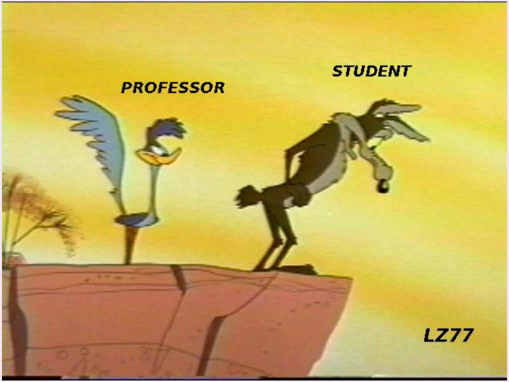
Indexing LZ77: The Next Step in Self-Indexing



Gonzalo Navarro
Department of Computer Science, University of Chile
gnavarro@dcc.uchile.cl

Part I: Why Jumping off the Cliff

Part I: Why Jumping off the Cliff

The Past Century

Self-Indexing: The Dawn of a New Era

A New Challenge: Fully Compressed Self-Indexes

Part I: Why Jumping off the Cliff

The Past Century

Self-Indexing: The Dawn of a New Era

A New Challenge: Fully Compressed Self-Indexes

Part II: Lempel-Ziv Self-Indexing

Current Lempel-Ziv Indexes

A LZ77 Self-Index

Conclusions

Part II: Lempel-Ziv Self-Indexing

Current Lempel-Ziv Indexes

A LZ77 Self-Index

Conclusions

Part II: Lempel-Ziv Self-Indexing

Part I: Why Jumping off the Cliff

The Past Century

Self-Indexing: The Dawn of a New Era

A New Challenge: Fully Compressed Self-Indexes

In the past century...

- Inverted indexes were the only serious solution for indexing large text collections.
- They even achieved index and text compression, without ruining good I/O performance.
- They were (and still are) the best developed solution for the problem.
- BUT... they work only for natural language texts.

In the past century...

- Applications such as
  - Computational biology
  - Music and multimedia processing
  - Software repositories
  - Text retrieval on Chinese and other oriental languages
  - ... and even some kinds of text retrieval on natural language!

  were not included in this framework.
- The only way to deal with those sequences was to treat them as strings.

In the past century...

- For example, the Human Genome, with 3G bases, easily fits in a 1 GB memory.
- But its suffix array requires 12 GB... and its suffix tree more than 30 GB!
- One can use secondary storage, but still this is much slower.
- In practice, usage of these structures was confined to handle not so large texts...
- ... where at least the simple search problem could be reasonably handled by sequential scanning!

Part I: Why Jumping off the Cliff

The Past Century

Self-Indexing: The Dawn of a New Era

A New Challenge: Fully Compressed Self-Indexes

Self-Indexing: The Dawn of a New Era

- In year 2000, several researchers simultaneously figured out how to compress suffix arrays.
- Initially, the idea was to provide a compressed data structure that replaced the suffix array [Grossi & Vitter].
- But soon it was realized that a more ambitious goal, dubbed self-indexing, was achievable:
  - Take space proportional to the compressed text.
  - Be able to reproduce any text substring (hence replacing the text).
  - Provide fast searching on the text (hence incorporating an index within the same space).
- The most famous self-index families appeared in year 2000, the Compressed Suffix Array [Sadakane] and the FM-index [Ferragina & Manzini].

Self-Indexing: The Dawn of a New Era

- A lot of research on these self-indexes has been carried out in this decade. Today, the best representatives offer:
  - Space close to the k-th order entropy of T, $H_k(T) + o(|T|)$. This is in practice as little as 30% of an English text.
  - Counting time $O(m(1 + \frac{\log \sigma}{\log \log n}))$ in theory, and very competitive with plain suffix arrays in practice, 1 Mchar/sec.
  - Locating time $O(log^{1+\epsilon} n)$ per occurrence in theory, and decent in practice, 100 Kocc/sec (yet this is much slower than suffix arrays — others get much closer but are not that small).
  - Extracting a text of length ℓ in $O(\log^{1+\epsilon} n + \ell(1 + \frac{\log \sigma}{\log \log n}))$ in theory, and decent in practice, 1 Mchar/sec.

