Compressed Suffix Trees in Practice

Simon Gog

Computing and Information Systems
The University of Melbourne

February 13th 2013

Outline

Succinct data structures (1)

Data structure D

representation of an object X + operations on X

Example: Rank-bit-vector

bit vector b of length n

`(0,1,0,1,1,0,1,1)`

`(0,0,1,1,2,3,3,4)`

in n bits space

\+ access $b[i]$ in $\mathcal{O}(1)$ time

$rank(i) = \sum_{j=0}^{i-1} b[j]$ in $\mathcal{O}(n)$ time

Succinct data structure D

Space of D is close the information theoretic lower bound to represent X, while operations can still be performed efficient.

Succinct data structures (1)

Data structure D

representation of an object X + operations on X

Example: Rank-bit-vector

bit vector b of length n

`(0,1,0,1,1,0,1,1)`

`(0,0,1,1,2,3,3,4)`

in n bits space

+

access $b[i]$ in $\mathcal{O}(1)$ time

$rank(i) = \sum_{j=0}^{i-1} b[j]$ in $\mathcal{O}(n)$ time

Succinct data structure D

Space of D is close the information theoretic lower bound to represent X, while operations can still be performed efficient.

Succinct data structures (1)

Data structure D

representation of an object X + operations on X

Example: Rank-bit-vector

bit vector b of length n
`(0,1,0,1,1,0,1,1)`
`(0,0,1,1,2,3,3,4)`
in $n + n\log n$ bits space

+

access $b[i]$ in $\mathcal{O}(1)$ time
$rank(i) = \sum_{j=0}^{i-1} b[j]$ in $\mathcal{O}(1)$ time

Succinct data structure D

Space of D is close the information theoretic lower bound to represent X, while operations can still be performed efficient.

Succinct data structures (1)

Data structure D

representation of an object X + operations on X

Example: Rank-bit-vector

bit vector b of length n
`(0,1,0,1,1,0,1,1)`
`(0,0,1,1,2,3,3,4)`
in $n + n \log n$ bits space

+

access $b[i]$ in $\mathcal{O}(1)$ time
$rank(i) = \sum_{j=0}^{i-1} b[j]$ in $\mathcal{O}(1)$ time

Succinct data structure D

Space of D is close the information theoretic lower bound to represent X, while operations can still be performed efficient.

Succinct data structures (2)

Can succinct data structures replace classic uncompressed data structures *in practice*?

- Less memory $\Rightarrow$ fewer CPU cycles !?
- Less memory $\Rightarrow$ less costs !?

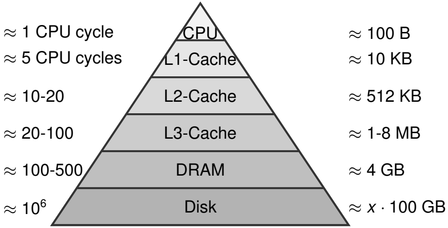
Problems:

- in theory
  - develop succinct data structures
- in practice
  - constants in $\mathcal{O}(1)$-time terms are large
  - $o(n)$-space term is not negligible
  - complex data structures are hard to implement

Succinct data structures (2)

Can succinct data structures replace classic uncompressed data structures *in practice*?

- Less memory $\Rightarrow$ fewer CPU cycles !?



- Less memory $\Rightarrow$ less costs !?

Problems:

- in theory
  - develop succinct data structures
- in practice
  - constants in $\mathcal{O}(1)$-time terms are large
  - $o(n)$-space term is not negligible
  - complex data structures are hard to implement

Succinct data structures (2)

Can succinct data structures replace classic uncompressed data structures *in practice*?

- Less memory $\Rightarrow$ fewer CPU cycles !?
- Less memory $\Rightarrow$ less costs !?

Instance name	main memory	price per hour
Micro	613.0 MB	0.02 US$
High-Memory Quadruple Extra Large	68.4 GB	2.00 US$

Pricing of Amazons Elastic Cloud Computing (EC2) service in July 2011.

Problems:

- in theory
  - develop succinct data structures
- in practice
  - constants in $\mathcal{O}(1)$-time terms are large
  - $o(n)$-space term is not negligible
  - complex data structures are hard to implement

Succinct data structures (2)

Can succinct data structures replace classic uncompressed data structures *in practice*?

- Less memory $\Rightarrow$ fewer CPU cycles !?
- Less memory $\Rightarrow$ less costs !?

Problems:

- in theory
  - develop succinct data structures
- in practice
  - constants in $\mathcal{O}(1)$-time terms are large
  - $o(n)$-space term is not negligible
  - complex data structures are hard to implement

The classic index data structure: The suffix tree (ST)

Let T be a text of length n over alphabet Σ of size σ.

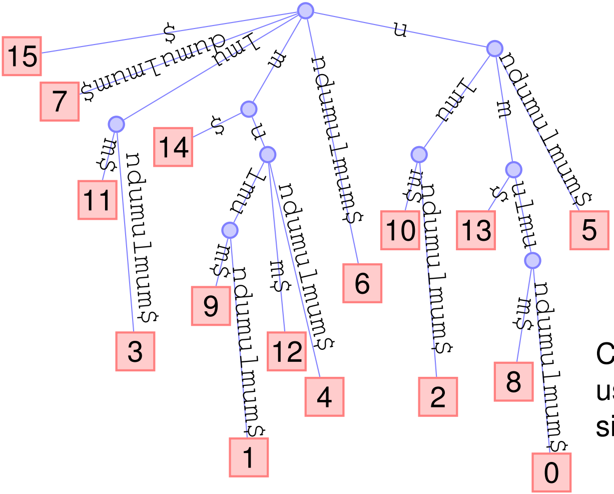
Suffix tree

- index data structure for T (construction $\mathcal{O}(n)$)
- can be used to solve many problems in optimal time complexity
  - bioinformatics
  - data compression
- uses $\mathcal{O}(n \log n)$ bits!
  In practice (ASCII-alphabet) $\geq$ 17 times the size of T

Can not handle „The Attack of Massive Data"

- DNA sequencing data (NGS)
- ...

Example: ST of T=`umulmundumulmum$`



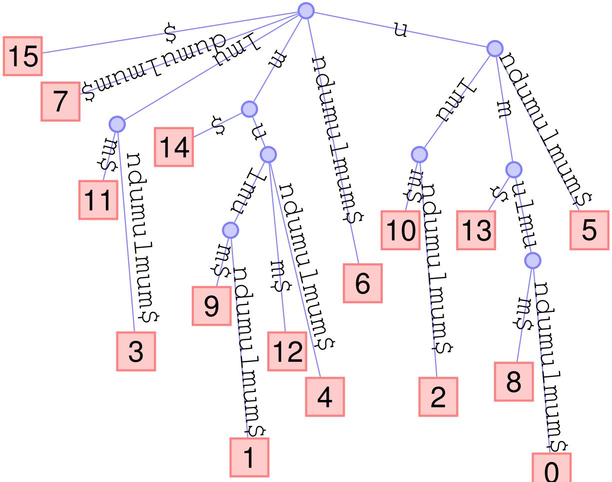
$n = 16$

$\Sigma = \{\$, \mathtt{d}, \mathtt{l}, \mathtt{m}, \mathtt{n}, \mathtt{u}\}$

$\sigma = 6$

Classic implementation uses pointers each of size 4 or 8 bytes!

Example: ST of T=`umulmundumulmum$`



Operations

root()

is_leaf(*v*)

parent(*v*)

degree(*v*)

child(*v*, *c*)

select_child(*v*, *i*)

depth(*v*)

edge(*v*, *d*)

lca(*v*, *w*)

sl(*v*)

wl(*v*, *c*)

CSTs

Goal of a CST implementation

Replace fastest uncompressed ST implementations in different scenarios

(a) both fit in RAM and we measure time
(b) both fit in RAM and we measure resource costs
(c) only CST fits in RAM and we measure time

Proposals

- Sadakane's CST `cst_sada`
- Fully Compressed Suffix Tree (Russo et al.)
- CSTs based on interval representation of nodes (Fischer et al. `cstY`, Ohlebusch et al. `cst_sct3`)

