Compressed Text Indexes with Fast Locate

Rodrigo González[1] and Gonzalo Navarro[1]

[1]Department of Computer Science
University of Chile

Compressed Text Indexes with Fast Locate, CPM 2007

Outline

Outline

1. Motivation
   - Results obtained
2. Compressing the Suffix Array
   - Exposing the runs
   - Re-Pair
   - Compressing Repair Rules
   - Faster compression
3. Experimental Results
   - Compression performance
   - Attached to a compressed self-index
   - Like a full-text index
   - Secondary memory
4. Conclusions and Future Work
   - Conclusions
   - Future Work

Outline

1. Motivation
   - Results obtained
2. Compressing the Suffix Array
   - Exposing the runs
   - Re-Pair
   - Compressing Repair Rules
   - Faster compression
3. Experimental Results
   - Compression performance
   - Attached to a compressed self-index
   - Like a full-text index
   - Secondary memory
4. Conclusions and Future Work
   - Conclusions
   - Future Work

Outline

Motivation

- In pattern matching, the main problem consists in searching a text T for a pattern P, where text and pattern are sequences of symbols 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 compressed self-indexes.
- In compressed self-indexes, locate the occurrences position in T where P occurs, is still hundreds to thousands of times slower than their classical counterparts. While classical indexes pay $O(occ)$ time to locate the occ occurrences, self-indexes pay $O(occ \log^{\varepsilon} n)$.

Motivation

- In pattern matching, the main problem consists in searching a text T for a pattern P, where text and pattern are sequences of symbols 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 compressed self-indexes.
- In compressed self-indexes, locate the occurrences position in T where P occurs, is still hundreds to thousands of times slower than their classical counterparts. While classical indexes pay $O(occ)$ time to locate the occ occurrences, self-indexes pay $O(occ \log^{\varepsilon} n)$.

Motivation

- In pattern matching, the main problem consists in searching a text T for a pattern P, where text and pattern are sequences of symbols 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 compressed self-indexes.
- In compressed self-indexes, locate the occurrences position in T where P occurs, is still hundreds to thousands of times slower than their classical counterparts. While classical indexes pay $O(occ)$ time to locate the occ occurrences, self-indexes pay $O(occ \log^{\varepsilon} n)$.

Outline

Results obtained

- A new suffix array compression technique that builds on well-known regularity properties that show up in suffix arrays when the text they index is compressible.
- With this compression we reduce the suffix array to 20–70% of its original size, depending on its compressibility.
- We prove that the compression ratio we achieve depends on the k-th order empirical entropy of T ($H_k \log(1/H_k)$)

Results obtained

- A new suffix array compression technique that builds on well-known regularity properties that show up in suffix arrays when the text they index is compressible.
- With this compression we reduce the suffix array to 20–70% of its original size, depending on its compressibility.
- We prove that the compression ratio we achieve depends on the k-th order empirical entropy of T ($H_k \log(1/H_k)$)

Results obtained

- A new suffix array compression technique that builds on well-known regularity properties that show up in suffix arrays when the text they index is compressible.
- With this compression we reduce the suffix array to 20–70% of its original size, depending on its compressibility.
- We prove that the compression ratio we achieve depends on the k-th order empirical entropy of T ($H_k \log(1/H_k)$)

Results obtained

- We can obtain a cell from the compressed suffix array (i.e., locate) only 2–20 times slower than the original *SA*.
- The access pattern is local, which makes the scheme suitable for secondary memory

