Compressed genomic sequences with fast access

Szymon Grabowski

Institute of Applied Computer Science,
Lodz University of Technology, Poland
sgrabow@kis.p.lodz.pl

August 6, 2016

This lecture was part of the 1st Summer School on Bioinformatics Data Structures, funded by BIRDS project (www.birdsproject.eu)
This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 690941

Big data app domains

Large Hadron Collider produced $\sim$15 PB of data in 2012.

http://home.cern/about/computing:
The Data Centre processes about 1 PB of data every day.

Big data app domains

Large Hadron Collider produced ∼15 PB of data in 2012.

http://home.cern/about/computing:
The Data Centre processes about 1 PB of data every day.

Large Synoptic Survey Telescope, around 2008:
the camera is expected to take over 200,000 pictures (1.28 PB uncompressed) per year (wikipedia).

In its planned 10-year run, the LSST will capture, process and store more than 30 TB of image data each night, yielding a 150 PB database.

Big data app domains

Large Hadron Collider produced ∼15 PB of data in 2012.

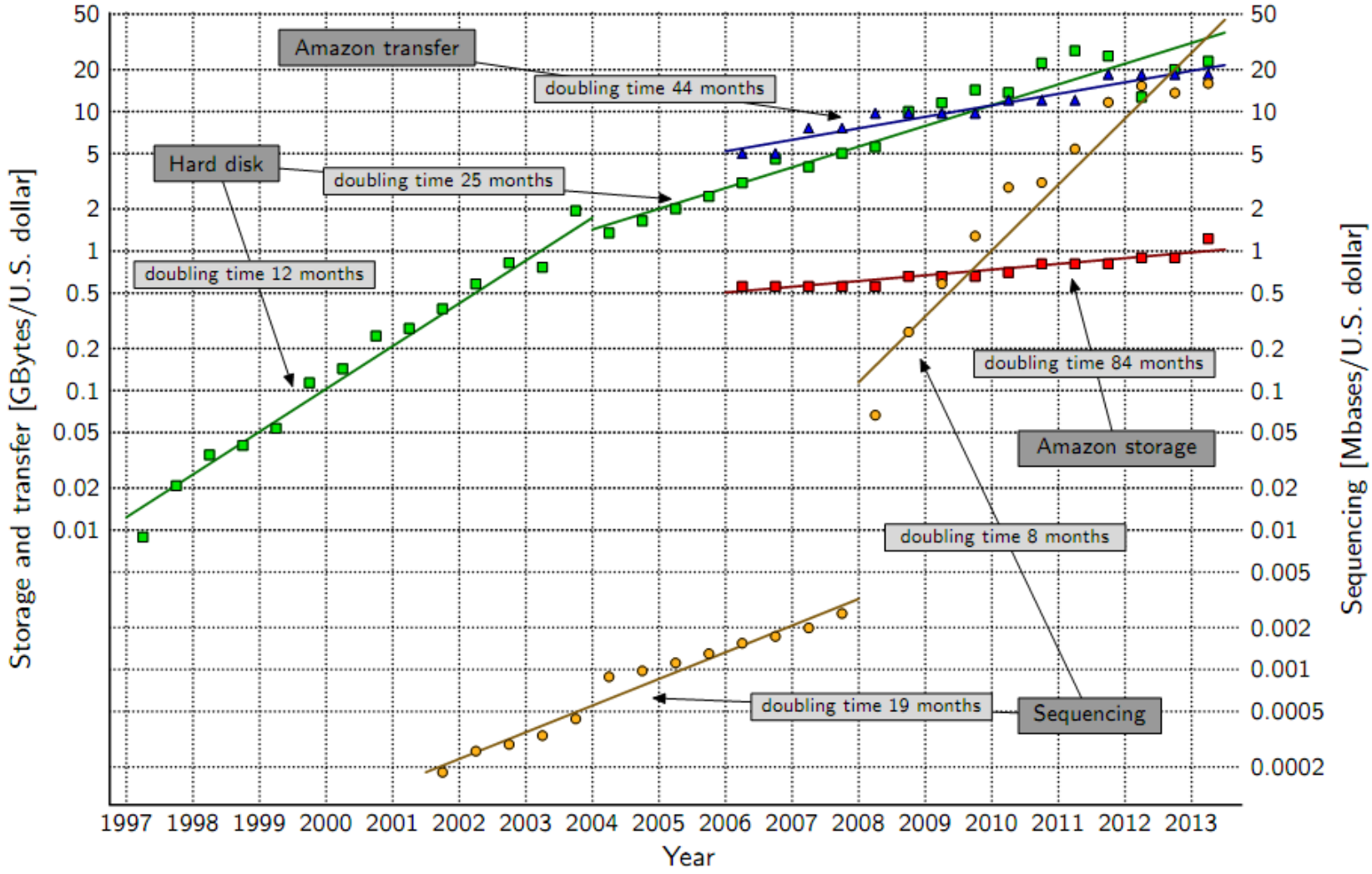
http://home.cern/about/computing:
The Data Centre processes about 1 PB of data every day.

Large Synoptic Survey Telescope, around 2008:
the camera is expected to take over 200,000 pictures (1.28 PB uncompressed) per year (wikipedia).

In its planned 10-year run, the LSST will capture, process and store more than 30 TB of image data each night, yielding a 150 PB database.

The Australian Square Kilometre Array Pathfinder (ASKAP) project currently acquires 7.5 TB/s (less than 1 GB/s stored) of sample image data, projected to increase 100-fold (˜25 ZB per year) by 2025.

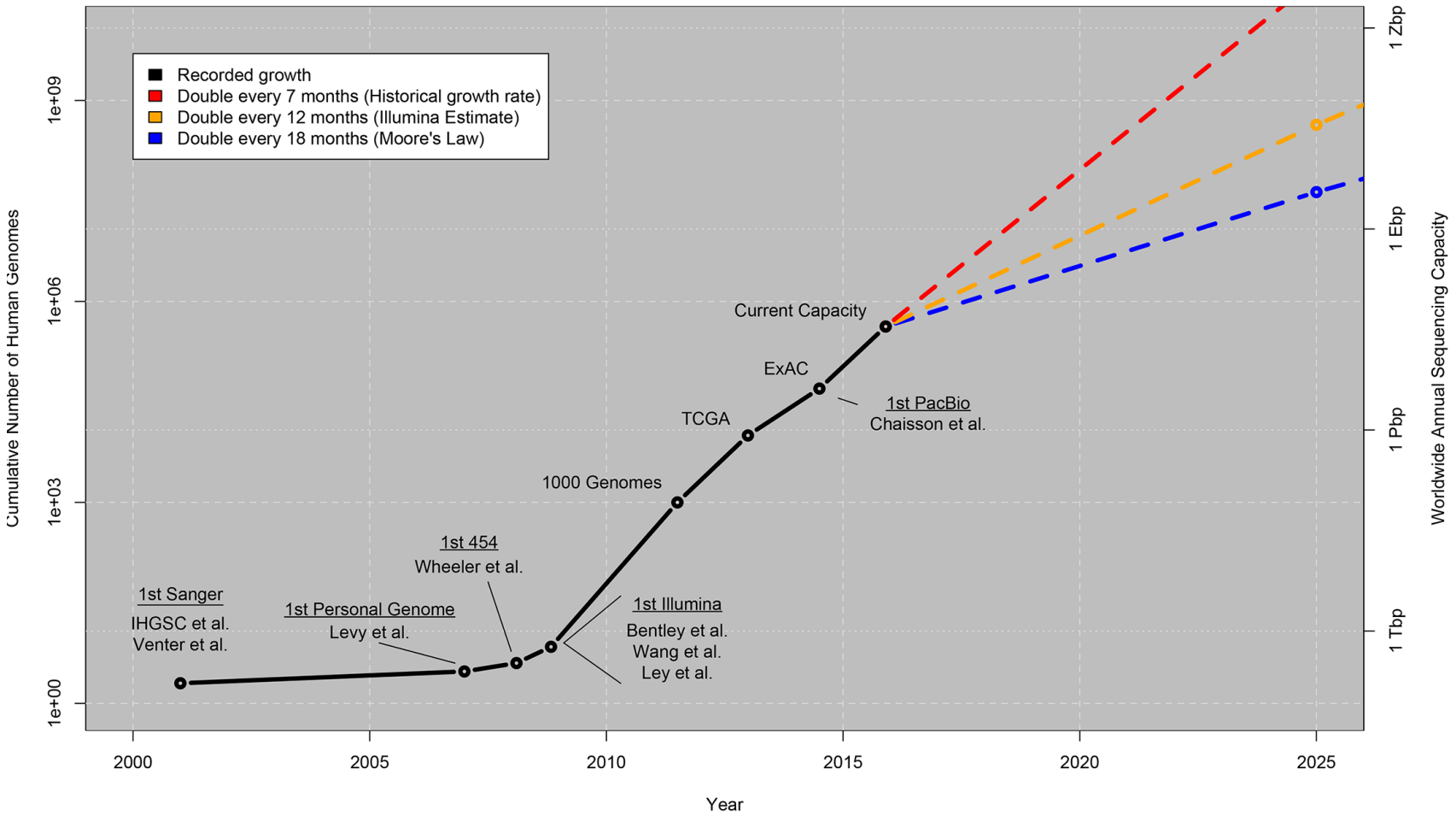
Bioinformatics: beyond Moore's law



[1]

[1] Deorowicz & Grabowski, ALMOB 2013

Growth of DNA sequencing, prediction



[2]

[2]Stephens et al., Big Data: Astronomical or Genomical?, Plos ONE 2015

Compression to the rescue

Problem overview

Genome sequences of the same species are very similar to each other. LZ77-type redundancy.
But: huge input, far distances between reference and currently compressed phrases.
So, we need an LZ77 variant (or a related method), working fast and in possibly small space.

Extra functionality

Fast random access to data (given S in compressed form, extract any $S[i]$ or $S[i \ldots j]$ possibly fast).

What doesn't work

Nice result...