CSTs

Goal of a CST implementation

Replace fastest uncompressed ST implementations in different scenarios

(a) both fit in RAM and we measure time

(b) both fit in RAM and we measure resource costs

(c) only CST fits in RAM and we measure time

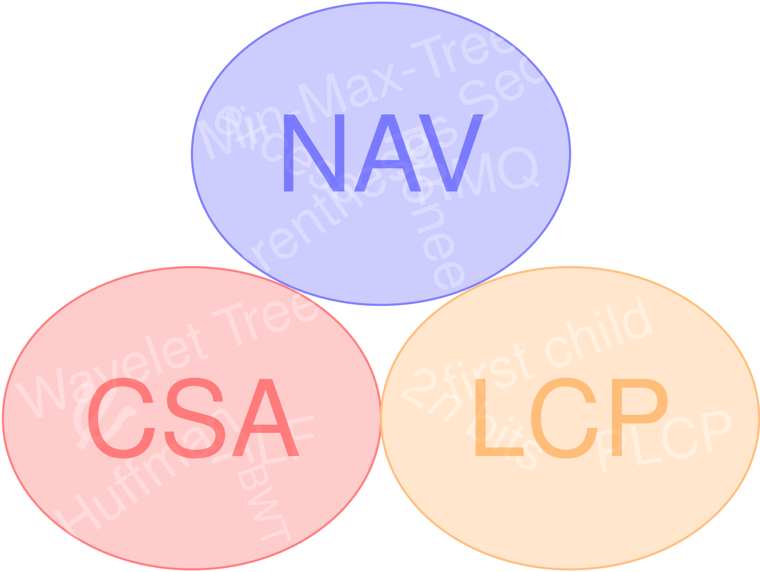
Proposals which might work for (a) and (b)

- Sadakane's CST `cst_sada`
- Fully Compressed Suffix Tree (Russo et al.)
- CSTs based on interval representation of nodes (Fischer et al. `cstY`, Ohlebusch et al. `cst_sct3`)

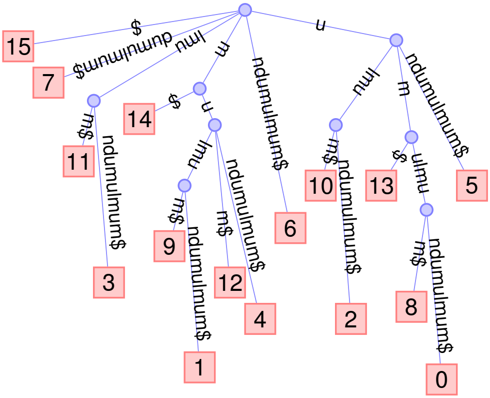
Outline

1. Introduction
   - Basic data structures
   - The suffix tree
2. CST design
   - NAV (tree topology and navigation)
   - CSA (lexicographic information)
   - LCP (longest common prefixes)
3. CST in practice
   - The *sdsl* library

Big picture of CST design



Big picture of CST design

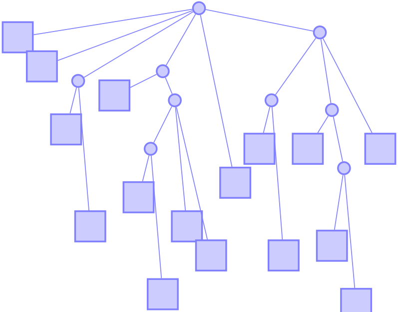


15	7	11	3	14	9	1	12	4	6	10	2	13	8	0	5

0	0	0	3	0	1	5	2	2	0	0	4	1	2	6	1

(()()(()())(()((()())()()))()((()())(()(()()))()))

Example: Compressing NAV

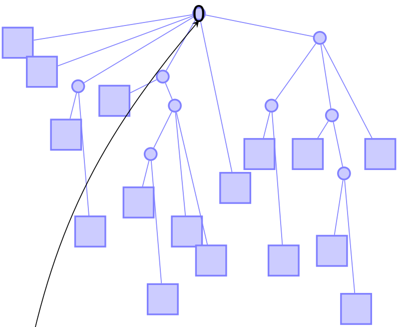


tree uncompressed
$\mathcal{O}(n \log n)$ bits

BPS_{dfs}= (()()(()())(()((()())()()))()((()())(()(()()))()))

compressed
$4n$ bits

Example: Compressing NAV

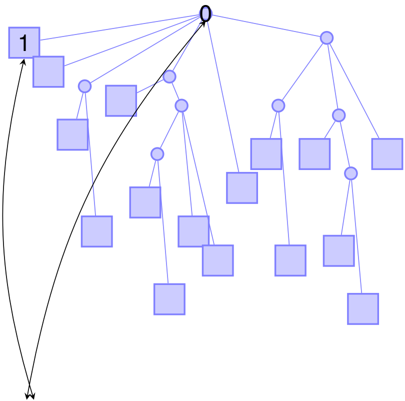


BPS$_{dfs}$= (()()(()())(()((()())()()))()((()())(()(()()))()))

tree uncompressed
$\mathcal{O}(n \log n)$ bits

compressed
$4n$ bits

Example: Compressing NAV

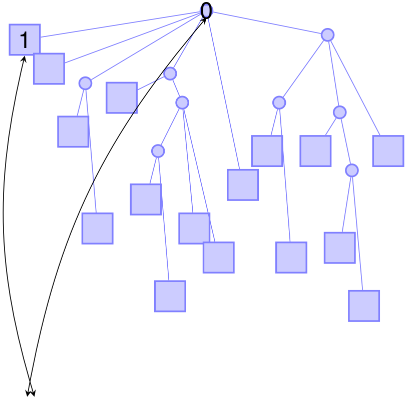


tree uncompressed
$\mathcal{O}(n \log n)$ bits

BPS$_{dfs}$= (()()(()())(()((()())()()))()((()())(()(()()))()))

compressed
$4n$ bits

Example: Compressing NAV

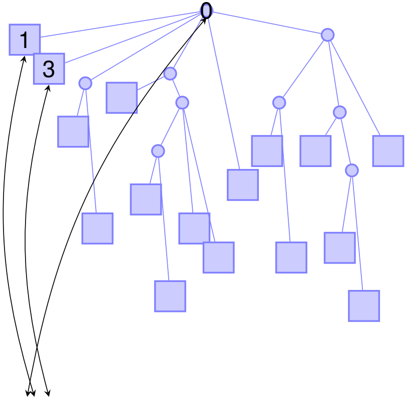


BPS$_{dfs}$= (()()(()())(()((()())()()))()((()())(()(()()))()))

tree uncompressed
$\mathcal{O}(n \log n)$ bits

compressed
$4n$ bits

Example: Compressing NAV

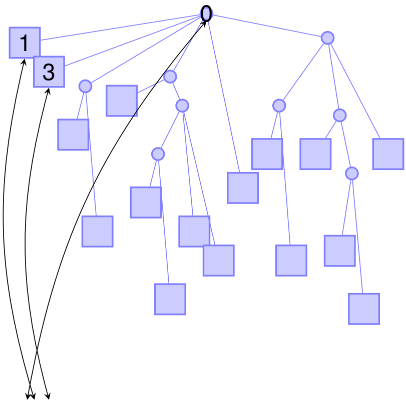


$\text{BPS}_{dfs}=$ (()()(()())(()((()())()()))()((()())(()(()()))()))

tree uncompressed
$\mathcal{O}(n \log n)$ bits

compressed
$4n$ bits

Example: Compressing NAV

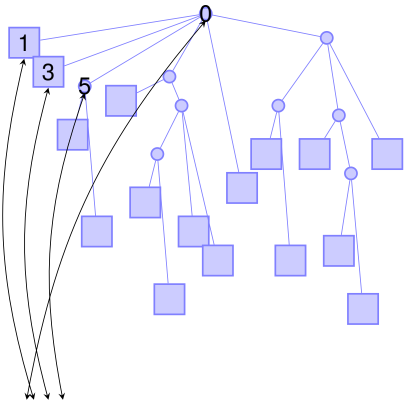


BPS$_{dfs}$= (()()(()())(()((()())()()))()((()())(()(()()))()))

tree uncompressed
$\mathcal{O}(n \log n)$ bits

compressed
$4n$ bits

Example: Compressing NAV

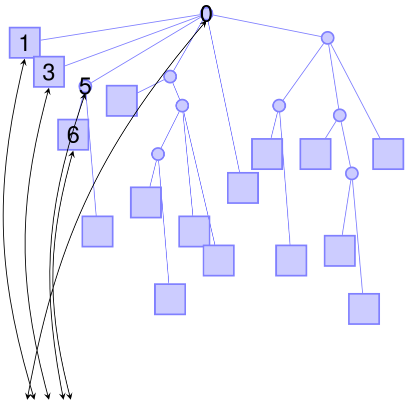


$\mathrm{BPS}_{dfs}=$ (()()(()())(()((()())()()))()((()())(()(()()))()))

tree uncompressed
$\mathcal{O}(n \log n)$ bits

compressed
$4n$ bits

Example: Compressing NAV

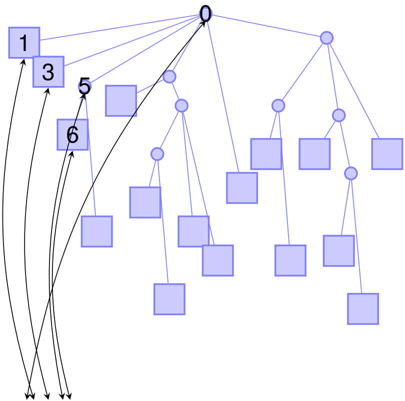


BPS_{dfs}= (()()(()())(()((()())()()))()((()())(()(()())))()))

