On Undetected Redundancy in the Burrows-Wheeler Transform

Uwe Baier

Institute of Theoretical Computer Science
Ulm University

Data Compression

Why should we compress our data?...

...people from informatics should know best themselves...

- huge amount of data, storage can be expensive
- not every method can be implemented using streaming and parallelism $\Rightarrow$ memory is an even more limited resource
- some compressed representations allow methods of string analysis to be performed much faster

In this talk

- BWT Preliminaries
- Tunneled BWT
- Practical Implementation
- Experimental Results
- Conclusion

Context-Based Compression

Observation: similar contexts tend to be succeeded (or preceded) by similar characters

- In english texts, the letter `q` always is followed by an `u`
- The string `eer` tends to be preceded by a `b`

Can we use this knowledge to compress data?

⇒ Burrows-Wheeler Transform [Burrows and Wheeler, 1994]

BWT and sorted suffixes of $S =$ `easypeasy$`

```
y   $
e   asy$
e   asypeasy$
p   easy$
$   easypeasy$
y   peasy$
a   sy$
a   sypeasy$
s   y$
s   ypeasy$
```

BWT - What is it?

"The BWT L *is a string generated by concatenating all cyclic preceding characters of the lexicographically sorted suffixes of a string S."*

BWT generation of $S = \texttt{easypeasy\$}$

prec. char.	suffixes
y	$
s	y$
a	sy$
e	asy$
p	easy$
y	peasy$
s	ypeasy$
a	sypeasy$
e	asypeasy$
$	easypeasy$

BWT - What is it?

"The BWT L *is a string generated by concatenating all cyclic preceding characters of the lexicographically sorted suffixes of a string S."*

BWT generation of $S = \texttt{easypeasy\$}$

prec. char.	suffixes		L	sorted suffixes
y	$		y	$
s	y$		e	asy$
a	sy$		e	asypeasy$
e	asy$	sort →	p	easy$
p	easy$		$	easypeasy$
y	peasy$		y	peasy$
s	ypeasy$		a	sy$
a	sypeasy$		a	sypeasy$
e	asypeasy$		s	y$
$	easypeasy$		s	ypeasy$

BWT - What is it?

"The BWT L *is a string generated by concatenating all cyclic preceding characters of the lexicographically sorted suffixes of a string S."*

BWT generation of $S = \texttt{easypeasy\$}$

prec. char.	suffixes
y	$
s	y$
a	sy$
e	asy$
p	easy$
y	peasy$
s	ypeasy$
a	sypeasy$
e	asypeasy$
$	easypeasy$

sort →

L	sorted suffixes
y	$
e	asy$
e	asypeasy$
p	easy$
$	easypeasy$
y	peasy$
a	sy$
a	sypeasy$
s	y$
s	ypeasy$

BWT - Use for Data Compression?

- BWT places characters preceding the same context near to each other

```
                        y   $
preceding characters    e   asy$          Context
                        e   asypeasy$
                        p   easy$
                        ⋮   ⋮
```

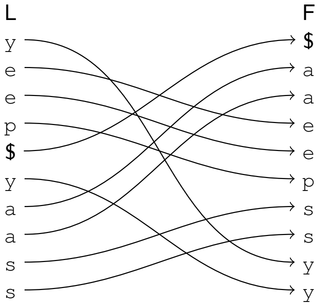
- character distribution of small portions of BWT is skew

Common Compression Approaches

- transform local to global skewness (MTF [Ryabko, 1980]) + entropy coding (Huffman-Coding [Huffman, 1952])
- run-length-encoding $\cdots\underbrace{\texttt{aaaaaa}}_{\text{6 times}}\cdots \Rightarrow \cdots\underbrace{\texttt{a01}}_{6=(101)_2}\cdots$

BWT - Inverting

- generate F (first characters of sorted suffixes) by sorting L



- *k*-th occurence of character *c* in L corresponds to *k*-th occurence of character *c* in F

⇒ collecting characters in L during a walk through L using correspondence yields the reversed original sequence