$nH_k(S) + o(n)$ bits to represent (replace) S, with reading any $\Theta(\log_\sigma n)$ successive symbols of S in constant time (Sadakane & Grossi, 2006; González & Navarro, 2006; Ferragina & Venturini, 2007).

What doesn't work

Nice result...

$nH_k(S) + o(n)$ bits to represent (replace) S, with reading any $\Theta(\log_\sigma n)$ successive symbols of S in constant time (Sadakane & Grossi, 2006; González & Navarro, 2006; Ferragina & Venturini, 2007).

...but not for our case

Great theoretical achievement, but are there any implementations? More importantly, k here must be small, namely $k = o(\log_\sigma n)$ (doesn't capture the LZ77 redundancy).

Folklore that kind of works, but...

Idea

Partition S into equal-length blocks.
LZ77-encode each block with reference to R.
Store offsets to encoded blocks.
Accessing $S[i]$: find the resp. block, decode it wholly (or up to position i), return the symbol.

Folklore that kind of works, but...

Idea

Partition S into equal-length blocks.
LZ77-encode each block with reference to R.
Store offsets to encoded blocks.
Accessing $S[i]$: find the resp. block, decode it wholly (or up to position i), return the symbol.

The ugliness exposed

- Max match length capped.

Folklore that kind of works, but...

Idea

Partition S into equal-length blocks.
LZ77-encode each block with reference to R.
Store offsets to encoded blocks.
Accessing $S[i]$: find the resp. block, decode it wholly (or up to position i), return the symbol.

The ugliness exposed

- Max match length capped.
- Last matches in blocks artificially truncated.

Folklore that kind of works, but...

Idea

Partition S into equal-length blocks.
LZ77-encode each block with reference to R.
Store offsets to encoded blocks.
Accessing $S[i]$: find the resp. block, decode it wholly (or up to position i), return the symbol.

The ugliness exposed

- Max match length capped.
- Last matches in blocks artificially truncated.
- Non constant time decoding.

Folklore that kind of works, but...

Idea

Partition S into equal-length blocks.
LZ77-encode each block with reference to R.
Store offsets to encoded blocks.
Accessing $S[i]$: find the resp. block, decode it wholly (or up to position i), return the symbol.

The ugliness exposed

- Max match length capped.
- Last matches in blocks artificially truncated.
- Non constant time decoding.
- Extra working space at decoding.

RLZ (R=Relative)
(Kuruppu, Puglisi & Zobel, SPIRE, 2010)

Idea

S is parsed into maximal phrases (substrings) taken from R.
Constant-time access to any $R[j]$ is assumed.
To access $S[i]$, we need to know two things:

- Where the LZ-phrase containing $S[i]$ starts.
- Where the source (in R) of the LZ-phrase to which $S[i]$ belongs is.

RLZ, cont'd

Implementation

- Compressed bit-vector $B[1; n]$ with $O(1)$-time rank/select, 1s mark the first symbol of each phrase in S.
- Array $Q[1; t]$ storing the start positions of the phrases' sources.

RLZ, cont'd

Implementation

- Compressed bit-vector $B[1; n]$ with $O(1)$-time rank/select, 1s mark the first symbol of each phrase in S.
- Array $Q[1; t]$ storing the start positions of the phrases' sources.

Access formula

$$S[j] = R[Q[B.rank(j)] + j - B.select(B.rank(j))]$$

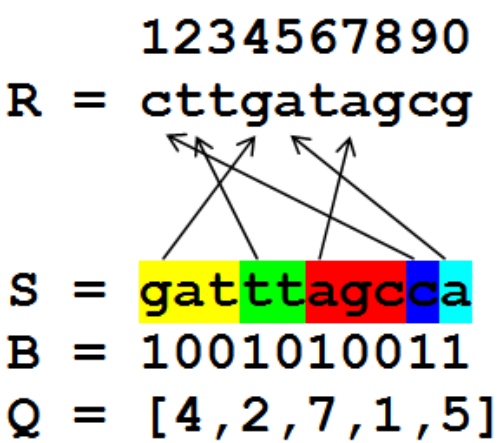
RLZ, cont'd

Implementation

- Compressed bit-vector $B[1; n]$ with $O(1)$-time rank/select, 1s mark the first symbol of each phrase in S.
- Array $Q[1; t]$ storing the start positions of the phrases' sources.

Access formula

$$S[j] = R[Q[B.rank(j)] + j - B.select(B.rank(j))]$$



S[8] = ?

B.rank(8) = 3

Q[3] = 7

B.select(B.rank(8)) = 6

Hence, S[8] = R[7 + (8 - 6)] = R[9]

GDC1 (Genome Differential Compressor) (Deorowicz & Grabowski, Bioinf. 2011)

Idea, inspired by (Kuruppu et al., Proc. ACSC, 2011)

- match offsets often form increasing sequences $\rightarrow$ differential encoding,

GDC1 (Genome Differential Compressor) (Deorowicz & Grabowski, Bioinf. 2011)

Idea, inspired by (Kuruppu et al., Proc. ACSC, 2011)

- match offsets often form increasing sequences $\rightarrow$ differential encoding,
- Huffman coding for various statistics,

GDC1 (Genome Differential Compressor) (Deorowicz & Grabowski, Bioinf. 2011)

Idea, inspired by (Kuruppu et al., Proc. ACSC, 2011)

- match offsets often form increasing sequences $\rightarrow$ differential encoding,
- Huffman coding for various statistics,
- block-based compression for random access (i.e., decode the block first),

GDC1 (Genome Differential Compressor) (Deorowicz & Grabowski, Bioinf. 2011)

Idea, inspired by (Kuruppu et al., Proc. ACSC, 2011)

- match offsets often form increasing sequences $\rightarrow$ differential encoding,
- Huffman coding for various statistics,
- block-based compression for random access (i.e., decode the block first),
- optionally (variant "ultra"): multiple reference sequences.

GDC1 results (cere), AMD Opteron 2.4 GHz

Table 1. Compression results for *S.cerevisiae* (39 genomes)

Method	Total		Avg. per seq.		c-sp	d-sp
	Size (MB)	Ratio	Size	Ratio	(MB/s)	(MB/s)
original	493.98	–	–	–	–	–
gzip*	132.00	3.7	3.385	3.7	3.65	91.92
bzip2*	122.66	4.0	3.145	4.0	6.22	16.08
7z*	4.97	99.3	0.052	244.7	0.54	163.03
RLCSA	40.13	12.3	–	–	0.57	1.75
Comrad	16.50	29.9	–	–	0.70	12.05
RLZ-opt*	9.33	52.9	0.167	76.1	1.60	189.99
RLZ-RePair*	6.48	76.2	–	–	0.27	160.91
GDC-fast	6.95	71.1	0.103	123.5	33.15	160.91
GDC-normal	6.88	71.8	0.103	123.5	33.15	161.43
GDC-advanced	6.66	74.2	0.097	131.0	38.90	161.43
GDC-advanced-R	6.66	74.2	0.097	131.0	29.06	161.43
GDC-ultra	4.57	108.2	0.042	303.5	2.53	161.43

The '*' character after compressor name means that EOLs were initially removed.

GDC1 results (human)

Table 3. Compression results for *H.sapiens* taken from Complete Genomics Inc. (70 genomes)

Method	Total		Avg. per seq.		c-sp	d-sp
	Size (MB)	Ratio	Size	Ratio	(MB/s)	(MB/s)
original	218 961.98	–	–	–	–	–
7z*	1 131.27	193.5	7.00	446.8	0.38	155.12
RLZ-opt*	1 731.31	126.5	15.26	204.9	1.34	130.38
GDC-fast	1 462.66	149.7	7.34	426.3	35.23	144.62
GDC-normal	1 201.15	182.3	7.34	426.3	35.90	146.16
GDC-advanced	1 201.25	182.3	7.34	426.3	36.36	147.05
GDC-advanced-R	1 201.25	182.3	7.34	426.3	26.71	146.65
GDC-ultra	910.13	240.6	3.12	1003.0	4.17	150.02

The '*' character after compressor name means that EOLs were initially removed.

GDC1 results, random access

Table 5. Random access time for 70 (top five rows) and 4 (bottom six rows) genomes of *H.sapiens*, for varying pattern length, m

	File size	RAM usage	$m = 10$	$m = 100$	$m = 1000$
GDC-fast/normal	1465.2 / 1204.2	–	–	–	–
GDC-ra-2^{11}-2^{13}	1761.1 / 1499.7	2124 / 1863	1180 / 3800	129 / 398	23 / 57
GDC-ra-2^{11}-2^{15}	1578.9 / 1317.4	1739 / 1477	1710 / 4390	188 / 457	58 / 64
GDC-ra-2^{11}-2^{17}	1501.6 / 1240.1	1582 / 1320	4040 / 6720	408 / 681	50 / 85
GDC-ra-2^{11}-2^{19}	1479.0 / 1217.5	1537 / 1275	11 750 / 14 580	1190 / 1474	129 / 165
RLCSA	5429.0	5685	33830	4698	1651
GDC-fast/normal	980.4 / 719.4	–	–	–	–
GDC-ra-2^{11}-2^{13}	997.1 / 736.6	1051 / 789	830 / 3360	90 / 351	18 / 52
GDC-ra-2^{11}-2^{15}	990.6 / 729.1	1038 / 774	1300 / 3810	140 / 404	22 / 56
GDC-ra-2^{11}-2^{17}	987.2 / 725.8	1031 / 770	3170 / 5770	320 / 593	41 / 75
GDC-ra-2^{11}-2^{19}	986.2 / 724.7	1027 / 766	10070 / 12830	1017 / 1280	108 / 143

All times are in nanosecond per character. File and memory sizes are in Megabytes. The complete collections are of size: 218 962 and 12 253 MB.

Simplified GDC1, with $O(1)$-time access (as described by Cox et al., SPIRE'16)

Idea

Three arrays used: $Q[1;t]$ like in RLZ. $M[1;t]$ with last character of each phrase. $B[1;n]$ (compressed bv with rank/select) with 1s marking the last char of each phrase.

