Document Retrieval on Repetitive Collections

Jouni Sirén, University of Chile
with
Gonzalo Navarro, University of Chile
Simon J. Puglisi, University of Helsinki

Document retrieval

- We have a collection of d documents (strings) of total length n.
- We want to list those documents that contain pattern P as a substring.
- We are interested on the actual performance and space usage of the algorithms on real data.
- This work was inspired by observations that dedicated methods are often worse than brute force.

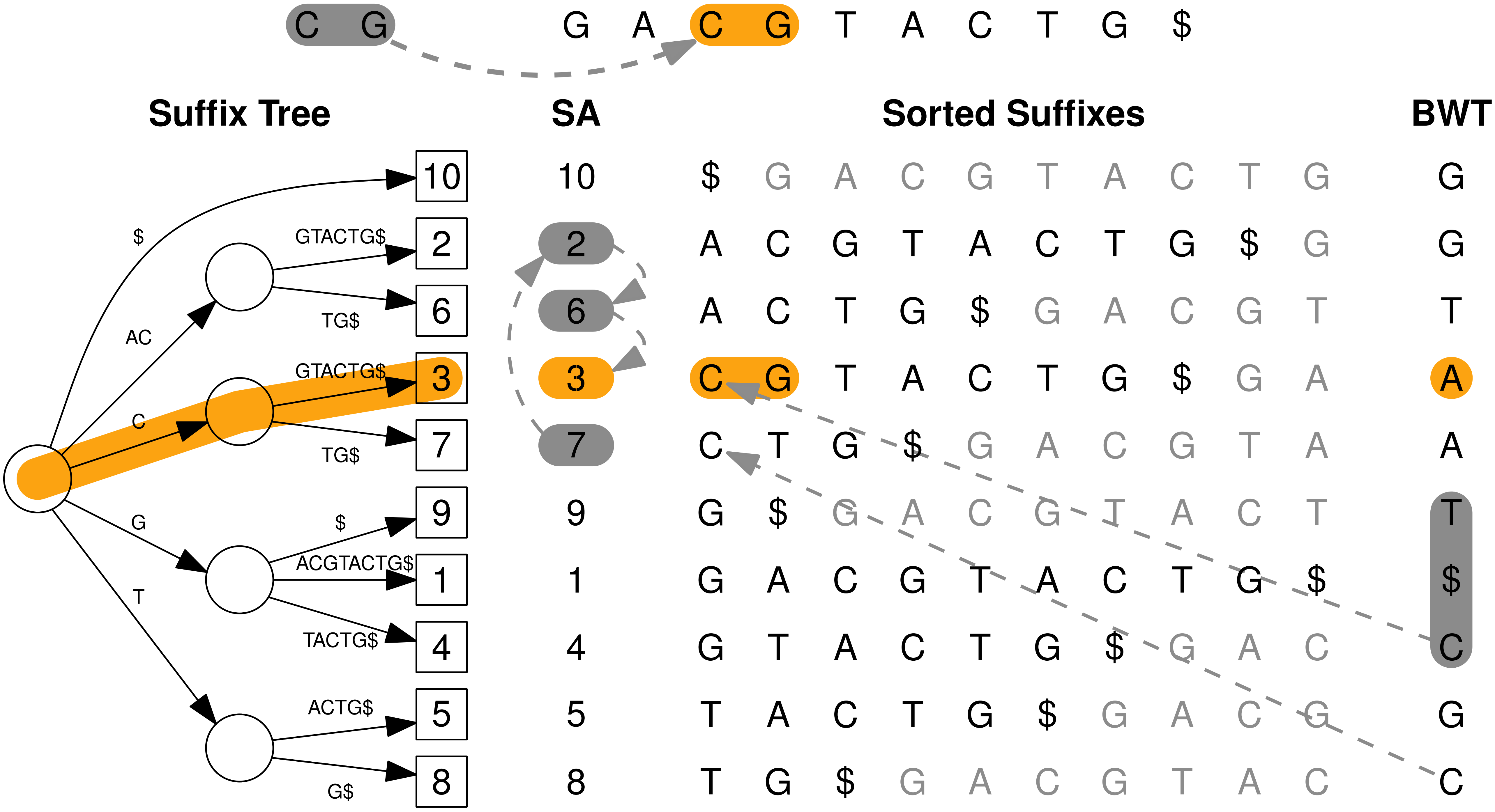
Query	**Definition**	**Ideal time complexity**
locate(P)	Find all occurrences of pattern P in all documents.	occ
list(P)	Find all documents that contain pattern P.	docc
topk(P, k)	Find the k documents that contain the most occurrences of P.	k