tree uncompressed
$\mathcal{O}(n \log n)$ bits

compressed
$4n$ bits

Example: Compressing NAV

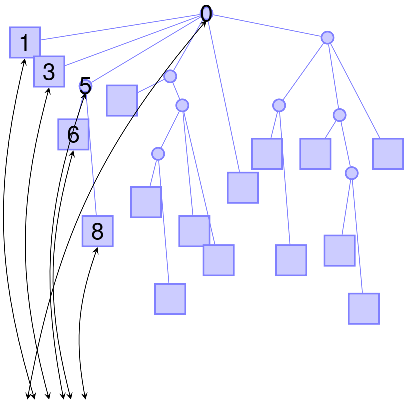


BPS$_{dfs}$= ((()()((()))(()((()()))(()))()((()))(()(()()))()))

tree uncompressed
$\mathcal{O}(n \log n)$ bits

compressed
$4n$ bits

Example: Compressing NAV

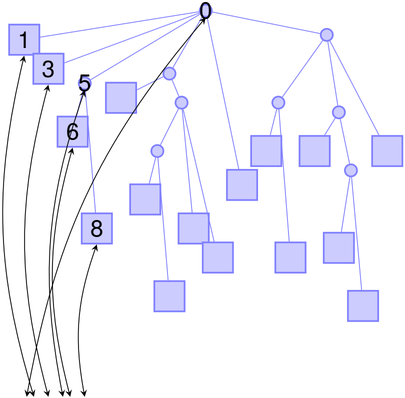


BPS$_{dfs}$= (()()(()(()((()(((()())()())()((()())(()(()()))()))

tree uncompressed
$\mathcal{O}(n \log n)$ bits

compressed
$4n$ bits

Example: Compressing NAV



BPS$_{dfs}$= (()()(()())(()((()())()()))()((()())(()(()()))()))

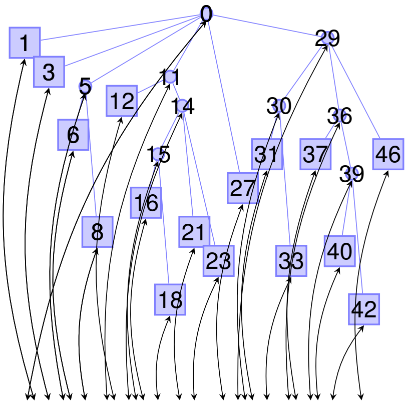
tree uncompressed

$\mathcal{O}(n \log n)$ bits

compressed

$4n$ bits

Example: Compressing NAV

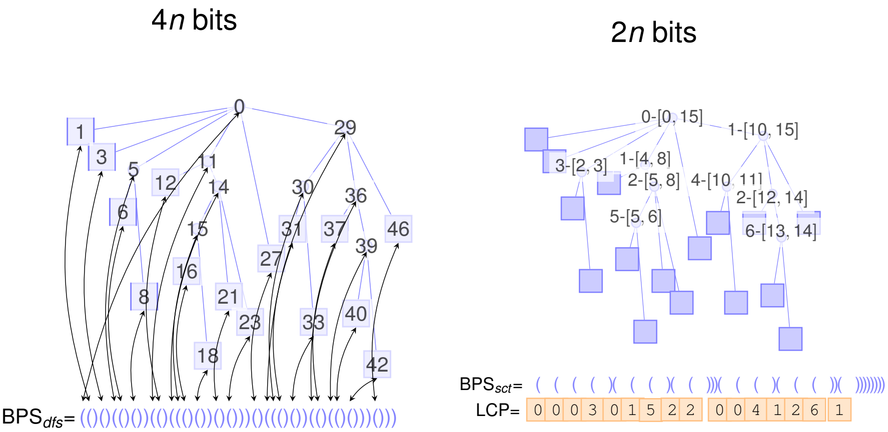


BPS_{dfs}= (()()(()())(()((()())()()))()((()())(()(()()))()))

tree uncompressed
$\mathcal{O}(n \log n)$ bits

compressed
$4n$ bits

NAV data structures (1)



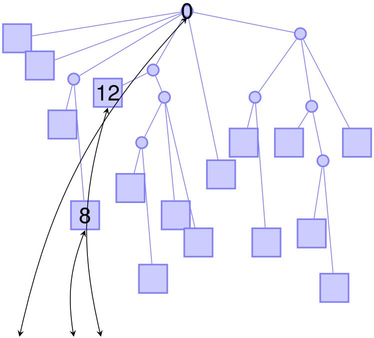
$+o(n)$ bits to answer *find_open*(i), *find_close*(i), *enclose*(i), *double_enclose*(i, j),*rank*(i, c), *select*(i, c),... in constant time

NAV data structures (2)

Comparison of different NAV structures

space in bits	`cst_sada` $4n + o(n)$	`cst_sct` $2n + o(n)$	`cst_sct3` $3n + o(n)$
$root()$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
$degree(v)$	$\mathcal{O}(\sigma)$	$\mathcal{O}(t_{\text{LCP}} \log \sigma)$	$\mathcal{O}(1)$
$depth(v)$	$\mathcal{O}(t_{\text{LCP}})$	$\mathcal{O}(t_{\text{LCP}})$	$\mathcal{O}(t_{\text{LCP}})$
$parent(v)$	$\mathcal{O}(1)$	$\mathcal{O}(t_{\text{LCP}} \log \sigma)$	$\mathcal{O}(1)$
$select\_child(v, i)$	$\mathcal{O}(i)$	$\mathcal{O}(t_{\text{LCP}})$	$\mathcal{O}(1)$
$sibling(v)$	$\mathcal{O}(1)$	$\mathcal{O}(t_{\text{LCP}})$	$\mathcal{O}(1)$
$sl(v)$, $lca(v, w)$	$\mathcal{O}(1)$	$\mathcal{O}(t_{\text{LCP}} \log \sigma)$	$\mathcal{O}(1)$
$child(v, c)$	$\mathcal{O}(t_{\text{SA}} \sigma)$	$\mathcal{O}(t_{\text{SA}} \log \sigma)$	$\mathcal{O}(t_{\text{SA}} \log \sigma)$

Example operations: $select\_leaf(i)$ and $lca(v, w)$ on NAV



BPS_{dfs}= (()()(()())(()((()())()()))()((()())(()(()()))()))

$$select\_leaf(4) = select(4, '10') = 8$$

$$select\_leaf(5) = select(5, '10') = 12$$

$$lca(8,12) = double\_enclose(8,12) = 0$$

Virtues of a CSA based on BWT

- Small size: |CSA| = |BWT| + $\frac{n \log n}{s_{SA}}$ bits where |BWT| can be chosen to be
  - $n \log \sigma$ bits
  - $nH_0(\mathrm{T})$ bits
  - $nH_k(\mathrm{T}) + \mathcal{O}(\sigma^k)$ bits
- pattern matching in time $\mathcal{O}(|P| \log \sigma)$ (even $\mathcal{O}(|P|)$ for $\sigma \in \mathit{polylog}(n)$) by backward search (Ferragina & Manzini)

H_k of the *Pizza&Chili* 200MB test cases

k	dblp.xml H_k	dblp.xml CT/n	dna H_k	dna CT/n	english H_k	english CT/n	proteins H_k	proteins CT/n	rand_k128 H_k	rand_k128 CT/n	sources H_k	sources CT/n
0	5.257	0.0000	1.974	0.0000	4.525	0.0000	4.201	0.0000	7.000	0.0000	5.465	0.0000
1	3.479	0.0000	1.930	0.0000	3.620	0.0000	4.178	0.0000	7.000	0.0000	4.077	0.0000
2	2.170	0.0000	1.920	0.0000	2.948	0.0001	4.156	0.0000	6.993	0.0001	3.102	0.0000
3	1.434	0.0007	1.916	0.0000	2.422	0.0005	4.066	0.0001	5.979	0.0100	2.337	0.0012
4	1.045	0.0043	1.910	0.0000	2.063	0.0028	3.826	0.0011	0.666	0.6939	1.852	0.0082
5	0.817	0.0130	1.901	0.0000	1.839	0.0103	3.162	0.0173	0.006	0.9969	1.518	0.0250
6	0.705	0.0265	1.884	0.0001	1.672	0.0265	1.502	0.1742	0.000	1.0000	1.259	0.0509
7	0.634	0.0427	1.862	0.0001	1.510	0.0553	0.340	0.4506	0.000	1.0000	1.045	0.0850
8	0.574	0.0598	1.834	0.0004	1.336	0.0991	0.109	0.5383	0.000	1.0000	0.867	0.1255
9	0.537	0.0773	1.802	0.0013	1.151	0.1580	0.074	0.5588	0.000	1.0000	0.721	0.1701
10	0.508	0.0955	1.760	0.0051	0.963	0.2292	0.061	0.5699	0.000	1.0000	0.602	0.2163