BWT - Inverting Example

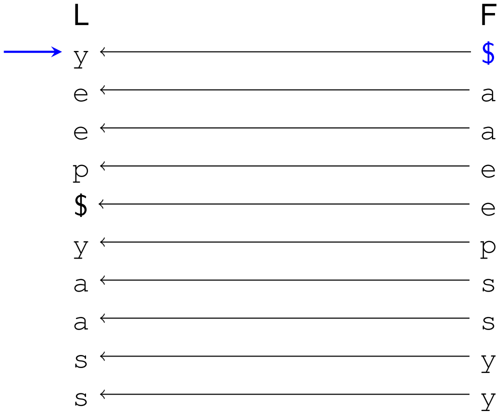
k-th occurence of character c in L *corresponds to*
k-th occurence of character c in F

L		F
y	←	$
e	←	a
e	←	a
p	←	e
$	←	e
y	←	p
a	←	s
a	←	s
s	←	y
s	←	y

$S =$ $

BWT - Inverting Example

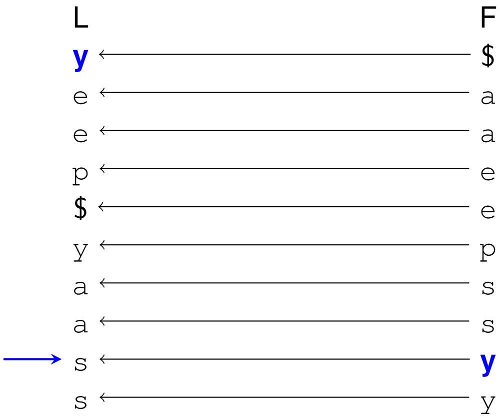
k-th occurence of character c in L *corresponds to*
k-th occurence of character c in F



$S =$ y$

BWT - Inverting Example

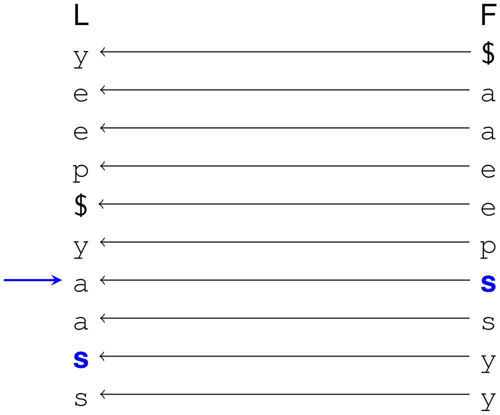
k-th occurence of character c in L *corresponds to*
k-th occurence of character c in F



$S =$ sy$

BWT - Inverting Example

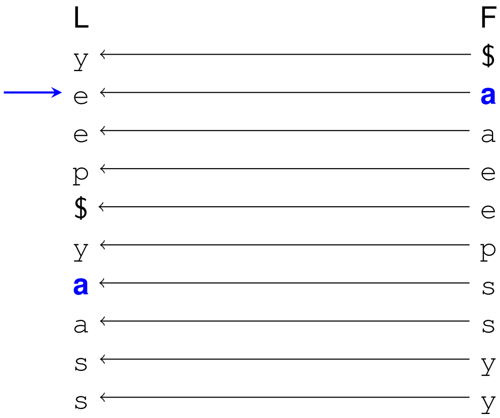
k-th occurence of character c in L *corresponds to*
k-th occurence of character c in F



$S =$ asy$

BWT - Inverting Example

k-th occurence of character c in L *corresponds to*
k-th occurence of character c in F



$S =$ `easy$`

BWT - Inverting Example

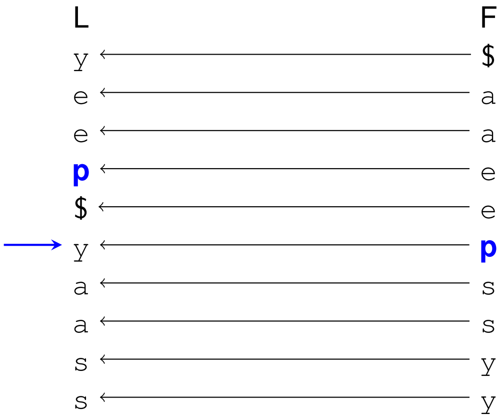
k-th occurence of character c in L *corresponds to*
k-th occurence of character c in F

