Data Structures based on Relative Lempel-Ziv parsing

Jouni Sirén, University of Chile
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

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.

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 $rank_0(i) = i - rank_1(i)$.

- $select_0$ and $select_1$ often require separate structures or 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.

```
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)).
- Space-efficient construction algorithms using 1.5n bytes are not that efficient for bitvectors.
- The incremental BWT construction of Hon et al. (Algorithmica, 2007) works better.
- 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 ($n = 2^{30}$)	Size	rank	$select_0$	$select_1$	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

Bitvector (CST for NA12878)	Plain	RRR	Sparse	RLZ
Wavelet tree (Huffman-shaped)	1247.9 MB 3.45 bpc	636 MB 1.76 bpc	1718 MB 4.75 bpc	721 MB 1.99 bpc
Sampled positions (d = 32)	536 MB 1.48 bpc	105 MB 0.29 bpc	90 MB 0.25 bpc	128 MB 0.35 bpc
Balanced parentheses tree	1819 MB 5.03 bpc	1320 MB 3.65 bpc	2634 MB 7.28 bpc	683 MB 1.89 bpc
Sadakane's LCP	1077 MB 2.98 bpc	717 MB 1.98 bpc	1443 MB 3.99 bpc	313 MB 0.86 bpc

Another relative FM-index

- The RLZ bitvector does not work too well in the wavelet tree of a BWT.

- We could try the original FM-index (Ferragina, Manzini, JACM, 2005) instead.

- We generalize the RLZ bitvector for larger alphabets, using a sparse bitvector for each character c to map between the occurrences of c and blocks.

- For DNA sequences, blocks of 2 to 4 phrases work well.

NA12878 (reference without chr Y)	Size	Queries
Plain FM-index	1090 MB 3.01 bpc	60 s
RRR FM-index	636 MB 1.76 bpc	347 s
Relative FM-index	181 MB 0.50 bpc	589 s
FM-index (RLZ bitvector)	721 MB 1.99 bpc	537 s
RLZ FM-index (block size 4)	474 MB 1.31 bpc	506 s

Differential LCP array

- The LCP array stores the lengths of the longest common prefixes of lexicographically adjacent suffixes of a text.

- If the dataset is repetitive, the differential LCP array LCP' (where LCP'[i] = LCP[i] – LCP[i – 1]) is grammar-compressible (Abeliuk et al., Algorithms, 2013).

- This should mean that if R and S are similar, LCP'(S) is RLZ-compressible relative to LCP'(R).

Relative LCP array

- On the first attempt, the RLZ parsing of the differential LCP array of the NA12878 genome took 2.20 bpc: 1.14 bpc for phrase starts, 0.27 bpc for phrase boundaries, and 0.79 bpc for mismatches.

- The mismatches use 26 bits each due to long runs of character N in the genome. This can be reduced to around 8 bits, for a total of roughly 1.65 bpc.

- Random access to LCP can be provided by using the mismatches as absolute samples.

Fischer's CST

- Fischer's compressed suffix tree (TCS, 2009) identifies suffix tree nodes as LCP intervals, using RMQ/PSV/NSV queries over the LCP array.

- We can support this with the relative LCP array by building a RMQ structure over the reference LCP and a balanced tree of minimal LCP values over the RLZ phrases.

- The total size of the relative CST should be at most |CSA| + 2n bits for human genomes.

Conclusions

- Sorting the suffixes in lexicographic order multiplies the differences between two texts, making RLZ less useful for compressing the structures.

- The best use of RLZ in text indexes seems to be compressing the differential LCP array.

- We can build a relative CST by using the relative FM-index and the relative LCP array.