Backward search

i	T^{BWT}	F
0	m	$
1	n	dumulmum$
2	u	lmum$
3	u	lmundumulmum$
4	u	m$
5	u	mulmum$
6	u	mulmundumulmum$
7	l	mum$
8	l	mundumulmum$
9	u	ndumulmum$
10	m	ulmum$
11	m	ulmundumulmum$
12	m	um$
13	d	umulmum$
14	$	umulmundumulmum$
15	m	undumulmum$

(a)

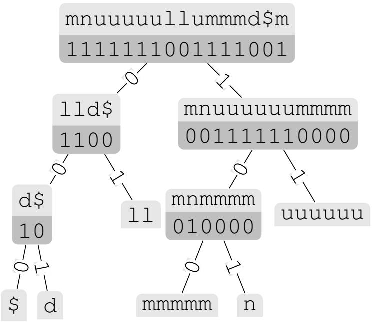
T^{BWT}	F
m	$
n	dumulmum$
u	lmum$
u	lmundumulmum$
u	m$
u	mulmum$
u	mulmundumulmum$
l	mum$
l	mundumulmum$
u	ndumulmum$
m	ulmum$
m	ulmundumulmum$
m	um$
d	umulmum$
$	umulmundumulmum$
m	undumulmum$

(b)

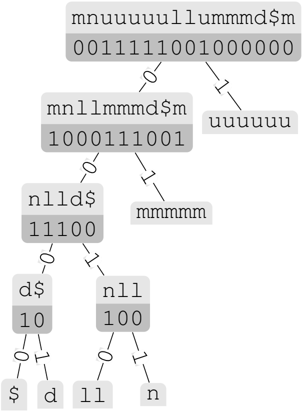
T^{BWT}	F
m	$
n	dumulmum$
u	lmum$
u	lmundumulmum$
u	m$
u	mulmum$
u	mulmundumulmum$
l	mum$
l	mundumulmum$
u	ndumulmum$
m	ulmum$
m	ulmundumulmum$
m	um$
d	umulmum$
$	umulmundumulmum$
m	undumulmum$

(c)

Wavelet tree: rank for character sequences



(a)



(b)

Example: Compressing CSA (practical approach)

i	SA	LF	T^{BWT}	T
0	15	4	m	$
1	7	9	n	dumulmum$
2	11	10	u	lmum$
3	3	11	u	lmundumulmum$
4	14	12	u	m$
5	9	13	u	mulmum$
6	1	14	u	mulmundumulmum$
7	12	2	l	mum$
8	4	3	l	mundumulmum$
9	6	15	u	ndumulmum$
10	10	5	m	ulmum$
11	2	6	m	ulmundumulmum$
12	13	7	m	um$
13	8	1	d	umulmum$
14	0	0	$	umulmundumulmum$
15	5	8	m	undumulmum$

Example: Compressing CSA (practical approach)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
15	7	11	3	14	9	1	12	4	6	10	2	13	8	0	5

SA uncompressed
$n \log n$ bits

m	n	u	u	u	u	u	l	l	u	m	m	m	d	$	m
$	d	l	l	m	m	m	m	m	n	u	u	u	u	u	u
15	7	11	3	14	9	1	12	4	6	10	2	13	8	0	5

compressed

$\text{CSA} = \text{T}^{\text{BWT}} + \text{SA samples:}$

$n \log \sigma + o(n \log \sigma)$ bits
$+$
$\frac{n \log n}{s_{\text{SA}}}$ bits

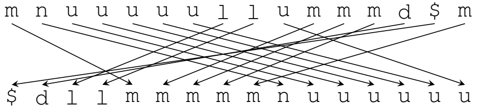
$s_{\text{SA}} = 3$
access LF[i] in time $\mathcal{O}(\log \sigma)$
access CSA[i] in time $\mathcal{O}(s_{\text{SA}} \log \sigma)$

Example: Compressing CSA (practical approach)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
15	7	11	3	14	9	1	12	4	6	10	2	13	8	0	5

SA uncompressed
$n \log n$ bits

SA[13]=



15	7	11	3	14	9	1	12	4	6	10	2	13	8	0	5

compressed

$\text{CSA} = \text{T}^{\text{BWT}} + \text{SA samples:}$

$n \log \sigma + o(n \log \sigma)$ bits

$+$

$\frac{n \log n}{s_{\text{SA}}}$ bits

$s_{\text{SA}} = 3$

access LF[i] in time $\mathcal{O}(\log \sigma)$

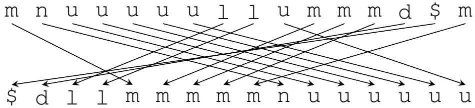
access CSA[i] in time $\mathcal{O}(s_{\text{SA}} \log \sigma)$

Example: Compressing CSA (practical approach)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
15	7	11	3	14	9	1	12	4	6	10	2	13	8	0	5

SA uncompressed
$n \log n$ bits

SA[13]=SA[1]+1



15	7	11	3	14	9	1	12	4	6	10	2	13	8	0	5

compressed

$\text{CSA} = \text{T}^{\text{BWT}} + \text{SA samples:}$

$n \log \sigma + o(n \log \sigma)$ bits
$+$
$\frac{n \log n}{s_{\text{SA}}}$ bits

$s_{\text{SA}} = 3$

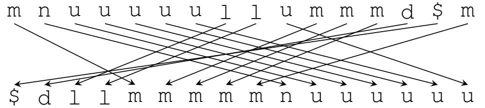
access LF[i] in time $\mathcal{O}(\log \sigma)$

access CSA[i] in time $\mathcal{O}(s_{\text{SA}} \log \sigma)$

Example: Compressing CSA (practical approach)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
15	7	11	3	14	9	1	12	4	6	10	2	13	8	0	5

SA[13]=SA[9]+2



15	7	11	3	14	9	1	12	4	6	10	2	13	8	0	5

$s_{SA} = 3$

access LF[i] in time $\mathcal{O}(\log\sigma)$

access CSA[i] in time $\mathcal{O}(s_{SA}\log\sigma)$

SA uncompressed
$n\log n$ bits

compressed

CSA $=$ T^{BWT} $+$ SA samples:

$n\log\sigma + o(n\log\sigma)$ bits

$+$

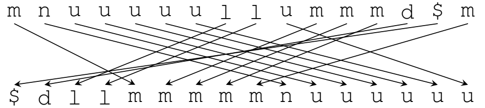
$\frac{n\log n}{s_{SA}}$ bits

Example: Compressing CSA (practical approach)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
15	7	11	3	14	9	1	12	4	6	10	2	13	8	0	5

SA uncompressed
$n \log n$ bits

SA[13]=6+2=8



15	7	11	3	14	9	1	12	4	6	10	2	13	8	0	5

compressed

$\mathrm{CSA} = \mathrm{T}^{\mathrm{BWT}} + \mathrm{SA}$ samples:

$n \log \sigma + o(n \log \sigma)$ bits

$+$

$\frac{n \log n}{s_{\mathrm{SA}}}$ bits

$s_{\mathrm{SA}} = 3$

access LF[i] in time $\mathcal{O}(\log \sigma)$

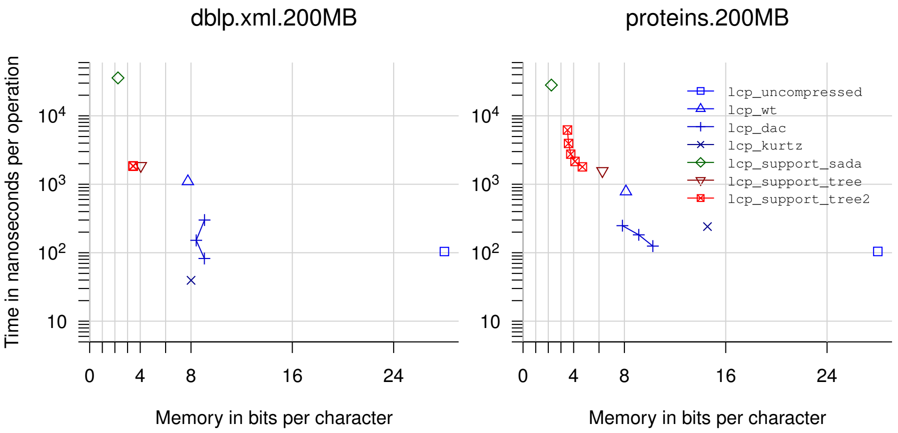
access CSA[i] in time $\mathcal{O}(s_{\mathrm{SA}} \log \sigma)$

