Compressed Suffix Trees for Repetitive Collections based on Block Trees

Manuel Cáceres

CLEI 2020, CLTM

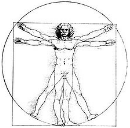
20/10/2020

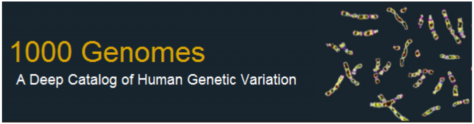
Context

- The amount of data is in constant growth

Context

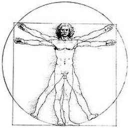
- The amount of data is in constant growth

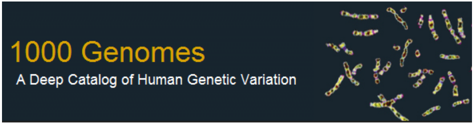




Context

- The amount of data is in constant growth

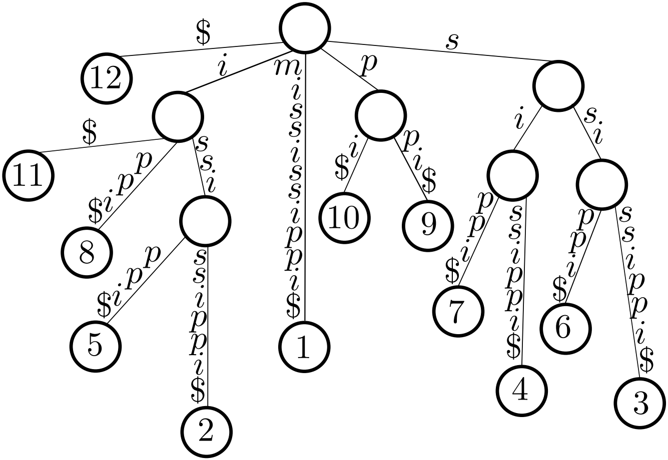




- Complex queries on these data are required

Suffix Tree

$$T = mississippi\$$$



Applications in Stringology/Bioinfomatics

Applications in Stringology/Bioinfomatics

- Approximate pattern matching

Applications in Stringology/Bioinfomatics

- Approximate pattern matching
- Longest common substring

Applications in Stringology/Bioinfomatics

- Approximate pattern matching
- Longest common substring
- Finding maximal repeats

Applications in Stringology/Bioinfomatics

- Approximate pattern matching
- Longest common substring
- Finding maximal repeats
- Computing matching statistics

Space Usage

- A human genome: $\sim 700MB$

Space Usage

- A human genome: $\sim 700MB$

- Suffix Tree: $\Theta(n \log n)$ bits

Space Usage

- A human genome: $\sim 700MB$

- Suffix Tree: $\Theta(n \log n)$ bits
    - Engineered implementation: ~ 80 bits per symbol (bps)

Space Usage

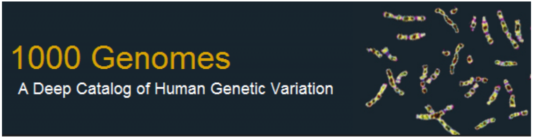
- A human genome: $\sim 700MB$
- Suffix Tree: $\Theta(n \log n)$ bits
  - Engineered implementation: ~ 80 bits per symbol (bps)
- Suffix Tree of one genome: $\sim 30GB$

Space Usage

- A human genome: $\sim 700MB$

- Suffix Tree: $\Theta(n \log n)$ bits
  - Engineered implementation: $\sim$ 80 bits per symbol (bps)

- Suffix Tree of one genome: $\sim 30GB$



$\sim 30TB$

Compressed Suffix Tree

Compressed Suffix Tree (CST)

Compressed Suffix Trees are formed by *Compact Data Structures*

Compressed Suffix Tree (CST)

Compressed Suffix Trees are formed by *Compact Data Structures*

- Compressed Suffix Array (CSA)

Compressed Suffix Tree (CST)

Compressed Suffix Trees are formed by *Compact Data Structures*

- Compressed Suffix Array (CSA)
- Compressed LCP

Compressed Suffix Tree (CST)

Compressed Suffix Trees are formed by *Compact Data Structures*

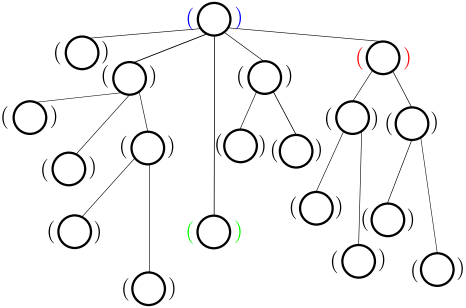
- Compressed Suffix Array (CSA)
- Compressed LCP
- Topology representation

Compressed Suffix Tree (CST)

Compressed Suffix Trees are formed by *Compact Data Structures*

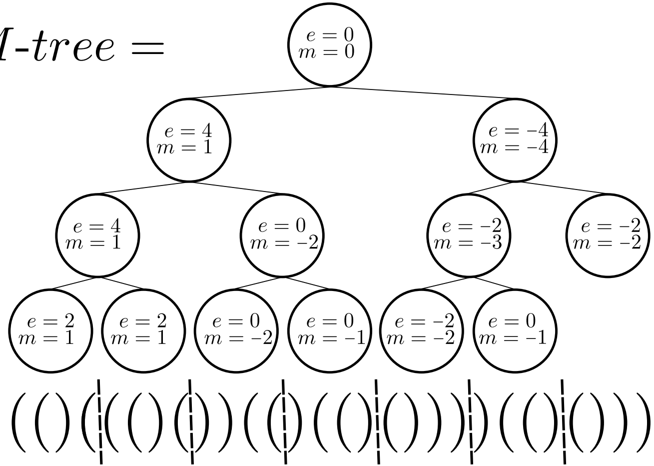
- Compressed Suffix Array (CSA)
- Compressed LCP
- Topology representation

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



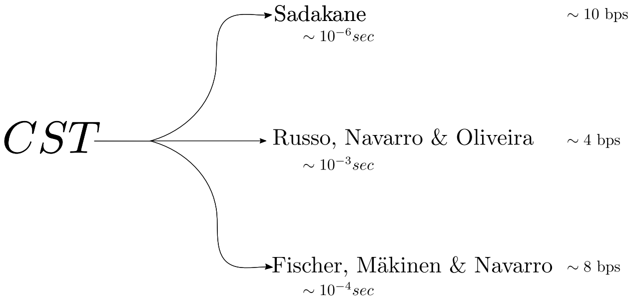
Range min-Max Tree

rmM-tree =



$P = (()((()())(()(()()))))(()()))$

State of the Art



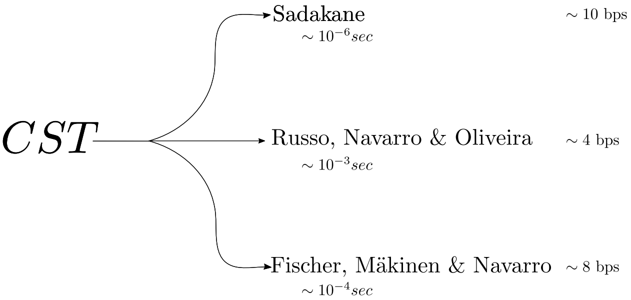
CST

Sadakane ~ 10 bps
$\sim 10^{-6} sec$

Russo, Navarro & Oliveira ~ 4 bps
$\sim 10^{-3} sec$

Fischer, Mäkinen & Navarro ~ 8 bps
$\sim 10^{-4} sec$

State of the Art



CST

Sadakane ~ 10 bps

$\sim 10^{-6} sec$

Russo, Navarro & Oliveira ~ 4 bps

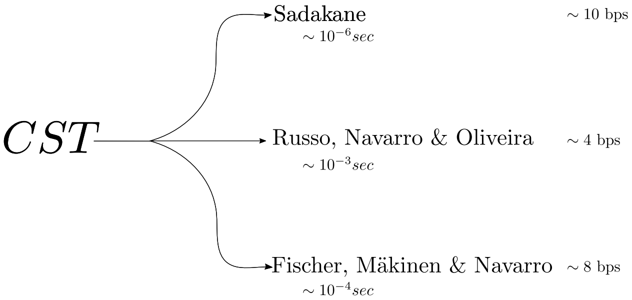
$\sim 10^{-3} sec$

Fischer, Mäkinen & Navarro ~ 8 bps

$\sim 10^{-4} sec$

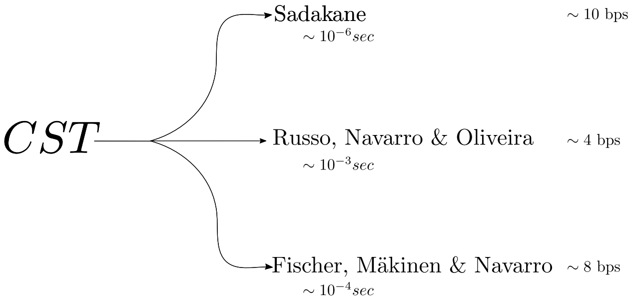
- Still a lot of space

State of the Art



- Still a lot of space
- Many collections are highly repetitive

State of the Art



- Still a lot of space
- Many collections are highly repetitive
- BWT-Runs, Lempel-Ziv and Grammar based indexes

Repetition-Aware CSTs

Repetition-Aware CSTs

- Abeliuk et. al

Repetition-Aware CSTs

- Abeliuk et. al
    - Run-length CSA (RLCSA)

Repetition-Aware CSTs

- Abeliuk et. al
    - Run-length CSA (RLCSA)
    - No parentheses topology

Repetition-Aware CSTs

- Abeliuk et. al
    - Run-length CSA (RLCSA)
    - No parentheses topology

Performance

It uses $\sim 1 - 2$ bps but operates in 10^{-3} sec.

Repetition-Aware CSTs

- Abeliuk et. al
    - Run-length CSA (RLCSA)
    - No parentheses topology

Performance

It uses $\sim 1 - 2$ bps but operates in 10^{-3} sec.