	L		F
	y	←	$
	e	←	a
	e	←	a
→	p	←	**e**
	$	←	e
	y	←	p
	a	←	s
	a	←	s
	s	←	y
	s	←	y

$S =$ `peasy$`

BWT - Inverting Example

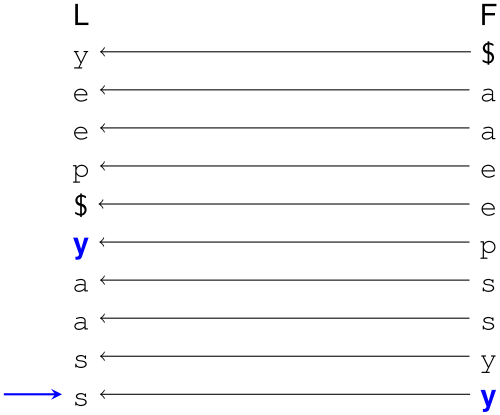
k-th occurence of character c in L *corresponds to*
k-th occurence of character c in F



$$S = \quad \texttt{ypeasy\$}$$

BWT - Inverting Example

k-th occurence of character c in L *corresponds to*
k-th occurence of character c in F



$S =$ sypeasy$

BWT - Inverting Example

k-th occurence of character c in L *corresponds to*
k-th occurence of character c in F

L		F
y	←	$
e	←	a
e	←	a
p	←	e
$	←	e
y	←	p
a	←	s
→ a	←	**s**
s	←	y
s	←	y

$S =$ asypeasy$

BWT - Inverting Example

k-th occurence of character c in L *corresponds to*
k-th occurence of character c in F

L	F
y	$
e	a
e	**a**
p	e
$	e
y	p
a	s
a	s
s	y
s	y

$$S = \texttt{easypeasy\$}$$

BWT - Inverting Example

k-th occurence of character c in L *corresponds to*
k-th occurence of character c in F

	L		F
	y	←	$
	e	←	a
	e	←	a
	p	←	e
end →	$	←	**e**
	y	←	p
	a	←	s
	a	←	s
	s	←	y
	s	←	y

$$S = \texttt{easypeasy\$}$$

Observation on Contexts

similar contexts tend to be preceded by the same character
$\Rightarrow$ similar contexts tend to be preceded by the same substrings

sorted suffixes and corr. prefixes of $S = \texttt{easypeasy\$}$

easypeasy	\$
easype	asy\$
e	asypeasy\$
easyp	easy\$
ε	easypeasy\$
easy	peasy\$
easypea	sy\$
ea	sypeasy\$
easypeas	y\$
eas	ypeasy\$

- Can we use this?

BWT ”Tunneling“

1. determine a set of blocks (≙ equal consecutive preceding substrings) to be tunneled

```
         ⋮       ⋮
        ea      sypeasy$
  easypeas      y$
       eas      ypeasy$
```

2. determine the corresponding columns in L and F for each block
3. cross out all entries from the columns in L and F, except for the uppermost ones
4. remove positions which were crossed out both in F and L
5. result: shortened L and two bitvectors cntL and cntF saving the remaining crosses

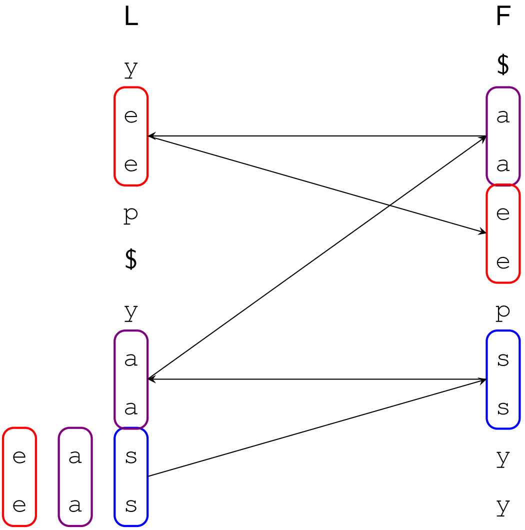
BWT Tunneling - Example

1. determine a set of blocks (≙ equal consecutive preceding substrings) to be tunneled

