1$.
- Note that level $k+1$ needs half the space of level k.
- For large enough $k = \ell = \Theta(\log\log n)$, the suffix array is stored explicitly.
- If $B_k(i) = 1$, $SA_k[i] = 2 \times SA_{k+1}[rank(B_k, i)]$.
- If $B_k(i) = 0$, $SA_k[i] = 2 \times SA_{k+1}[rank(B_k, \Psi_k(i))] - 1$.
- This permits computing $SA[i] = SA_0[i]$ in $O(\log\log n)$ time.
- Hence searching can be done at $O((m + \log\log n)\log n)$ time.
- We store the B_k and Ψ_k explicitly.

- Function $rank(B_k, \cdot)$ is computed in constant time as explained.
- The Ψ_k vector is stored with differential plus delta encoding, with an absolute each $\log n$ entries.
- A two-level structure like that for $rank$ computes Ψ_k in constant time.
- The delta encoding dominates space, and it is good because of the same rep property.
- By dividing the array by 2^c instead of by 2, for $c = \varepsilon \log \log n$, we get S constant time.



B_1 1 1 1 1 1 0 0 0 0 0

Ψ_1 7 6 5 8 9 0 3 4 2 1

	0	1	2	3	4
SA_2	3	5	2	1	4
B_2	0	0	1	0	1
Ψ_2	4	3	0	2	1

	0	1
SA_3	1	2

alabar a la alabarda$

D = 110010000000010110010

C = $_abdlr

Compressed Suffix Arrays without Text

Sad

- Use only $\Psi = \Psi_0$ and C.
- The text can be discarded: $rank$ over a bit vector D with a 1 for each cha
  first character pointed by $SA[i]$.
- Using Ψ_0 we obtain successive text characters.
- Overall space is $n(H_0 + 8 + 3\log_2 H_0)$ bits, text included.
- In practice, this index takes $0.6n$ to $0.7n$ bytes.
- Performance is good for counting, but showing text contexts is slow.
- Search complexity is $O(m \log n + R)$.
- Recently shown how to build in little space.

Building on the Burrows-Wheeler Transform: FM-Index

Ferragina & M

- Based on the Burrows-Wheeler transform.
- The characters preceding each suffix are collected from the suffix array.
- The result is a more compressible permuted text.
- These are coded with move to front, run-length and δ-coding.
- The transformation is reversible.
- Given a position in the permuted text (last column), we can find the position letter preceding it in the original text.
- The trick is that we know which letter is at each position in the sorted array of (first column).
- And letters in the first column follow those in the last column.
- Given a letter in the last column, which is the k-th c, we easily find its posi the first column, and hence the character preceding it.
- Starting at the character "$", we can obtain the text backwards.

alabar a la alabarda$

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

araadl ll$ bbaar aaaa BWT

alabar a la alabarda$	$alabar a la alabarda
labar a la alabarda$a	a la alabarda$alabar
abar a la alabarda$al	alabarda$alabar a la
bar a la alabarda$ala	la alabarda$alabar a
ar a la alabarda$alab	a$alabar a la alabard
r a la alabarda$alaba	a alabarda$alabar a l
a la alabarda$alabar	a la alabarda$alabar
a la alabarda$alabar	abar a la alabarda$al
la alabarda$alabar a	abarda$alabar a la al
la alabarda$alabar a	alabar a la alabarda$
a alabarda$alabar a l	alabarda$alabar a la
alabarda$alabar a la	ar a la alabarda$alab
alabarda$alabar a la	arda$alabar a la alab
labarda$alabar a la a	bar a la alabarda$ala
abarda$alabar a la al	barda$alabar a la ala
barda$alabar a la ala	da$alabar a la alabar
arda$alabar a la alab	la alabarda$alabar a
rda$alabar a la alaba	labar a la alabarda$a
da$alabar a la alabar	labarda$alabar a la a
a$alabar a la alabard	r a la alabarda$alaba
$alabar a la alabarda	rda$alabar a la alaba

1st "a"

1st "d"

2nd "r"

9th "a"

- The index also has a cumulative letter frequency array C.
- As well as $Occ[c, i]$ = number of occurrences of c before position i in the per text.
- If we start at a position i in the permuted text, with character c, the previous is at position $C[c] + Occ[c, i]$.
- The search for pattern $p_1 \ldots p_m$ is done backwards, in optimal $O(m)$ time.
- First we take interval for p_m, $[l, r) = [C[p_m], C[p_m + 1])$.
- Interval for $p_{m-1}p_m$ is $[l, r) = [C[p_{m-1}] + Occ[p_{m-1}, l], C[p_{m-1}] + Occ[p_{m-1}$
- C is small and Occ is cumulative, so it is easy to store with blocks and superb
- The in-block computation of Occ is done by scanning the permuted text.
- We can show text contexts by walking the permuted text as explained.
- A problem is how to know which text position are we at!
- Some suffix array pointers are stored, we walk the text until we find one.