- *Grammar-Compressed Suffix Tree* (GCST)

Repetition-Aware CSTs

- Abeliuk et. al
  - Run-length CSA (RLCSA)
  - No parentheses topology

Performance

It uses $\sim 1-2$ bps but operates in 10^{-3} sec.

- *Grammar-Compressed Suffix Tree* (GCST)
  - Run-length CSA (RLCSA)

Repetition-Aware CSTs

- Abeliuk et. al
  - Run-length CSA (RLCSA)
  - No parentheses topology

Performance

It uses $\sim 1 - 2$ bps but operates in 10^{-3} sec.

- *Grammar-Compressed Suffix Tree* (GCST)
  - Run-length CSA (RLCSA)
  - Topology: *Grammar-Compressed Topology* (GCT)

Repetition-Aware CSTs

- Abeliuk et. al
    - Run-length CSA (RLCSA)
    - No parentheses topology

Performance

It uses $\sim 1 - 2$ bps but operates in 10^{-3} sec.

- *Grammar-Compressed Suffix Tree* (GCST)
    - Run-length CSA (RLCSA)
    - Topology: *Grammar-Compressed Topology* (GCT)

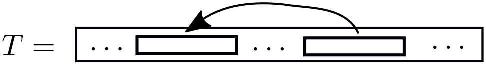
Performance

It uses ~ 2 bps and operates in 10^{-5} sec.

Block Tree

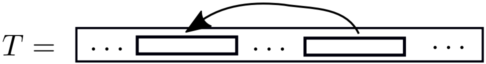
Block Tree

- *Lempel-Ziv* bounded structure

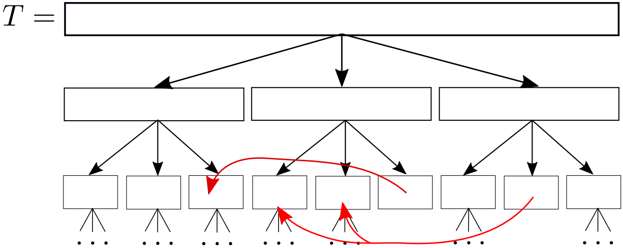


Block Tree

- *Lempel-Ziv* bounded structure

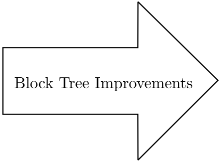


- It divides the text into blocks and uses *back pointers* to previous occurrences



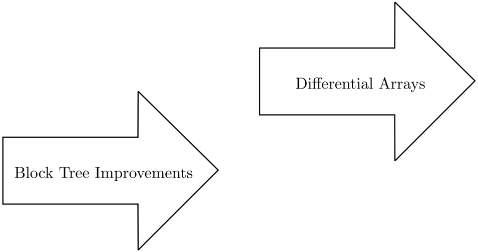
Work Done

Work Done



Block Tree Improvements

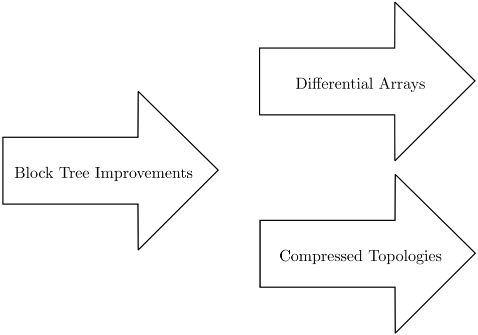
Work Done



Differential Arrays

Block Tree Improvements

Work Done

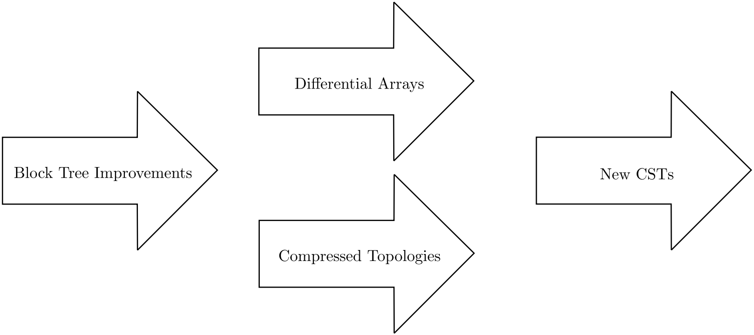


Differential Arrays

Block Tree Improvements

Compressed Topologies

Work Done



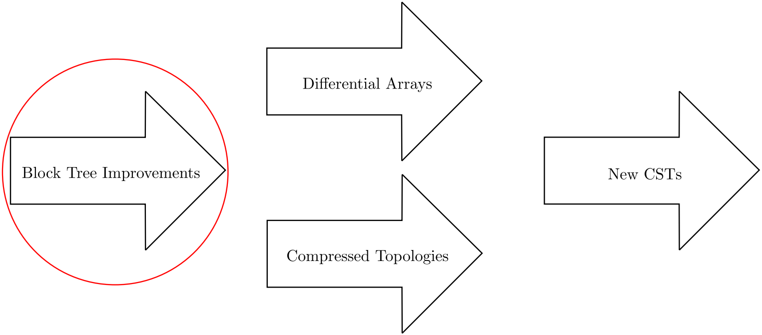
Block Tree Improvements

Differential Arrays

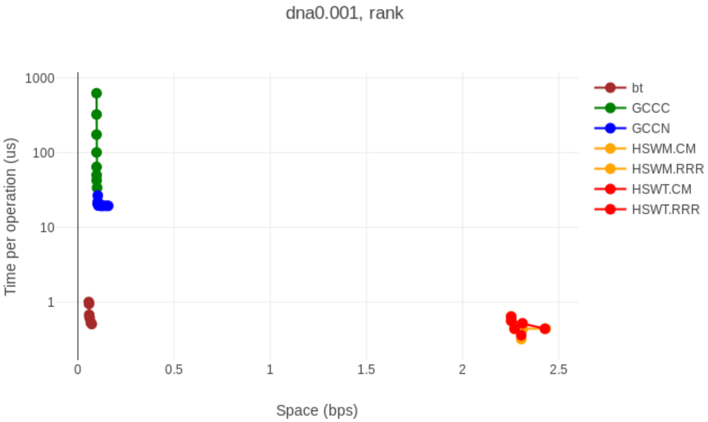
Compressed Topologies

New CSTs

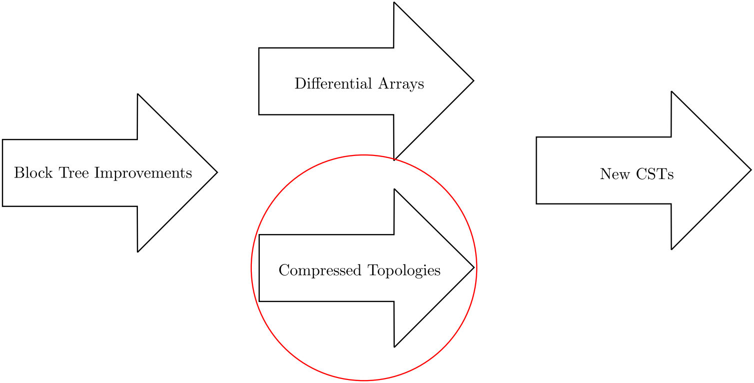
Block Tree Improvements



Results – Compared to State-of-the-art



Compressed Topologies



Block Tree Improvements

Differential Arrays

Compressed Topologies

New CSTs

Compressed Topologies

Block Tree Compressed Topology (BT-CT)

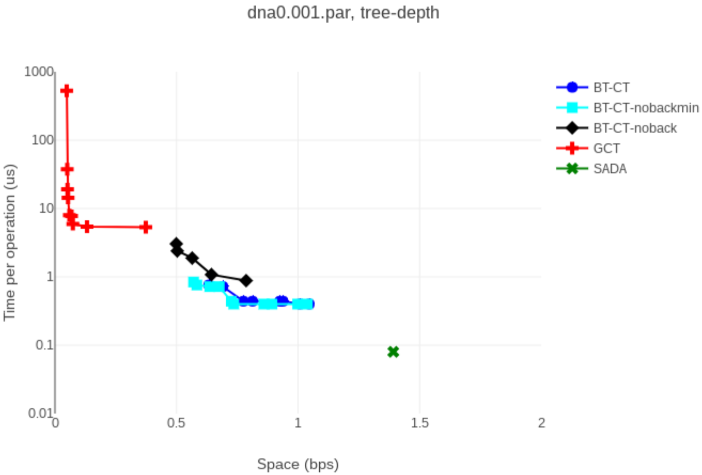
- Augmentation of Block Tree nodes with *leaf-rank*, *excess* and *min-excess* fields

Compressed Topologies

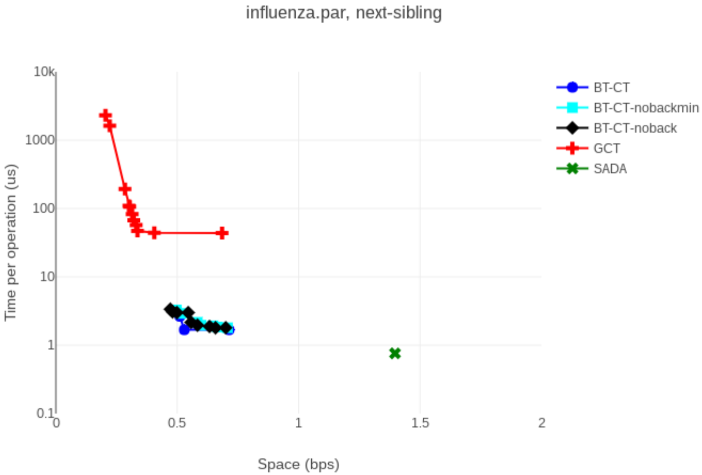
Block Tree Compressed Topology (BT-CT)

- Augmentation of Block Tree nodes with *leaf-rank*, *excess* and *min-excess* fields
- Implementation of Primitives Parentheses Operations
  - $excess(i)$
  - $leaf\text{-}rank(i)$
  - $leaf\text{-}select(j)$
  - $fwd\text{-}search(i, d)$
  - $bwd\text{-}search(i, d)$
  - $min\text{-}excess(i, j)$