Overview of LCP data structures

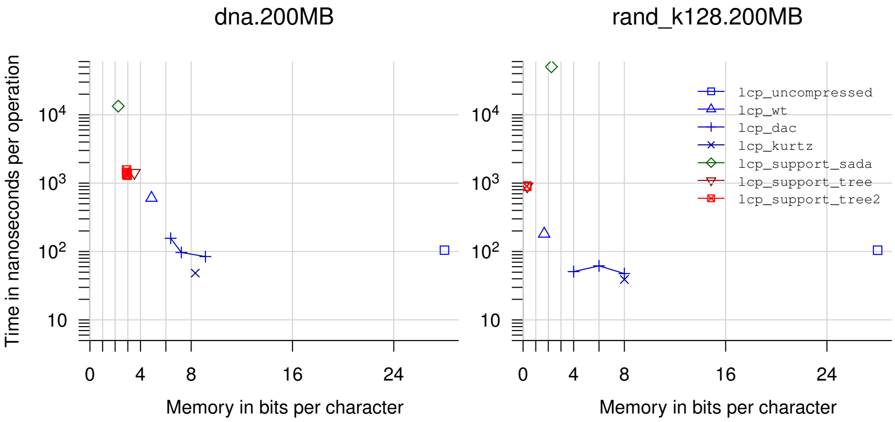
data structure	uses	access	memory in bits
lcp_uncompressed	-	$\mathcal{O}(1)$	$n \log n$
lcp_support_sada	CSA	$\mathcal{O}(t_{SA})$	$2n + o(n)$
lcp_kurtz	-	$\mathcal{O}(\log n)$ or $\mathcal{O}(1)$	$8n_1 + 2n_2 \log n$
...	...	...	...
lcp_support_tree	NAV	$\mathcal{O}(\log \sigma')$	$H_0 q_1 + q_2 \log n$
lcp_support_tree2	NAV & LF	$\mathcal{O}(s_{LCP} \log \sigma')$	$H_0 q_1 + (q_2 \log n)/s_{LCP}$

with $n_1 + n_2 = n$
and $q_1 + q_2 = q < n$
q number of inner nodes of the ST

Runtime for random access to the LCP array (1)



Runtime for random access to the LCP array (2)



Runtime for random access to the LCP array (3)

Outline

1 Introduction
- Basic data structures
- The suffix tree

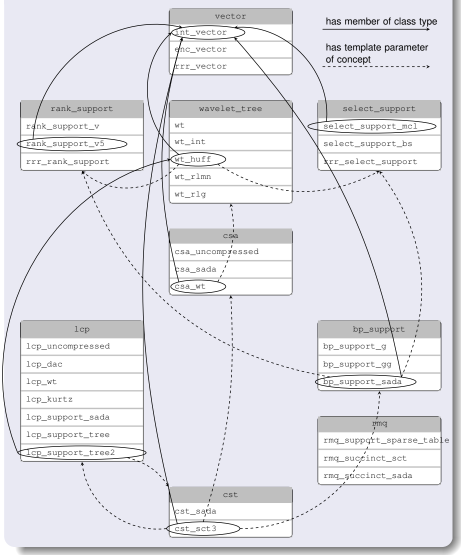
2 CST design
- NAV (tree topology and navigation)
- CSA (lexicographic information)
- LCP (longest common prefixes)

3 CST in practice
- The *sdsl* library

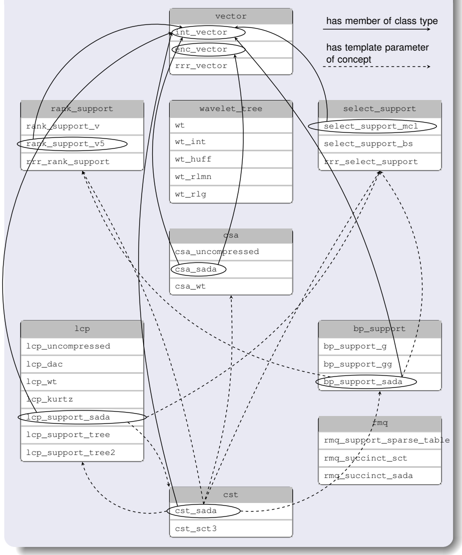
The succinct data structure library *sdsl*

- Provides basic and advanced succinct data structures
- Easy to use (very similar to C++ STL)
- Fast and space-efficient construction of data structures
- 64-bit implementation
- Well-optimized implementation (e.g. now using hardware POPCOUNT operation,..)
- Easy configuration of myriads of CSAs, CSTs with many time-space trade-offs
- Fast prototyping of other complex succinct data structures

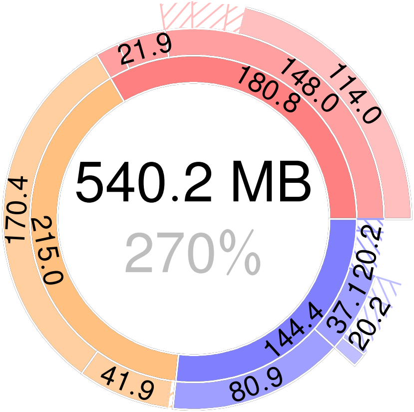
cst_sada<csa_sada<>,lcp_..



cst_sct3<csa_wt<wt_huff<>..



CST space in practice (for `english.200MB`)



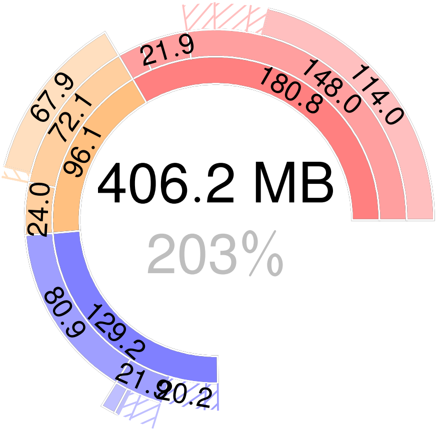
CSA: 180.8 MB
- wt: 148 MB
  - data: 114 MB
  - rank: 7.1 MB
  - select 1: 14.4 MB
  - select 0: 12.5 MB
- sa_sample: 21.9 MB
- isa_sample: 10.9 MB

lcp: 215 MB
- lcp values: 170.4 MB
- overflow mark: 41.9 MB
- rank: 2.6 MB

nav: 144.4 MB
- BPS_{dfs}(`bit_vector`): 80.9 MB
- bp_support: 37.1 MB
  - small block: 5.7 MB
  - medium block: 1.6 MB
  - bp rank: 20.2 MB
  - bp select: 9.6 MB
- rank_support10: 20.2 MB
- select_support10: 6.1 MB

```
cst_sada<csa_wt<wt_huff<> >,lcp_dac<> >
```

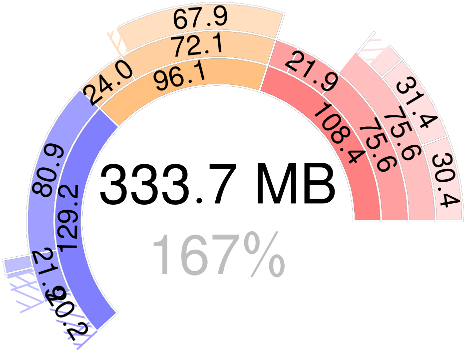
CST space in practice (for `english.200MB`)



- CSA: 180.8 MB
  - wt: 148 MB
    - data: 114 MB
    - rank: 7.1 MB
    - select 1: 14.4 MB
    - select 0: 12.5 MB
  - sa_sample: 21.9 MB
  - isa_sample: 10.9 MB
- lcp: 96.1 MB
  - small lcp: 72.1 MB
    - data: 67.9 MB
    - rank: 4.2 MB
  - big lcp: 24 MB
- nav: 129.2 MB
  - bp: 80.9 MB
  - bp_support: 21.9 MB
    - small block: 5.7 MB
    - medium block: 1.6 MB
    - bp rank: 5.1 MB
    - bp select: 9.6 MB
  - rank_support10: 20.2 MB
  - select_support10: 6.1 MB

```
cst_sada<csa_wt<wt_huff<> >,lcp_support_tree2<> >
```

CST space in practice (for `english.200MB`)



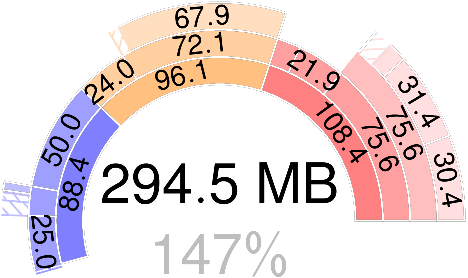
CSA: 108.4 MB
- wt: 75.6 MB
  - data: 75.6 MB
    - bt: 30.4 MB
    - btnr: 31.4 MB
    - btnrp: 6.7 MB
    - rank samples: 6.9 MB
    - invert: 0.2 MB
  - rank: 0 MB
  - select 1: 0 MB
  - select 0: 0 MB
- sa_sample: 21.9 MB
- isa_sample: 10.9 MB

lcp: 96.1 MB
- small lcp: 72.1 MB
  - data: 67.9 MB
  - rank: 4.2 MB
- big lcp: 24 MB

nav: 129.2 MB
- BPS_{dfs}(`bit_vector`): 80.9 MB
- bp_support: 21.9 MB
  - small block: 5.7 MB
  - medium block: 1.6 MB
  - bp rank: 5.1 MB
  - bp select: 9.6 MB
- rank_support10: 20.2 MB
- select_support10: 6.1 MB

```
cst_sada<csa_wt<wt_huff<rrr_vector<> >
>,lcp_support_tree2<> >
```