- Overall space is $5H_kn$ bits, text included, for any k.
- In practice, 0.3 to 0.8 times the text size, and includes the text.
- Counting the number of occurrences is amazingly fast.
- Reporting their positions and text contexts is very slow, however.
- Search complexity is $O(m + R\log^{\varepsilon} n)$.
- Construction needs to start from the suffix array.
- Some (theoretical) provisions for dynamism, search time becomes $O(m \log^3 n$

alabar a la alabarda$

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

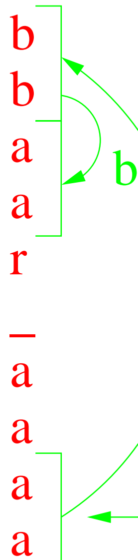
araadl ll$ bbaar aaaa BWT

2,6,1,0,5,6,5,1,0,5,2,6,0,5,0,6,3,2,0,0,0 MTF

C[$]=0, C[_]=1, C[a]=4, C[b]=13, C[d]=15, C[l]=16, C[r]=19

Occ[$] = 0,0,0,0,0,0,0,0,0,1,1,1,1,1,1,1,1,1,1,1,1
Occ[_] = 0,0,0,0,0,0,1,1,1,1,2,2,2,2,2,2,3,3,3,3,3
Occ[a] = 1,1,2,3,3,3,3,3,3,3,3,3,3,4,5,5,5,6,7,8,9
Occ[b] = 0,0,0,0,0,0,0,0,0,0,0,1,2,2,2,2,2,2,2,2,2
Occ[d] = 0,0,0,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1
Occ[l] = 0,0,0,0,0,1,1,2,3,3,3,3,3,3,3,3,3,3,3,3,3
Occ[r] = 0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,2,2,2,2,2,2

F	L
$	a
	r
	a
	a
a	d
a	l
a	_
a	l
a	l
a	$
a	_
a	b
a	b
b	a
b	a
d	r
l	_
l	a
l	a
r	a
r	a



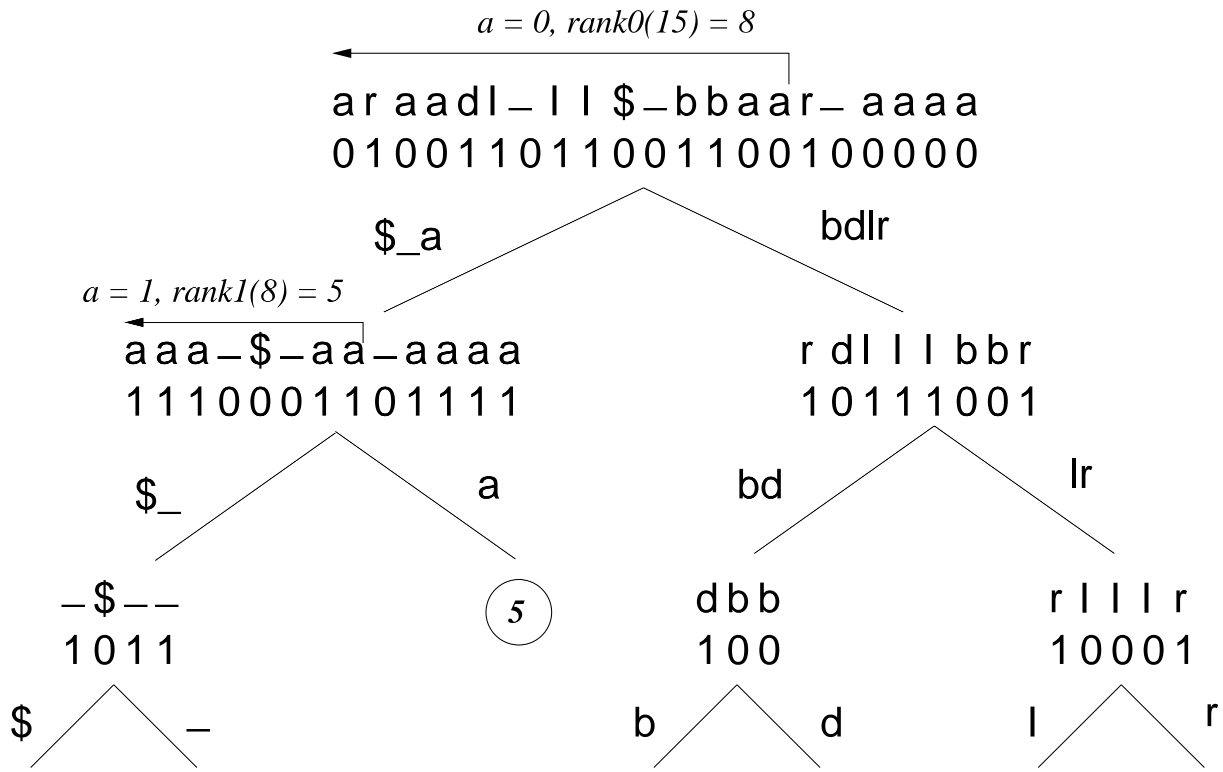
FM-Index Implementations

- The main issue is to implement $Occ[c,i]$ efficiently...
- while reaching space close to $nH_k(T)$.
- The original FM-index achieves this for constant-size alphabets...
- but it is completely impractical otherwise.
- The best current solutions to handle larger alphabets:
  - Succinct Suffix Array
  - Run-Length FM-Index
  - Alphabet-Friendly FM-Index