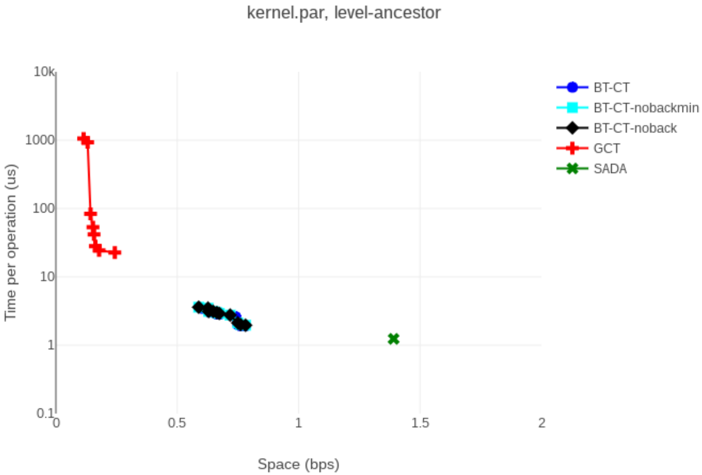
Results – *tree-depth*



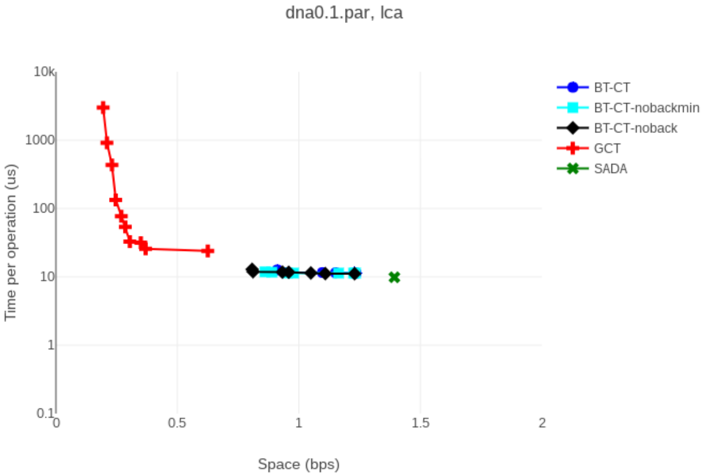
Results – *next-sibling*



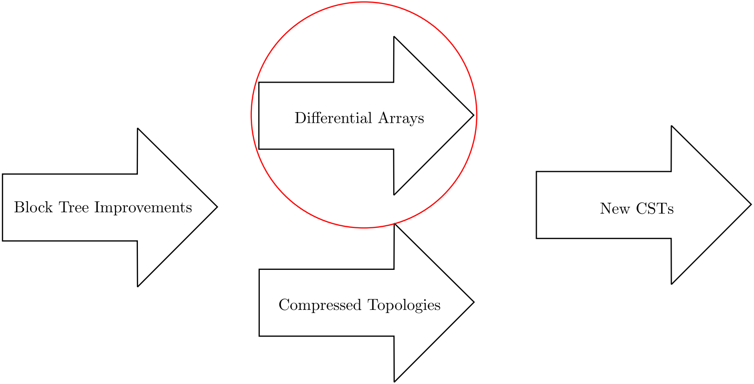
Results – *level-ancestor*



Results – *lca*


dna0.1.par, lca
Time per operation (us)
10k
1000
100
10
1
0.1
0
0.5
1
1.5
2
Space (bps)
BT-CT
BT-CT-nobackmin
BT-CT-noback
GCT
SADA

Differential Arrays



Differential Arrays

- Store the differences $A[i] - A[i-1]$, and a sampling

Differential Arrays

- Store the differences $A[i] - A[i-1]$, and a sampling
- $A[i] = A[s] + \sum_{j=s+1}^{i} A[j] - A[j-1]$

Differential Arrays

- Store the differences $A[i] - A[i-1]$, and a sampling
- $A[i] = A[s] + \sum_{j=s+1}^{i} A[j] - A[j-1]$
- Differential encodings of the suffix array A, its inverse A^{-1} and the LCP inherits the repetitiveness from its input

Differential Arrays

- Store the differences $A[i] - A[i-1]$, and a sampling
- $A[i] = A[s] + \sum_{j=s+1}^{i} A[j] - A[j-1]$
- Differential encodings of the suffix array A, its inverse A^{-1} and the LCP inherits the repetitiveness from its input
- We adapted Block Trees and Grammar structures
  - Similar to rank
  - Replace rank fields by partial sums of differences

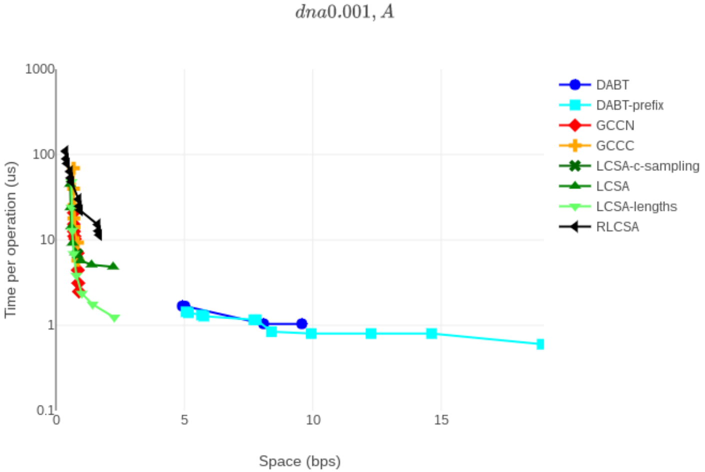
Differential Arrays

- Analogous adaptation of Grammar GCC structure

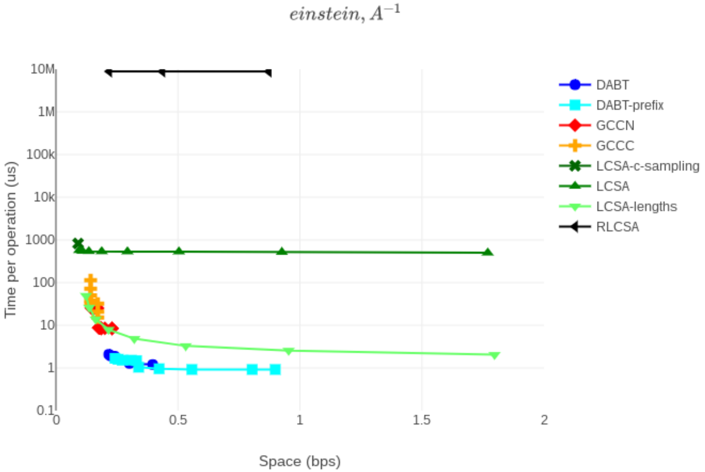
Differential Arrays

- Analogous adaptation of Grammar GCC structure
- New variant and Augmentation of *Locally Compressed Suffix Array* (LCSA)
  - *LCSA-c-sampling*
  - *LCSA-lengths*

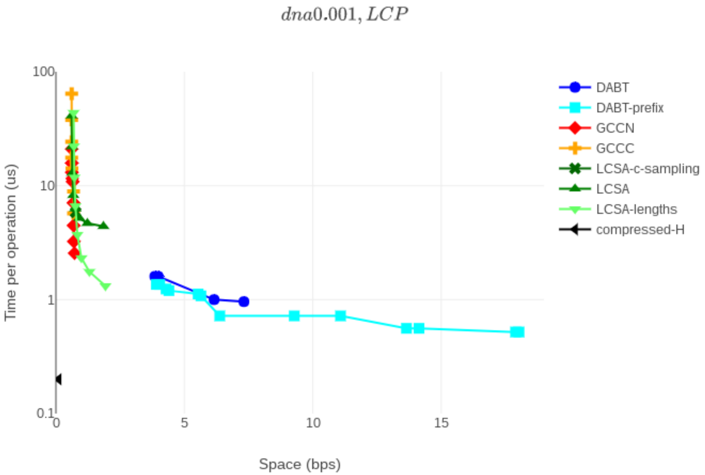
Results – *A*


*dna*0.001, *A*
Time per operation (us)
1000
100
10
1
0.1
0
5
10
15
Space (bps)
DABT
DABT-prefix
GCCN
GCCC
LCSA-c-sampling
LCSA
LCSA-lengths
RLCSA

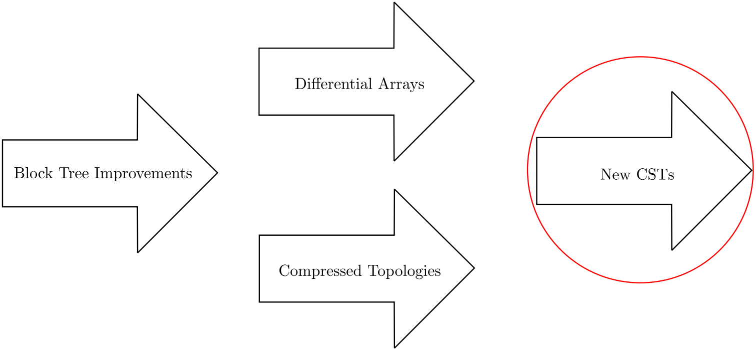
Results – A^{-1}


einstein, A^{-1}
Time per operation (us)
10M
1M
100k
10k
1000
100
10
1
0.1
0
0.5
1
1.5
2
Space (bps)
DABT
DABT-prefix
GCCN
GCCC
LCSA-c-sampling
LCSA
LCSA-lengths
RLCSA

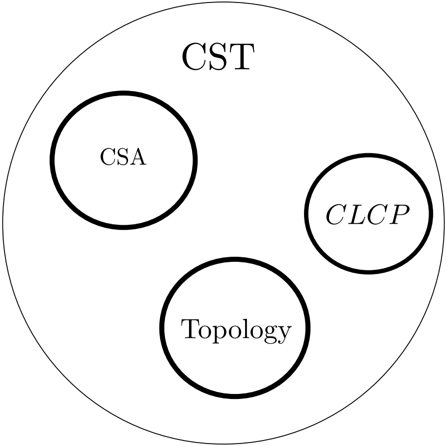
Results – LCP


$dna0.001, LCP$
Time per operation (us)
100
10
1
0.1
0
5
10
15
Space (bps)
DABT
DABT-prefix
GCCN
GCCC
LCSA-c-sampling
LCSA
LCSA-lengths
compressed-H