list vs. topk

- In list, the performance of dedicated methods depends on the occ/docc ratio. This depends on the documents themselves, but not on the size of the collection.

- In topk, the relevant ratio is docc/k, which depends on the size of the collection. Large collections demand different methods than small ones.

- I will concentrate on list in this talk.



SA	Sorted Suffixes	BWT
10	$ G A C G T A C T G	G
2	A C G T A C T G $ G	G
6	A C T G $ G A C G T	T
3	C G T A C T G $ G A	A
7	C T G $ G A C G T A	A
9	G $ G A C G T A C T	T
1	G A C G T A C T G $	$
4	G T A C T G $ G A C	C
5	T A C T G $ G A C G	G
8	T G $ G A C G T A C	C

Solving locate

- The suffix tree and the suffix array are much larger than the original text, limiting their usefulness. They solve locate essentially in $O(|P| + occ)$ time.

- The FM-index (Ferragina and Manzini, 2005) and the compressed suffix array (Grossi and Vitter, 2005) are based on the Burrows-Wheeler transform. Using $O(n/s)$ words of extra space in addition to the compressed BWT, they solve locate in $O(|P| + s \cdot occ)$ time.

- Most solutions for list use a CSA or an FMI as the basic structure.

Bitvectors

- Bitvectors are the main building block of succinct and compressed data structures. They consist of a binary sequence with extra structures to support rank and select.

- rank(B, i) = ∑ B[1,i], while select(B, i) is the inverse.

- The number of rank/select operations predicts actual performance quite well.

- Many different encodings: plain, entropy-compressed, gap encoded, run-length encoded, grammar-compressed.

Brute-L and Brute-D

- We use gap encoded bitvector B marking document boundaries to convert text positions into document identifiers. This takes $O(d \log n)$ bits.

- Brute-L uses the bitvector to convert the results of locate, and then filters out duplicates. Overall time complexity is $O(|P| + s \cdot occ + sort(occ))$.

- Brute-D converts the suffix array into the document array DA. It solves list in $O(|P| + sort(occ))$ time using $n \log d$ bits of extra space.

Muthukrishnan's algorithm

- Muthukrishnan's algorithm (2002) finds the first occurrence of each document in the query range.
- C[i] points to the previous occurrence of DA[i]. If C[i] is outside the query range, DA[i] is the first occurrence of that document identifier.
- Uses range minimum queries over C to find the smallest values recursively.
- Time complexity is $O(|P| + docc)$.

Query list(”m”)

1. Find the query range: [14, 20].
2. Find the minimum value in C[14, 20]: C[14].
3. If C[14] ≥ 14 (original sp), stop.
4. Report D[14].
5. Continue to [14, 13] and [15, 20].

Row	C	D	Suffix
1	0	1	$
2	0	2	$
3	0	3	$
4	2	2	al$
5	3	3	e$
6	4	2	imal$
7	5	3	imize$
8	1	1	imum$
9	6	2	inimal$
10	7	3	inimize$
11	8	1	inimum$
12	10	3	ize$
13	9	2	l$
14	11	1	m$
15	13	2	mal$
16	15	2	minimal$
17	12	3	minimize$
18	14	1	minimum$
19	17	3	mize$
20	18	1	mum$
21	16	2	nimal$
22	19	3	nimize$
23	20	1	nimum$
24	23	1	um$
25	22	3	ze$

Sadakane's improvements

- Sadakane (2007) improved the space usage of Muthukrishnan's algorithm.
- Array C is not needed, if the recursion is done in preorder from left to right. DA[i] is the first occurrence, if it has not been encountered before.
- Document array can be replaced by bitvector B.
- RMQ needs just $2n + o(n)$ bits (Fischer, 2010).

