Aligning DNA sequences on compressed collections of genomes

Part 2. Compressed indexing

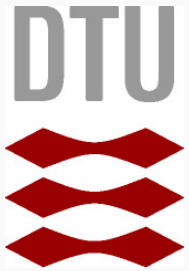
The CODATA-RDA Research Data Science Applied workshop on Bioinformatics

ICTP, Trieste - Italy

July 24-28, 2017

Nicola Prezza

Technical University of Denmark
DTU Compute
DK-2800 Kgs. Lyngby
Denmark



As seen in the previous lecture, any two Human genomes are 99.9% similar.

This suggests that storing a collection of genomes in plain format is not a smart solution

We need to resort to **data compression**

Compression

A compressor is an algorithm C that takes as input a file F and, exploiting repetitions and regularities in F, produces a file $C(F)$ that is often much smaller than F.

Decompression

Importantly, the process must be reversible: there must exist an algorithm D (a decompressor) such that $D(C(F)) = F$

Examples

- WinZip
- 7-Zip
- Bzip2
- ...

As seen, 1000 Human genomes $G_1, G_2, ..., G_{1000}$ require 3 TB to be stored. This can be expressed as

$$|G_1, G_2, ..., G_{1000}| \approx 3\ TB$$

Recall, however, that each genome carries only 3 MB of new information...

Our first **goal** is to find a good compressor C such that

$$|C(G_1, G_2, ..., G_{1000})| \approx 6\ GB$$

later, we will focus on indexing this compressed data

Compression of repetitive text collections

The research field studying the compression of repetitive text collections is very active. The main compression techniques used are:

- Lempel-Ziv parsing (LZ77)
- Grammar compression
- Burrows-Wheeler transform (BWT)

We will focus on BWT. First however, let's take a quick look at the other two techniques

Lempel-Ziv parsing

LZ77

- Lempel, Ziv, 1977. At the core of WinZip and 7-Zip
- Idea: divide the text into *phrases*. Each phrase is the shortest string that does not appear before in the text
- Compress each phrase storing position from where we copy it, length and last character

Example

$$\texttt{G = ACTACTACTACTACTACTACTGACTT}$$

$$\texttt{LZ77(G)} = \texttt{A|C|T|ACTACTACTACTACTACTG|ACTT|} \xrightarrow{compress} \langle \texttt{-,0,A} \rangle \langle \texttt{-,0,C} \rangle \langle \texttt{-,0,T} \rangle \langle \texttt{1,18,G} \rangle \langle \texttt{1,3,T} \rangle$$

$|\texttt{G}| = 26$ Bytes. $|\texttt{LZ77(G)}| = 15$ Bytes.

Lempel-Ziv parsing

How do we decompress $|\mathtt{LZ77(G)}|$ to obtain G?

Exercise

Decompress $\langle \mathtt{-,0,A} \rangle \langle \mathtt{-,0,T} \rangle \langle \mathtt{2,10,C} \rangle$

Grammar compression

The idea here is to replace groups of characters with new symbols, until we obtain only 1 symbol S (start symbol)

Example: text T = abababbabbababbabbabb

abababbabbababbabbabb	replace $X \rightarrow ab$
XXXbXbXXXbXb	replace $Y \rightarrow Xb$
XXYYXXYY	replace $Z \rightarrow XX$
ZYYZYY	replace $W \rightarrow ZY$
WYWY	replace $K \rightarrow WY$
KK	replace $S \rightarrow KK$
S	stop

grammar(T) = $\{S \rightarrow KK, X \rightarrow ab, Y \rightarrow Xb, Z \rightarrow XX, W \rightarrow ZY, K \rightarrow WY\}$

|grammar(T)| = 18 Bytes

Grammar compression

To decompress, just apply the rules backward from the start symbol. Stop expanding when there is no rule for a character.

Exercise

Decompress the following set of rules:

- $S \to KKG$
- $K \to XY$
- $X \to TA$
- $Y \to XX$

What is the resulting genome?

Exercise

Compress with LZ77 the genome obtained from the previous exercise. Which compressor compresses better the genome?

The Burrows-Wheeler transform

- Michael Burrows and David J Wheeler, 1994
- BWT is at the basis of today's most successful DNA compressed indexes: Bowtie, BWA, SOAP2, ...

Idea: we "mix" the letters in our genome G and obtain a string BWT(G) of the same length with the following interesting properties:

- We can reconstruct G from BWT(G)
- BWT(G) is "more compressible" than G

... How do we do this?

First try

The first idea could be to put the letters in alphabetic order:

CTCTCTCTCTCTCTCTCCTG$ $\rightarrow$ $CCCCCCCCCCCGTTTTTTTTT

$CCCCCCCCCCCGTTTTTTTTT is **very** compressible (can you suggest a technique to compress it?)