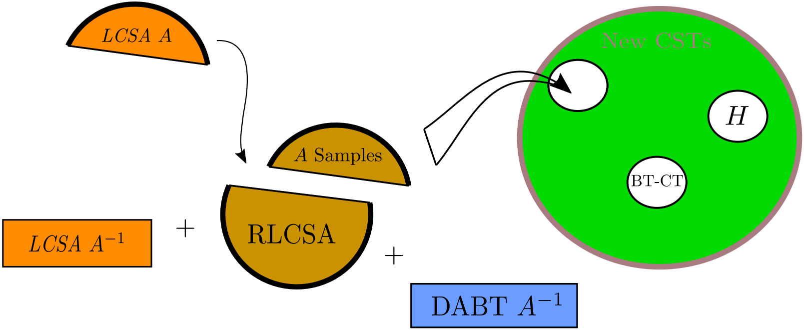
New CSTs


Block Tree Improvements
Differential Arrays
Compressed Topologies
New CSTs

New CSTs

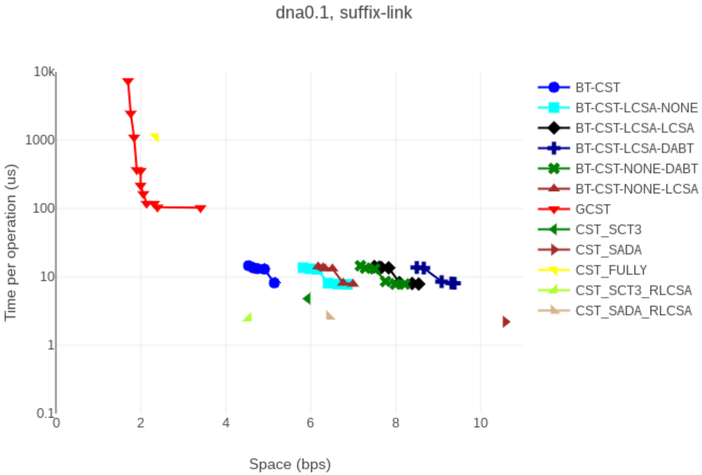


New CSTs

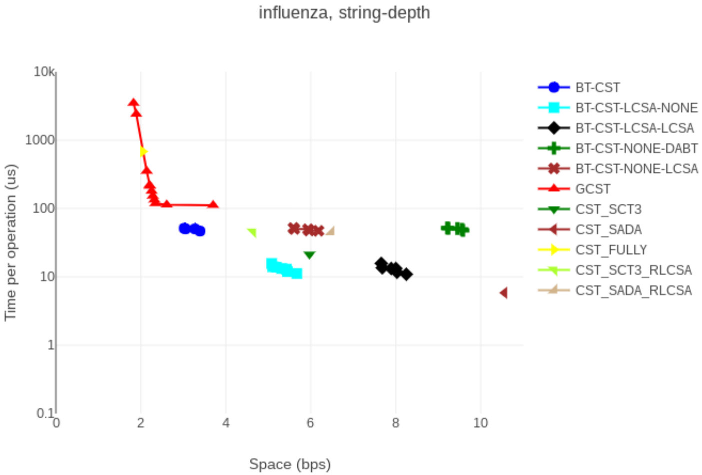


BT-CST-{LCSA, NONE}-{LCSA, DABT, NONE}

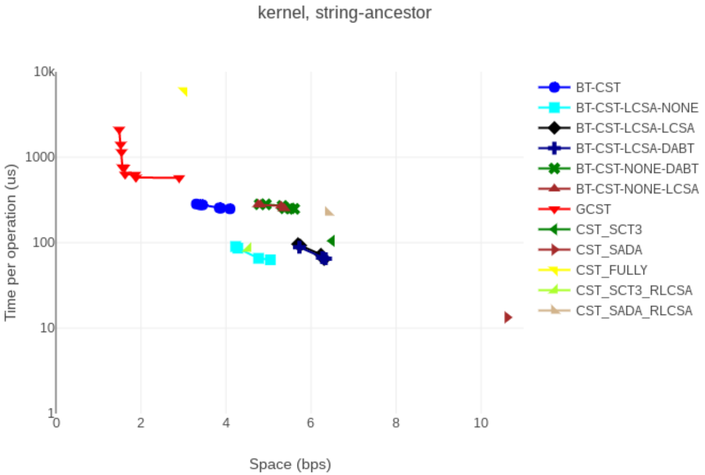
Results – *suffix-link*



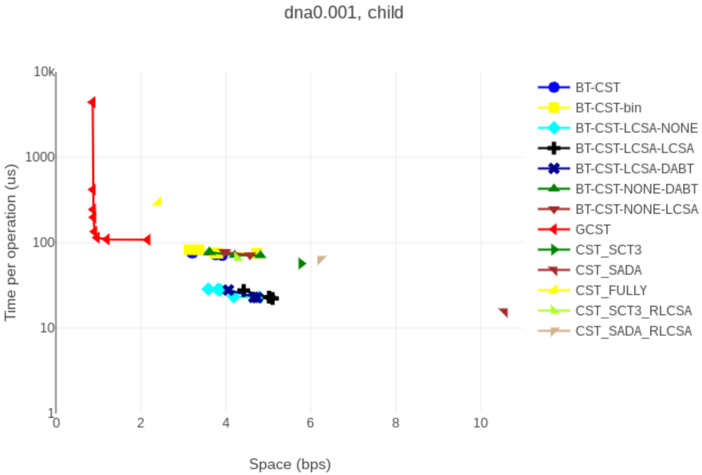
Results – *string-depth*



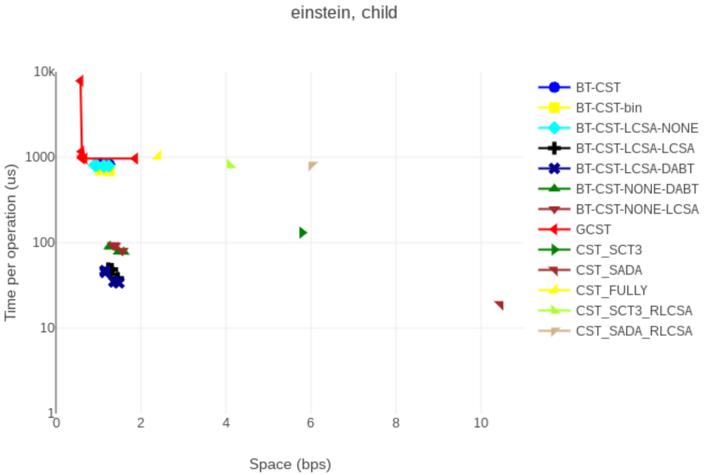
Results – *string-ancestor*



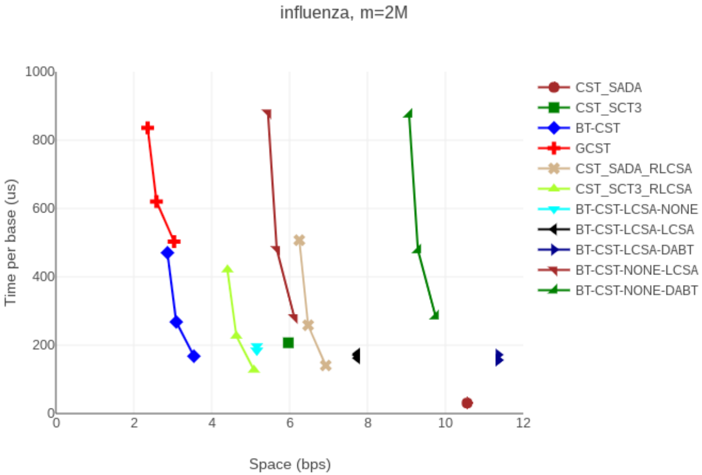
Results – *child*



Results – *child*



Results – Maximal Substrings



Conclusions

Conclusions

- Practical and theoretical enrichment of Block Trees

Conclusions

- Practical and theoretical enrichment of Block Trees
- New structures for repetitive differential encodings

Conclusions

- Practical and theoretical enrichment of Block Trees
- New structures for repetitive differential encodings
- Fastest repetition-aware parenthesis topology

Conclusions

- Practical and theoretical enrichment of Block Trees
- New structures for repetitive differential encodings
- Fastest repetition-aware parenthesis topology
- Fastest repetition-aware compressed suffix tree

Conclusions

- Practical and theoretical enrichment of Block Trees
- New structures for repetitive differential encodings
- Fastest repetition-aware parenthesis topology
- Fastest repetition-aware compressed suffix tree
- Public available code for researchers and practitioners

