Relative Compression of Data Structures

Jouni Sirén
with
Djamal Belazzougui, University of Helsinki
Travis Gagie, University of Helsinki
Simon Gog, Karlsruhe Institute of Technology
Giovanni Manzini, University of Eastern Piedmont
Gonzalo Navarro, University of Chile
Simon J. Puglisi, University of Helsinki

1. Algorithms vs. data structures
2. Relative data structures
3. Relative FM-index
4. Relative Lempel-Ziv bitvector

Algorithms

- Most problems in CS (and elsewhere) are algorithmic: given some data, we want to answer some questions about the data.

- Computer scientists like well-defined algorithmic problems.

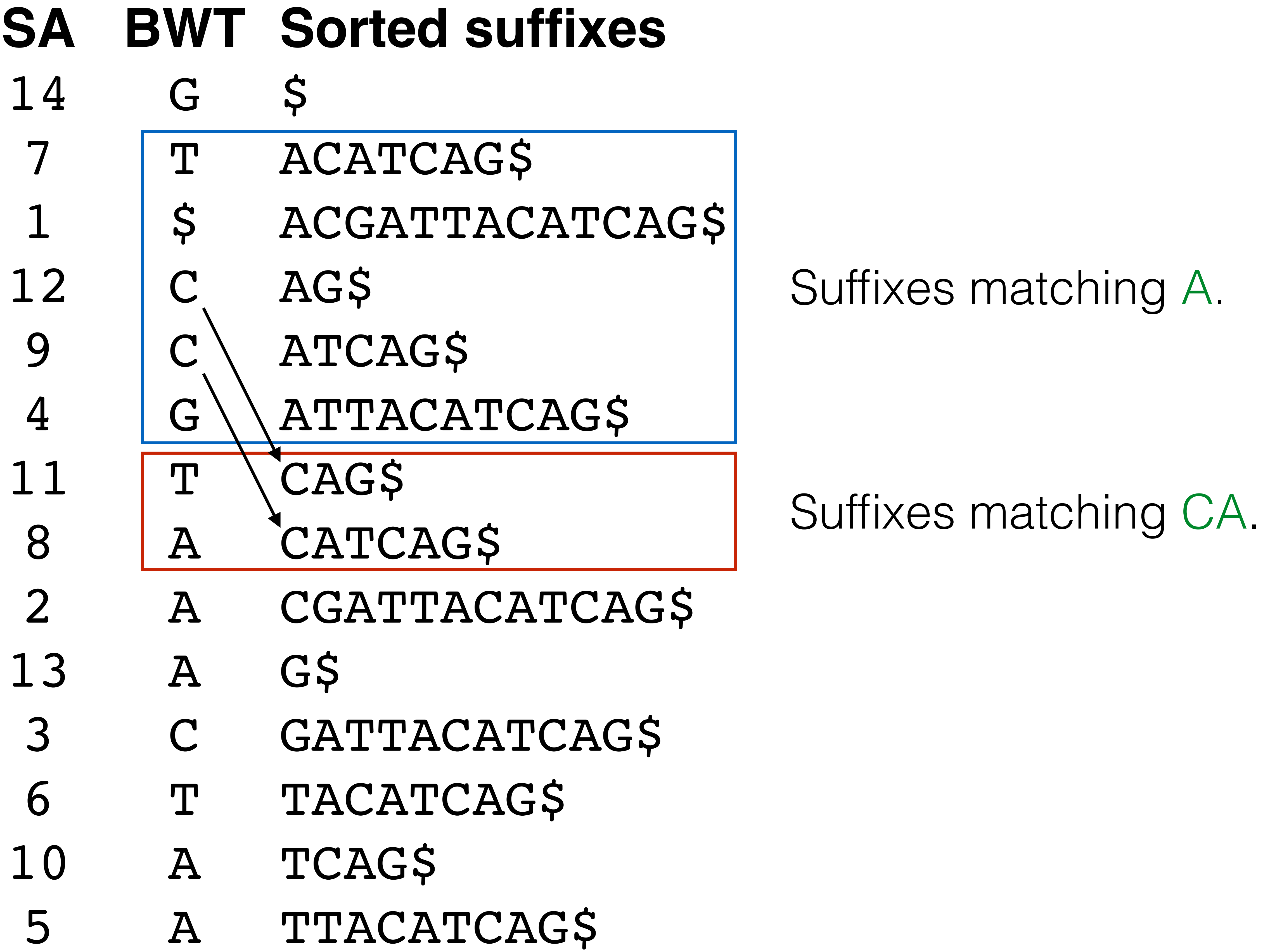
- Solutions can be evaluated by resource usage (e.g. time, space, communication), accuracy, reliability…

Data structures

- Given some data, we want to store the data in a way that makes some queries efficient.