Unfortunately, we cannot reconstruct the original string from $CCCCCCCCCCCGTTTTTTTTT! Why?

note: we add a special character '$' at the end for reasons that will be explained later. $ is smaller than all other alphabet characters.

Definition: right-rotation of a word

we obtain a right-rotation of a word W if we take the last character of W and move it at the begin of W

Example

The right-rotation of `bioinformatics` is `sbioinformatic`

Rotations of a word

If you repeat this process, you obtain all rotations of the word

Imagine taking all rotations of our text and sorting them in alphabetic order ... what we just did was to take the first column (**F**)

$CTCTCTCTCTCTCTCTCCTG
CCTG$CTCTCTCTCTCTCTCT
CTCCTG$CTCTCTCTCTCTCT
CTCTCCTG$CTCTCTCTCTCT
CTCTCTCCTG$CTCTCTCTCT
CTCTCTCTCCTG$CTCTCTCT
CTCTCTCTCTCCTG$CTCTCT
CTCTCTCTCTCTCCTG$CTCT
CTCTCTCTCTCTCTCCTG$CT
CTCTCTCTCTCTCTCTCCTG$
CTG$CTCTCTCTCTCTCTCTC
G$CTCTCTCTCTCTCTCTCCT
TCCTG$CTCTCTCTCTCTCTC
TCTCCTG$CTCTCTCTCTCTC
TCTCTCCTG$CTCTCTCTCTC
TCTCTCTCCTG$CTCTCTCTC
TCTCTCTCTCCTG$CTCTCTC
TCTCTCTCTCTCCTG$CTCTC
TCTCTCTCTCTCTCCTG$CTC
TCTCTCTCTCTCTCTCCTG$C
TG$CTCTCTCTCTCTCTCTCC

Note: character '$' guarantees that all rotations are distinct and, therefore, their ordering is well defined.

The Burrows-Wheeler transform

What if we take the **last** column? (**L**)

$CTCTCTCTCTCTCTCTCCTG
CCTG$CTCTCTCTCTCTCTCT
CTCCTG$CTCTCTCTCTCTCT
CTCTCCTG$CTCTCTCTCTCT
CTCTCTCCTG$CTCTCTCTCT
CTCTCTCTCCTG$CTCTCTCT
CTCTCTCTCTCCTG$CTCTCT
CTCTCTCTCTCTCCTG$CTCT
CTCTCTCTCTCTCTCCTG$CT
CTCTCTCTCTCTCTCTCCTG$
CTG$CTCTCTCTCTCTCTCTC
G$CTCTCTCTCTCTCTCTCCT
TCCTG$CTCTCTCTCTCTCTC
TCTCCTG$CTCTCTCTCTCTC
TCTCTCCTG$CTCTCTCTCTC
TCTCTCTCCTG$CTCTCTCTC
TCTCTCTCTCCTG$CTCTCTC
TCTCTCTCTCTCCTG$CTCTC
TCTCTCTCTCTCTCCTG$CTC
TCTCTCTCTCTCTCTCCTG$C
TG$CTCTCTCTCTCTCTCTCC

This was exactly what Burrows and Wheeler proposed to do in 1994.

Last column = Burrows-Wheeler transform BWT(G) of G

BWT(G) = GTTTTTTTTT$CTCCCCCCCCC

$CTCTCTCTCTCTCTCTCCTG
CCTG$CTCTCTCTCTCTCTCT
CTCCTG$CTCTCTCTCTCTCT
CTCTCCTG$CTCTCTCTCTCT
CTCTCTCCTG$CTCTCTCTCT
CTCTCTCTCCTG$CTCTCTCT
CTCTCTCTCTCCTG$CTCTCT
CTCTCTCTCTCTCCTG$CTCT
CTCTCTCTCTCTCTCCTG$CT
CTCTCTCTCTCTCTCTCCTG$
CTG$CTCTCTCTCTCTCTCTC
G$CTCTCTCTCTCTCTCTCCT
TCCTG$CTCTCTCTCTCTCTC
TCTCCTG$CTCTCTCTCTCTC
TCTCTCCTG$CTCTCTCTCTC
TCTCTCTCCTG$CTCTCTCTC
TCTCTCTCTCCTG$CTCTCTC
TCTCTCTCTCTCCTG$CTCTC
TCTCTCTCTCTCTCCTG$CTC
TCTCTCTCTCTCTCTCCTG$C
TG$CTCTCTCTCTCTCTCTCC

BWT(G) = GTTTTTTTTT$CTCCCCCCCCCC

Notice anything? BWT(G) is not as good as the first column ($CCCCCCCCCCCGTTTTTTTTT), but very close!