Compressed Suffix Trees for Repetitive Collections based on Block Trees

Manuel Cáceres

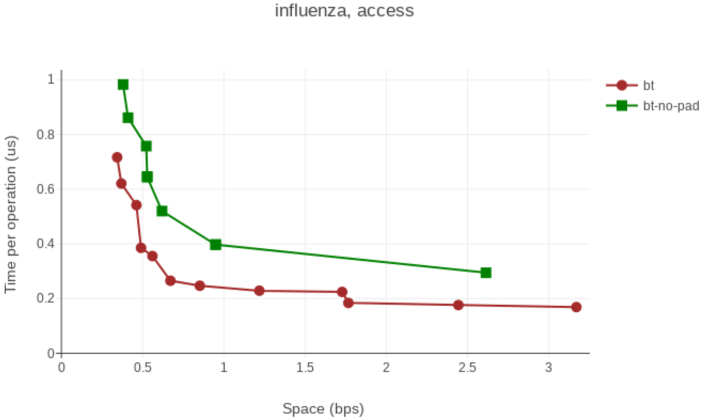
CLEI 2020, CLTM

20/10/2020

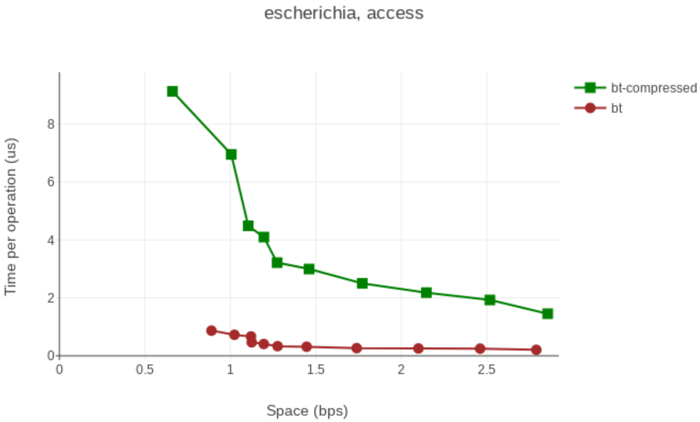
Other heuristic improvements

- Padding versus No Padding
- Compressed components versus Plain components

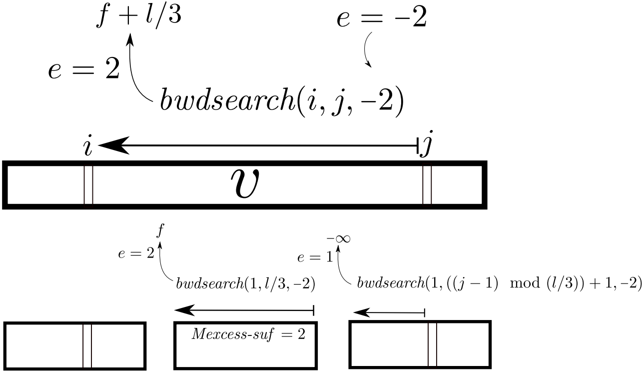
Results – Padding



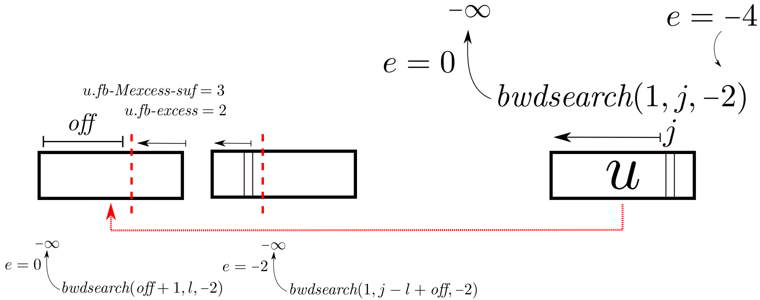
Results – Compressed Components



$bwd\text{-}search(i, d \leq 0)$

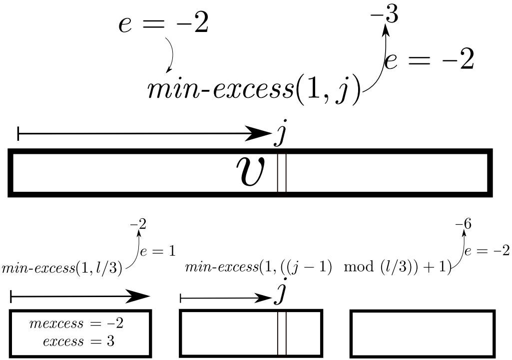


$bwd\text{-}search(i, d \leq 0)$



$O(1)$ time

$min\text{-}excess(i, j)$



$e = -2$

-3

$e = -2$

$min\text{-}excess(1, j)$

j

v

-2

$e = 1$

$min\text{-}excess(1, l/3)$

-6

$e = -2$

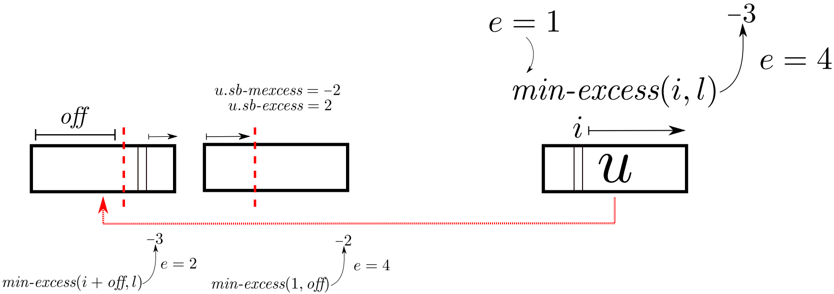
$min\text{-}excess(1, ((j-1) \mod (l/3)) + 1)$

j

$mexcess = -2$
$excess = 3$

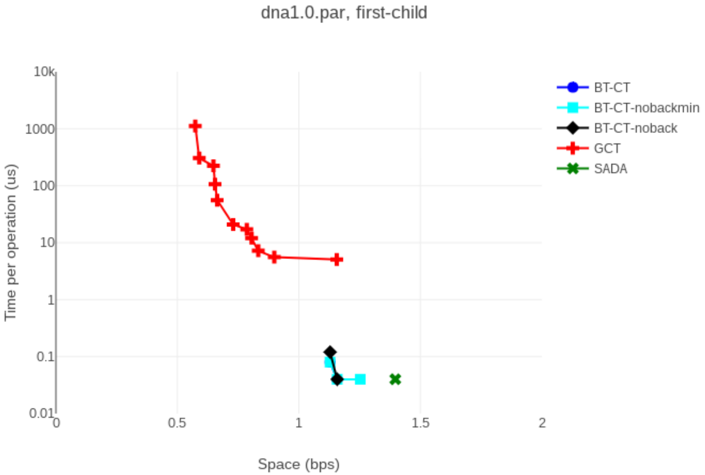
$O(1)$ time

$min\text{-}excess(i, j)$

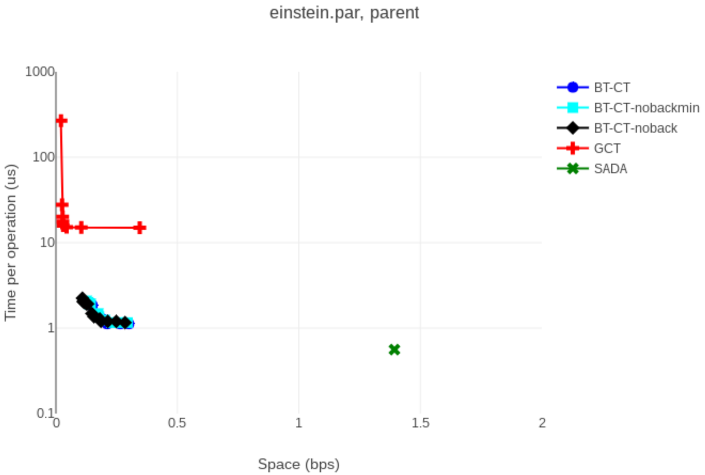


$O(1)$ time

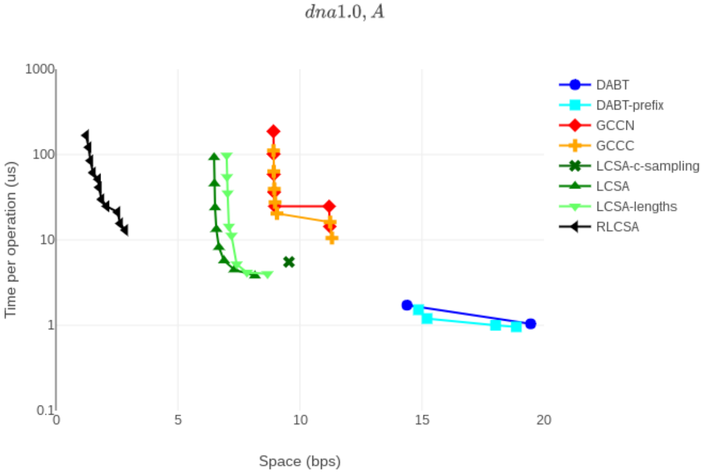
Results – *first-child*



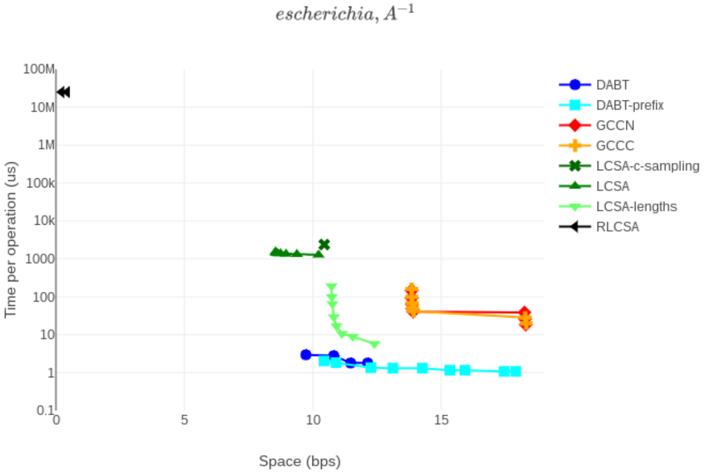
Results – *parent*



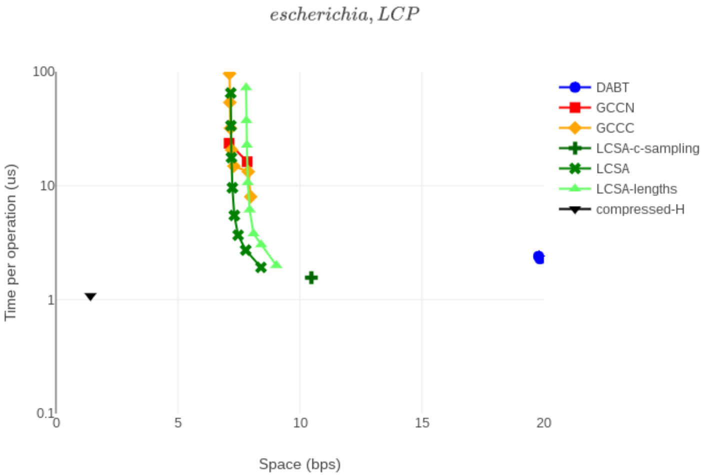
Results – A


$dna1.0$, A
Time per operation (us)
1000
100
10
1
0.1
0
5
10
15
20
Space (bps)
DABT
DABT-prefix
GCCN
GCCC
LCSA-c-sampling
LCSA
LCSA-lengths
RLCSA