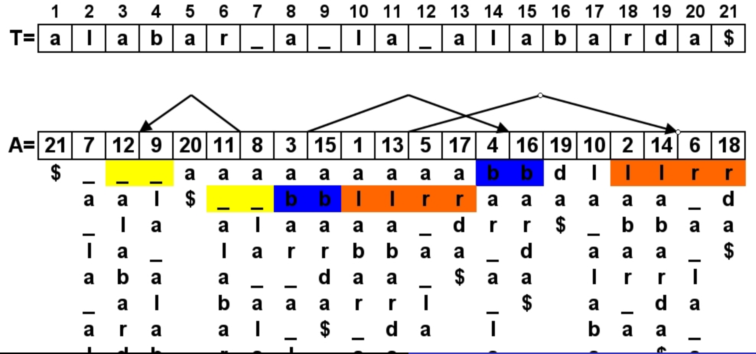
Results obtained

- We can obtain a cell from the compressed suffix array (i.e., locate) only 2–20 times slower than the original *SA*.
- The access pattern is local, which makes the scheme suitable for secondary memory

Outline

Exposing the runs

- The suffix array A can be partitioned into *runs*. A run is a maximal segment that appears repeated (shifted by 1) elsewhere, ie, $A[j+s] = A[i+s] + 1$ for $0 \leq s \leq \ell$. The number of such runs is at most $nH_k + \sigma^k$ for any k.



Exposing the runs

- We convert these runs into true repetitions, representing A in differential form: $A'[i] = A[i] - A[i-1]$ if $i > 1$. So $A'[j+s] = A'[i+s]$ for $1 \leq s \leq \ell$.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
T=	a	l	a	b	a	r	_	a	_	l	a	_	a	l	a	b	a	r	d	a	$
A=	21	7	12	9	20	11	8	3	15	1	13	5	17	4	16	19	10	2	14	6	18
A'=		-14	5	-3	11	-9	-3	-5	12	-14	12	-8	12	-13	12	3	-9	-8	12	-8	12

Outline

Re-Pair

Re-Pair is a dictionary-based compression method:

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

Pairs **Rules**

-8	12	3 times
12	-8	2 times

C=	-14	5	-3	11	-9	-3	-5	12	-14	12	13	-13	12	3	-9	13	13

Pairs **Rules**

13 → -8 12

Re-Pair

- The result of the compression is the table of rules (call it R) plus the sequence of (original and new) symbols into which A' has been compressed (call it C).
- Any portion of C can be easily decompressed in optimal time and fast in practice, by adding some absolute samples.
- We can limit the size of any compressed symbol in C.
- We can limit the size of the dictionary.

Re-Pair

- The result of the compression is the table of rules (call it R) plus the sequence of (original and new) symbols into which A' has been compressed (call it C).
- Any portion of C can be easily decompressed in optimal time and fast in practice, by adding some absolute samples.
- We can limit the size of any compressed symbol in C.
- We can limit the size of the dictionary.

Re-Pair

- The result of the compression is the table of rules (call it R) plus the sequence of (original and new) symbols into which A' has been compressed (call it C).
- Any portion of C can be easily decompressed in optimal time and fast in practice, by adding some absolute samples.
- We can limit the size of any compressed symbol in C.
- We can limit the size of the dictionary.

Re-Pair

- The result of the compression is the table of rules (call it R) plus the sequence of (original and new) symbols into which A' has been compressed (call it C).
- Any portion of C can be easily decompressed in optimal time and fast in practice, by adding some absolute samples.
- We can limit the size of any compressed symbol in C.
- We can limit the size of the dictionary.

Outline

Compressing Repair Rules

- We introduce a novel technique to compress the Re-Pair dictionary, which might be of independent interest.
- In principle storing each rule requires 2 integers.
- We design a technique to save space in the dictionary R.

Rules

A	→ a	b
B	→ a	A
C	→ A	d
D	→ B	c
E	→ a	D

Rules

A	1
B	4
C	7
D	10
E	13

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
B=	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0
S=		a	b		a	1		1	d		4	c		a	10

Compressing Repair Rules

- We introduce a novel technique to compress the Re-Pair dictionary, which might be of independent interest.
- In principle storing each rule requires 2 integers.
- We design a technique to save space in the dictionary *R*.