Simplified GDC1, with $O(1)$-time access (as described by Cox et al., SPIRE'16)

Idea

Three arrays used: $Q[1;t]$ like in RLZ. $M[1;t]$ with last character of each phrase. $B[1;n]$ (compressed bv with rank/select) with 1s marking the last char of each phrase.

Access formula

$$S[j] = \begin{cases} M[B.rank(j)] & \text{if } B[j] = 1, \\ R[Q[B.rank(j)+1] + j - B.select(B.rank(j)) - 1] & \text{O/W} \end{cases}$$

Simplified GDC1, with $O(1)$-time access, example

```
    1234567890
R = cttgatagcg
```

```
S = gatttagccaa
B = 00010000101
Q = [4,6,5]
M = [t,c,a]
```

```
S[9] = ?
B.rank(9) = 2
B[9] = 1 -> S[9] = M[2] = c

S[7] = ?
B.rank(7) = 1
B[7] = 0 ->
  B.select(B.rank(7)) = 4
  S[9] = R[Q[1+1]+7-4-1] =
    R[6+7-4-1] = R[8] = g
```

LZ-End (Kreft & Navarro, 2010)

Idea

LZ77 variant in which the source of each phrase is a suffix of previous phrases. A trailing character follows.

LZ-End (Kreft & Navarro, 2010)

Idea

LZ77 variant in which the source of each phrase is a suffix of previous phrases. A trailing character follows.

Format

- $L[1; z]$ encodes the trailing characters,

LZ-End (Kreft & Navarro, 2010)

Idea

LZ77 variant in which the source of each phrase is a suffix of previous phrases. A trailing character follows.

Format

- $L[1; z]$ encodes the trailing characters,
- $source[1; z]$ (using $z \log z$ bits) encodes the phrase ID where the source ends,

LZ-End (Kreft & Navarro, 2010)

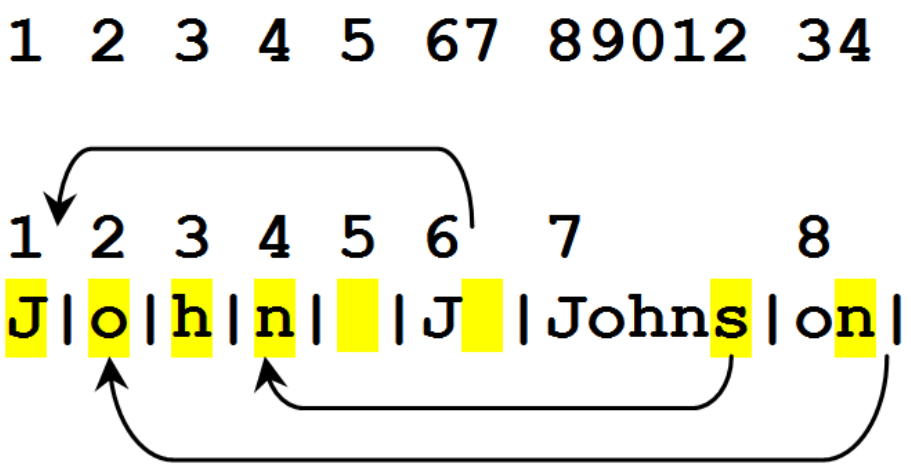
Idea

LZ77 variant in which the source of each phrase is a suffix of previous phrases. A trailing character follows.

Format

- $L[1; z]$ encodes the trailing characters,
- $source[1; z]$ (using $z \log z$ bits) encodes the phrase ID where the source ends,
- $B[1; n]$ is a bit-vector marking the end positions of phrases in T

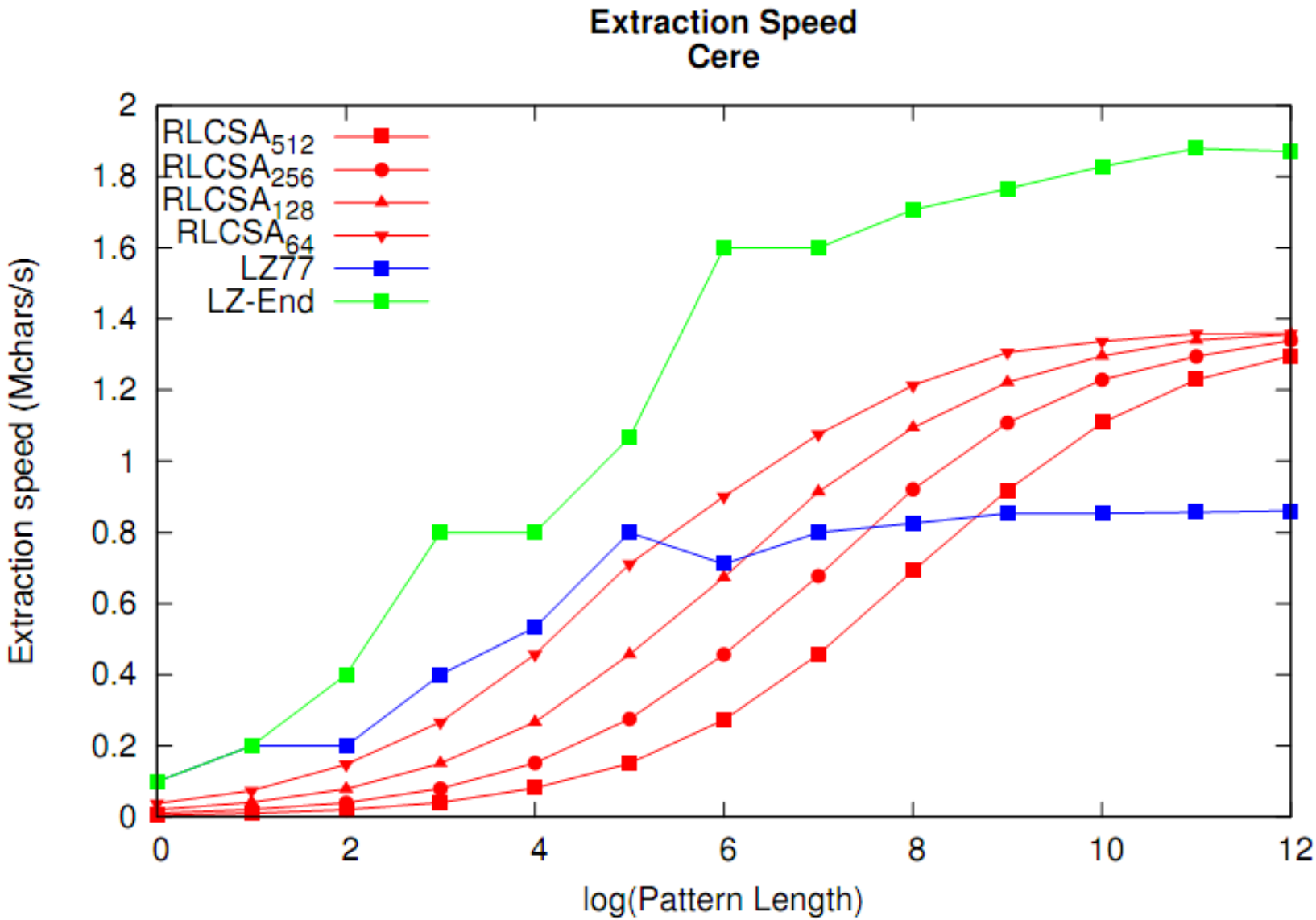
LZ-End, extract example



extract(T[9; 12]) (ohns)

Check that $T[12]$ is marked, read $L[7]$, i.e., s ($= T[12]$).
$T[11]$ is not marked, use the source phrase (its id $= 4$),
extract its last symbol ($L[4]$), i.e., n.
If $|source[4] > 1|$, we'd recursively refer to its source.
But here $|source[4]| = 1$, so we extract the last symbol of the prev phrase, i.e., read $L[3]$. Etc.
Linear time if the substring to extract ends at a phrase boundary.

LZ-End, extraction speed



Grammar compression: RePair (Larsson & Moffat, 2000)

AGGTC#AGTTC#AGGGC

R_1 -> AG
(AGGTC#AGTTC#AGGGC --> R_1GTC#R_1TTC#R_1GGC)

R_2 -> R_1G
(R_1GTC#R_1TTC#R_1GGC --> R_2TC#R_1TTC#R_2GC)

R_3 -> TC
(R_2TC#R_1TTC#R_2GC --> R_2R_3#R_1TR_3#R_2GC)

R_4 -> R_3#
(R_2R_3#R_1TR_3#R_2GC --> $R_2R_4R_1$TR_4R_2GC)

Grammar compression: RePair (Larsson & Moffat, 2000)

AGGTC#AGTTC#AGGGC

R_1 -> AG
(AGGTC#AGTTC#AGGGC --> R_1GTC#R_1TTC#R_1GGC)

R_2 -> R_1G
(R_1GTC#R_1TTC#R_1GGC --> R_2TC#R_1TTC#R_2GC)

R_3 -> TC
(R_2TC#R_1TTC#R_2GC --> R_2R_3#R_1TR_3#R_2GC)

R_4 -> R_3#
(R_2R_3#R_1TR_3#R_2GC --> $R_2R_4R_1$TR_4R_2GC)

Output: rule set $\mathcal{R}$, compressed sequence $\mathcal{C}$.

RePair on a bitmap B, with rank (Navarro, Puglisi & Valenzuela, 2013)

Augmenting the RePair representation

Store length of each expanded nonterminal Z:
if $Z \to XY$, then $\ell(Z) = \ell(X) + \ell(Y)$.
Similarly, store $r(X)$ being # of 1s in each expanded X.
Again, $r(Z) = r(X) + r(Y)$, if $Z \to XY$.

RePair on a bitmap B, with rank (Navarro, Puglisi & Valenzuela, 2013)

Augmenting the RePair representation

Store length of each expanded nonterminal Z:
if $Z \to XY$, then $\ell(Z) = \ell(X) + \ell(Y)$.
Similarly, store $r(X)$ being # of 1s in each expanded X.
Again, $r(Z) = r(X) + r(Y)$, if $Z \to XY$.