		L	F
		y	$
		e	a
		e	a
		p	e
		$	e
		y	p
		a	s
		a	s
e	a	s	y
e	a	s	y

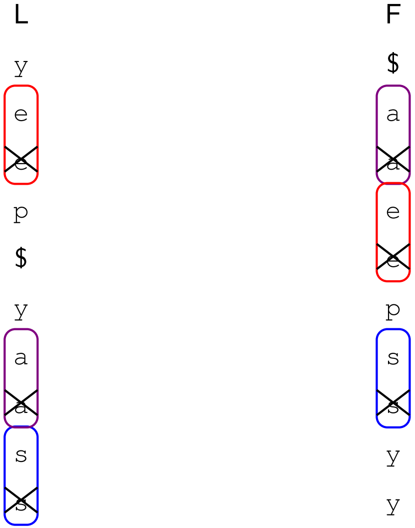
BWT Tunneling - Example

2. determine the corresponding columns in L and F for each block



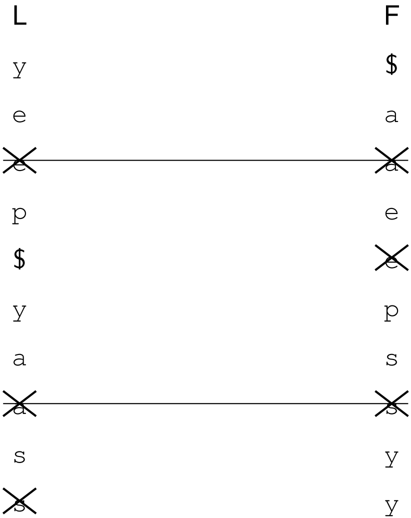
BWT Tunneling - Example

3. cross out all entries from the columns in L and F, except for the uppermost ones



BWT Tunneling - Example

4. remove positions which were crossed out both in F and L



BWT Tunneling - Example

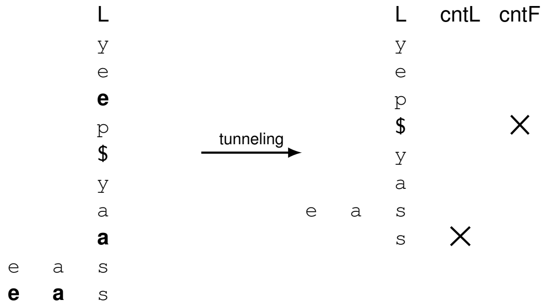
5. result: shortened L and two bitvectors cntL and cntF saving the remaining crosses

L	cntL	cntF
y		
e		
p		
$		×
y		
a		
s		
s	×	

Tunneling - Recap

tunneling removes all entries from a block except for

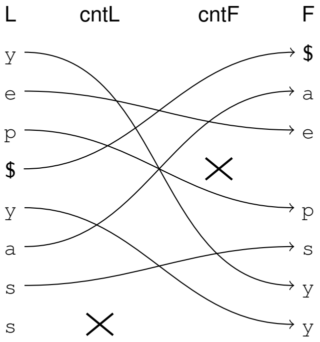
- the uppermost row
- the rightmost column



- tunneling reduces run-lengths in L at cost of increasing the number of runs in cntL and cntF - is it worth it?
- Can we invert a tunneled BWT?