Rules

A	→	a	b
B	→	a	A
C	→	A	d
D	→	B	c
E	→	a	D

Rules

A	1
B	4
C	7
D	10
E	13

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
B=	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0
S=		a	b		a	1		1	d		4	c		a	10

Compressing Repair Rules

- We introduce a novel technique to compress the Re-Pair dictionary, which might be of independent interest.
- In principle storing each rule requires 2 integers.
- We design a technique to save space in the dictionary *R*.

Rules

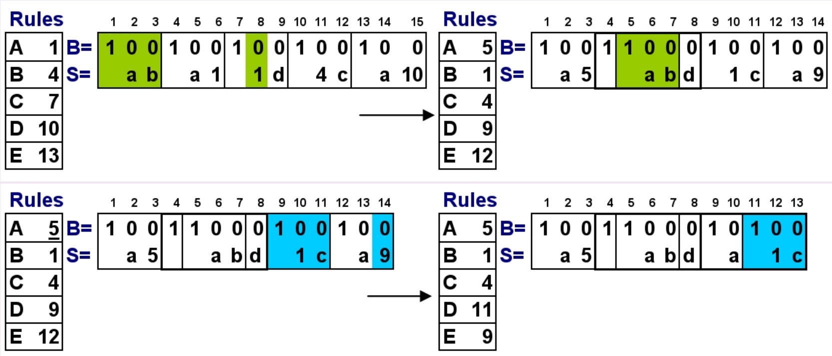
A	→	a	b
B	→	a	A
C	→	A	d
D	→	B	c
E	→	a	D

Rules

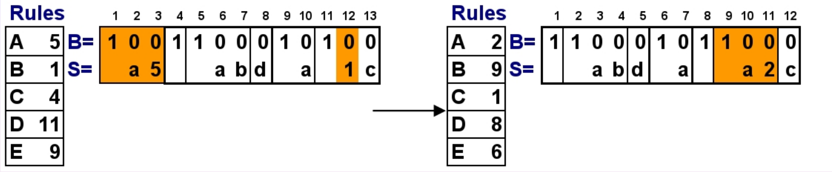
A	1
B	4
C	7
D	10
E	13

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
B=	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0
S=		a	b		a	1		1	d		4	c		a	10

Compressing Repair Rules



Compressing Repair Rules



- We gain almost one integer per rule, if the rule is used by any other rule.
- We design an algorithm to carry out this compression in time $O(|R|)$.
- Larsson and Moffat use different compression methods for Re-Pair dictionary. Ours still permits accessing it at random without decompressing it.

Outline

Faster compression

- The original Re-Pair algorithm runs in $O(n)$ time, but requires $5n$ integers of extra space.
- We design a version of Re-Pair that requires less than $2n$ integers, but it is rather slow.
- Alternatively we show how to use specific properties of suffix arrays to obtain much faster compression losing only 1%–14% of compression.
- The idea to replace the original algorithm to select pairs by another guided by the function Ψ.

Faster compression

- The original Re-Pair algorithm runs in $O(n)$ time, but requires $5n$ integers of extra space.
- We design a version of Re-Pair that requires less than $2n$ integers, but it is rather slow.
- Alternatively we show how to use specific properties of suffix arrays to obtain much faster compression losing only 1%–14% of compression.
- The idea to replace the original algorithm to select pairs by another guided by the function Ψ.

Faster compression

- The original Re-Pair algorithm runs in $O(n)$ time, but requires $5n$ integers of extra space.
- We design a version of Re-Pair that requires less than $2n$ integers, but it is rather slow.
- Alternatively we show how to use specific properties of suffix arrays to obtain much faster compression losing only 1%–14% of compression.
- The idea to replace the original algorithm to select pairs by another guided by the function Ψ.

Faster compression

- The original Re-Pair algorithm runs in $O(n)$ time, but requires $5n$ integers of extra space.
- We design a version of Re-Pair that requires less than $2n$ integers, but it is rather slow.
- Alternatively we show how to use specific properties of suffix arrays to obtain much faster compression losing only 1%–14% of compression.
- The idea to replace the original algorithm to select pairs by another guided by the function Ψ.