- Computer scientists like to think that data structures implement well-defined interfaces.

- Data structures implementing the same interface are typically compared by query performance and space usage.



SA	**BWT**	**Sorted suffixes**
14	G	$
7	T	ACATCAG$
1	$	ACGATTACATCAG$
12	C	AG$
9	C	ATCAG$
4	G	ATTACATCAG$
11	T	CAG$
8	A	CATCAG$
2	A	CGATTACATCAG$
13	A	G$
3	C	GATTACATCAG$
6	T	TACATCAG$
10	A	TCAG$
5	A	TTACATCAG$

Suffixes matching A.

Suffixes matching CA.

BWT-based indexes

- FM-index (FMI) (Ferragina, Manzini, JACM 2005)
- Compressed suffix array (CSA) (Grossi, Vitter, SIAM J. Comput. 2005)
- The index consists of the BWT, an additional structure to support $\text{rank}_c(\text{BWT}, i)$ queries, and some sampled SA values.
- Rarely the best data structures for a specific problem, but often almost as good.

The original interface

count	$P \longmapsto ep + 1 - sp$
locate	$P \longmapsto SA[sp, ep]$
extract	$(i, j) \longmapsto T[i, j]$

More exact matching

find	$P \longmapsto (sp, ep)$
locate	$i \longmapsto SA[i]$
locate	$(sp, ep) \longmapsto SA[sp, ep]$

People always find new ways of using a data structure. Fixed interfaces rarely capture the full functionality of the structure.

Low-level interface

LF	$(sp_i, ep_i, c) \longmapsto (sp_{i-1}, ep_{i-1})$
LF	$i \longmapsto SA^{-1}[SA[i] - 1]$
Ψ	$i \longmapsto SA^{-1}[SA[i] + 1]$
char	$i \longmapsto T[SA[i]]$

Really low-level interface

rank	$(c, i) \longmapsto rank_c(BWT, i)$
select	$(c, i) \longmapsto select_c(BWT, i)$
access	$i \longmapsto BWT[i]$
next	$(c, i) \longmapsto (\min j \geq i : BWT[j] = c, rank(c, j))$
next	$(c, i, r) \longmapsto (\min j > i : BWT[j] = c, r + 1)$

Compressed data structures

- Provide the functionality of a data structure, while using similar space as a compressed representation of the data.

- Take advantage of the repetitiveness of the data to store it in less space than the information-theoretic minimum.

- Are always trade-offs between space usage and query performance.

Relative data compression

- Individual genomes can be represented by their differences to the reference genome.

- Version control systems store revisions of a document as insertions and deletions to older revisions.

- Relative Lempel-Ziv parsing (Kuruppu et al., SPIRE 2010) represents a string as a concatenation of substrings of another string.

Relative data structures

- Let R and S be similar datasets.
- If we build data structure D for the datasets, D(R) and D(S) should also be similar.
- We can encode D(S) relative to D(R) as D(S | R).
- Given D(R) and D(S | R), we can query D(S) efficiently or decompress it.

Relative data structures	Compressed data structures
Individual data structures for each of the datasets.	A single data structure containing all the datasets.
The encoding of S depends only on S and R.	The encoding of S may depend on all the other datasets.
Compression depends on the similarity between S and R.	Compression may take advantage of the similarities between all the datasets.
Construction for multiple datasets is easy to distribute.	Construction for multiple datasets may require significant resources and specialized algorithms.
Datasets can be added and removed easily.	Insertion and deletion require specialized algorithms.

BWT construction

- Space-efficient merging algorithm (Hon et al., Algorithmica 2007) adds a new sequence S to the BWT of collection C.

- Build BWT(S); search for S in BWT(C) to find the positions for the suffixes of S; merge the BWTs.