CST space in practice (for `english.200MB`)



- CSA: 108.4 MB
  - wt: 75.6 MB
    - data: 75.6 MB
      - bt: 30.4 MB
      - btnr: 31.4 MB
      - btnrp: 6.7 MB
      - rank samples: 6.9 MB
      - invert: 0.2 MB
    - rank: 0 MB
    - select 1: 0 MB
    - select 0: 0 MB
  - sa_sample: 21.9 MB
  - isa_sample: 10.9 MB
- lcp: 96.1 MB
  - small lcp: 72.1 MB
    - data: 67.9 MB
    - rank: 4.2 MB
  - big lcp: 24 MB
- nav: 88.4 MB
  - bp: 50 MB
  - bp_support: 13.4 MB
    - small block: 3.5 MB
    - medium block: 0.8 MB

```
cst_sct3<csa_wt<wt_huff<rrr_vector<> >
>,lcp_support_tree2<> >
```

Experimental setup

$0 \triangleq$ `cst_sada<csa_sada<>, lcp_dac<> >`

$1 \triangleq$ `cst_sada<csa_sada<>, lcp_support_tree2<> >`

$2 \triangleq$ `cst_sada<csa_wt<>, lcp_dac<> >`

$3 \triangleq$ `cst_sada<csa_wt<>, lcp_support_tree2<> >`

$4 \triangleq$ `cst_sct3<csa_sada<>, lcp_dac<> >`

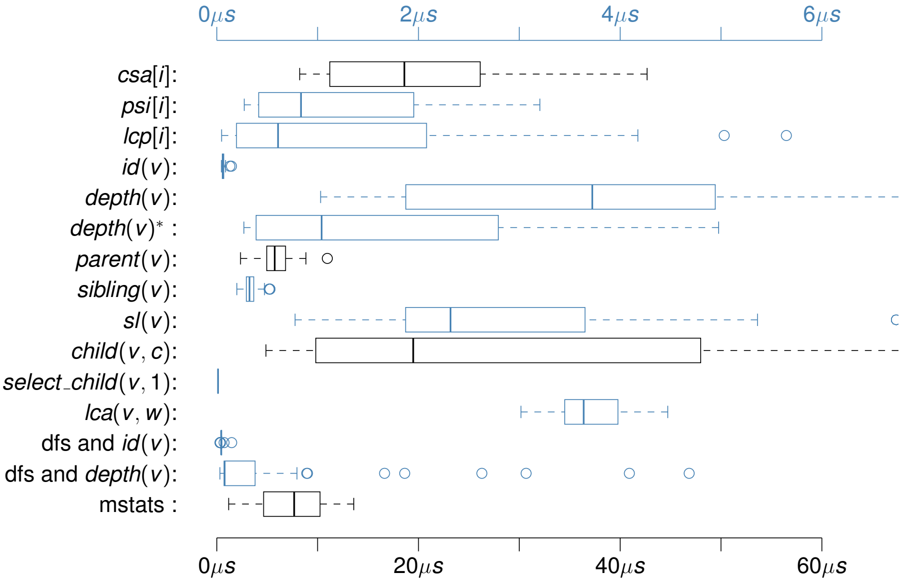
$5 \triangleq$ `cst_sct3<csa_sada<>, lcp_support_tree2<> >`

$6 \triangleq$ `cst_sct3<csa_wt<>, lcp_dac<> >`

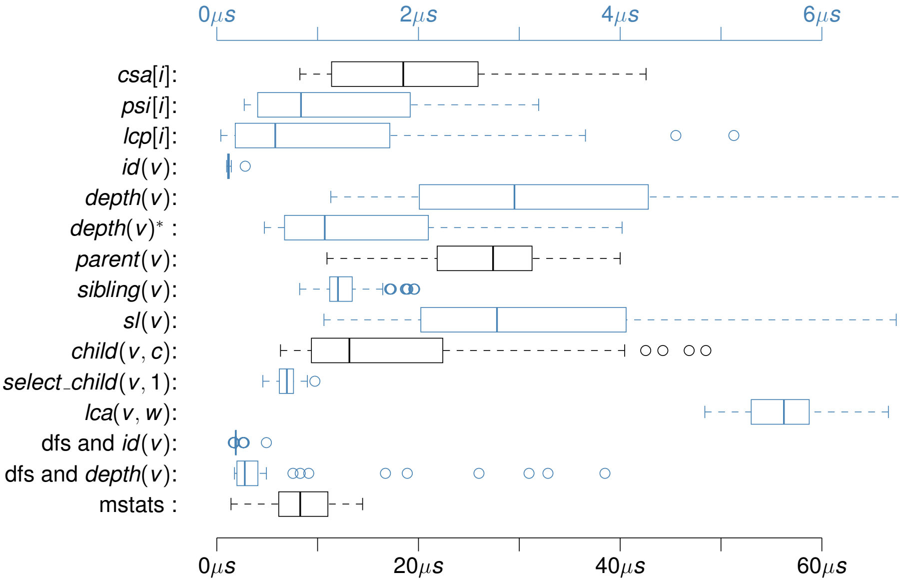
$7 \triangleq$ `cst_sct3<csa_wt<>, lcp_support_tree2<> >`

The same basic data structures are used, i.e. its a very fair comparison

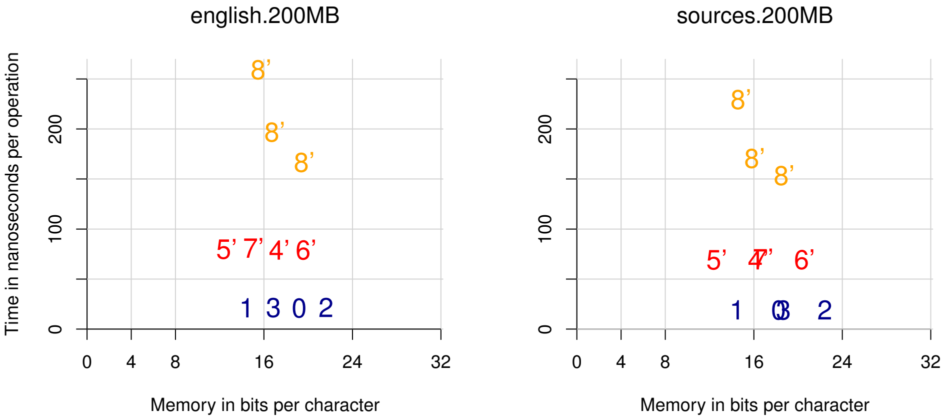
Runtime of operations of `cst_sada`



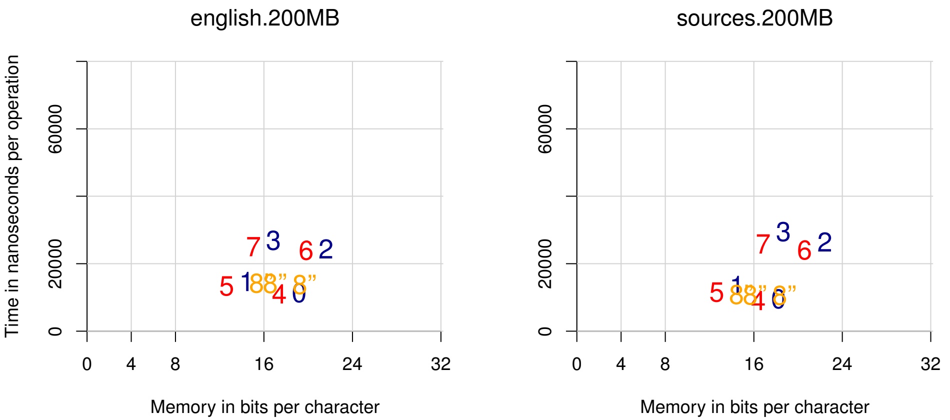
Runtime of operations of `cst_sct3`



Time-space trade-off for $select\_child(v, 1)$



Time-space trade-off for *child*(v, c)