Sampling B

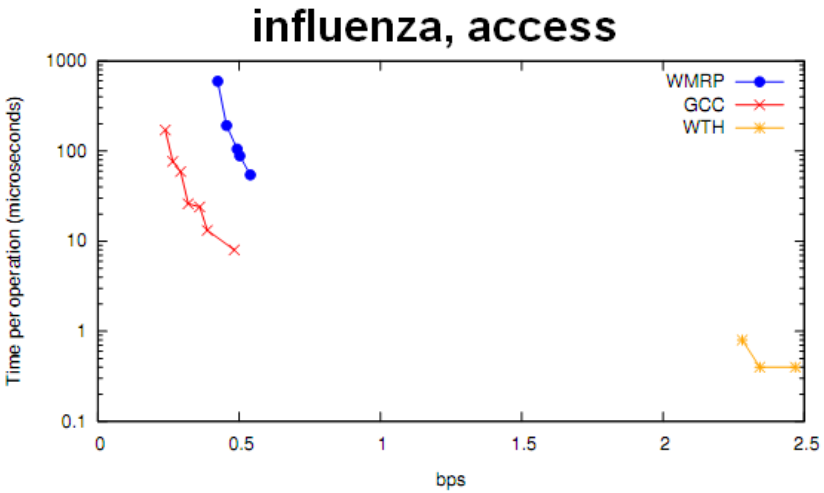
We also sample B every s bits. For each $B[i \cdot s]$ we store $P[i] = (p, o, r)$, where:
$C[p]$ is the p-th phrase of $\mathcal{C}$ containing $B[i \cdot s]$,
o is the offset within this phrase,
r is the rank up to that phrase.

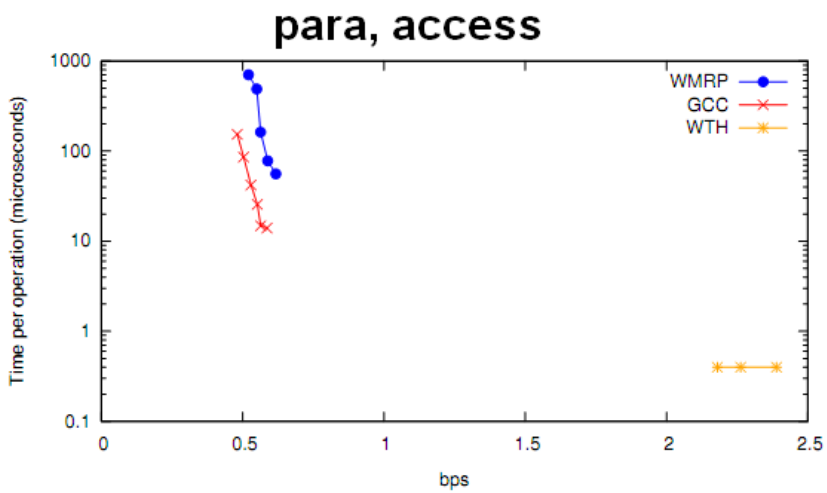
Computing $rank_1(B, i)$

$j \leftarrow \lfloor i/s \rfloor$;
$r \leftarrow P[j].r$;
$p' \leftarrow P[j].p$;
$l \leftarrow i - P[j].o$;
while $p' \leq |\mathcal{C}|$ **do**
  $Z \leftarrow \mathcal{C}[p']$;
  if $l + l(Z) < i$ **then**
    $p' \leftarrow p' + 1$;
    $r \leftarrow r + r(Z)$;
    $l \leftarrow l + l(Z)$;
  else
    return $Expand(Z, l, r)$ [3]

[3] Navarro, Puglisi & Valenzuela, JEA 2014

RePair-based compr., access (Navarro & Ordóñez, 2014)





dataset	n	σ	H_0	RP	LZ
para	429	5	1.12	0.37	0.19
influenza	322	16	1.98	0.23	0.15

CPU: Intel Xeon(R) E5620 2.4 GHz

LZ77 is an overkill

Idea

S is expected to be similar to R (R patched with some SNPs and (usually) short indels). While LZ77 searches for matches everywhere. Waste of compression time, memory and codespace.

LZ77 is an overkill

Idea

S is expected to be similar to R (R patched with some SNPs and (usually) short indels). While LZ77 searches for matches everywhere. Waste of compression time, memory and codespace.

An explicit incarnation

(Chern, Ochoa, Manolakos, No, Venkat & Weissman, 2012)
Starting from a position in S, find the longest matching string within a fixed window in R. Then encode (pos, len) of the match in R and the first unmatched symbol that follows.
(...) we want to be conservative and avoid excessive shifts, and to allow the algorithm to be agnostic to bursty insertions and deletions.

RLZ with compressed pointers (Ferrada, Gagie, Gog & Puglisi, SPIRE 2014)

Idea (first, absolute pointers)

Assumption: SNPs dominate. RLZ (Kuruppu et al.) is like LZSS.
Here: follow a match with a literal, like in LZ77.
Represent S with a seq of triples: $\langle \ell_r, p_r, c_r \rangle$,
with the meaning: copy of $R[p_r \ldots p_r + \ell_r - 1]c_r$.
In particular, $\ell_r = 0$ denotes a literal (p_r irrelevant then).
Use a few structures: a bv B_1 marking phrase beginnings,
array P of pointers, array C of mismatch literals.

RLZ with compressed pointers (Ferrada, Gagie, Gog & Puglisi, SPIRE 2014)

Idea (first, absolute pointers)

Assumption: SNPs dominate. RLZ (Kuruppu et al.) is like LZSS.
Here: follow a match with a literal, like in LZ77.
Represent S with a seq of triples: $\langle \ell_r, p_r, c_r \rangle$,
with the meaning: copy of $R[p_r \ldots p_r + \ell_r - 1]c_r$.
In particular, $\ell_r = 0$ denotes a literal (p_r irrelevant then).
Use a few structures: a bv B_1 marking phrase beginnings,
array P of pointers, array C of mismatch literals.

Access $S[i]$

$r = B_1.rank(i)$ is the index of the phrase containing the query.
If $B_1[i+1] = 1$, return $C[r]$.
Otherwise, return $R[P[r] + i - B_1.select(r)]$.

RLZ with compressed pointers, cont'd

Relative (not compressed yet) pointers

Use $P'[0 \ldots z-1] = [p_0 - h_0, \ldots, p_{z-1} - h_{z-1}]$,
where h_r is the starting position in S of phrase r.

RLZ with compressed pointers, cont'd

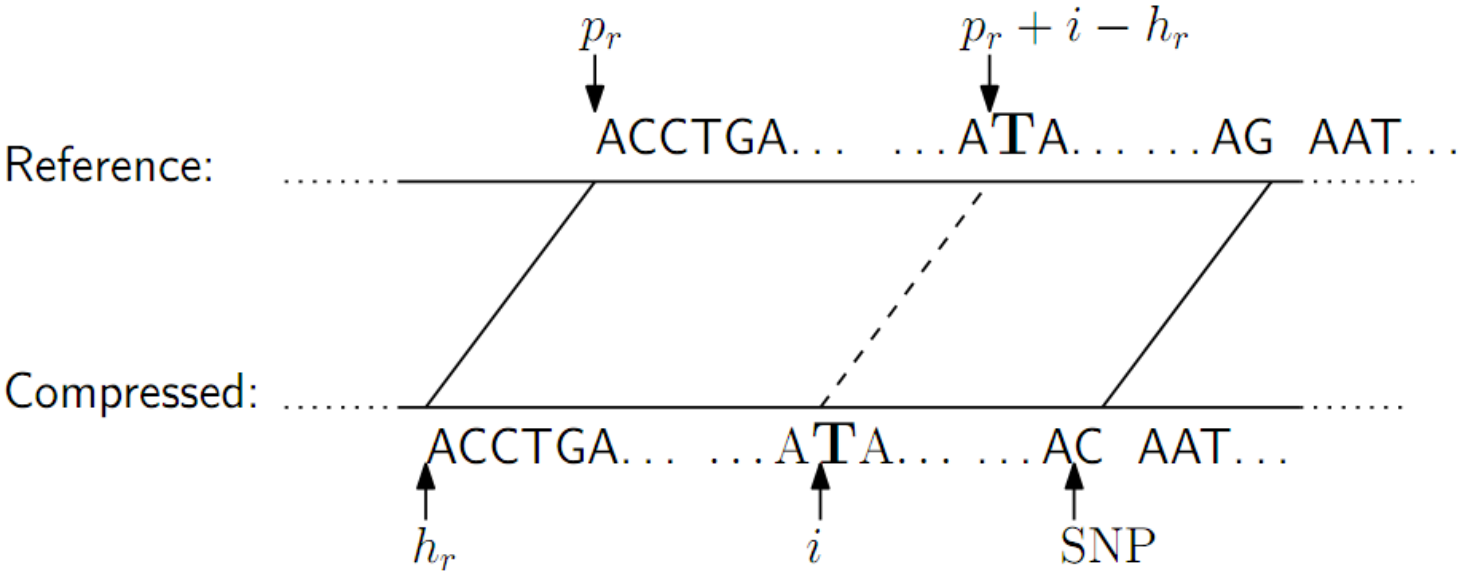
Relative (not compressed yet) pointers

Use $P'[0 \ldots z-1] = [p_0 - h_0, \ldots, p_{z-1} - h_{z-1}]$,
where h_r is the starting position in S of phrase r.

Access $S[i]$

Like previously, but no need to select! It's (sort of) precomputed.
I.e., replace
return $R[P[r] + i - B_1.select(r)]$
with
return $R[P'[r] + i]$.

RLZ with compressed pointers, cont'd



4

Yes, now compressed pointers

Idea: store (in P'') only those (relevant) pointers from P' which differ to their preceding (relevant) pointers.
Use bv B_2 with 1s for those pointers that are kept in P''.