Sada-X-Y

- Sada-C-L solves list in $O(|P| + s \cdot docc)$ time using $2n + o(n) + O(d \log n)$ bits of extra space.

- Sada-C-D solves list in $O(|P| + docc)$ time using $2n + o(n) + n \log d$ bits of extra space.

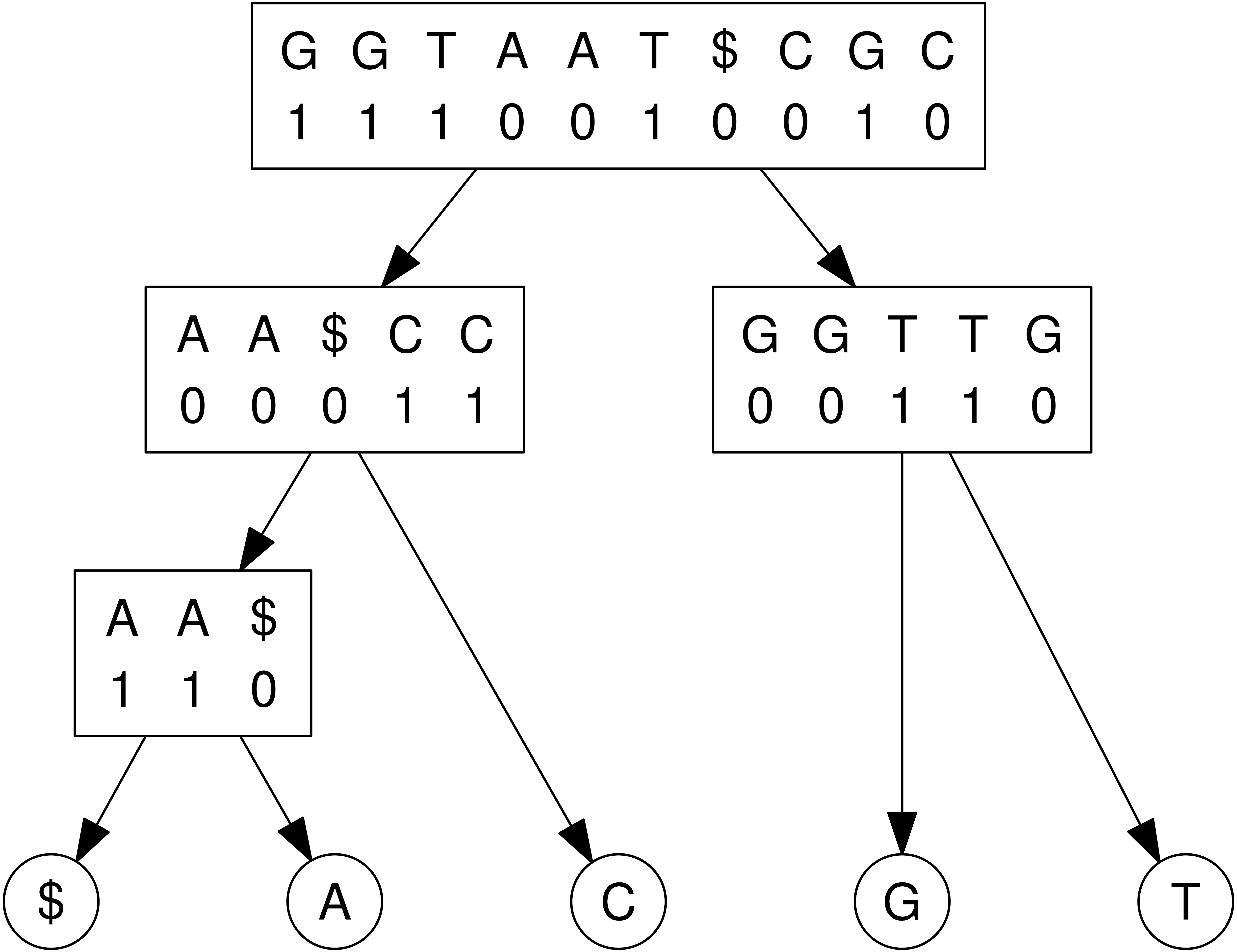
- Sada-I-L and Sada-I-D replace C with another array (Gagie et al., 2013) that is more compressible when the documents are similar to each other.