Wavelet Trees and Succinct Suffix Arrays

Sadakane; Mäkinen & N

- A tree data structure built on the alphabet domain.
- Every subtree handles a subset of Σ.
- Every node contains a bit array telling to which child does each text position b
- It can retrieve any string character in time $O(\log \sigma)$.
- It can solve Occ queries over the string in time $O(\log \sigma)$.
- By using multi-ary nodes we can get constant time for $\sigma = O(\text{polylog } n)$.
- The SSA just uses a wavelet tree over $bwt(T)$ to implement $Occ(c, i)$.
- Yet, the space is $nH_0(T)$.



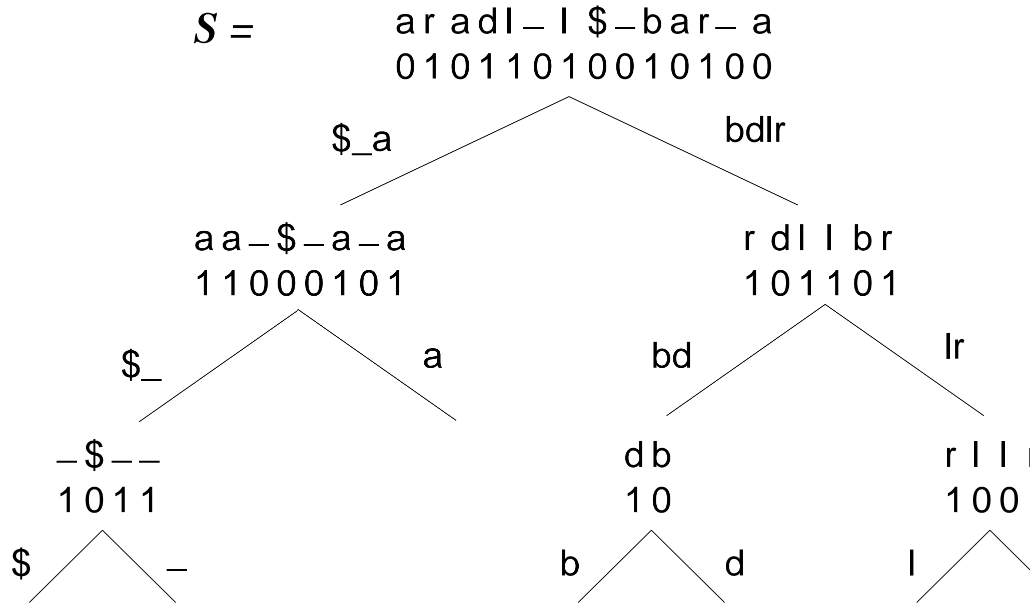
a = 0, rank0(15) = 8
ar aadl_ l l $_bbaar_ aaaa
010011011001100100000
$_a
bdlr
a = 1, rank1(8) = 5
aaa_$_aa_aaaa
1110001101111
r dl l l bbr
10111001
$_
a
bd
lr
_$__
1011
5
dbb
100
r l l l r
10001
$
_
b
d
l
r

Run-Length FM-Index

Mäkinen & N

- Consider the runs in $bwt(T)$.
- We prove that there are $O(nH_k(T))$ runs.
- We proceed as on the SSA, over the string of run heads.
- We manage to obtain Occ over the original $bwt(T)$ using the runs.
- Now the space is $nH_k(T) \log \sigma$.
- Time complexities are as in the SSA.

B'	T^{bwt}	*B*	*S*
1	a	1	a
1	r	1	r
1	a	1	a
1	a	0	
1	d	1	d
1	l	1	l
0	_	1	_
1	l	1	l
0	l	0	
1	$	1	$
0	_	1	_
0	b	1	b
0	b	0	
1	a	1	a
0	a	0	
1	r	1	r
1	_	1	_
1	a	1	a
0	a	0	
1	a	0	
1	a	0	



Alphabet-Friendly FM-Index

Ferragina, Manzini, Mäkinen & Na

- Another way to get $nH_k(T)$ space from the SSA.
- Split $bwt(T)$ in pieces and build an SSA on each piece.
- Let Rodrigo explain it better in the next talk...

State of the Art and Open Challenges

- There exist now several competing self-indexes.
- Sometimes their theoretical and practical versions differ a lot.
- Sometimes they are not even likely to be implemented.
- They have achieved attractive complexities for
  - Counting the number of occurrences of a substring in a text.
  - Locate those occurrences in the text.
  - Extract a text context around each occurrence.
  - Display any text substring.
- In practice, they are not always that competitive.

- The concept of self-indexing is fascinating...
- ... but it has a long way ahead.
  - Practical implementations.
  - Competitive times against classical solutions.
  - Construction in succinct space.
  - Handling of dynamism (inserting and deleting texts).
  - Handling of secondary memory.

If interested don't miss implementations, test beds, and survey at

http://pizzachili.dcc.uchile.cl