Results – A^{-1}

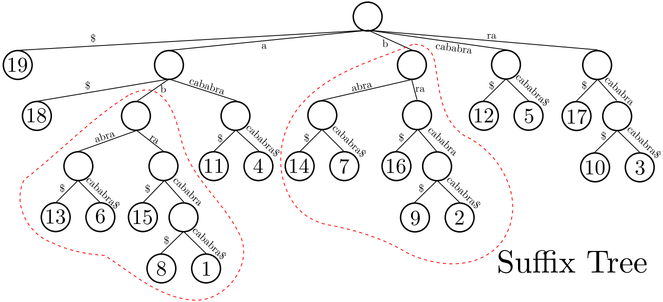

escherichia, A^{-1}
Time per operation (us)
100M
10M
1M
100k
10k
1000
100
10
1
0.1
0
5
10
15
Space (bps)
DABT
DABT-prefix
GCCN
GCCC
LCSA-c-sampling
LCSA
LCSA-lengths
RLCSA

Results – LCP


$escherichia, LCP$
Time per operation (us)
100
10
1
0.1
0
5
10
15
20
Space (bps)
DABT
GCCN
GCCC
LCSA-c-sampling
LCSA
LCSA-lengths
compressed-H

Repetitiveness

$$T = abracababracababra\$$$



Suffix Tree

$$P = (()(()((()())(()(()())))(()()))((()())(()(()())))(()())(()(()())))$$

$\Delta A =$

19	−1	−5	−7	9	−7	−7	10	−7	10	−7	9	−7	−7	10	−7	12	−7	−7

$\Delta A^{-1} =$

7	7	5	−10	7	−12	7	−5	7	5	−10	7	−12	7	−5	7	5	−15	−1

$\Delta LCP =$

0	0	1	5	−4	2	7	−10	7	−8	5	−4	2	7	−10	7	−7	2	7

Maximal Substrings Problem

Find all maximal substrings of $S[1, m]$ that are also substrings of a text $T[1, n]$

- Solved in $O(m)$ using the *suffix tree* of T
    - The algorithm maintains two integers i, j representing a substring $S[i, j]$
    - It uses *child* to advance j, when no possible outputs a maximal substring and starts applying *suffix-link* to advance i until an application of *child* is possible again

Real Pruning

To avoid dependency issues we need to eliminate these expansions in a *postorder right-to-left* traversal of the Block Tree. Moreover, when analyzing a block, it is enough to check if its children are all leaves, because if they were unnecessary expansions they would have been already processed in the traversal and turned into BackBlocks.

Theoretical Work on Block Trees

- At each level l (where the root is at level 0) the blocks are of lengths either $\left\lceil \frac{n}{r^l} \right\rceil$ or $\left\lfloor \frac{n}{r^l} \right\rfloor$
- Block Trees are well defined, that is, BackBlocks point to a well-defined block or pair of blocks in the same level, containing its leftmost occurrence. Even more, this block or pair of blocks are InternalBlocks
- The Block Tree can be implemented using $O\left(zrh_{bt}\log n\right)$ bits of space
- Pruning as alternative definition

Real Block Tree definition

A node v, representing $v.blk = T[i, i+b-1]$ can be of three types:

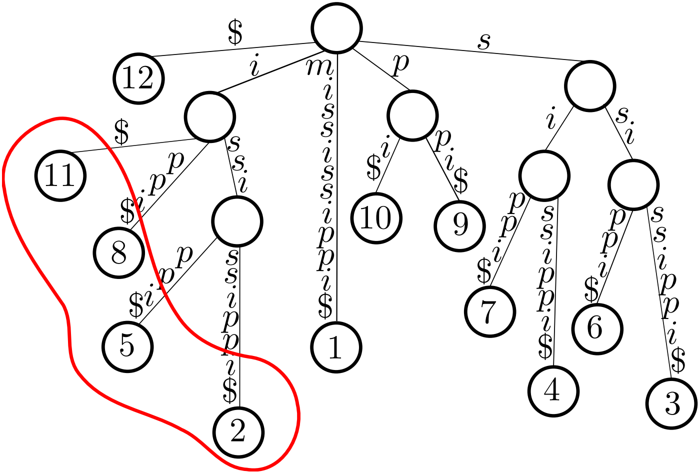
LeafBlock: If $b \leq mll$, where mll is a parameter, then v is a leaf of the Block Tree

BackBlock: Otherwise, if $T[i-b, i+b-1]$ and $T[i, i+2b-1]$ are not their leftmost occurrences in T, then the block is replaced by its leftmost occurrence in T

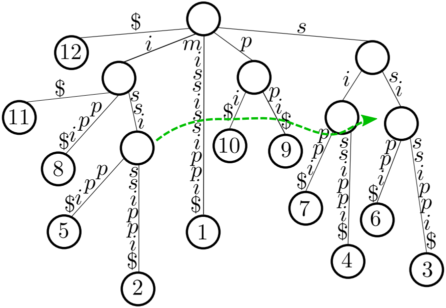
InternalBlock: Otherwise, the block is split into r blocks of size $\left\lceil \frac{b}{r} \right\rceil$ and $\left\lfloor \frac{b}{r} \right\rfloor$

Secondary Memory

$$T = mississippi\$$$

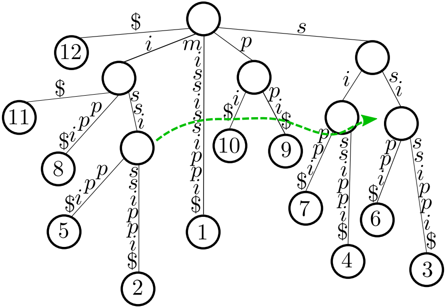


Suffix Array (SA)



12	11	8	5	2	1	10	9	7	4	6	3

Suffix Array (SA)

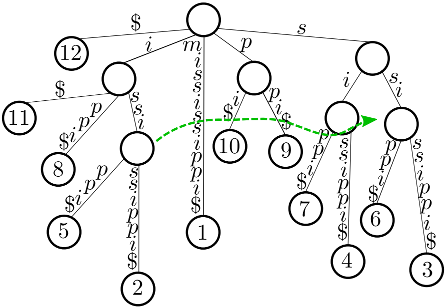


12	11	8	5	2	1	10	9	7	4	6	3

Longest Common Prefix (LCP)

$$LCP[i] = lcp\left(T[SA[i-1 \ldots n], T[SA[i \ldots n]\right)$$

Suffix Array (SA) → 48 bps

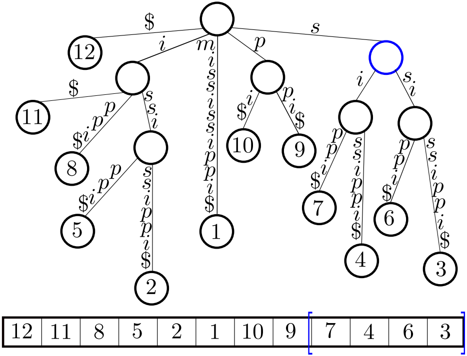


12	11	8	5	2	1	10	9	7	4	6	3

Longest Common Prefix (LCP)

$$LCP[i] = lcp\,(T[SA[i-1\ldots n], T[SA[i\ldots n])$$

Interval-based topology representation



12	11	8	5	2	1	10	9	7	4	6	3

State of the Art

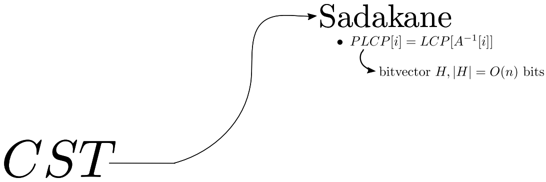
CST

Sadakane

- $PLCP[i] = LCP[A^{-1}[i]]$
  - bitvector $H, |H| = O(n)$ bits

State of the Art

- Parenthesis topology: $4n + o(n)$ bits



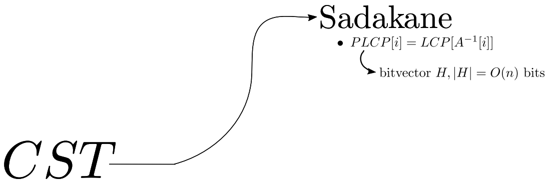
CST

Sadakane

- $PLCP[i] = LCP[A^{-1}[i]]$
  - bitvector $H, |H| = O(n)$ bits

State of the Art

- Parenthesis topology: $4n + o(n)$ bits
- Bitvector H: $2n + o(n)$ bits



CST

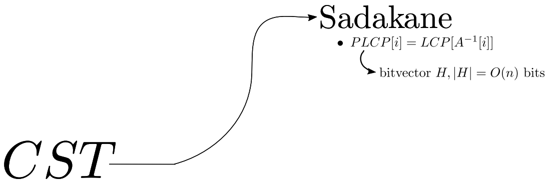
Sadakane

- $PLCP[i] = LCP[A^{-1}[i]]$

bitvector $H, |H| = O(n)$ bits

State of the Art

- Parenthesis topology: $4n + o(n)$ bits
- Bitvector H: $2n + o(n)$ bits
- $lcp(T[SA[i] \dots n], T[SA[j] \dots n]) =$ $LCP[rank_{()}(P, rmq_{excess(P)}(rank_{()}(j), rank_{()}(i)))]$



CST

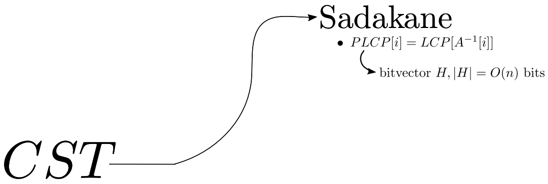
Sadakane

- $PLCP[i] = LCP[A^{-1}[i]]$

bitvector $H, |H| = O(n)$ bits

State of the Art

- Parenthesis topology: $4n + o(n)$ bits
- Bitvector H: $2n + o(n)$ bits
- $lcp(T[SA[i] \ldots n], T[SA[j] \ldots n]) = LCP[rank_{()}(P, rmq_{excess(P)}(rank_{()}(j), rank_{()}(i)))]$



CST

Sadakane

- $PLCP[i] = LCP[A^{-1}[i]]$

  bitvector $H, |H| = O(n)$ bits

Performance

Implementations based on Sadakane's CST use $\sim$ 10 bps and operate in the order of microseconds.