Construction of a CST

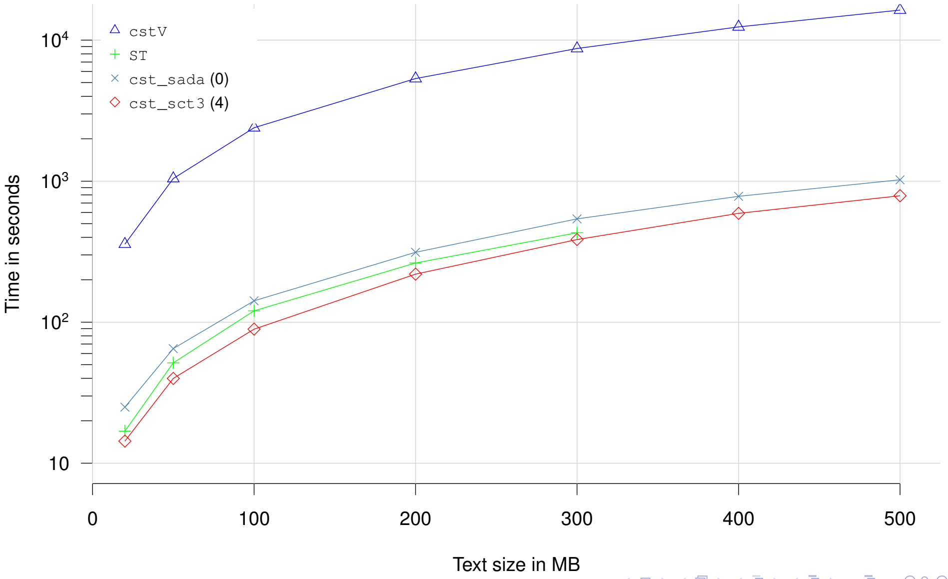
```
#include <sdsl/suffixtrees.hpp>
#include <sdsl/util.hpp>

using namespace sdsl;

typedef cst_sct3<> tCST;

int main(int argc, char* argv[]){
  tCST cst;
  construct_cst( argv[1], cst );
}
```

Runtime for construction (prefixes of `english` text)

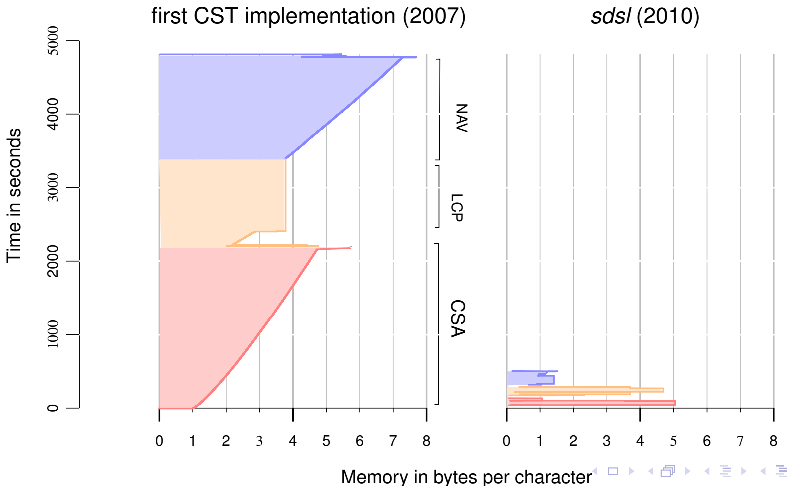


CST construction – resources comparison

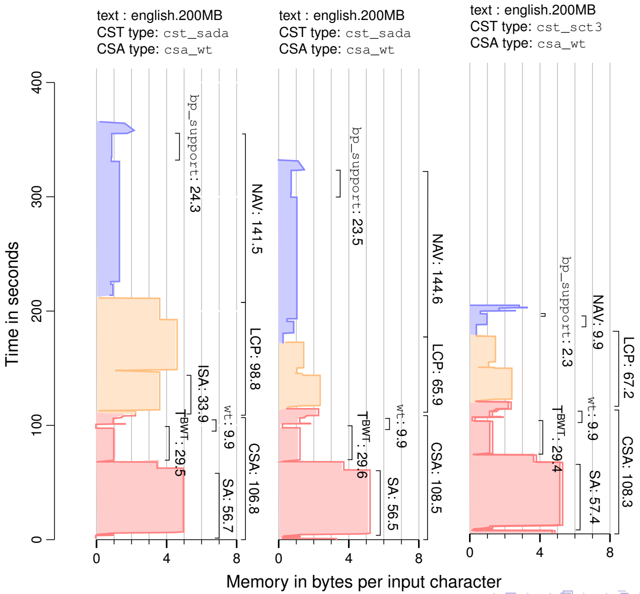
text: `english.200MB`
σ: 226

CST type: `cst_sada`
LCP type: `lcp_sada`

CSA type: `csa_wt`



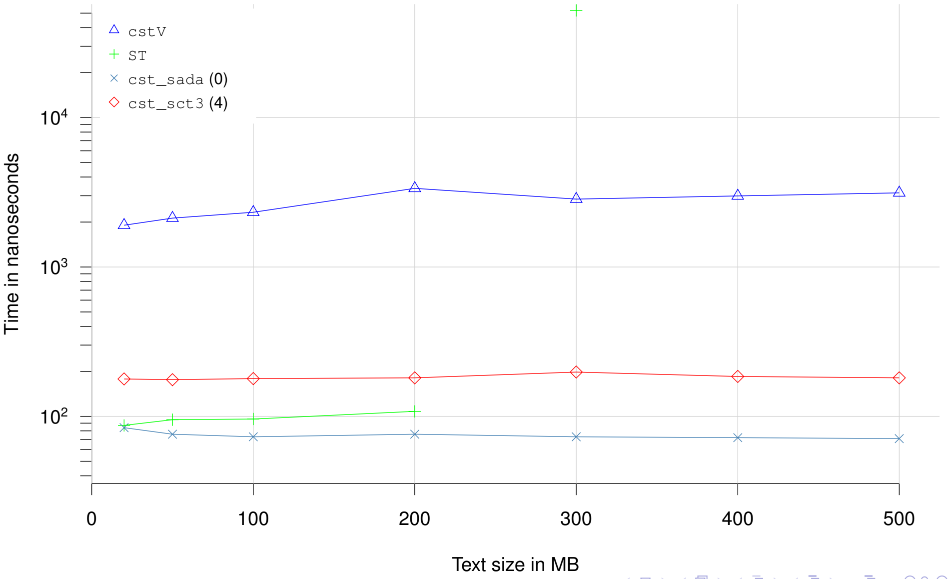
Detailed resources for the construction of CSTs



Depth first search traversal in a CST

```
template<class Cst>
void test_cst_dfs_iterator_and_depth(Cst &cst){
  typedef typename Cst::const_iterator iterator;
  long long cnt = 0;
  for(iterator it=cst.begin();it!=cst.end();++it){
    if( !cst.is_leaf(*it) )
        cnt += cst.depth(*it);
  }
  cout << cnt << endl;
}
```

Runtime of dfs on CST (prefixes of `english` text)



Conclusion

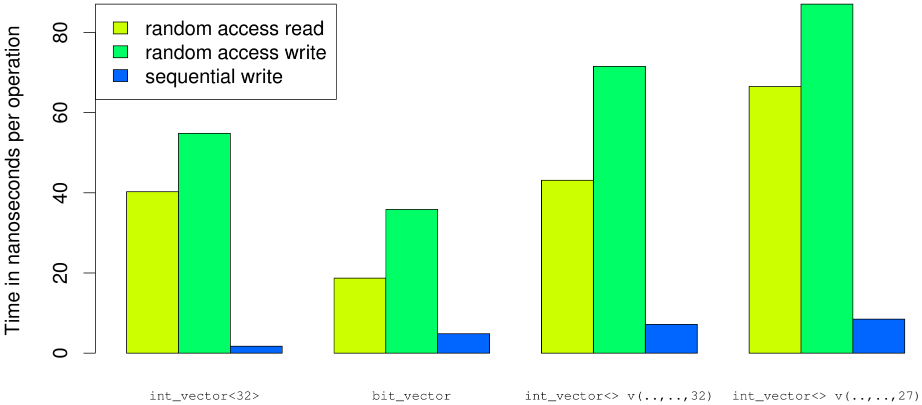
CSTs ...

- ... can be build fast and space-efficient
- ... provide a rich set of functionality
  - fast operations: basic navigation, access LF or Ψ
  - slow operations: $\mathit{child}(v, c)$, access to SA

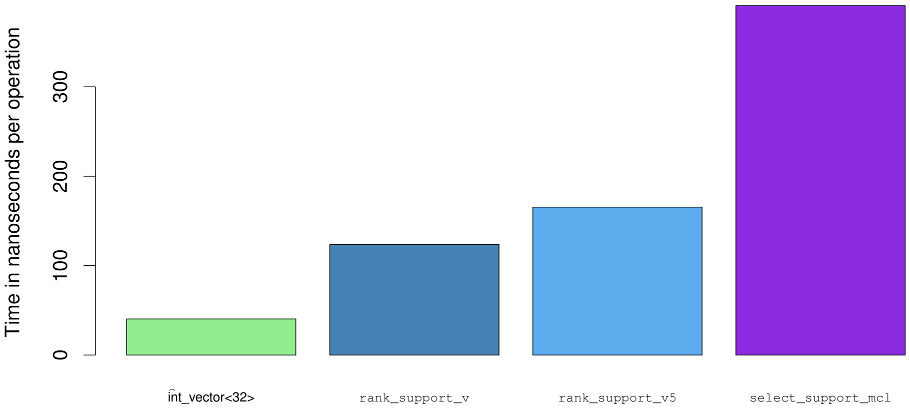
You can use the *sdsl* library to configure a CSTs which fits your needs

Thank you!