Wavelet trees

- Wavelet trees (Grossi et al., 2003) are a versatile data structure for sequences. They can, for example, list the distinct characters in a substring quickly.

- When built over DA, a wavelet tree can solve list(P) in $O(|P| + docc \log d)$ time with $n \log d + o(n \log d)$ bits of extra space (Gagie et al., 2009).

- WT uses different encodings for the bitvectors in the wavelet tree (Navarro and Valenzuela, 2012).


G G T A A T $ C G C
1 1 1 0 0 1 0 0 1 0
A A $ C C
0 0 0 1 1
G G T T G
0 0 1 1 0
A A $
1 1 0
$
A
C
G
T

Precomputed document listing

- PDL (Gagie et al., 2013) covers the SA by subtrees of the suffix tree having at most b (e.g. 256) leaves.

- We store the answers for list for the roots of the selected subtrees, as well as for some higher-level nodes.

- Queries below the selected nodes use Brute-L, while the answers for higher-level nodes are computed as unions of stored answers.

- PDL-BC uses a web graph compressor (Hernandez and Navarro, 2012) to store the answers, while PDL-RP uses Re-Pair (Larsson and Moffat, 2000).

Grammar-compressed index

- Grammar (Claude and Munro, 2013) is based on a grammar-compressed text index (Claude and Navarro, 2012).
- It uses Re-Pair to parse the text. For each nonterminal, it stores the set of documents where the nonterminal is used. The sets are also compressed with Re-Pair.
- Grammar is conceptually similar to PDL.
- Does not need a CSA/FMI.

Lempel-Ziv index

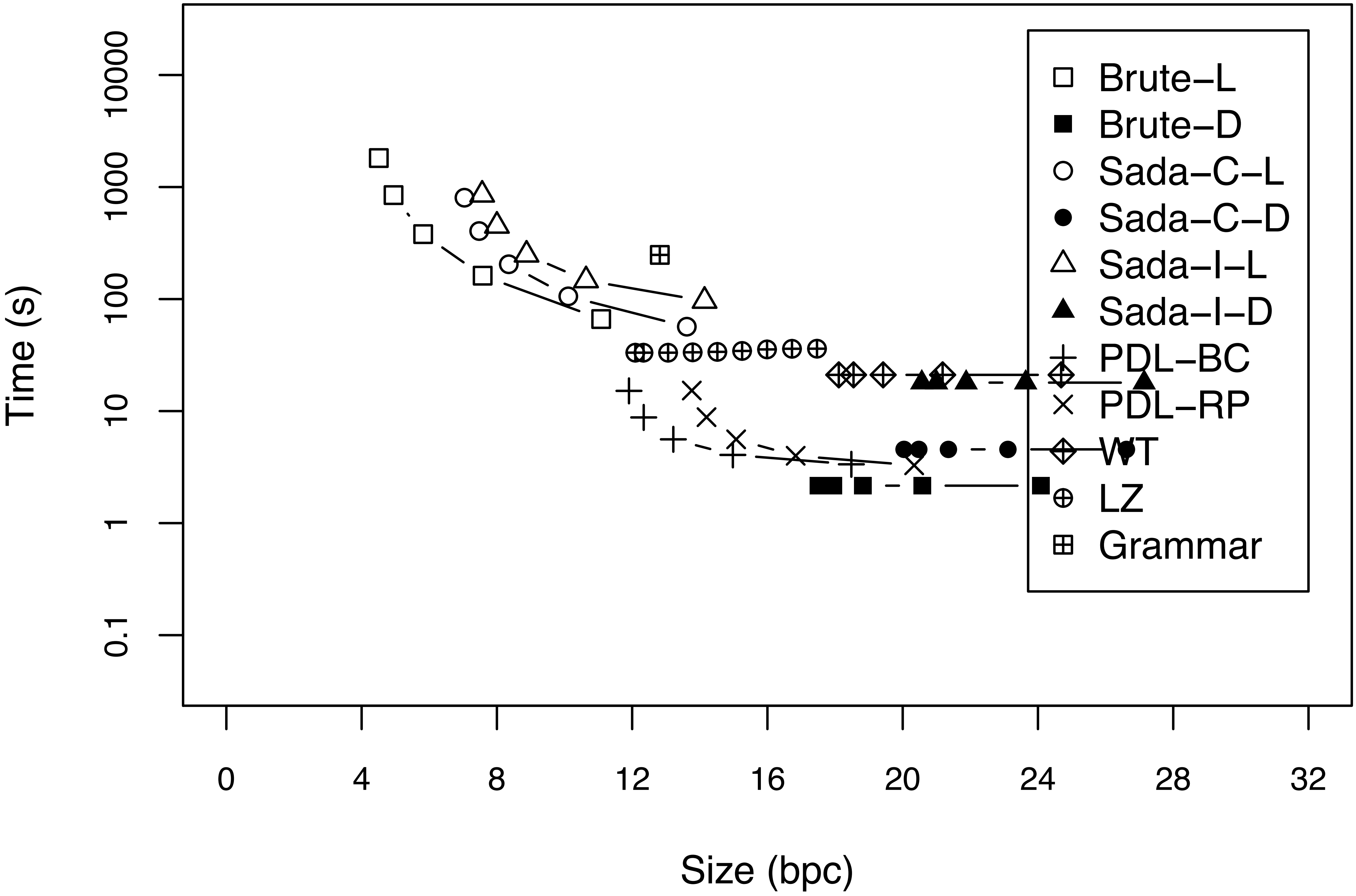
- LZ (Ferrada and Navarro, 2013) is based on the Lempel-Ziv 78 parsing of DA.