State of the Art

$CST \longrightarrow$ Russo et. al

- Sampling of nodes

State of the Art

- FM-index as CSA. Uses $|CSA| + o(n)$ bits

CST → Russo et. al

  - Sampling of nodes

State of the Art

- FM-index as CSA. Uses $|CSA| + o(n)$ bits
- Sample of *near* nodes through *suffix links*

CST → Russo et. al

- Sampling of nodes

State of the Art

- FM-index as CSA. Uses $|CSA| + o(n)$ bits
- Sample of *near* nodes through *suffix links*
- $LCSA(u, v)$ = lowest common sampled ancestor of u and v

CST → Russo et. al

- Sampling of nodes

State of the Art

- FM-index as CSA. Uses $|CSA| + o(n)$ bits
- Sample of *near* nodes through *suffix links*
- $LCSA(u, v)$ = lowest common sampled ancestor of u and v

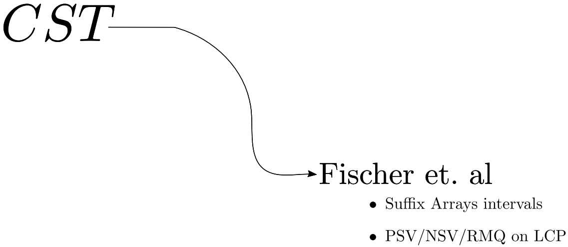
$CST \longrightarrow$ Russo et. al

- Sampling of nodes

Performance

Implementations based on Russo's CST use ~ 4 bps but operate in the order of milliseconds.

State of the Art



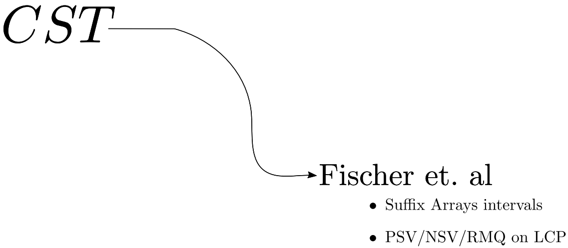
CST

Fischer et. al

- Suffix Arrays intervals
- PSV/NSV/RMQ on LCP

State of the Art

- *Run Length encoding* of bitvector H



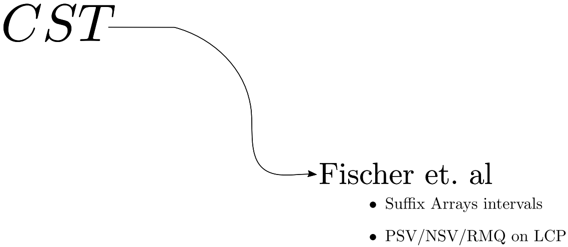
CST

Fischer et. al

- Suffix Arrays intervals
- PSV/NSV/RMQ on LCP

State of the Art

- *Run Length encoding* of bitvector H
- Does not use topology. Suffix tree nodes are represented as suffix array intervals



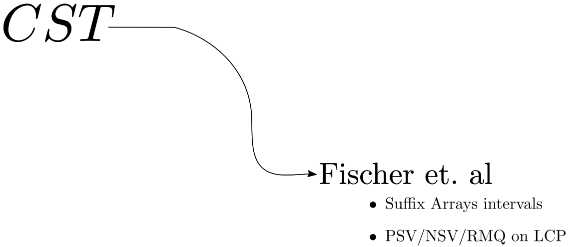
CST

Fischer et. al

- Suffix Arrays intervals
- PSV/NSV/RMQ on LCP

State of the Art

- *Run Length encoding* of bitvector H
- Does not use topology. Suffix tree nodes are represented as suffix array intervals
- rmq, psv/nsv over LCP simulate the operations



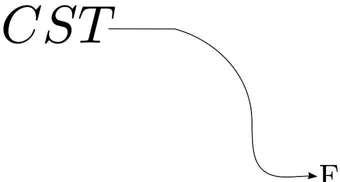
CST

Fischer et. al

- Suffix Arrays intervals
- PSV/NSV/RMQ on LCP

State of the Art

- *Run Length encoding* of bitvector H
- Does not use topology. Suffix tree nodes are represented as suffix array intervals
- rmq, psv/nsv over LCP simulate the operations



CST

Fischer et. al

- Suffix Arrays intervals
- PSV/NSV/RMQ on LCP

Performance

Implementations based on Fischer's CST use ~ 8 bps but operate in the order of hundred of milliseconds.

State of the Art

Sadakane

- $PLCP[i] = LCP[A^{-1}[i]]$
  - bitvector $H, |H| = O(n)$ bits

CST

Primitive Parentheses Operations

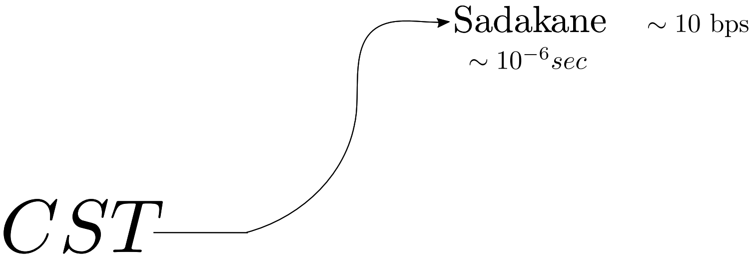
- $excess(i) = rank_{(}(i) - rank_{)}(i)$
- $leaf\text{-}rank(i) = rank_{()}(i) = |\{1 \leq j \leq i-1 \mid P[j] = (\wedge P[j+1] =)\}|$
- $leaf\text{-}select(j) = select_{()}(j) = \min(\{i \mid leaf\text{-}rank(i+1) = j\} \cup \{\infty\})$
- $fwd\text{-}search(i,d) = \min(\{j > i \mid excess(j) = excess(i) + d)\} \cup \{\infty\})$
- $bwd\text{-}search(i,d) = \max(\{j < i \mid excess(j) = excess(i) + d)\} \cup \{-\infty\})$
- $min\text{-}excess(i,j) = \min(\{excess(k) - excess(i-1) \mid i \leq k \leq j\} \cup \{\infty\})$

Some reductions

$$tree\text{-}depth(v) = excess(v)$$

$$next\text{-}sibling(v) = fwd\text{-}search(v,-1) + 1$$

State of the Art



CST

Sadakane ~ 10 bps

$\sim 10^{-6} sec$

State of the Art

$CST \longrightarrow$ Russo et. al

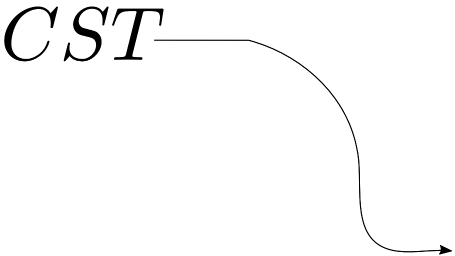
- Sampling of nodes

State of the Art

$CST \longrightarrow$ Russo et. al ~ 4 bps

$\sim 10^{-3} sec$

State of the Art

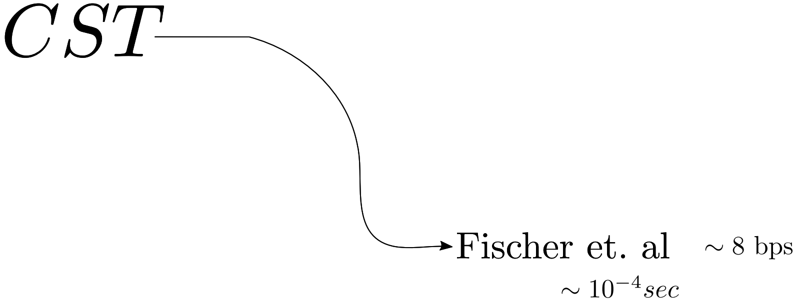


CST

Fischer et. al

- Suffix Arrays intervals
- PSV/NSV/RMQ on LCP

State of the Art

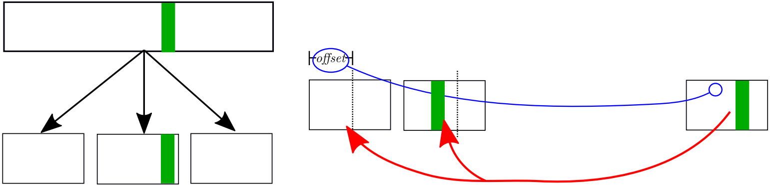


CST

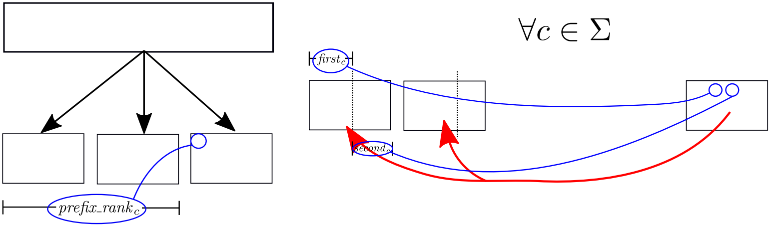
Fischer et. al ~ 8 bps

$\sim 10^{-4} sec$

Block Tree: Access



Block Tree: Rank/Select



$\forall c \in \Sigma$

Block Tree Improvements

- Introduced in “Queries on LZ-Bounded Encodings” (QLZBE)
    - We proved properties stated in the publication
- Implemented by Ordoñez