Tunneled BWT - Inverting

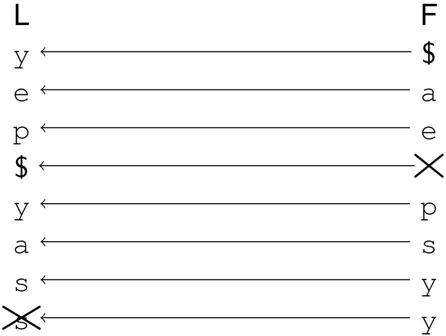
- sort regular characters in L to free places in F



- *k*-th occurence of character *c* in L corresponds to *k*-th occurence of character *c* in F
- use uppermost row of a tunnel for all rows of a block
- when entering a tunnel, save offset to uppermost row to get back to correct “lane” after tunnel

Tunneled BWT - Inverting Example

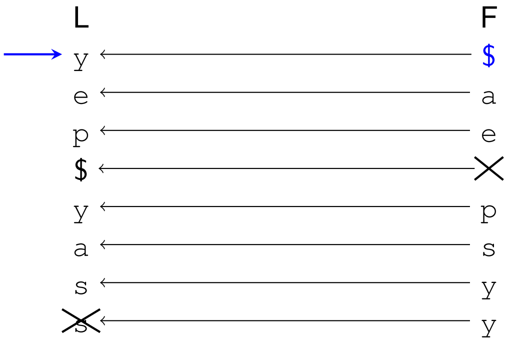
k-th occurence of character c in L *corresponds to*
k-th occurence of character c in F



$S =$ $

Tunneled BWT - Inverting Example

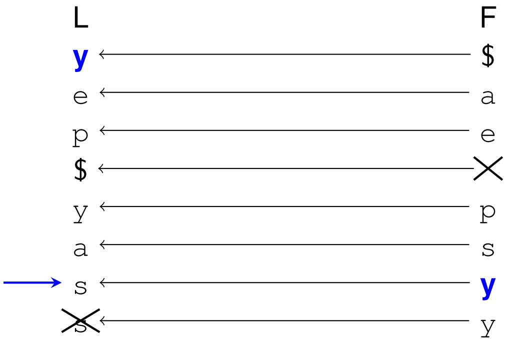
k-th occurence of character c in L *corresponds to*
k-th occurence of character c in F



$S =$ y\$

Tunneled BWT - Inverting Example

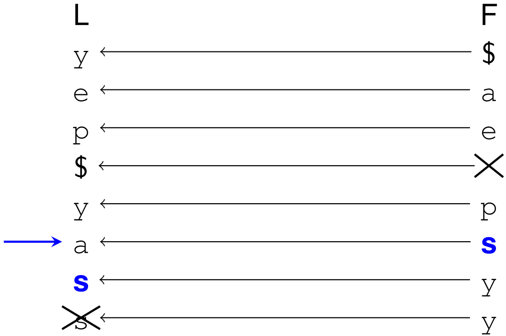
k-th occurence of character c in L *corresponds to*
k-th occurence of character c in F



$S =$ sy$

Tunneled BWT - Inverting Example

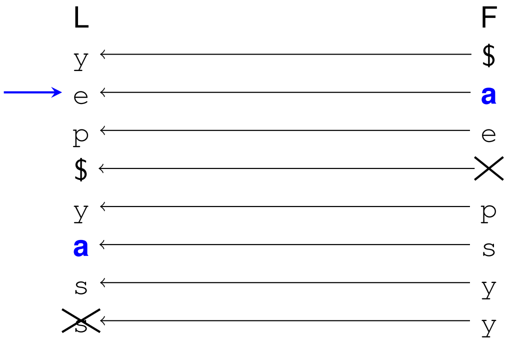
k-th occurence of character c in L *corresponds to*
k-th occurence of character c in F



$S =$ asy$

Tunneled BWT - Inverting Example

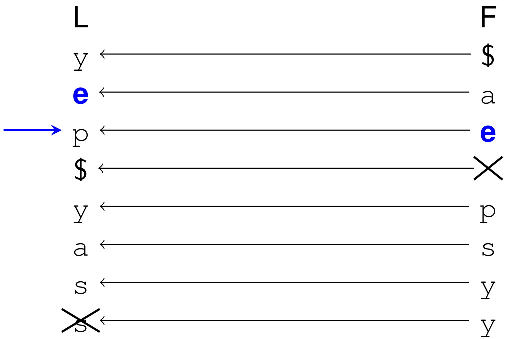
k-th occurence of character c in L *corresponds to*
k-th occurence of character c in F