- DA is parsed into phrases (x, c), where x is an earlier substring, and c is the following identifier. Sada-C-D is used over the sequence of identifiers c in different ways to solve list.

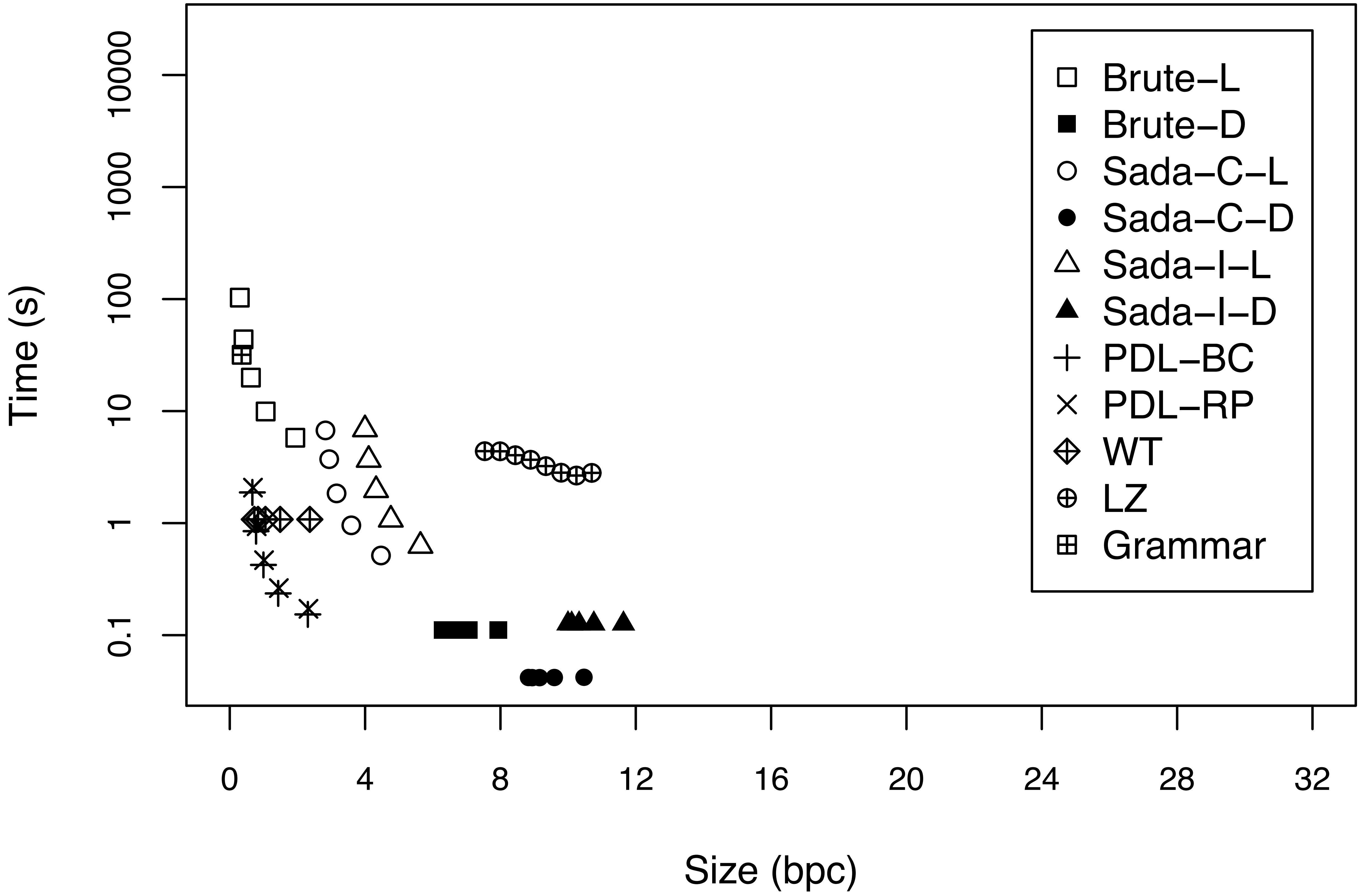
- Does not need a CSA/FMI.

Dataset	**Description**
Enwiki	Pages from a snapshot of the English language Wikipedia.
Page	Pages from the Finnish language Wikipedia. All revisions of a page are concatenated into a single document.
Revision	As Page, but each revision is a separate document.
dna_00100	Short synthetic DNA sequences generated from 100 base sequences.