[4] Ferrada et al., Relative Lempel-Ziv with constant-time..., SPIRE 2014

RLZ with compressed pointers, results

Table 1. Random access time (for extraction lengths $m = 8, 64, 512, 4096$ characters) for RLZ and GDC for a collection of 36 *S. cerevisiae* genomes, totalling 464 MB before compression. The two GDC rows correspond to different settings for the R-block size and D-block size (see [2]). Times are given in nanoseconds per extracted character averaged over 10 million extractions.

Method	Size (MB)	$m = 8$	$m = 64$	$m = 512$	$m = 4096$
RLZ	11	50	7	3	2
GDC-ra-2^8-2^8	14	500	68	15	8
GDC-ra-2^{12}-2^{12}	10	750	102	19	8

GDC2 (Deorowicz, Danek & Niemiec, Sci. Rep., 2015)

Idea ("LZ on LZ")

1st level factoring: apply LZSS to S^k with R as ref, obtain L^k.
2nd lvl: apply LZSS to L^k where phrase sources are in $L^j, j < k$.

input data

Position	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
R	A	C	T	G	A	C	C	G	T	C	G	A	T	T	T	A	A	C	C	C		
S^1	A	A	C	T	G	A	C	C	G	T	A	A	G	A	T	T	T	A	A	T	G	C
S^2	A	C	T	A	A	C	C	G	T	C	C	A	T	T	T	A	A	T	G	C		
S^3	A	C	T	A	A	C	C	G	T	C	A	A	T	T	T	A	A	T	G	C		
S^4	A	A	A	C	T	G	A	T	C	G	T	C	G	T	T	A	A	G	T	G	C	
S^5	A	C	T	G	A	T	C	G	T	C	G	T	T	A	A	C	C	C				
S^6	A	C	T	G	A	T	C	G	T	C	G	T	T	A	A	G	C	C				

first level factoring

L^1	(L1M, 16, 3) (L1M, 3, 7) (L1L, A) (L1L, A) (L1M, 11, 7) (L1L, T) (L1L, G) (L1L, C)
L^2	(L1M, 1, 3) (L1L, A) (L1M, 5, 6) (L1L, C) (L1M, 12, 6) (L1L, T) (L1L, G) (L1L, C)
L^3	(L1M, 1, 3) (L1L, A) (L1M, 5, 6) (L1L, A) (L1M, 12, 6) (L1L, T) (L1L, G) (L1L, C)
L^4	(L1L, A) (L1M, 16, 3) (L1M, 3, 3) (L1L, T) (L1M, 7, 5) (L1M, 13, 2) (L1M, 16, 2) (L1L, G) (L1L, T) (L1L, G) (L1L, C)
L^5	(L1M, 1, 5) (L1L, T) (L1M, 7, 5) (L1M, 13, 2) (L1M, 16, 5)
L^6	(L1M, 1, 5) (L1L, T) (L1M, 7, 5) (L1M, 13, 2) (L1M, 16, 2) (L1L, G) (L1M, 19, 2)

GDC2, cont'd

first level factoring

L^1	(L1M, 16, 3) (L1M, 3, 7) (L1L, A) (L1L, A) (L1M, 11, 7) (L1L, T) (L1L, G) (L1L, C)
L^2	(L1M, 1, 3) (L1L, A) (L1M, 5, 6) (L1L, C) (L1M, 12, 6) (L1L, T) (L1L, G) (L1L, C)
L^3	(L1M, 1, 3) (L1L, A) (L1M, 5, 6) (L1L, A) (L1M, 12, 6) (L1L, T) (L1L, G) (L1L, C)
L^4	(L1L, A) (L1M, 16, 3) (L1M, 3, 3) (L1L, T) (L1M, 7, 5) (L1M, 13, 2) (L1M, 16, 2) (L1L, G) (L1L, T) (L1L, G) (L1L, C)
L^5	(L1M, 1, 5) (L1L, T) (L1M, 7, 5) (L1M, 13, 2) (L1M, 16, 5)
L^6	(L1M, 1, 5) (L1L, T) (L1M, 7, 5) (L1M, 13, 2) (L1M, 16, 2) (L1L, G) (L1M, 19, 2)

second level factoring ↓

D^1	(L1M, 16, 3) (L1M, 3, 7) (L1L, A) (L1L, A) (L1M, 11, 7) (L1L, T) (L1L, G) (L1L, C)
D^2	(L1M, 1, 3) (L1L, A) (L1M, 5, 6) (L1L, C) (L1M, 12, 6) **(L2M, 1, 6, 3)**
D^3	**(L2M, 2, 1, 3)** (L1L, A) **(L2M, 2, 5, 4)**
D^4	(L1L, A) (L1M, 16, 3) (L1M, 3, 3) (L1L, T) (L1M, 7, 5) (L1M, 13, 2) (L1M, 16, 2) (L1L, G) **(L2M, 1, 6, 3)**
D^5	(L1M, 1, 5) **(L2M, 4, 4, 3)** (L1M, 16, 5)
D^6	**(L2M, 5, 1, 4) (L2M, 4, 7, 2)** (L1M, 19, 2)

GDC2, cont'd

first level factoring

L^1	(L1M, 16, 3) (L1M, 3, 7) (L1L, A) (L1L, A) (L1M, 11, 7) (L1L, T) (L1L, G) (L1L, C)
L^2	(L1M, 1, 3) (L1L, A) (L1M, 5, 6) (L1L, C) (L1M, 12, 6) (L1L, T) (L1L, G) (L1L, C)
L^3	(L1M, 1, 3) (L1L, A) (L1M, 5, 6) (L1L, A) (L1M, 12, 6) (L1L, T) (L1L, G) (L1L, C)
L^4	(L1L, A) (L1M, 16, 3) (L1M, 3, 3) (L1L, T) (L1M, 7, 5) (L1M, 13, 2) (L1M, 16, 2) (L1L, G) (L1L, T) (L1L, G) (L1L, C)
L^5	(L1M, 1, 5) (L1L, T) (L1M, 7, 5) (L1M, 13, 2) (L1M, 16, 5)
L^6	(L1M, 1, 5) (L1L, T) (L1M, 7, 5) (L1M, 13, 2) (L1M, 16, 2) (L1L, G) (L1M, 19, 2)

second level factoring ↓

D^1	(L1M, 16, 3) (L1M, 3, 7) (L1L, A) (L1L, A) (L1M, 11, 7) (L1L, T) (L1L, G) (L1L, C)
D^2	(L1M, 1, 3) (L1L, A) (L1M, 5, 6) (L1L, C) (L1M, 12, 6) **(L2M, 1, 6, 3)**
D^3	**(L2M, 2, 1, 3)** (L1L, A) **(L2M, 2, 5, 4)**
D^4	(L1L, A) (L1M, 16, 3) (L1M, 3, 3) (L1L, T) (L1M, 7, 5) (L1M, 13, 2) (L1M, 16, 2) (L1L, G) **(L2M, 1, 6, 3)**
D^5	(L1M, 1, 5) **(L2M, 4, 4, 3)** (L1M, 16, 5)
D^6	**(L2M, 5, 1, 4) (L2M, 4, 7, 2)** (L1M, 19, 2)

Results (compression ratio and speed)

H.sapiens 9557:1 (vs 2262 for GDC-ultra, 2065 for FRESCO (Wandelt & Leser, 2013)),
A.thaliana 587:1 (vs 245 for GDC-ultra, 179 for FRESCO),
(Multi-thr.) compr. speed ~200 MB/s, decomp. speed 1 GB/s.

In practice, we can often change the problem. Use a VCF db

Reality escapes

If the problem is easy/closed, it is rarely useful.
If the problem is hard/intractable, we can often replace it with a simpler one, which may be even more relevant in practice.

In practice, we can often change the problem. Use a VCF db

Reality escapes

If the problem is easy/closed, it is rarely useful.
If the problem is hard/intractable, we can often replace it with a simpler one, which may be even more relevant in practice.

New genome representation

Rather than storing genomes in FASTA (i.e., raw sequences), we can refer to e.g.

- VCF (Danecek et al., 2011) used in the 1000GP,
- general feature format (GFF) used in the PGP.

That is, use a ref genome R and a db of m variants. Represent each genome from the collection as m bits (if the i-th variant occurs in it or not; assuming bi-allelic sites).

TGC (Thousands Genomes Compression) (Deorowicz, Danek & Grabowski, Bioinf. 2013)

Data	VCFmin	VDBV		VDBV + gzip		VDBV + 7z		TGC	
	size	size	c-time	size	c-time	size	c-time	size	c-time
				H. sapiens					
Chr. 1	13,249	890.6	136	368.9	311	55.9	1,070	**32.3**	690
Chr. 11	8,351	560.1	86	238.7	200	33.1	598	**20.2**	392
Chr. 21	2,286	153.0	25	66.8	54	10.4	107	**6.3**	96
Complete	173,505	11,679.9	1,819	4,813.6	4,075	733.8	11,176	**431.6**	8,364
				A. thaliana					
Chr. 1	4,945	405.8	64	117.3	121	34.7	395	**26.1**	231
Chr. 4	3,386	281.4	42	79.7	92	23.7	271	**18.2**	120
Complete	20,756	1,722.5	270	489.0	526	143.5	1,673	**109.7**	815

Sizes in MB, times in sec, c-time is compression time.
VDBV = variant db + byte vector
Compr. var-db incl. in TGC: 51.0 MB (H.sap), 12.5 MB (A.th).