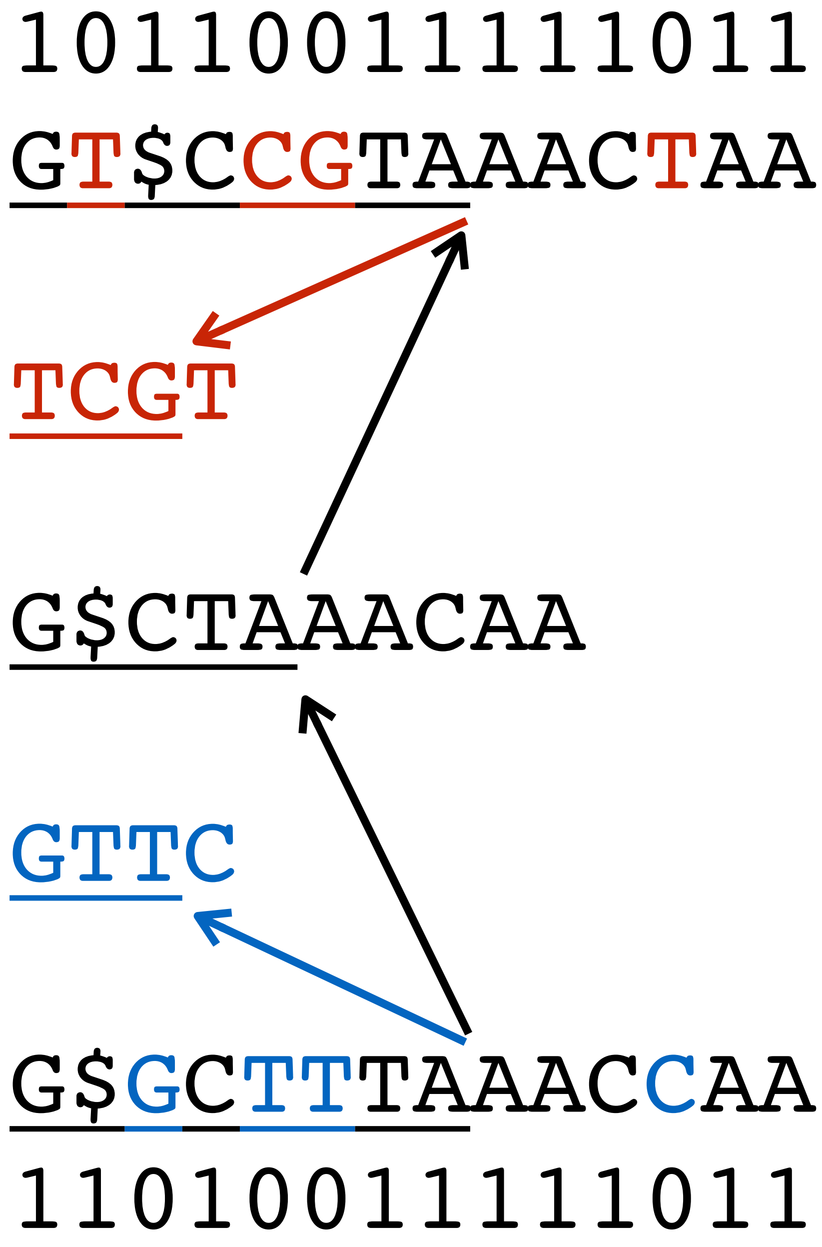
- A similar algorithm can remove sequences from the collection and modify the existing sequences.

- What if we skip the merging step?

ACGATTACATCAG$	**BWT**	**BWT**	**ACGACTATATCAG$**
$	G	G	$
ACATCAG$	T	$	ACGACTATATCAG$
ACGATTACATCAG$	$	G	ACTATATCAG$
AG$	C	C	AG$
ATCAG$	C	T	ATATCAG$
ATTACATCAG$	G	T	ATCAG$
CAG$	T	T	CAG$
CATCAG$	A	A	CGACTATATCAG$
CGATTACATCAG$	A	A	CTATATCAG$
G$	A	A	G$
GATTACATCAG$	C	C	GACTATATCAG$
TACATCAG$	T	C	TATATCAG$
TCAG$	A	A	TATCAG$
TTACATCAG$	A	A	TCAG$

ACGATTACATCAG$	**BWT**	**BWT**	**ACGACTATATCAG$**
$	G	G	$
ACATCAG$	T	$	ACGACTATATCAG$
ACGATTACATCAG$	$	G	ACTATATCAG$
AG$	C	C	AG$
ATCAG$	C	T	ATATCAG$
ATTACATCAG$	G	T	ATCAG$
CAG$	T	T	CAG$
CATCAG$	A	A	CGACTATATCAG$
CGATTACATCAG$	A	A	CTATATCAG$
G$	A	A	G$
GATTACATCAG$	C	C	GACTATATCAG$
TACATCAG$	T	C	TATATCAG$
TCAG$	A	A	TATCAG$
TTACATCAG$	A	A	TCAG$

Computing $\mathrm{rank}_c(i)$



	10110011111011	$\mathrm{select}_1(5) = 8$
BWT(R)	GT$CCGTAAACTAA	$\mathrm{rank}_C(8) = 2$
BWT(R) – LCS	TCGT	$\mathrm{rank}_C(3) = 1$
LCS	G$CTAAACAA	$\mathrm{rank}_C(5) = 1$
BWT(S) – LCS	GTTC	$\mathrm{rank}_C(3) = 0$
BWT(S)	G$GCTTTAAACCAA	$\mathrm{rank}_C(8) = 1$
	11010011111011	$\mathrm{rank}_1(8) = 5$

Relative FM-index

- Sequences BWT(R) – LCS and BWT(S) – LCS with rank support (e.g. as wavelet trees [Grossi et al., SODA 2003]).
- Compressed bitvectors marking LCS in BWT(R) and BWT(S).
- We use SDSL (Gog et al., SEA 2014) in the implementation.