Enwiki

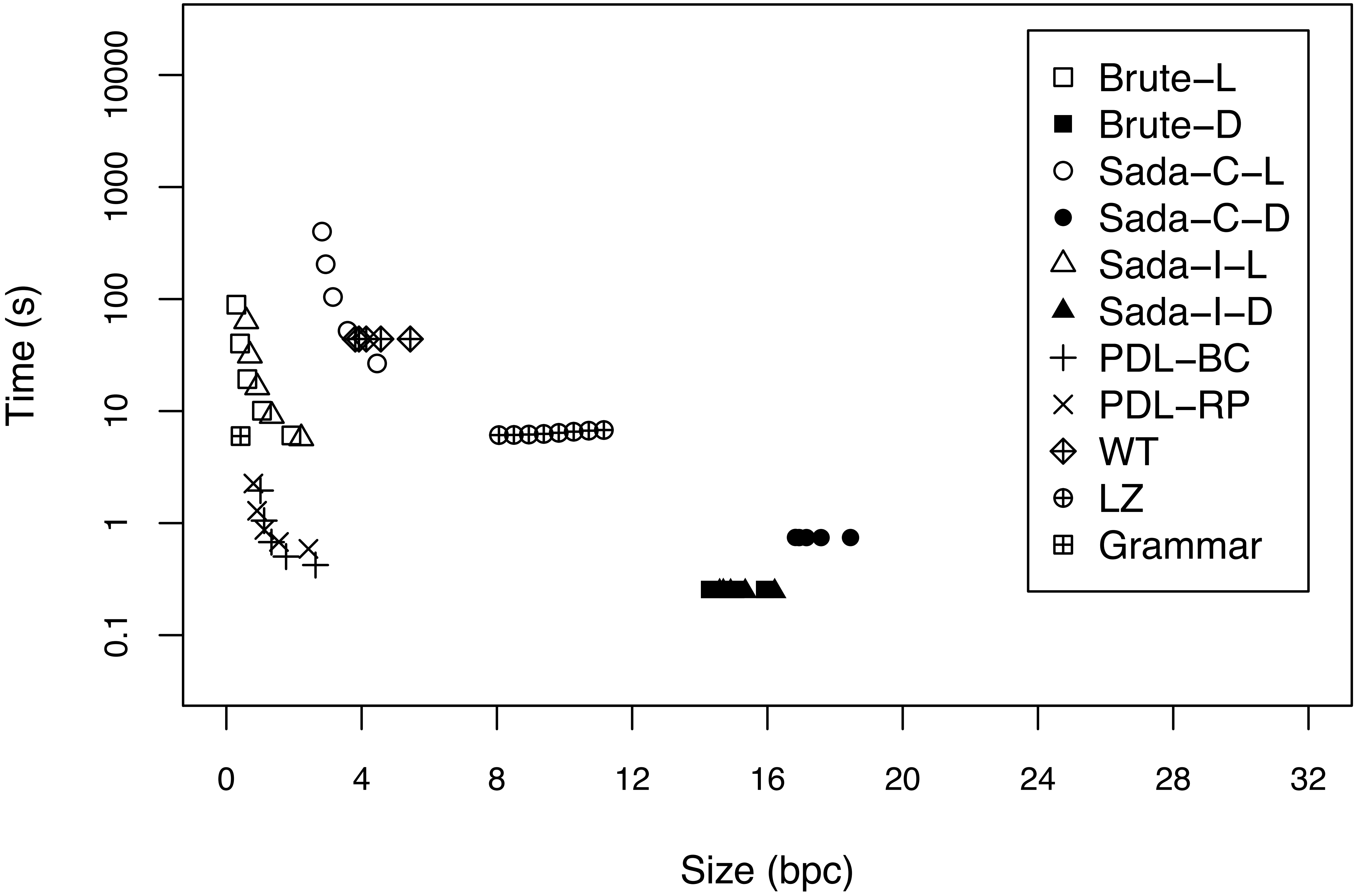

Time (s)
10000
1000
100
10
1
0.1
0
4
8
12
16
20
24
28
32
Size (bpc)
Brute–L
Brute–D
Sada–C–L
Sada–C–D
Sada–I–L
Sada–I–D
PDL–BC
PDL–RP
WT
LZ
Grammar