Self-Indexing: The Dawn of a New Era

- Of course there are still many challenges ahead, some of these partially solved and others not solved at all:
  - How to manage them in secondary memory, when even compressed they do not fit in RAM.
  - How to build them within a space close to their final compressed representation.
  - How to handle updates to the text collection.
  - How to provide more powerful searches.
- But the solutions of most of those challenges are under way, and one can be in general extremely satisfied with, and optimistic about, this technology.

Part I: Why Jumping off the Cliff

A New Challenge: Full Compression

- But there is one further challenge that may hit a fundamental limit of this technology in its current form.
- It is about a compressibility measure many were happy with in the beginning:

$$nH_k(T) \;+\; \frac{n \log \sigma \log \log n}{\log n}$$

- Note it has a sublinear term that is not compressible.
- Note that the compressible part refers to k-th order entropy, which is far from capturing all the relevant sources of compressibility that arise in applications.

A New Challenge: Full Compression

- In particular, applications handling very repetitive collections, such as
  - Databases of genomes and proteins.
  - Code repositories containing multiple versions.
  - Temporal textual databases containing versions of documents.

  do not benefit from the H_k model.
- Recall the empirical entropy definition (similar to the classical one but using T itself as the model)

$$nH_k(T) \;=\; \sum_{i=k}^{n} \log \frac{occ(T, t_{i-k} \ldots t_{i-1})}{occ(T, t_{i-k} \ldots t_i)}$$

- It holds $H_k(TT) \approx H_k(T)$, thus H_k is totally insensitive to repetitions that are farther than k symbols in the past.

A New Challenge: Full Compression

Application Scenario: Computational Biology

- Sequencing genomes is becoming cheap and fast.
- We are not far from the day where we will have databases of thousands or millions of genomes.
- The applications of such a database are unimaginable, BUT...
- 1 million uncompressed genomes $\Longrightarrow$ about 3 petabytes
- a classical suffix tree $\Longrightarrow$ 30 petabytes
- compressed with current self-indexes $\Longrightarrow$ 750 terabytes
- just the sublinear part we mentioned $\Longrightarrow$ 200 terabytes
- Overall, the best we can do requires close to 1 petabyte.

A New Challenge: Full Compression

Application Scenario: Computational Biology

- However, those genomes may be up to 99.9% identical.
- This means (very roughly) that 99.9% of the substrings of one genome can be found in another genome.
- If we were able of exploiting these repetitions, our petabyte would become an inoffensive terabyte.
- However, the H_k measure is totally unable of spotting these regularities.

A New Challenge: Full Compression

Application Scenario: Computational Biology

- With Sirén, Välimäki and Mäkinen we studied another compressibility measure: the number of runs in Ψ.
- We also aimed at largely reducing the uncompressible part.
- This turned to be more sensitive to large repetitions, and even better than LZ78.
- However, we found that the approach was inferior to LZ77, both in theory and in practice.
- In theory, a single difference can produce $\sqrt{n}$ new runs in Ψ, but only one new phrase in LZ77.
- In practice, p7zip compressed our genomes 10 times better than our indexes.

A New Challenge: Full Compression

- We can call our improved index fully compressed, that is, with no or very mild incompressible term in the space.
- This is a first necessary step towards handling very repetitive collections.
- We expect that the full-compression concept will spread in self-indexing in the next years.
- However, the index does not achieve space linear in the number of differences between the texts, only LZ77 compression achieved this.
- This seems to be essential to achieve an order of magnitud less space.