$S =$ easy$

Tunneled BWT - Inverting Example

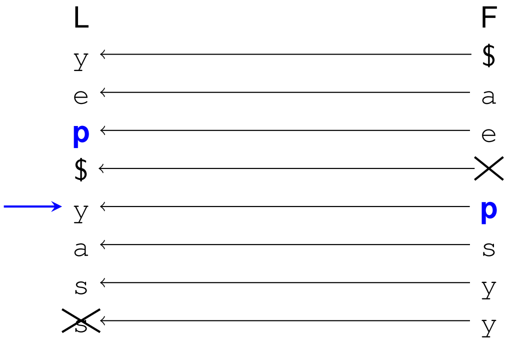
k-th occurence of character c in L *corresponds to*
k-th occurence of character c in F



$S =$ peasy$

Tunneled BWT - Inverting Example

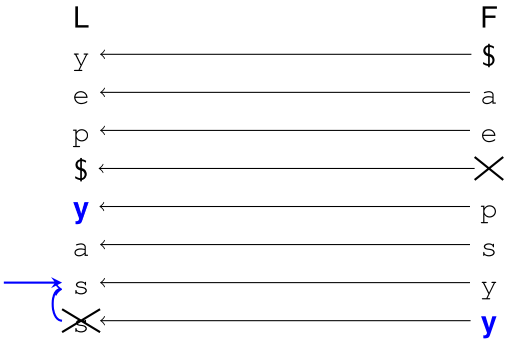
k-th occurence of character c in L *corresponds to*
k-th occurence of character c in F



$S =$ ypeasy$

Tunneled BWT - Inverting Example

k-th occurence of character c in L *corresponds to*
k-th occurence of character c in F

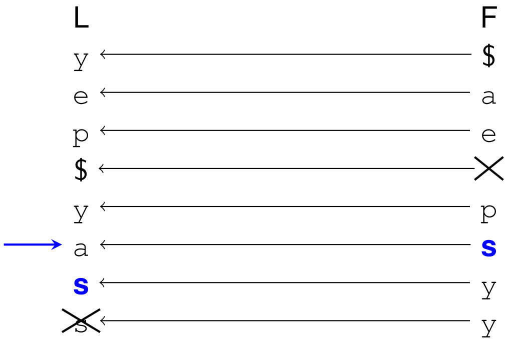


tunnel start detected ⇒ switch to uppermost row
offset = 1

$S =$ `sypeasy$`

Tunneled BWT - Inverting Example

k-th occurence of character c in L *corresponds to*
k-th occurence of character c in F

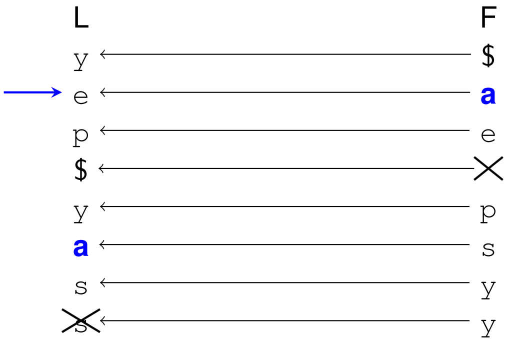


offset = 1

$S =$ asypeasy$

Tunneled BWT - Inverting Example

k-th occurence of character c in L *corresponds to k-th occurence of character c in* F

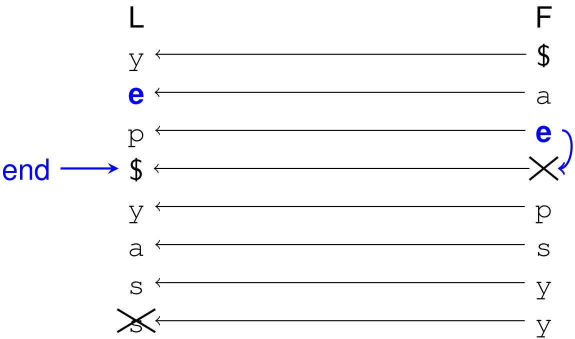


offset = 1

$S =$ easypeasy$

Tunneled BWT - Inverting Example

k-th occurence of character c in L *corresponds to*
k-th occurence of character c in F