Faster compression

- We note that the function Ψ ($A[\Psi(i)] = A[i] + 1$), which is similar to the suffix links of the suffix tree, can be used to obtain a much faster compression algorithm.
- We navigate A' using Ψ, adding symbols to the dictionary and replacing the repeated pairs.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
A'=		-14	5	-3	11	-9	-3	-5	12	-14	12	-8	12	-13	12	3	-9	-8	12	-8	12
Psi=	10	7	11	17	1	3	4	14	15	18	19	20	21	12	13	5	6	8	9	2	16
A=	21	7	12	9	20	11	8	3	15	1	13	5	17	4	16	19	10	<u>2</u>	14	6	18

Faster compression

- We note that the function Ψ ($A[\Psi(i)] = A[i] + 1$), which is similar to the suffix links of the suffix tree, can be used to obtain a much faster compression algorithm.
- We navigate A' using Ψ, adding symbols to the dictionary and replacing the repeated pairs.

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

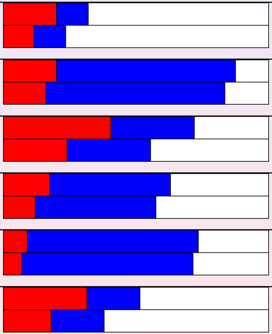
Outline

Compression performance

Collection, Size (MB)	Re-Pair Compression	Re-Pair Time (s)	Re-Pair Ψ Time (s)
xml, 100		25986	260
dna, 100		11150	546
english, 100		93421	485
pitches, 50		15371	180
proteins, 100		3143	641
sources, 100		106173	377

Compression performance

Collection	Re-Pair Compression
xml	
dna	
english	
pitches	
proteins	
sources	



Outline

Attached to a compressed self-index

- This structure can be used attached to a compressed self-index. The self-index identifies the segment of the (virtual) suffix array where the occurrences lie.
- In this case our compressed *SA* is the location mechanism of a new self-index.
- We compare the combination AF-FMI + our suffix array against existing self-indexes, giving them all the same space to operate.

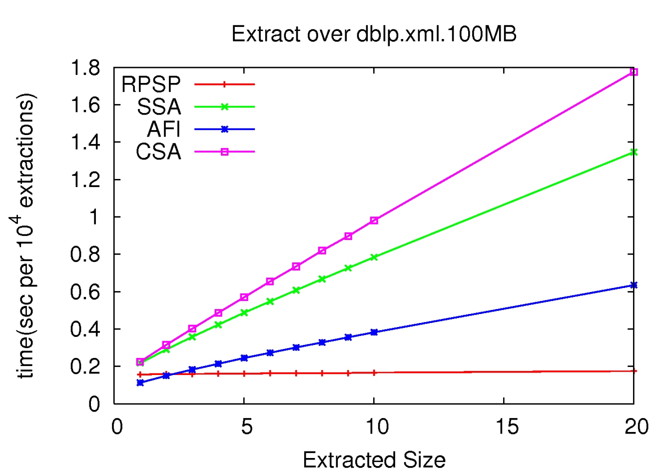
Attached to a compressed self-index

- This structure can be used attached to a compressed self-index. The self-index identifies the segment of the (virtual) suffix array where the occurrences lie.
- In this case our compressed *SA* is the location mechanism of a new self-index.
- We compare the combination AF-FMI + our suffix array against existing self-indexes, giving them all the same space to operate.