Page

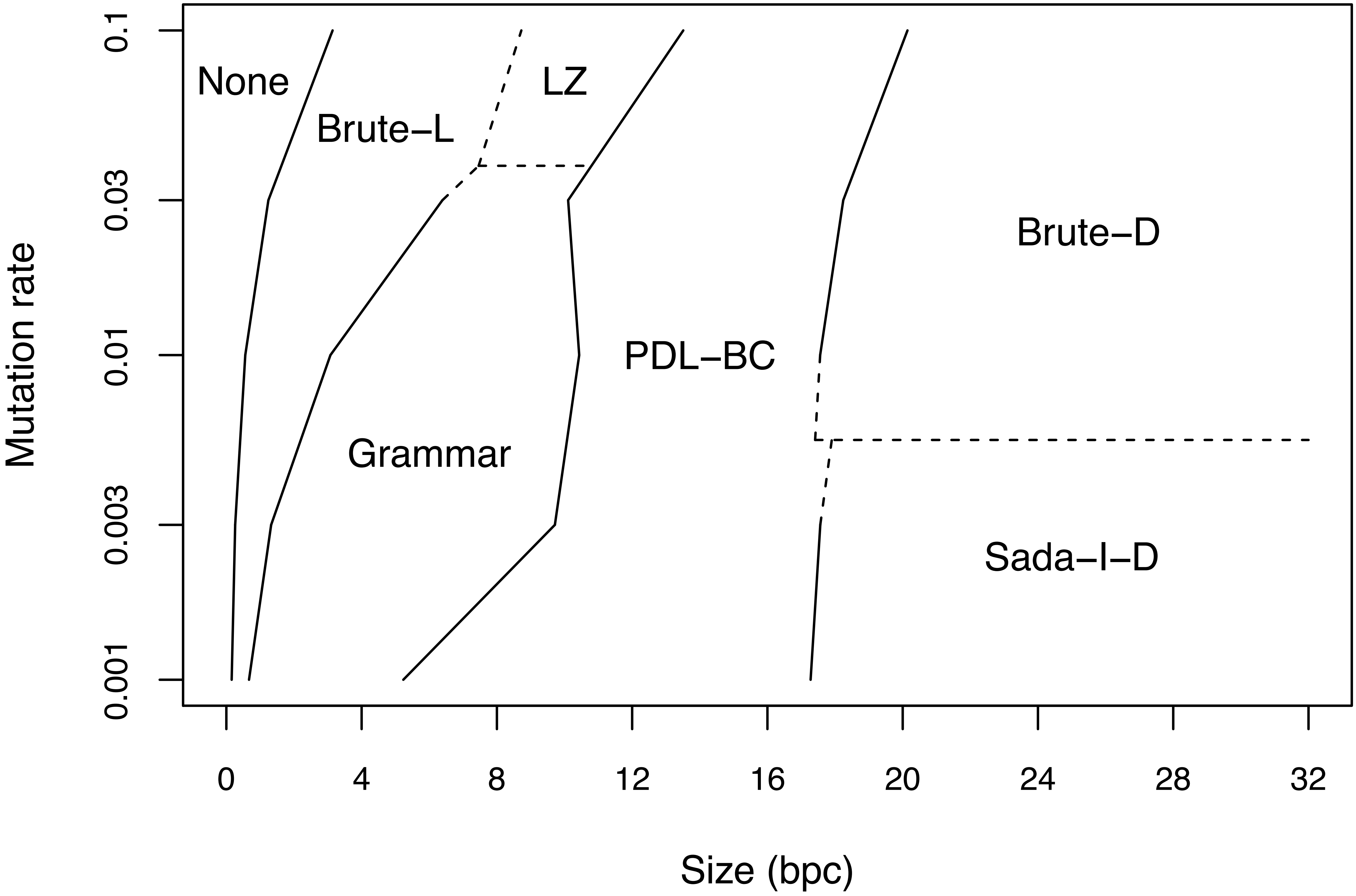


Time (s)
10000
1000
100
10
1
0.1
0
4
8
12
16
20
24
28
32
Size (bpc)
□ Brute–L
■ Brute–D
○ Sada–C–L
● Sada–C–D
△ Sada–I–L
▲ Sada–I–D
+ PDL–BC
× PDL–RP
◈ WT
⊕ LZ
⊞ Grammar

Revision


Time (s)
10000
1000
100
10
1
0.1
0
4
8
12
16
20
24
28
32
Size (bpc)
□ Brute–L
■ Brute–D
○ Sada–C–L
● Sada–C–D
△ Sada–I–L
▲ Sada–I–D
+ PDL–BC
× PDL–RP
WT
⊕ LZ
⊞ Grammar

dna_00100


None
Brute–L
LZ
Grammar
PDL–BC
Brute–D
Sada–I–D
Mutation rate
0.1
0.03
0.01
0.003
0.001
Size (bpc)
0
4
8
12
16
20
24
28
32

Simplified observations

- Brute-L is quite fast, especially with repetitive data.
- When more space is available, PDL is much faster. PDL-BC works better with non-repetitive collections, while PDL-RP is easier to build.
- Brute-D is usually even faster, while using more space.
- There is no clear winner.

Opportunistic data structures

- A term from the original paper on the FM-index (Ferragina and Manzini, 2000).

- Standard algorithm design concentrates on the worst case. We dig into the hard core of the problem, ignoring all properties that could make that particular instance easy.

- Compressed data structures are opportunistic: they are designed for the easy cases.

- Different inputs are easy for different methods.

Thanks!