A New Challenge: Full Compression

- However, LZ77 is a compression method, not a self-index.
- We are thus faced to the challenge of building a text index that:
  - Is a self-index.
  - Is fully compressed.
  - If the collection can be split into s pieces, so that each piece appears somewhere in previous text, the index takes space proportional to s.
- Such a kind of index does not exist today.

Part II: Lempel-Ziv Self-Indexing

Current Lempel-Ziv Indexes

A LZ77 Self-Index

Conclusions

Part II: Lempel-Ziv Self-Indexing

Current Lempel-Ziv Indexes

A LZ77 Self-Index

Conclusions

Current Lempel-Ziv Indexes

- LZ76, LZ77, LZ78... compressors converge to nH_k, but slowly: $k = o(\log_\sigma n)$ for the extra terms to be $o(n \log \sigma)$.
- On the other hand, they can be break the nH_k bound by far.
- For typical texts, they are indeed not the best, but on repetitive texts they could be much better.
- Interestingly, Lempel-Ziv indexes predate other compressed text indexes.
- The *sparse suffix tree* [Kärkkäinen 1996] indexed only LZ77(-like) phrase beginnings, achieving $O(nH_k) + |T|$.
- It has been the first index achieving space proportional of the k-th order entropy, yet it was not a self-index.
- It was able to locate each occurrence in $O(\log n)$ time after an $O(m^2 + m \log n)$ initial cost. No counting is supported.

Current Lempel-Ziv Indexes

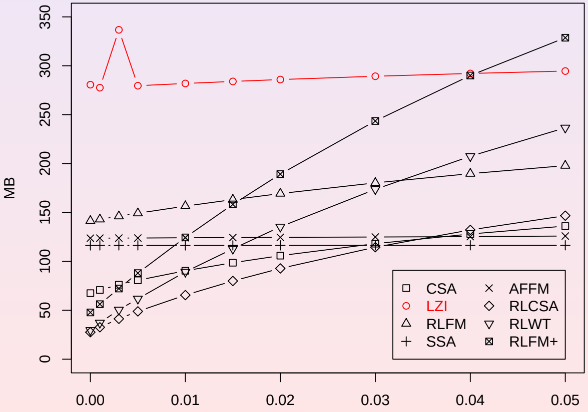
- Several self-indexes followed, building on Kärkkäinen's basic LZ-index design.
  - [Ferragina and Manzini 2001] use LZ78 parsing combined with an FM-index to get $O(nH_k \log^\gamma n)$ bits of space and $O(m(1 + \frac{\log \sigma}{\log \log n}) + occ)$ locating time.
  - [N. 2002] uses LZ78 parsing to get $4nH_k(1 + o(1))$ bits and $O(m^3 \log \sigma + (m + occ) \log n)$ time.
  - [Russo and Oliveira 2006] use a modified LZ78 parsing (maximal parsing) to achieve $5nH_k(1 + o(1))$ bits and $O((m + occ) \log n)$ time.
  - [Arroyuelo and N. 2006] use LZ78 parsing to achieve $(2 + \epsilon)nH_k(1 + o(1))$ bits and $O(m^2 + (m + occ) \log n)$ time.
  - [Arroyuelo and N. 2007] use LZ78 parsing plus an FM-index to achieve $(3 + \epsilon)nH_k(1 + o(1))$ bits and $O((m + occ) \log n)$ time.

Current Lempel-Ziv Indexes

- Notice some important things:
  - These are fully-compressed indexes, as they have no incompressible extra space complexity terms.
  - Although they built on the sparse suffix tree idea, no one ever again tried to build on LZ77, but on LZ78.
  - They cannot count efficiently (unless you add a compressed suffix array of some kind).