TGC algorithm

- LZSS style,

TGC algorithm

- LZSS style,
- byte-oriented,

TGC algorithm

- LZSS style,
- byte-oriented,
- parsing into matches and literals,

TGC algorithm

- LZSS style,
- byte-oriented,
- parsing into matches and literals,
- matches: $\langle 1, ref_seq, len \rangle$, literals: $\langle 0, byte_val \rangle$,

TGC algorithm

- LZSS style,
- byte-oriented,
- parsing into matches and literals,
- matches: $\langle 1, ref_seq, len \rangle$, literals: $\langle 0, byte_val \rangle$,
- several contextual models for the components,

TGC algorithm

- LZSS style,
- byte-oriented,
- parsing into matches and literals,
- matches: $\langle 1, ref_seq, len \rangle$, literals: $\langle 0, byte_val \rangle$,
- several contextual models for the components,
- arithmetic coding used,

TGC algorithm

- LZSS style,
- byte-oriented,
- parsing into matches and literals,
- matches: $\langle 1, ref_seq, len \rangle$, literals: $\langle 0, byte_val \rangle$,
- several contextual models for the components,
- arithmetic coding used,
- no random access. :-(

Constant-time variant detection in a sequence

```
R          : 11110000110110010111

S1         : 11110111010110111101
bv(S1)     : 1....1...1....111111

S2         : 11110000110100000111
bv(S2)     : 1..........11.11...

S3         : 11000010101100111111
bv(S3)     : 1.1.1..1.1......11..
```

$S_i[j] = R[j]$ **if** $bv(S_i).rank(j) \mod 2 = 1$ **else** $1 - R[j]$

Constant-time variant detection in a sequence, cont'd

Unfortunately, it's not so good in practice
(1000GP data: 2184 genomes, ~37M variant sites).
Fraction of set bits: almost 10% $\rightarrow H_0 \approx 0.47$. Weak compression.

Constant-time variant detection in a sequence, cont'd

Unfortunately, it's not so good in practice
(1000GP data: 2184 genomes, ~37M variant sites).
Fraction of set bits: almost 10% $\rightarrow H_0 \approx 0.47$. Weak compression.

Simple alternative

Divide each bit-vector into snippets of b bits and Huff-compress.
Add another bit-vector (B_i) with 1s telling the Huffman codeword beginnings. Compress it, add select.
To access $S_i[j]$: calculate $B_i.select(j/b)$, decode the corresponding Huffman codeword etc.

Constant-time variant detection in a sequence, cont'd

Unfortunately, it's not so good in practice
(1000GP data: 2184 genomes, ~37M variant sites).
Fraction of set bits: almost 10% $\rightarrow H_0 \approx 0.47$. Weak compression.

Simple alternative

Divide each bit-vector into snippets of b bits and Huff-compress.
Add another bit-vector (B_i) with 1s telling the Huffman codeword beginnings. Compress it, add select.
To access $S_i[j]$: calculate $B_i.select(j/b)$, decode the corresponding Huffman codeword etc.

Simple tradeoff

B_i, even compressed, are relatively large. Solution: set 1 for the beginning of every kth Huffman codeword (fewer 1s, better compression of B_i). Access time grows from $O(1)$ to $O(k)$ (or to $O(1 + kb/\log n)$ in theory).

Better than Huffman (in this app)

Huffman coding is optimal among the codes with a codebook.
But here we have Huffman + another bit string (marking the beginnings).
So maybe we can do better?

Better than Huffman (in this app)

Huffman coding is optimal among the codes with a codebook.
But here we have Huffman + another bit string (marking the beginnings).
So maybe we can do better?

Fredriksson & Nikitin, 2007; Ferragina & Venturini, 2007

As the codeword boundaries are known (from another bit string), use the codespace as densely as possible.
I.e., $0, 1, 00, 01, 10, 11, 000, \ldots$.

Better than Huffman (in this app)

Huffman coding is optimal among the codes with a codebook.
But here we have Huffman + another bit string (marking the beginnings).
So maybe we can do better?

Fredriksson & Nikitin, 2007; Ferragina & Venturini, 2007

As the codeword boundaries are known (from another bit string), use the codespace as densely as possible.
I.e., $0, 1, 00, 01, 10, 11, 000, \ldots$.

But now we can't mark every kth codeword

A lame hybrid: take a small k, Huffman-encode the first $k-1$ snippets and use the dense encoding for the last snippet (only).

Huffman or not, some results

Huffman

$b = 16$, avg Huffman codeword length: 5.30 bpc.
Set $k = 10$, we obtain a companion bv of length $(5.30/16)n$ bits, where the fraction of 1s is $1/53 = 1.89\%$, i.e. $H_0 = 0.135$.
Total: $n \times 5.30/16 \times (1 + 0.135 * 1.3) = 0.39n$ bits
(assuming 1.3 expansion factor for the RRR-compressed bv).

Huffman or not, some results

Huffman

$b = 16$, avg Huffman codeword length: 5.30 bpc.
Set $k = 10$, we obtain a companion bv of length $(5.30/16)n$ bits, where the fraction of 1s is $1/53 = 1.89\%$, i.e. $H_0 = 0.135$.
Total: $n \times 5.30/16 \times (1 + 0.135 * 1.3) = 0.39n$ bits
(assuming 1.3 expansion factor for the RRR-compressed bv).

Dense coding

$b = 16$, avg dense codeword length: 3.19 bpc.
A companion bv practically incompressible, so we skip compr.
Total: $n \times 3.19/16 \times (1 + 1 * 1.3) = 0.46n$ bits.

Huffman or not, some results, cont'd

Hybrid, $b = 16$, $k = 4$ or $k = 10$

$k = 4$

$5.30 + 5.30 + 5.30 + 3.19 = 19.09$ bits on avg for 64 input bits.

Plus a bv where fraction of 1s is $1/19.09 = 5.2\%$. $H_0 = 0.296$.

Total: $n \times 19.09/64 \times (1 + 0.296 * 1.3) = 0.413n$ bits.

$k = 10$

Total: $0.375n$ bits.

Back to RLZ-like compression; coarse granulation

Maybe matches with bit precision are not a good idea?

Use 'symbols' of $b > 1$ bits.
Pro: b times shorter bit vector.
Con: Mismatch phrases have to be stored explicitly.

Let $b = 4$

R : F0D975C032AB78145

S_1 : F36975B033CB76505
bv(S_1) : 11.1..11.1.1.1..1

mismatch phrases: (3,6) (B) (3,C) (6,5,0)

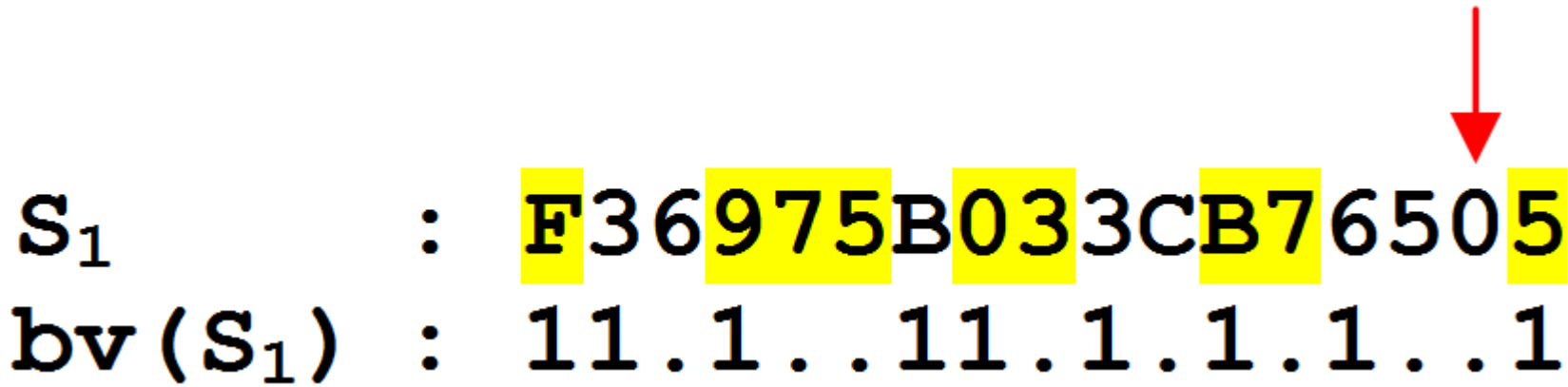
Apply a (Compressed) Prefix Sum ds

Raman, Raman & Rao, SODA 2002

n non-neg. integers summing up to m can be represented in $\mathcal{B}(n, m+n) + o(n)$ bits and support $O(1)$-time partial sum queries.

Back to our example

Prefix Sum ds built for $X = \{2, 1, 2, 3\}$.
Let's query $S_1[63]$. If $bv(S_1).rank(1 + j/b) = 2c$, we compute $sum(c-1, X)$.
That is, $bv(S_1).rank(1 + 63/4) = 8$, so we read $sum(3, X) = 5$.



One more tweak and some results

Mismatch phrases are compressible too

We Huffman-compress them and adapt the prefix sum structure appropriately.