Attached to a compressed self-index

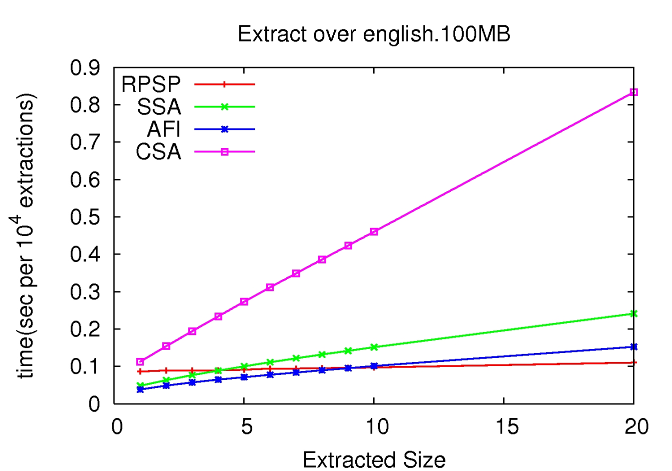
- This structure can be used attached to a compressed self-index. The self-index identifies the segment of the (virtual) suffix array where the occurrences lie.
- In this case our compressed *SA* is the location mechanism of a new self-index.
- We compare the combination AF-FMI + our suffix array against existing self-indexes, giving them all the same space to operate.

Decompression performance



Time to extract positions from SA.

Decompression performance



Time to extract positions from SA.

Outline

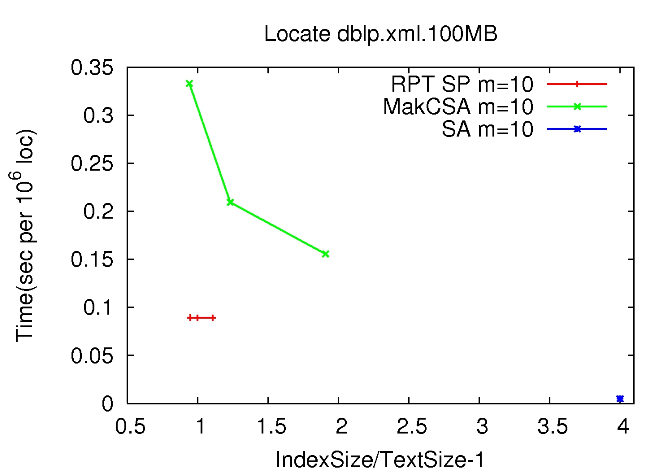
Like a full-text index

- A simpler way to use our structure is like a replacement of the classical suffix array. We use the absolute samples to boost the binary searching.
- We compare against Mäkinen's Compact Suffix Array, which is similar in spirit (compressed SA separately from text).

Like a full-text index

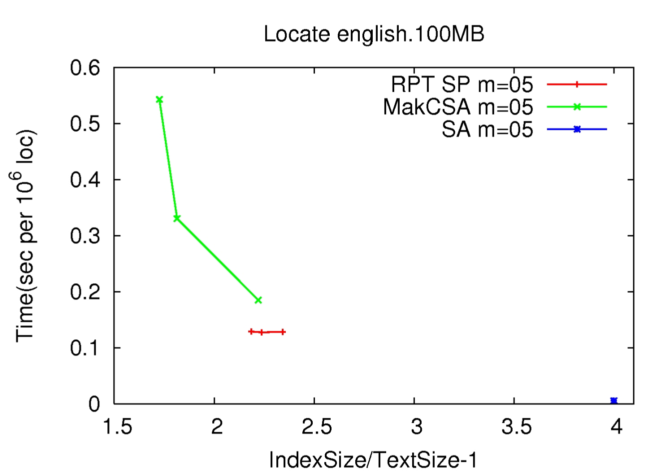
- A simpler way to use our structure is like a replacement of the classical suffix array. We use the absolute samples to boost the binary searching.
- We compare against Mäkinen's Compact Suffix Array, which is similar in spirit (compressed SA separately from text).

