# Faster Repetition-Aware Compressed Suffix Trees based on Block Trees

Manuel Cáceres / Gonzalo Navarro

SPIRE

09/09/2019

# Context

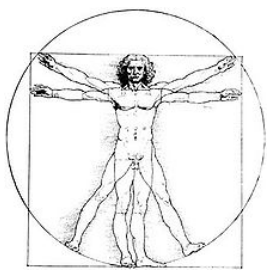
- The amount of data is in constant growth

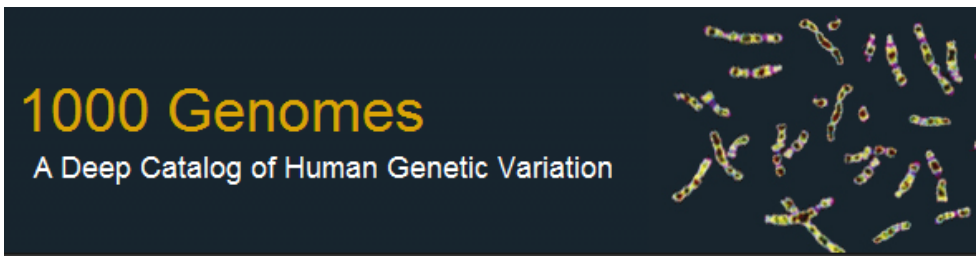
# Context

- The amount of data is in constant growth
- Complex queries on these data are required

# Context

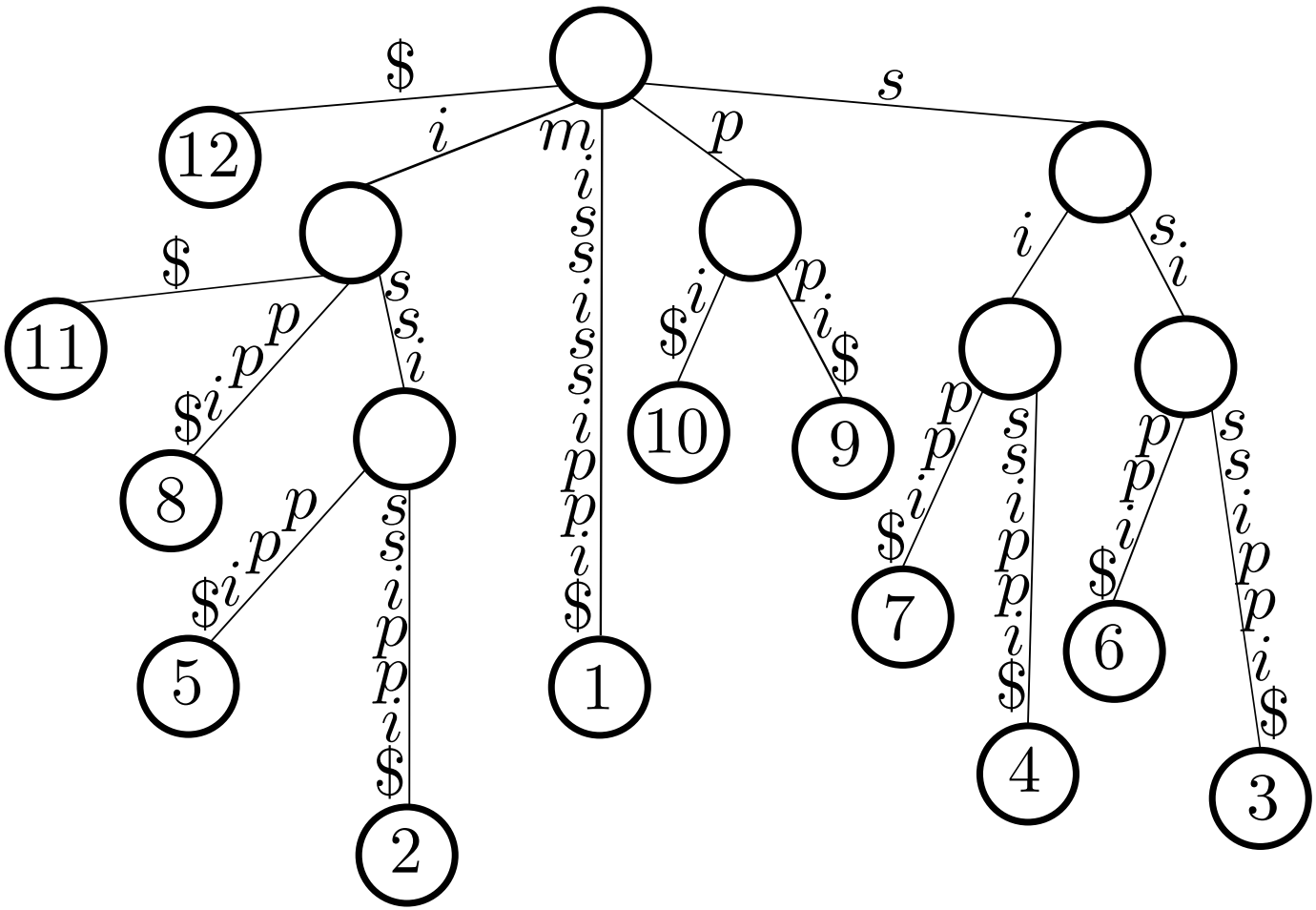
- The amount of data is in constant growth
- Complex queries on these data are required






# Suffix Tree

$$T = mississippi\$$$



# Space Usage

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

# Space Usage

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

- Engineered implementation: $\sim 80$ bits per symbol (bps)

# Space Usage

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

- Engineered implementation: $\sim 80$ bits per symbol (bps)
- A human genome: $\sim 700MB$