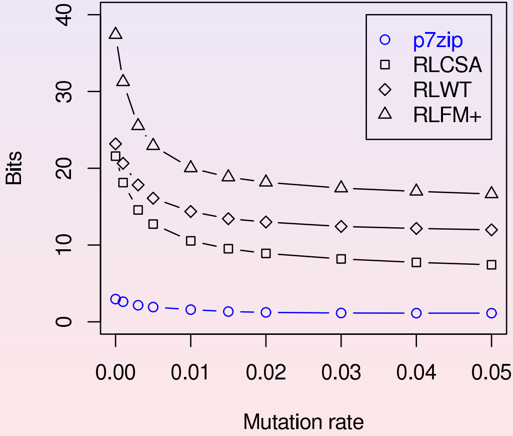
Current Lempel-Ziv Indexes

- Question: Why then trying to index LZ77, if LZ78 is easier to handle?
- Answer: LZ78 is too weak to profit from highly repetitive texts.



Current Lempel-Ziv Indexes

- Instead, LZ77 is extremely promising:



Part II: Lempel-Ziv Self-Indexing

A LZ77 Self-Index

- We came back to the original LZ77-based index and "modernized" it.
- We used compact data structures to achieve the minimum space we could.
- We are trying to convert it into a self-index.
- I will show you now what we have and where are we stuck.
- This is joint work with Diego Arroyuelo, Veli Mäkinen, Luis Russo, ... and hopefully anyone else able of getting us off this mess!

A LZ77 Self-Index

- From now on let $T[1, u]$ be the text, partitioned into n LZ77 phrases.
- We call primary occurrences those that span more than one phrase.
- We call secondary occurrences those included in a phrase.
- We find first the primary and from those the secondary occurrences.

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

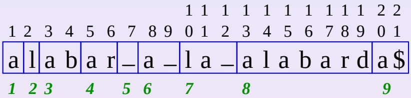
a | l | a b | a r | _ | a _ | l a _ | a l a b a r d | a $

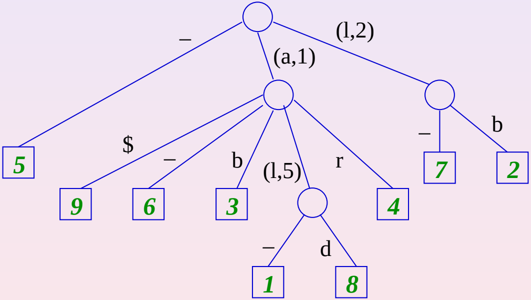
(0,0,a) (0,0,l) (1,1,b) (1,1,r) (0,0,_) (1,1,_) (2,2,−)(1,6,d) (1,1,$)

A LZ77 Self-Index

- A sparse suffix tree indexes phrase beginnings, n leaves.
- It is represented with at most
  - $4n + o(n)$ bits for parentheses (DFUDS representation)
  - $2n \log \sigma$ bits for letters
  - $n \log n$ bits for the phrase identifiers
- Skips are not stored (could require too much space), we see later how to recover them.
- Allows navigation to child labeled x in constant time, apart from several tree operations.

A LZ77 Self-Index





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

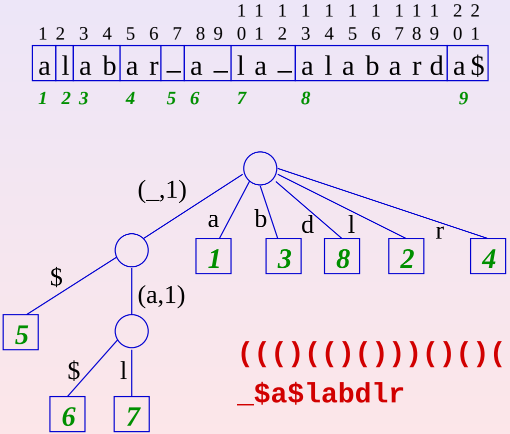
_a$_bl_drl_b

5,9,6,3,1,8,4,7,2

A LZ77 Self-Index

- A reverse trie indexes reversed phrases but the last, $n-1$ leaves.
- It is represented with at most
  - $4n + o(n)$ bits for parentheses (DFUDS representation)
  - $2n \log \sigma$ bits for letters
- Skips, again, are not stored.
- Allows navigation to child labeled x in constant time, apart from several tree operations.