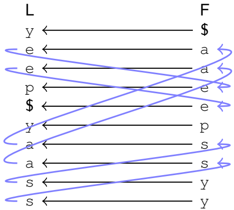
tunnel end detected ⇒ switch back using offset

offset = 1

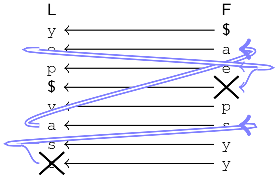
$S = \texttt{easypeasy\$}$

Tunneled BWT Inverting - Recap

Normal BWT



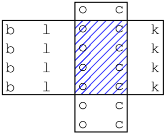
Tunneled BWT



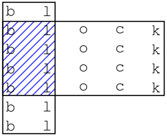
- uppermost row is used for all rows of a block
- offset is stored to ”get back“ to correct lane

Block Collisions

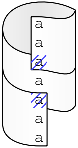
Compensable



Critical



Critical



- compensable collisions: cross overlay
- offset is stored on a stack

Practice: Considered Blocks

Consider only width-maximal run-based blocks:
block height is equal to the height of runs it starts and ends in

run $\hat{=}$ length-maximal repeat of same character

$$\cdots\texttt{aaxx}\underbrace{\texttt{xxxx}}_{\text{no run}}\texttt{xbbc}\underbrace{\texttt{aaaaaa}}_{\text{run}}\texttt{cc}\cdots$$

Result:

- only compensable collisions
- bitvectors cntL and cntF can be merged to one vector aux with alphabet size 3
- aux can be shortened to work run-based: only 1 symbol per run required

Practice: Block Choice

- choice depends on compression of L and aux
- L and aux come frome the same source
  $\Rightarrow$ compress both with same BWT backend encoder
- allows to abstract choice from used backend encoder

Greedy run-length-encoding strategy

- encoding size of run-length-encoded L and aux can be estimated
- greedy strategy: assign each block a score ($\hat{=}$ number of bits removed from L-encoding)
  - choose block with highest score
  - decrease score of colliding blocks with lower score
- result: “sorted list” of blocks
- tunnel score-highest blocks which give best tradeoff between benefit and aux encoding size
- works good as long as backend encoders also use run-length-encoding (or something similar)

Experiments: Overview

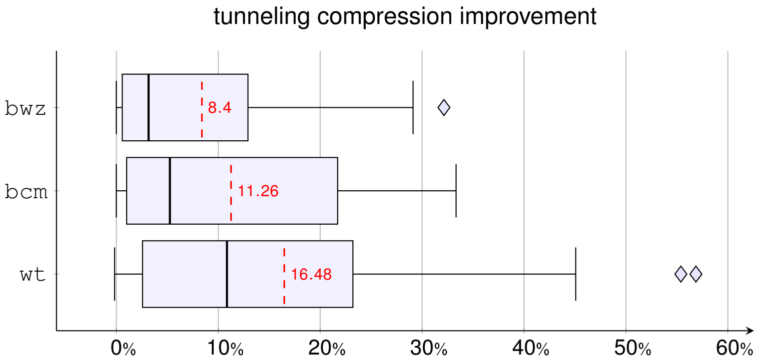
BWT compressors enhanced with tunneling

- `bwz`: original scheme by Burrows & Wheeler ($\approx$ `bzip2`)
- `bcm`: one of the best open-source BWT compressors
- `wt`: wavelet tree using hybrid bitvectors

Test Data

- Silesia Corpus: contains 12 files (6 - 49 MB)
- Pizza & Chili Corpus: contains 6 files (54 - 1130 MB)
- Repetitive Corpus: contains 9 files (45 - 446 MB)

Comparison: normal vs. tunneled BWT



- average encoding size decrease about 8 – 16 %
- peak encoding size decrease about 33 – 58 %