Runtime of `int_vector` access

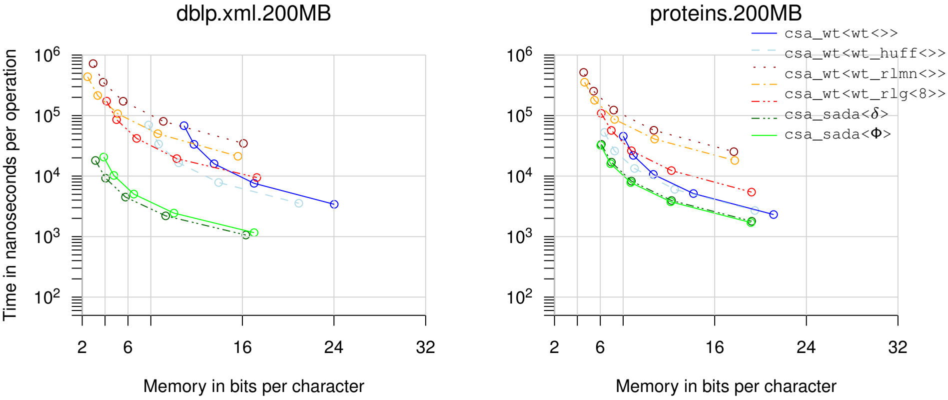


Operation runtime of basic data structures

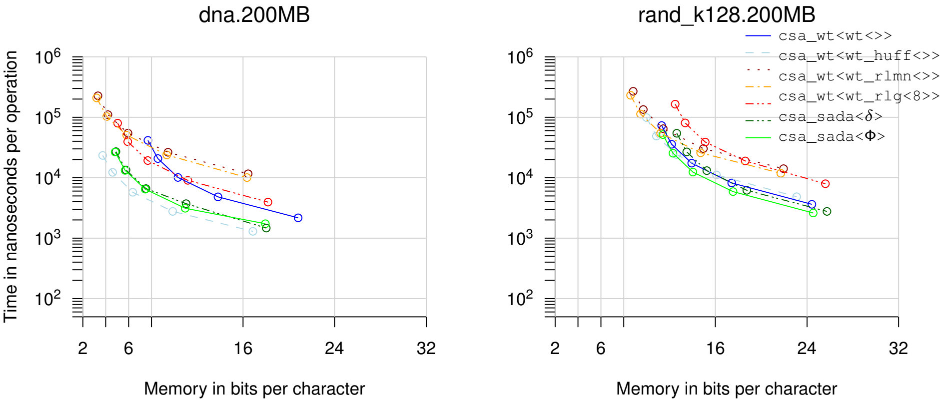


(random access)

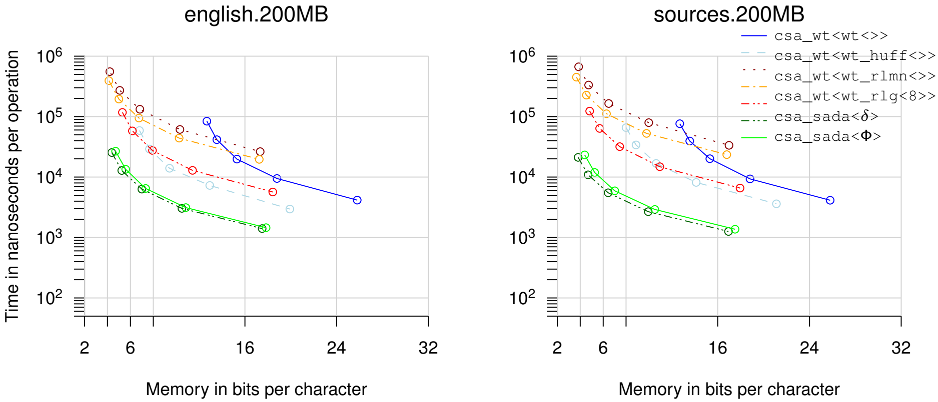
Runtime of CSA access (1)



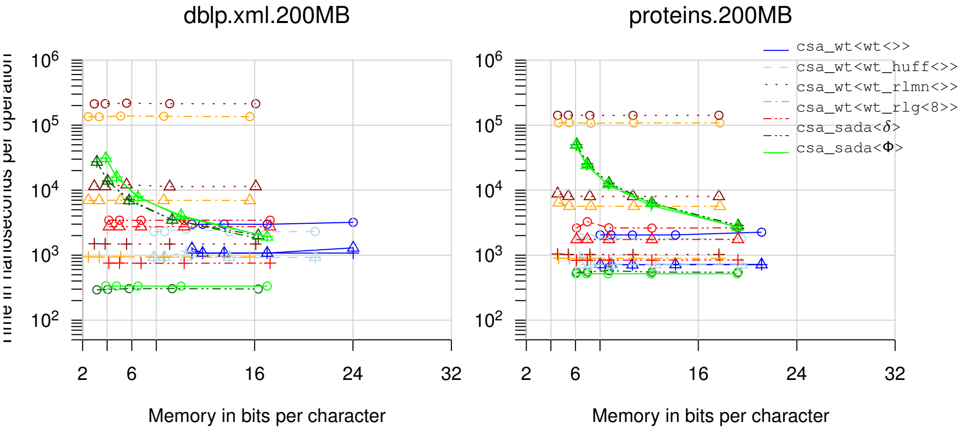
Runtime of CSA access (2)



Runtime of CSA access (3)

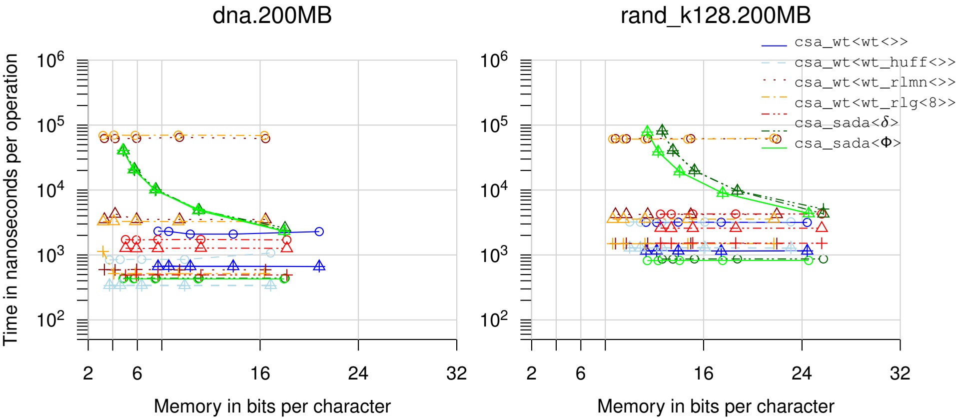


Runtime of CSA operations (1)



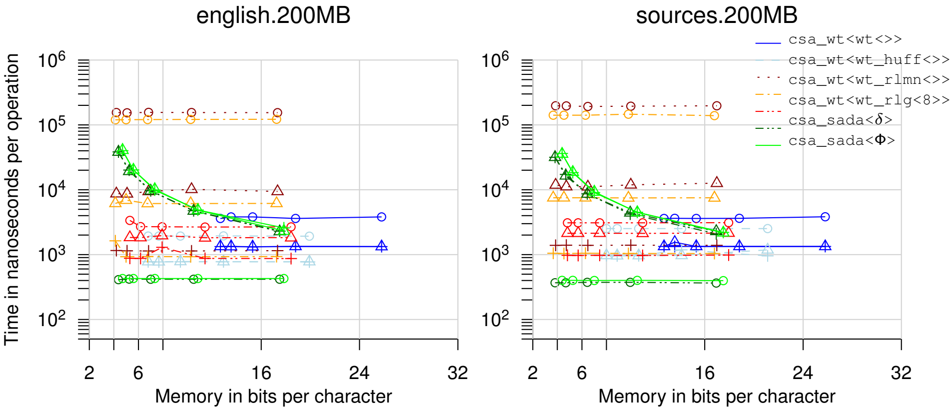
$\circ$ =psi[i], $\triangle$ =psi(i)=LF[i], $+$ =bwt[i]

Runtime of CSA operations (2)



○ =psi[i], △ =psi(i)=LF[i], + =bwt[i]

Runtime of CSA operations (3)



$\circ$ =psi[i], $\triangle$ =psi(i)=LF[i], $+$ =bwt[i]