A LZ77 Self-Index



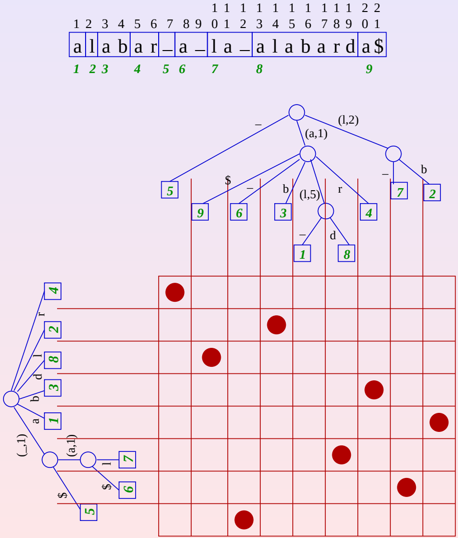
((()(()()))()()()()())

_$a$labdlr

A LZ77 Self-Index

- A range structure connects both trees: the suffix starting at phrase k with the reverse phrase $k-1$.
- Requires $n \log n + O(n \log\log n)$ bits of space.
- Allows range counting in $O(\log n)$ time and reporting each point in $O(\log n)$ time as well.
- Implemented with a wavelet tree.

A LZ77 Self-Index



A LZ77 Self-Index

- Partition $P[1,m]$ into $P[1,i]$ and $P[i+1,m]$ for each $1 \leq i < m$.
- Search the sparse suffix tree for $P[i+1,m]$ and the reverse trie for $(P[1,i])^{rev}$.
- The search gives two preorder intervals $[r_1, r_2]$ and $[l_1, l_2]$, respectively.
- Extract the points in the range data structure to get all the primary occurrences (phrase numbers, using the identifiers we store).

A LZ77 Self-Index

- Tries can be traversed in constant time per symbol using DFUDS.
- But we miss skip information: go to leftmost and rightmost leaves, extract symbols from there until they differ, and this gives the skip.
- Assuming that can be done in constant time per symbol, total search time is $O(m^2 + m \log n + occ \cdot \log n)$.
- We obtain the phrase numbers and offsets where each occurrence starts.
- We now introduce other data structures to convert these into text positions and also solve secondary occurrences.

A LZ77 Self-Index

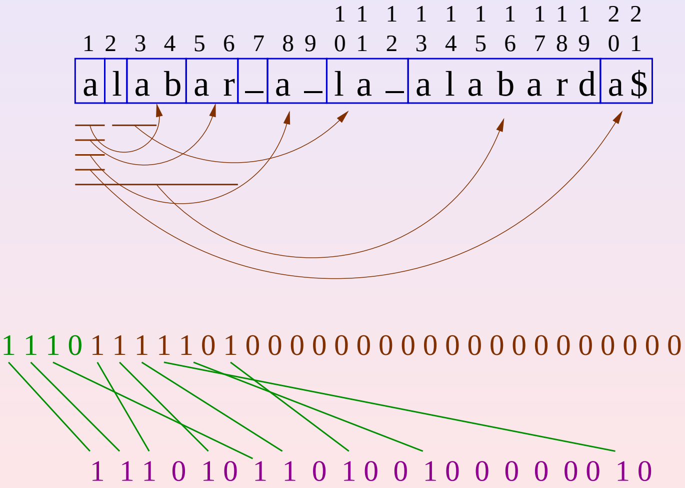
- With $n \log \frac{u}{n} + O(n \log\log \frac{u}{n})$ extra bits we convert phrase id into text position and vice versa in $O(\log n)$ time.
- This is through rank and select operation on the bitmap.
- Thus time for reporting stays $O(\log n)$ per (primary) occurrence.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
a	l	a	b	a	r	_	a	_	l	a	_	a	l	a	b	a	r	d	a	$
1	1	1	0	1	0	1	1	0	1	0	0	1	0	0	0	0	0	0	1	0