Block Tree Improvements

- Introduced in “Queries on LZ-Bounded Encodings” (QLZBE)
    - We proved properties stated in the publication
- Implemented by Ordoñez
    - Differs from theoretical proposal
    - Does not ensure the Lempel-Ziv bound
    - We implemented Block Trees following the theoretical proposal

Block Tree Improvements

- Introduced in “Queries on LZ-Bounded Encodings” (QLZBE)
  - We proved properties stated in the publication
- Implemented by Ordoñez
  - Differs from theoretical proposal
  - Does not ensure the Lempel-Ziv bound
  - We implemented Block Trees following the theoretical proposal
- A lot of new versions (not LZ bounded) emerge while working on the paper version
  - `block_tree`, `block_tree_no_clean`, `pruning_c_block_tree`, `heuristic_block_tree`, `heuristic_concatenate_block_tree`, `liberal_heuristic_block_tree`, `conservative_heuristic_block_tree`, `back_front_block_tree`, among others

Block Tree Improvements

- Construction algorithm
    - Fix construction algorithm given at QLZBE

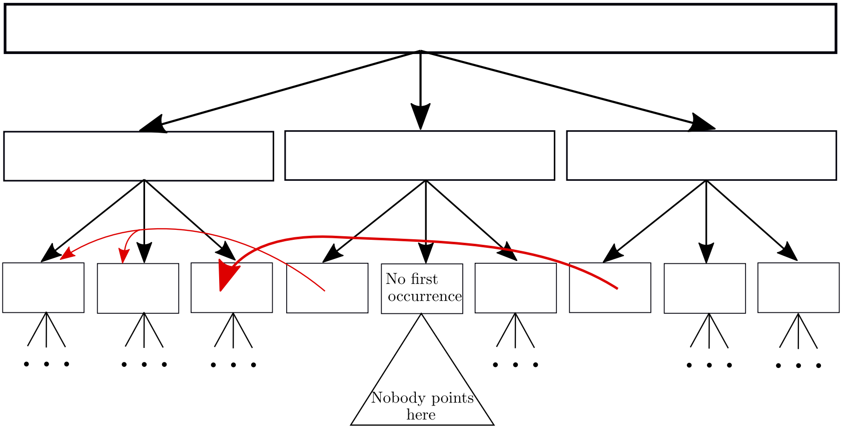
Block Tree Improvements

- Construction algorithm
    - Fix construction algorithm given at QLZBE
- Pruning improvement
    - Removal of *unnecessary expansions*

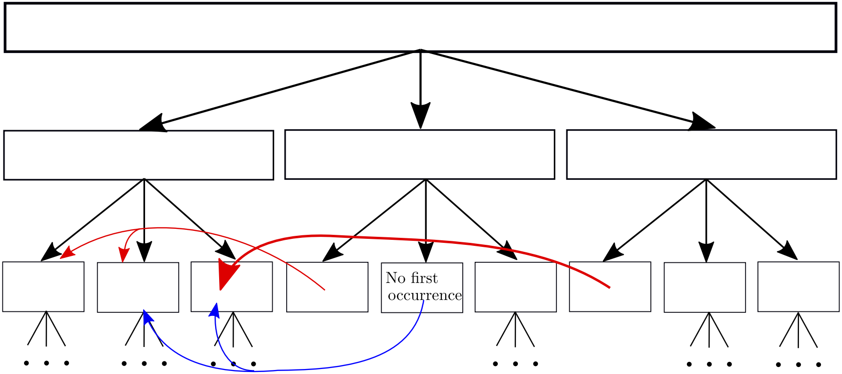
Block Tree Improvements

- Construction algorithm
    - Fix construction algorithm given at QLZBE
- Pruning improvement
    - Removal of *unnecessary expansions*
- New fields for the blocks

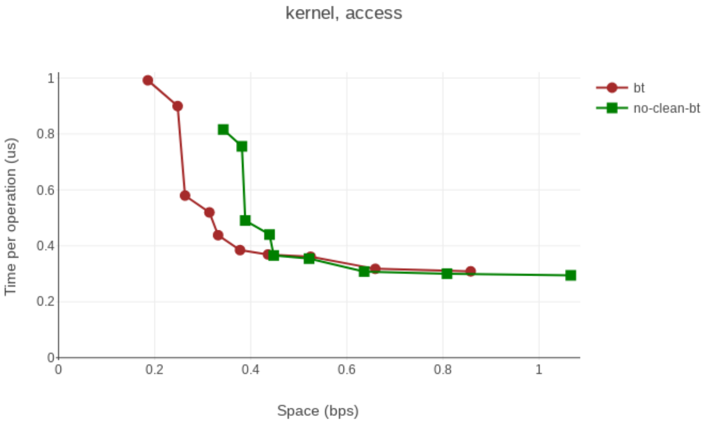
Unnecessary Expansions



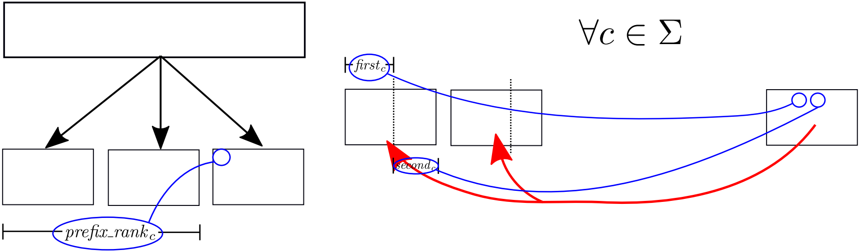
Unnecessary Expansions



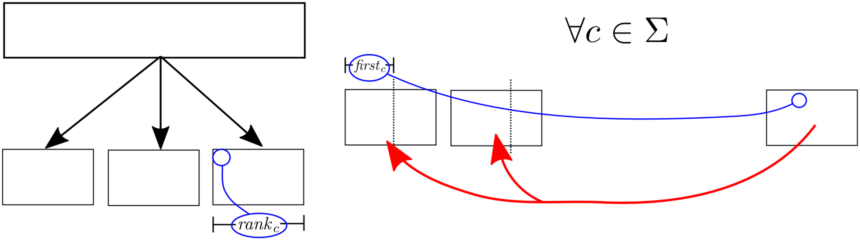
Results – Pruning



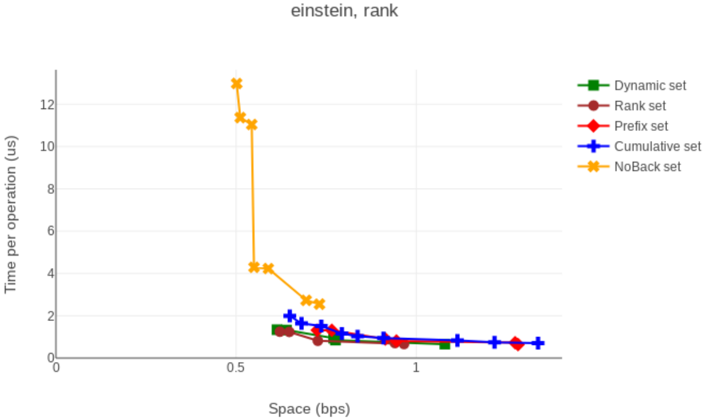
Prefix Set



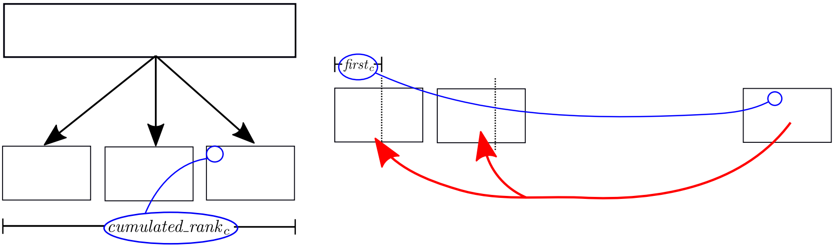
Rank Set



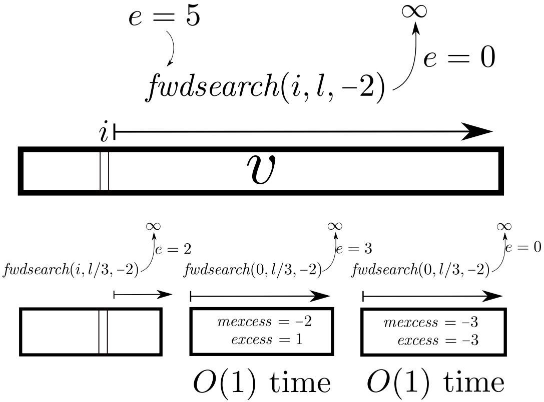
Results – Fields for rank/select



Better Version



$\mathit{fwd\text{-}search}(i, d \leq 0)$



$e = 5$

∞

$e = 0$

$\mathit{fwdsearch}(i, l, -2)$

i

v

∞ $e = 2$

$\mathit{fwdsearch}(i, l/3, -2)$

∞ $e = 3$

$\mathit{fwdsearch}(0, l/3, -2)$

∞ $e = 0$

$\mathit{fwdsearch}(0, l/3, -2)$

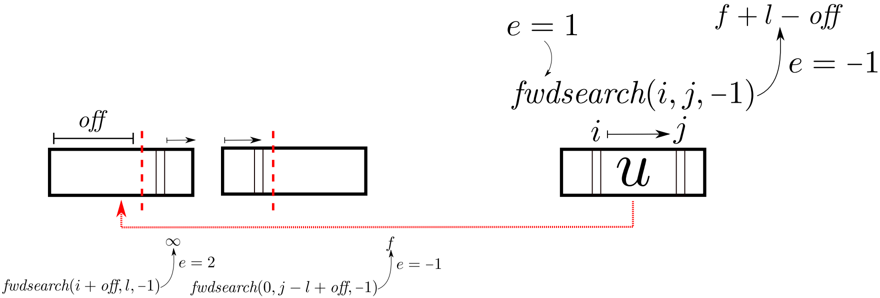
$\mathit{mexcess} = -2$
$\mathit{excess} = 1$

$\mathit{mexcess} = -3$
$\mathit{excess} = -3$

$O(1)$ time

$O(1)$ time

$fwd\text{-}search(i, d \le 0)$



DABT