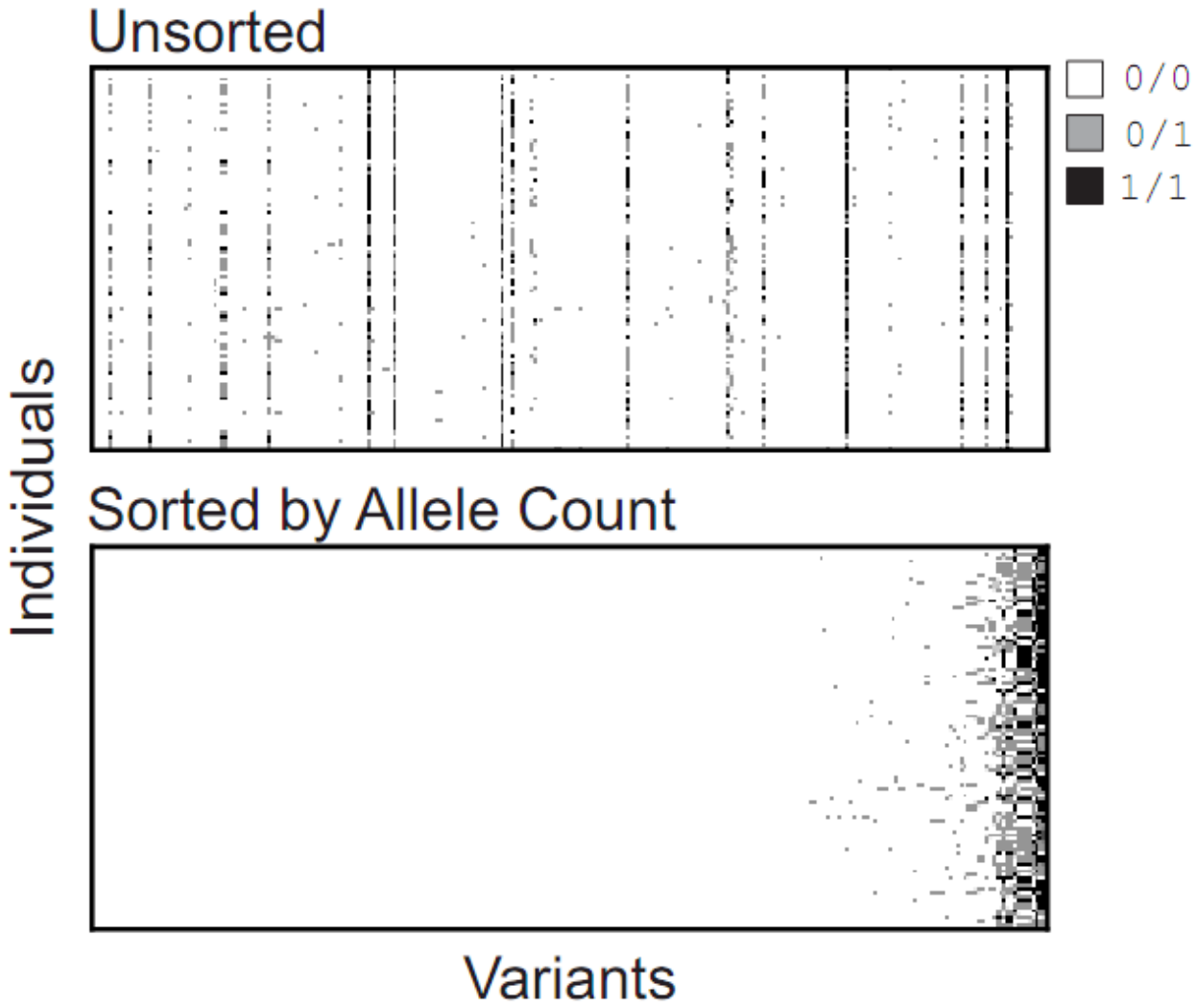
Estimated results (from a sample)

$b = 8$. Bv of length $n/8$, with 29% of 1s. $H_0 = 0.87(n/8)$.
31% of the bv are mismatch phrases, but their # is 14.5%.

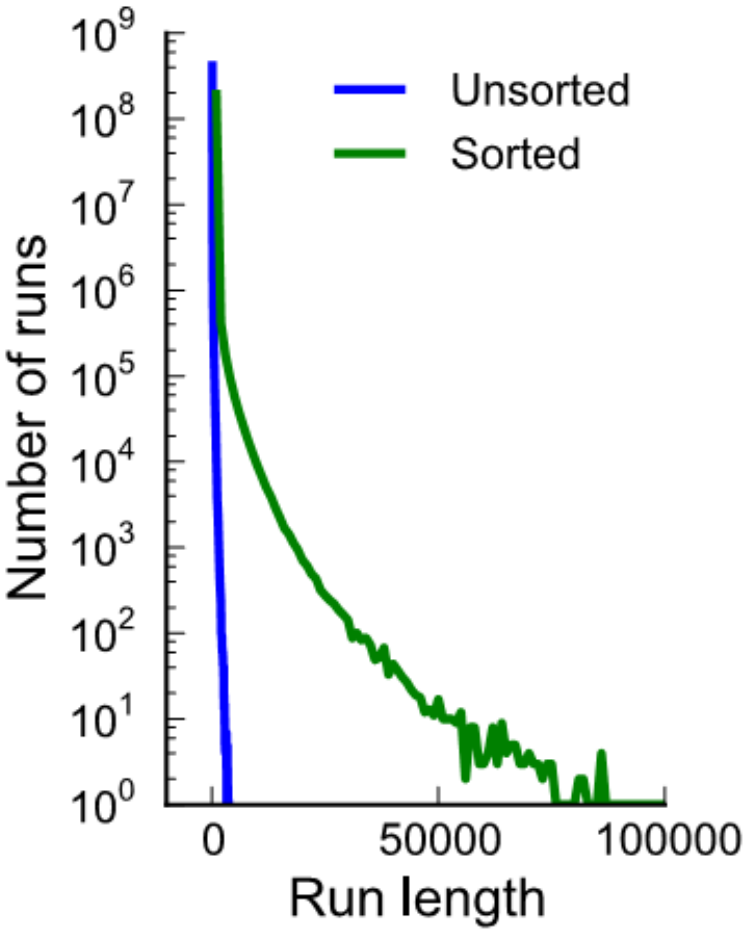
Mismatch phrases not compressed

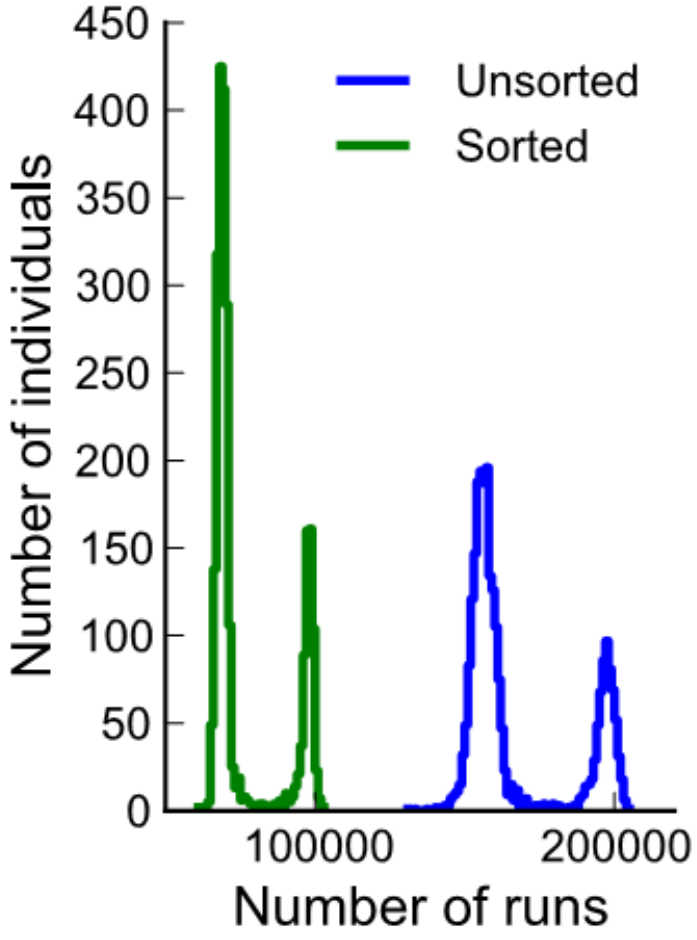
The prefix sum ds: $0.145(n/8)\log((0.145 + 0.31)n/8)$,
plus "$o(n/8)$", i.e., 14 MBit plus the $o(\cdot)$ term.
In total (in bits): 4.6M + 0.31M*8 + 14M = 21.1M, i.e. 57.3% (not incl. the lower-order terms) of the original bit-vector. :-(

Sorting variants by allele freq improves compression (Layer et al., Nature Meth. 2016)



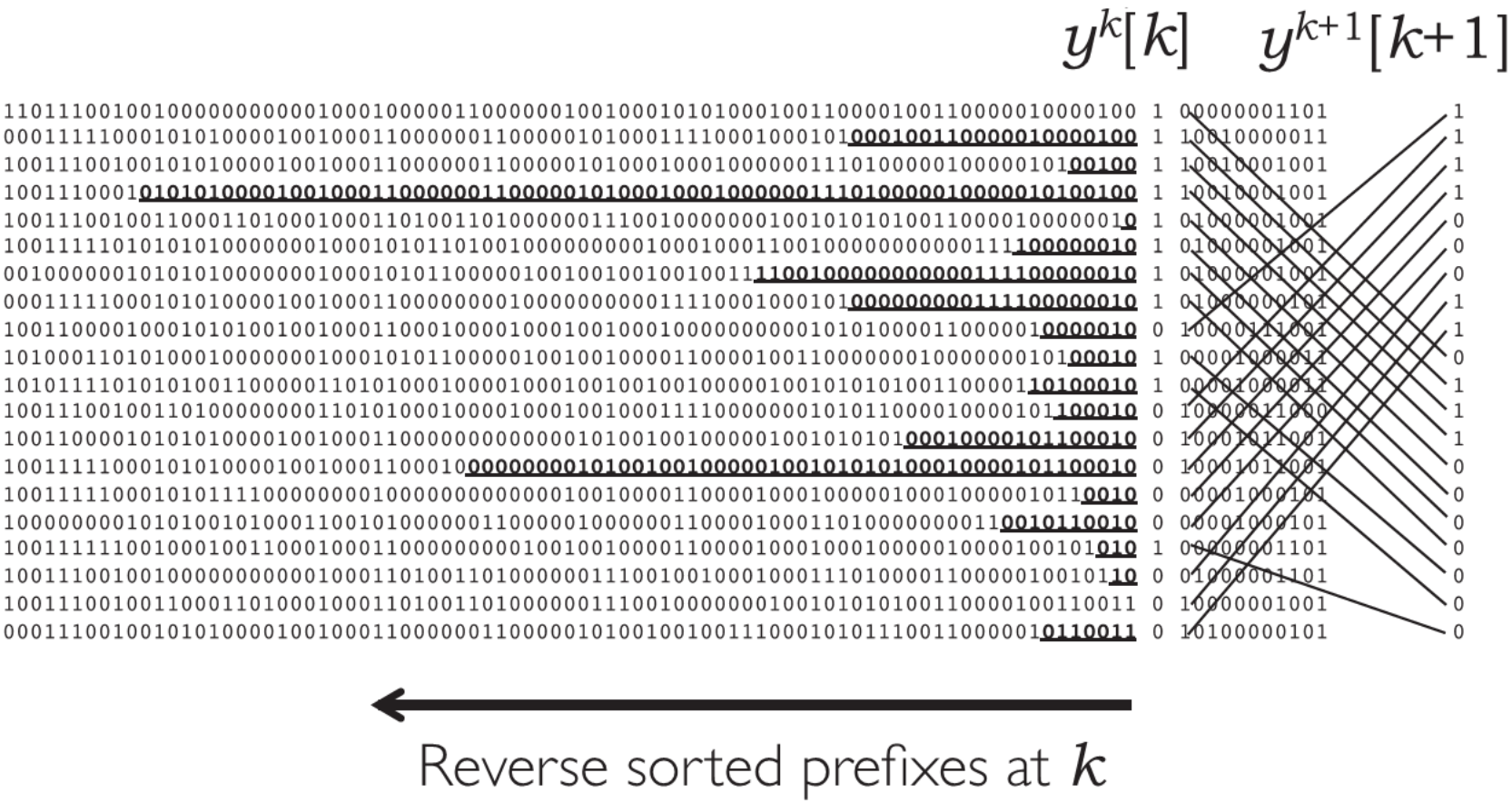
Runs in rows (individuals)





Positional BWT (Durbin, Bioinf. 2014)

N rows (samples), M columns (sites).
Reorder the rows M times, for each column.
Can be used for imputation and phasing (for ex., via finding all set-maximal matches within the matrix in linear time).