A LZ77 Self-Index

- For secondary occurrences we need another bitmap and a permutation π.
- The second bitmap marks beginning of phrase sources with 1s, and change to the next text position with a 0.
- It also provides rank and select in $O(\log n)$ time.
- Sources starting at the same position are ordered from shortest to longest.
- The permutation maps 1s in the bitmap of targets to 1s in the bitmap of sources.
- We add data to compute π^{-1} in $O(\log n)$ time.
- Total space added is
  $n \log \frac{u+n}{n} + O(n \log \log \frac{u+n}{n}) + n \log n + O(n) =$
  $n \log u + O(n \log \log \frac{u}{n})$

A LZ77 Self-Index



A LZ77 Self-Index

- For each primary occurrence found, we find the 0 of its starting position in the bitmap of sources.
- We consider the 1s preceding it backwards, one by one (disregarding 0s).
- We map each such 1 to the target, find out its length, and see if the source covers the primary occurrence.
- If it does, report a secondary occurrence.
- If it does not, stop and consider the next primary occurrence.
- Repeat the process with the secondary occurrences found, until no more occurrences are reported.

A LZ77 Self-Index

- The total time is $O(\log n)$ per occurrence reported.
- But it works only if no source strictly contains another source (strictly from left and right extremes)
- This can be enforced in the parsing as in Kärkkäinen's proposal.
- Our example does not obey this rule! (it is a pure LZ77 parsing).
- There is another proposal by Kärkkäinen that permits LZ77 parsing and uses a more complicated structure for mapping sources to targets.
- We have also considered compact variants of that one, omitted here.

A LZ77 Self-Index

- What is missing is the ability to extract a text substring, both for displaying and for supporting the Patricia tree search.
- We go to the target bitmap, and using rank, find out the phrases to output.
- The last symbol of each phrase is obtained directly (by storing them, $n \log \sigma$ more bits).
- For the other symbols of each involved phrase, use π to find the source positions, obtain the last symbols of the included phrases, and so on until all the symbols are discovered.
- Each step takes $O(\log n)$ time...
- ... but we cannot bound the number of steps to carry out!

A LZ77 Self-Index

- Total space is $2n \log u + n \log n + O(n(\log \sigma + \log \log u))$ bits.
- Total locating time is $O(m^2 + (m + occ) \log u)$...
- ... plus m^2 times the cost to extract a text symbol, which we cannot bound!
- We could store the skips to partially avoid this:
  - $n \log u$ more bits for the sparse suffix tree.
  - $n \log \frac{u}{n}$ more bits for the reverse trie.
  - Make just one final check for any point in the range, for the Patricia search.
  - But this requires extracting m symbols from T.
- A self index needs to extract arbitrary text positions, so this problem is central anyway.

A LZ77 Self-Index

- For example, what about using a denser sampling of letters.
- Build a dependency forest with the u letters of T.
- Ensure every path of length, say, $\log u$, contains a sampled letter.
- Each letter could then be extracted in $O(\log u)$ time.
- But if the tree is a root with $\sqrt{u}$ children, each with a chain of length $\sqrt{u}$, then we need $u/\log u$ letters stored, too much.
- Such can happen, e.g. with text $1\ 12\ 123\ 1234\ 12345 \ldots$.

Part II: Lempel-Ziv Self-Indexing

Current Lempel-Ziv Indexes

A LZ77 Self-Index

Conclusions

Conclusions

- An LZ77-based fully compressed self-index is extremely attractive for highly repetitive text collections.
- We have modernized an old compressed index proposal to convert it into a fully compressed self-index.
- Asymptotically it could be about 1.5 times the size of the file compressed with LZ77 (this is probably a bit optimistic).
- But we are stuck in how to give worst-case time guarantees for text extraction.
- Ideas very welcome!