Like a full-text index



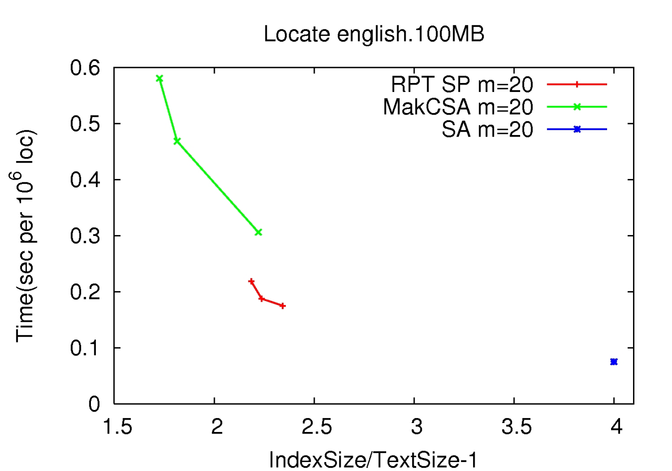
Time to binary search and locate the occ, simulating a classical SA.

Like a full-text index



Time to binary search and locate the occ, simulating a classical SA.

Like a full-text index

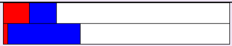
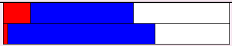


Time to binary search and locate the occ, simulating a classical SA.

Outline

Secondary memory

Thanks to its local decompression properties, on average, if the compression ratio is $0 \leq c \leq 1$, we perform $\lceil \frac{c \cdot occ}{B} \rceil$ disk accesses for locating the *occ* occurrences, being B the disk block size measured in integers.

Collection	Compression with dictionary of 2%
xml	
dna	
english	
pitches	
proteins	
sources	

Outline

1. Motivation
   - Results obtained
2. Compressing the Suffix Array
   - Exposing the runs
   - Re-Pair
   - Compressing Repair Rules
   - Faster compression
3. Experimental Results
   - Compression performance
   - Attached to a compressed self-index
   - Like a full-text index
   - Secondary memory
4. Conclusions and Future Work
   - Conclusions
   - Future Work

Conclusions

- New method to compress a suffix array, permitting fast local decompression.
- New self-index with a much faster locate.
- A viable alternative to classical suffix arrays.
- A secondary-memory version.
- Improvements to the original Re-Pair method.

Conclusions

- New method to compress a suffix array, permitting fast local decompression.
- New self-index with a much faster locate.
- A viable alternative to classical suffix arrays.
- A secondary-memory version.
- Improvements to the original Re-Pair method.

Conclusions

- New method to compress a suffix array, permitting fast local decompression.
- New self-index with a much faster locate.
- A viable alternative to classical suffix arrays.
- A secondary-memory version.
- Improvements to the original Re-Pair method.

Conclusions

- New method to compress a suffix array, permitting fast local decompression.
- New self-index with a much faster locate.
- A viable alternative to classical suffix arrays.
- A secondary-memory version.
- Improvements to the original Re-Pair method.

Conclusions

- New method to compress a suffix array, permitting fast local decompression.
- New self-index with a much faster locate.
- A viable alternative to classical suffix arrays.
- A secondary-memory version.
- Improvements to the original Re-Pair method.

Outline

1. Motivation
   - Results obtained
2. Compressing the Suffix Array
   - Exposing the runs
   - Re-Pair
   - Compressing Repair Rules
   - Faster compression
3. Experimental Results
   - Compression performance
   - Attached to a compressed self-index
   - Like a full-text index
   - Secondary memory
4. Conclusions and Future Work
   - Conclusions
   - Future Work

Future work

- Improve construction time without worsening the compression ratios achieved.
- Improve and implement the secondary memory index, which is right now a theoretical proposal.
- Study the improvements on Re-Pair on their own value.

Future work

- Improve construction time without worsening the compression ratios achieved.
- Improve and implement the secondary memory index, which is right now a theoretical proposal.
- Study the improvements on Re-Pair on their own value.

Future work

- Improve construction time without worsening the compression ratios achieved.
- Improve and implement the secondary memory index, which is right now a theoretical proposal.
- Study the improvements on Re-Pair on their own value.