- Property 1: the Burrows-Wheeler transform tends to produce **clusters of letters** if the text is repetitive
- Property 2: **BWT is reversible**. If I give you BWT(G), then you can reconstruct G (we will prove this later)

Run-length compression

$$\text{BWT(G)} = \text{GTTTTTTTTT\$CTCCCCCCCCC}$$

These clusters are called **equal-letter runs** (or just runs). In the example above, the number r of runs is $r = 6$

Run-length compressed Burrows-Wheeler transform

To compress: encode each run with its length and character:

$\langle G, 1\rangle,\ \langle T, 8\rangle,\ \langle \$, 1\rangle,\ \langle C, 1\rangle,\ \langle T, 1\rangle,\ \langle C, 9\rangle$

Compressed size = 12 Bytes. Original size = 21 Bytes

Repetitive texts

It can be proved that if the text has a lot of repetitions, the BWT has few runs (and the other way round).

As a result, we can say that a text is **repetitive** if the number of runs r in its BWT is small with respect to the text's length n:

- Repetitiveness of the text: $100 \cdot (1 - r/n)$
- Size of the compressed text: r

In the previous example, repetitiveness $= 100 \cdot (1 - 12/21) = 42\%$.

Exercise

- Compute the repetitiveness of your name
- Who has the most repetitive name?

Inverting the BWT

How to invert the BWT?

LF mapping

Property: the i-th character equal to x in column L and the i-th character equal to x in column F are the same position in the text

Example

In the example below, the second 'C' in column L (position 6) corresponds, in the text, to the second 'C' in column F (position 4) : both are followed by "TAT$".

1	$CCTAT
2	AT$CCT
3	CCTAT$
4	CTAT$C
5	T$CCTA
6	TAT$CC

Inverting the BWT

Note: after jumping from position 6 in column L to position 4 in column F, go to position 4 in column L. We just walked back of one position in the text.

1	$CCTAT
2	AT$CCT
3	CCTAT$
4	CTAT$C
5	T$CCTA
6	TAT$CC

Idea: to invert BWT, start from character $ and apply the LF mapping until we see $ again.

Note: we have only column L = BWT. We can however easily reconstruct also column F.

Inverting the BWT

1	$	...	T
2	A	...	T
3	C	...	$
4	C	...	C
5	T	...	A
6	T	...	C

- Begin: position 3 in column L (where $ is)
- LF mapping: $3 \rightarrow 1$

$

Inverting the BWT

1	$ ...	T
2	A ...	T
3	C ...	$
4	C ...	C
5	T ...	A
6	T ...	C

- In position 1 of L we read T.
- LF mapping: $1 \rightarrow 5$

T$

Inverting the BWT

1	$	...	T
2	A	...	T
3	C	...	$
4	C	...	C
5	T	...	A
6	T	...	C

- In position 5 of L we read A.
- LF mapping: $5 \rightarrow 2$

AT$

Inverting the BWT

1	$	...	T
2	A	...	T
3	C	...	$
4	C	...	C
5	T	...	A
6	T	...	C

- In position 2 of L we read T.
- LF mapping: $2 \rightarrow 6$

TAT$

Inverting the BWT

1	$	...	T
2	A	...	T
3	C	...	$
4	C	...	C
5	T	...	A
6	T	...	C

- In position 6 of L we read C.
- LF mapping: $6 \rightarrow 4$

CTAT$

Inverting the BWT

1	$	...	T
2	A	...	T
3	C	...	$
4	C	...	C
5	T	...	A
6	T	...	C

- In position 4 of L we read C.
- LF mapping: $4 \rightarrow 3$
- characters read until now: `CCTAT$`

CCTAT$

Inverting the BWT

1	$	...	T
2	A	...	T
3	C	...	$
4	C	...	C
5	T	...	A
6	T	...	C

- In position 3 of L we read $. STOP

inverse of BWT(TT$CAC) = **CCTAT$**

Inverting the BWT

Exercise

Invert the following BWT: `elnwleod$`

Compressed text indexes

BWT as an index

Is this all? is the BWT useful only for compression?