PBWT, compression and access

Compression

The columns, with their bits sorted in order of reversed prefixes, are strongly run-length compressible (local correlation in values due to linkage disequilibrium).

PBWT, compression and access

Compression

The columns, with their bits sorted in order of reversed prefixes, are strongly run-length compressible (local correlation in values due to linkage disequilibrium).

Access

Constant-time access to a RL-compressed bv $\rightarrow$ prefix sum.
But the bits in columns are permuted! How to read $x_i[k]$, where i is the original row index?

PBWT, compression and access

Compression

The columns, with their bits sorted in order of reversed prefixes, are strongly run-length compressible (local correlation in values due to linkage disequilibrium).

Access

Constant-time access to a RL-compressed bv $\rightarrow$ prefix sum.
But the bits in columns are permuted! How to read $x_i[k]$, where i is the original row index?

Simple idea

Store the permutation (in $M \log M$ bits) every $s = \omega(\log M)$ columns. Then scan over up to $s - 1$ following columns until $x_i[k]$ is recovered.

PBWT in BGT format (Heng Li, Bioinf 2015)

Critique of GQT (Layer et al.)

While it is very fast for selecting a subset of samples and for traversing all sites, it discards phasing, is inefficient for region query and is not compressed well.

We generated the BGT database for the first release of Haplotype Reference Consortium (HRC; http://bit.ly/HRC-org). The input is a BCF containing 32,488 samples across 39.2 million SNPs on autosomes. The BGT file size is 7.4GB, 11% of the genotype-only BCF, or 8% of GQT. Decoding the genotypes of all samples across 142k sites in a 10Mbp region takes 11 CPU seconds, which amounts to decoding 420 million genotypes per second. This speed is even faster than computing allele counts and outputting VCF.

How fast is $O(1)$-time?

We often use a compressed bit-vector with rank/select.
Access time approx. proportional to the number of cache misses.

2 misses: divide B into fixed-length blocks,
1st level: ranks of block beginnings and offsets to compressed blocks;
2nd level: the compressed blocks.

Question

Can we have < 2 cache misses on avg?

rank-cf (Grabowski & Raniszewski, 2016)

Obvious trick

Mono-block: block containing only 0s or only 1s.

Let f be the fraction of mono-blocks in B.

We have about $2 - f$ cache misses per rank, on avg.

rank-cf (Grabowski & Raniszewski, 2016)

Obvious trick

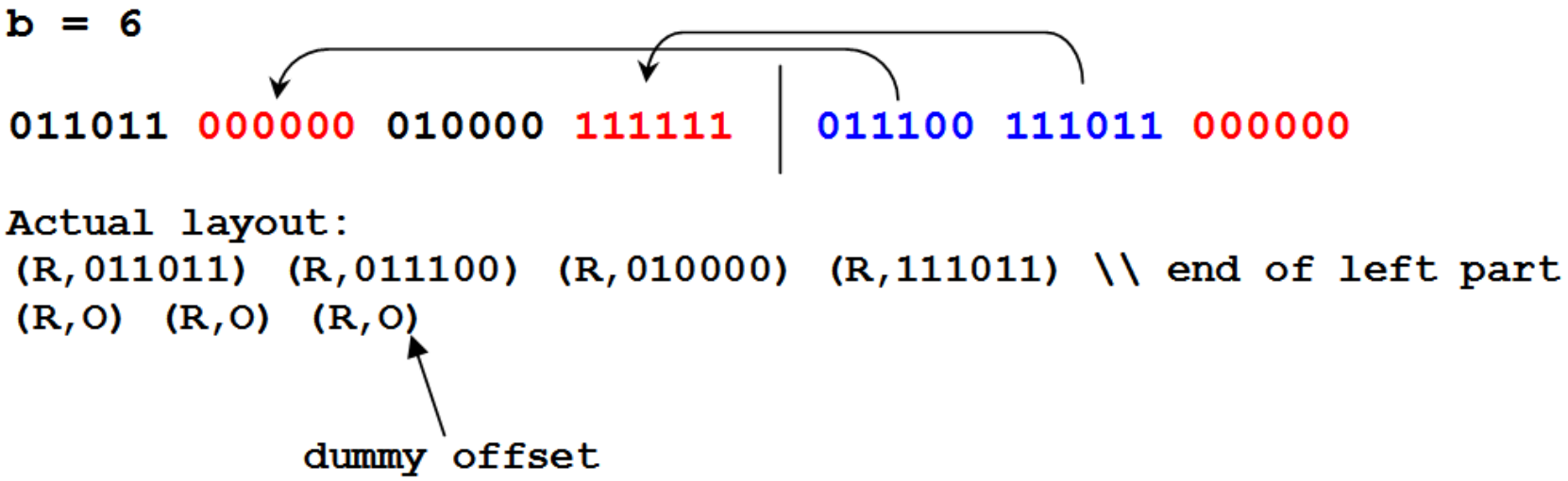
Mono-block: block containing only 0s or only 1s.
Let f be the fraction of mono-blocks in B.
We have about $2 - f$ cache misses per rank, on avg.

cf variant

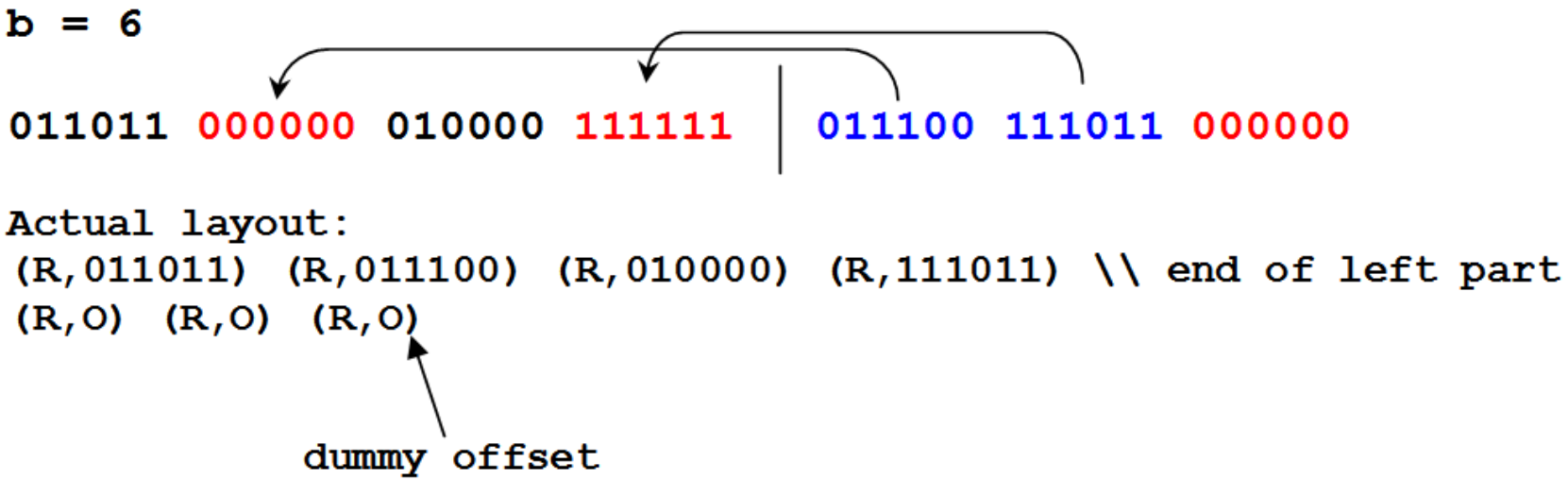
We scan B from left to right on block basis.
$\mathcal{L} = B[1 \ldots j]$, $\mathcal{R} = B[j+1 \ldots n]$ is the current split.
Find such j that # of mono-blocks in $\mathcal{L}$ equals to # of non-mono-blocks in $\mathcal{R}$. Store the content of non-mono-blocks from $\mathcal{R}$ in the holes of $\mathcal{L}$.

https://arxiv.org/abs/1605.01539

rank-cf, cont'd



rank-cf, cont'd



Benefit

Assuming that the mono-blocks are uniformly distributed over B: the expected # of c.m. is
$f \times 1 + (1-f)(1-f) \times 1 + f(1-f) \times 2 = 1 + f - f^2 \leq 2 - f$,
where the equality holds only for $f = 1$.

Conclusions

- Bioinformatics problems are often specific...

Conclusions

- Bioinformatics problems are often specific...
- thus (too) general algorithms are rarely competitive.

Conclusions

- Bioinformatics problems are often specific...
- thus (too) general algorithms are rarely competitive.
- Input representation matters!

Conclusions

- Bioinformatics problems are often specific...
- thus (too) general algorithms are rarely competitive.
- Input representation matters!
- “Constant time” is a flexible term.