Comparison to other Compressors

- `xz`: uses LZMA, similar to `7-zip`
- `zpaq`: uses context mixing
- all values are measured in bits per symbol

Compressor	Silesia Corpus			Pizza & Chili Corpus			Repetitive Corpus		
	nci (32 MB)	samba (21 MB)	webster (40 MB)	proteins (1130 MB)	dna (386 MB)	english (1024 MB)	coreutils (196 MB)	para (410 MB)	world-leaders (45 MB)
`bwz`	0.34	1.81	1.48	2.29	1.83	1.84	0.23	0.31	0.12
`bwz-tunneled`	0.33	1.75	1.48	2.00	1.81	1.66	0.17	0.21	0.11
`bcm`	0.29	1.49	1.24	2.33	1.72	1.56	0.23	0.32	0.13
`bcm-tunneled`	**0.28**	1.42	1.24	**1.95**	**1.70**	**1.34**	0.16	0.21	0.11
`wt`	0.61	2.70	2.08	3.97	2.05	2.45	0.69	0.49	0.40
`wt-tunneled`	0.54	2.45	2.07	2.72	2.03	1.99	0.38	0.42	0.29
`xz`	0.35	1.38	1.61	2.22	1.78	1.93	**0.14**	**0.11**	**0.09**
`zpaq`	0.36	**1.20**	**1.21**	2.61	1.86	1.64	0.62	1.85	0.09

Conclusion

Tunneling works nice...

- natural way to extend context-based compression to longer strings
- significant BWT compression improvement
- same or less resource requirements for decoding BWT

... but has some problems:

- block choice under collisions is not always optimal
- current block choice strategy is too complicated
- heavy resource requirements for encoding (memory peak and time double)

Future research goals

- try simpler block choice strategies
- examine hardness of optimal block choice
- prepare tunneling for text indexing

Questions

?

References I

Uwe Baier.
Tunneled BWT Implementation and Benchmark.
https://github.com/waYne1337/tbwt.
last visited January 2018.

Uwe Baier.
On Undetected Redundancy in the Burrows-Wheeler Transform.
https://arxiv.org/abs/1804.01937, 2018.

Michael Burrows and David J Wheeler.
A block-sorting lossless data compression algorithm.
Technical Report 124, Digital Equipment Corporation, 1994.

References II

Sebastian Deorowicz.
Silesia Corpus.
http://sun.aei.polsl.pl/~sdeor/index.php?page=silesia.
last visited January 2018.

Paolo Ferragina and Gonzalo Navarro.
Pizza & Chili Corpus.
http://pizzachili.dcc.uchile.cl/texts.html.
last visited January 2018.

Paolo Ferragina and Gonzalo Navarro.
Repetitive Corpus.
http://pizzachili.dcc.uchile.cl/repcorpus.html.
last visited January 2018.

References III

Luca Foschini, Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter.

When Indexing Equals Compression: Experiments with Compressing Suffix Arrays and Applications.

ACM Transactions on Algorithms, 2(4):611–639, 2006.

Simon Gog.

`sdsl-lite` Library.

`https://github.com/simongog/sdsl-lite`.

last visited January 2018.

David A. Huffman.

A Method for the Construction of Minimum-Redundancy Codes.

Proceedings of the IRE, 40(9):1098–1101, 1952.

References IV

- Juha Kärkkainen, Dominik Kempa, and Simon J. Puglisi.
  Hybrid Compression of Bitvectors for the FM-Index.
  In Proceedings of the 2014 Data Compression Conference, DCC '14, pages 302–311, 2014.

- Matt Mahoney.
  `zpaq` File Compressor.
  `http://mattmahoney.net/dc/zpaq.html`.
  last visited January 2018.

- Ilya Muravyov.
  `bcm` File Compressor.
  `https://github.com/encode84/bcm`.
  last visited January 2018.

References V

B. Ya Ryabko.
Data compression by means of a “book stack”.
Problems of Information Transmission, 16:265–269, 1980.

Tukaani.
`xz` File Compressor.
https://tukaani.org/xz/.
last visited January 2018.