Absolutely not! the BWT is a **full-text index**

BWT as an index

Suppose you want to find all occurrences of `ACT`. Note: they form a **range** in the first columns! (rows 2-4)

1	$ACTAGTACTGACTGCTGCGGT
2	ACTAGTACTGACTGCTGCGGT$
3	ACTGACTGCTGCGGT$ACTAGT
4	ACTGCTGCGGT$ACTAGTACTG
5	AGTACTGACTGCTGCGGT$ACT
6	CGGT$ACTAGTACTGACTGCTG
7	CTAGTACTGACTGCTGCGGT$A
8	CTGACTGCTGCGGT$ACTAGTA
9	CTGCGGT$ACTAGTACTGACTG
10	CTGCTGCGGT$ACTAGTACTGA
11	GACTGCTGCGGT$ACTAGTACT
12	GCGGT$ACTAGTACTGACTGCT
13	GCTGCGGT$ACTAGTACTGACT
14	GGT$ACTAGTACTGACTGCTGC
15	GT$ACTAGTACTGACTGCTGCG
16	GTACTGACTGCTGCGGT$ACTA
17	T$ACTAGTACTGACTGCTGCGG
18	TACTGACTGCTGCGGT$ACTAG
19	TAGTACTGACTGCTGCGGT$AC
20	TGACTGCTGCGGT$ACTAGTAC
21	TGCGGT$ACTAGTACTGACTGC
22	TGCTGCGGT$ACTAGTACTGAC

BWT as an index

Number of occurrences of ACT = 4-2+1 = 3

1	$ACTAGTACTGACTGCTGCGGT
2	ACTAGTACTGACTGCTGCGGT$
3	ACTGACTGCTGCGGT$ACTAGT
4	ACTGCTGCGGT$ACTAGTACTG
5	AGTACTGACTGCTGCGGT$ACT
6	CGGT$ACTAGTACTGACTGCTG
7	CTAGTACTGACTGCTGCGGT$A
8	CTGACTGCTGCGGT$ACTAGTA
9	CTGCGGT$ACTAGTACTGACTG
10	CTGCTGCGGT$ACTAGTACTGA
11	GACTGCTGCGGT$ACTAGTACT
12	GCGGT$ACTAGTACTGACTGCT
13	GCTGCGGT$ACTAGTACTGACT
14	GGT$ACTAGTACTGACTGCTGC
15	GT$ACTAGTACTGACTGCTGCG
16	GTACTGACTGCTGCGGT$ACTA
17	T$ACTAGTACTGACTGCTGCGG
18	TACTGACTGCTGCGGT$ACTAG
19	TAGTACTGACTGCTGCGGT$AC
20	TGACTGCTGCGGT$ACTAGTAC
21	TGCGGT$ACTAGTACTGACTGC
22	TGCTGCGGT$ACTAGTACTGAC

With some more effort, we can also find the corresponding text positions (we will not see how).

BWT as an index

How do we find the range? again LF mapping: the **backward search algorithm**

The backward search algorithm

- Recall that we store only the first and last columns
- We search backwards (from last to first character of ACT)
- START: ACT. range of T: 17-22 (we find it looking at column F)

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

The backward search algorithm

Now we want to extend with C: `ACT`

Idea: letters in column L in range 17-22 are those that precede T's in the text.

Map with LF only the 'C'

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

The backward search algorithm

We found the range of CT! (in orange)

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

The backward search algorithm

Now we want to extend with A: ACT

Again, letters in range 7-10 are those that precede CT in the text.

Map the 'A' inside range 7-10 from L to F

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

The backward search algorithm

Result: range of `ACT`.

STOP.

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

Compressed indexing

Key points

- Note that we store in memory only columns L (the BWT) and F
- Most importantly, both columns are stored **compressed**
- We can store very small additional data structures so that function LF runs very fast (i.e. few nanoseconds)

Compressed indexing

In practice this means that a BWT index on the Human genome (3Gbp) takes approximately 1 GB and accelerates pattern matching by millions of times (with respect to simply scanning the text)

Compressed indexing

Result

The BWT (with small additional structures) is a compressed full-text index.

This index is known as **FM-index** and was first described in:

Ferragina, Paolo, and Giovanni Manzini. "Opportunistic data structures with applications." Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on. IEEE, 2000.

Compressed indexes for DNA alignment

The FM index stands at the core of the most recent DNA aligners:

- BWA
- Bowtie
- ERNE 2
- SOAP 2
- ...

In the practical section of today we will take a closer look to these DNA aligners

Current research

Until this year, the most advanced versions of the FM index were not yet ready to handle repetitive texts:

- Space proportional to $r + n/s$ (where s is a parameter)
- Locate time proportional to $m + occ \cdot s$

The problem has been recently (May 2017) solved: the **r-index**[1]

- Space proportional to r
- Locate time proportional to $m + occ$ (optimal)

[1] Travis Gagie, Gonzalo Navarro, Nicola Prezza. "Optimal-Time Text Indexing in BWT-runs Bounded Space". arXiv:1705.10382. 2017