Index construction

- Finding the longest common subsequence (LCS) of two long strings is expensive.

- Myers' algorithm (Algorithmica, 1986) takes $O(nD)$ time, where $nD \approx n^2/50$ for the BWTs of human genomes.

- We approximate the LCS by partitioning the BWTs and using Myers' algorithm for each pair of partitions.

$	G	G	$
ACATCAG$	T	$	ACGACTATATCAG$
ACGATTACATCAG$	$	G	ACTATATCAG$
AG$	C	C	AG$
ATCAG$	C	T	ATATCAG$
ATTACATCAG$	G	T	ATCAG$
CAG$	T	T	CAG$
CATCAG$	A		
CGATTACATCAG$	A	A	CGACTATATCAG$
		A	CTATATCAG$
G$	A	A	G$
GATTACATCAG$	C	C	GACTATATCAG$
TACATCAG$	T	C	TATATCAG$
		A	TATCAG$
TCAG$	A	A	TCAG$
TTACATCAG$	A		

Dataset	LCS	Relative FM-index construction	Plain FM-index	RRR FM-index	Relative FM-index
NA12878 vs. 1000GP reference	2992M 96.65% 98.54%	589 s	1090 MB 3.01 bpc 57.31 s	636 MB 1.76 bpc 325.48 s	218 MB 0.60 bpc 619.81 s
NA12878 (reference without chr Y)	2991M 98.51% 98.51%				181 MB 0.50 bpc

Relative Lempel-Ziv

- Relative Lempel-Ziv (RLZ) parsing (Kuruppu et al., SPIRE 2010) encodes string S as a concatenation of substrings of string R.

- Because pointers are large and the differences between S and R are often single-character substitutions, we add the first mismatching character to the end of each phrase.

- See Ferrada et al., SPIRE 2014, for techniques for supporting fast random access.

Bitvectors

- A bitvector is a binary sequence supporting rank_0, rank_1, select_0, and select_1.
- rank_1 is the basic operation, with $\text{rank}_0(i) = i - \text{rank}_1(i)$.
- select_0 and select_1 either require separate structures or use binary search with rank_1.
- Common encodings include plain, entropy-compressed, sparse/gap-encoded, and run-length encoded.

RLZ bitvector

- Each phrase $P_i = (s_i, l_i, c_i)$ consists of the starting position s_i in R, the length l_i, and the mismatching character c_i.
- We use relative encoding with optional run-length encoding for the starting positions (Ferrada et al., SPIRE 2014).
- A sparse bitvector maps between the positions in S and the phrases they belong in.
- Another sparse bitvector does the same for 1-bits.
- If fast select_0 is needed, a third bitvector is used for 0-bits.

```cpp
uint64_t
RLZVector::rank(uint64_t i) const
{
  if(i >= this->size()) { return this->items(); }

  uint64_t phrase = this->blocks.blockFor(i);
  if(phrase == 0)
  {
    return this->oneBits(this->phrases.decode(0, 0), i);
  }
  uint64_t text_pos = this->blocks.itemsAfter(phrase - 1);

  return this->ones.itemsAfter(phrase - 1) +
    this->oneBits(this->phrases.decode(phrase, text_pos),
                  i - text_pos);
}
```

RLZ parsing

- RLZ parsing uses SA(R) or CSA(reverse(R)).
- The merging algorithm of Hon et al. (Algorithmica, 2007) works here as well.
- With block size $m = 2^{26}$, parsing speed is roughly 1 Mb/s, or 5 Mb/s by reusing a prebuilt index.
- In principle, memory usage is just 8m bytes in addition to the reference.

Bitvector (n = 2	Size	rank	select	select	access
Plain	190 MB 1.49 bpc	52 ns	272 ns	280 ns	15 ns
RRR	137 MB 1.07 bpc	451 ns	784 ns	769 ns	430 ns
RLZ 0.001	6.1 MB 0.05 bpc	510 ns	899 ns	887 ns	255 ns
RLZ 0.003	17 MB 0.13 bpc	639 ns	996 ns	1026 ns	348 ns
RLZ 0.010	57 MB 0.45 bpc	865 ns	1128 ns	1167 ns	447 ns
RLZ 0.030	151 MB 1.18 bpc	1019 ns	1311 ns	1329 ns	541 ns

Conclusions

- Given a collection of similar datasets, relative data compression can be used to store data structures for individual datasets space-efficiently.
- Compared to compressed data structures for the entire collection, relative data structures are larger but easier to handle.
- The relative FM-index stores the symmetric difference between BWT(R) and BWT(S), while the RLZ bitvector represents S as a concatenation of pieces of R.