# Space Usage

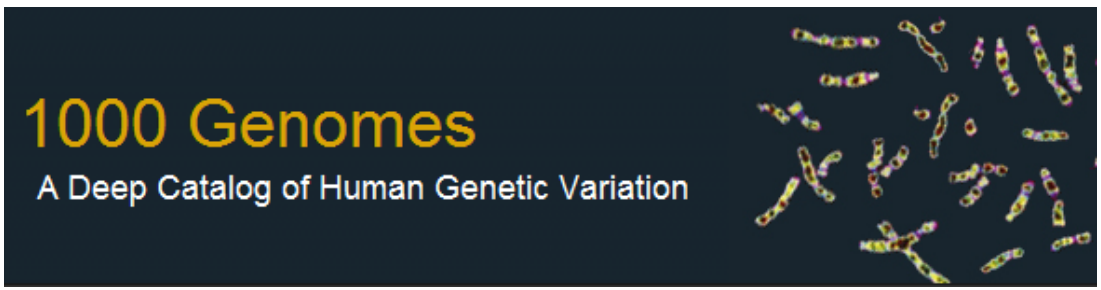
Suffix Tree: $\Theta(n \log n)$ bits

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

# Space Usage

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

- Engineered implementation: $\sim 80$ bits per symbol (bps)
- A human genome: $\sim 700MB$
- 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)

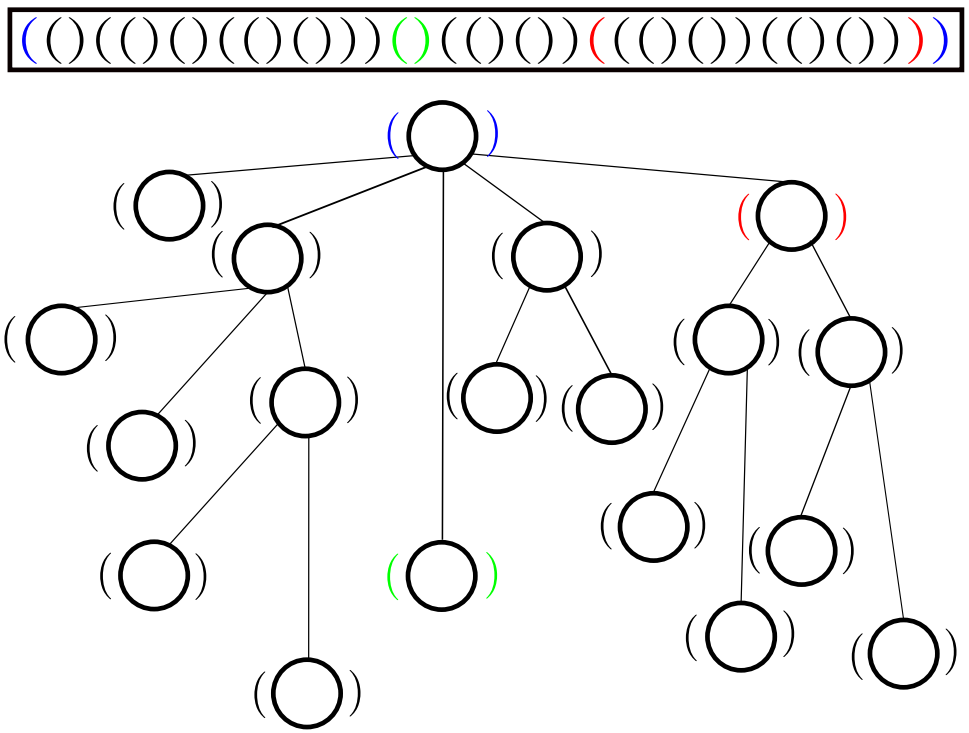
Compressed Suffix Trees are formed by *Compact Data Structures*

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

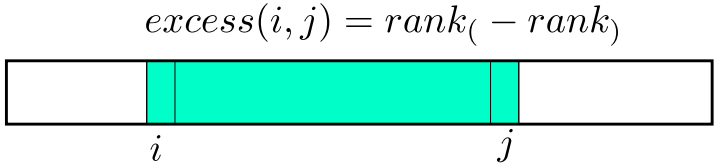
# Compressed Suffix Tree (CST)

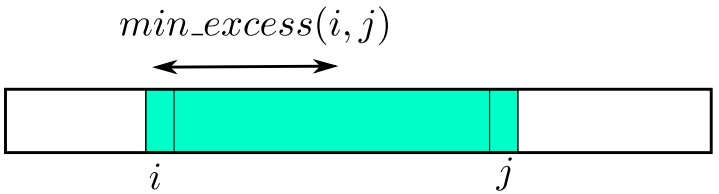
Compressed Suffix Trees are formed by *Compact Data Structures*

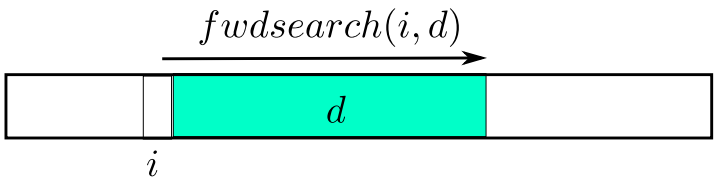
- Compressed Suffix Array (CSA)
- Compressed LCP
- Topology representation

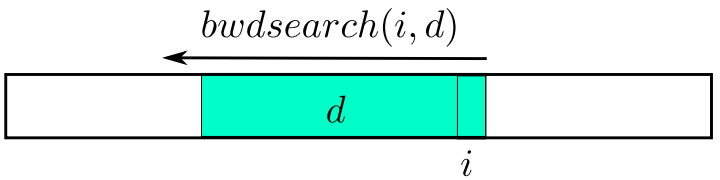


# Primitives

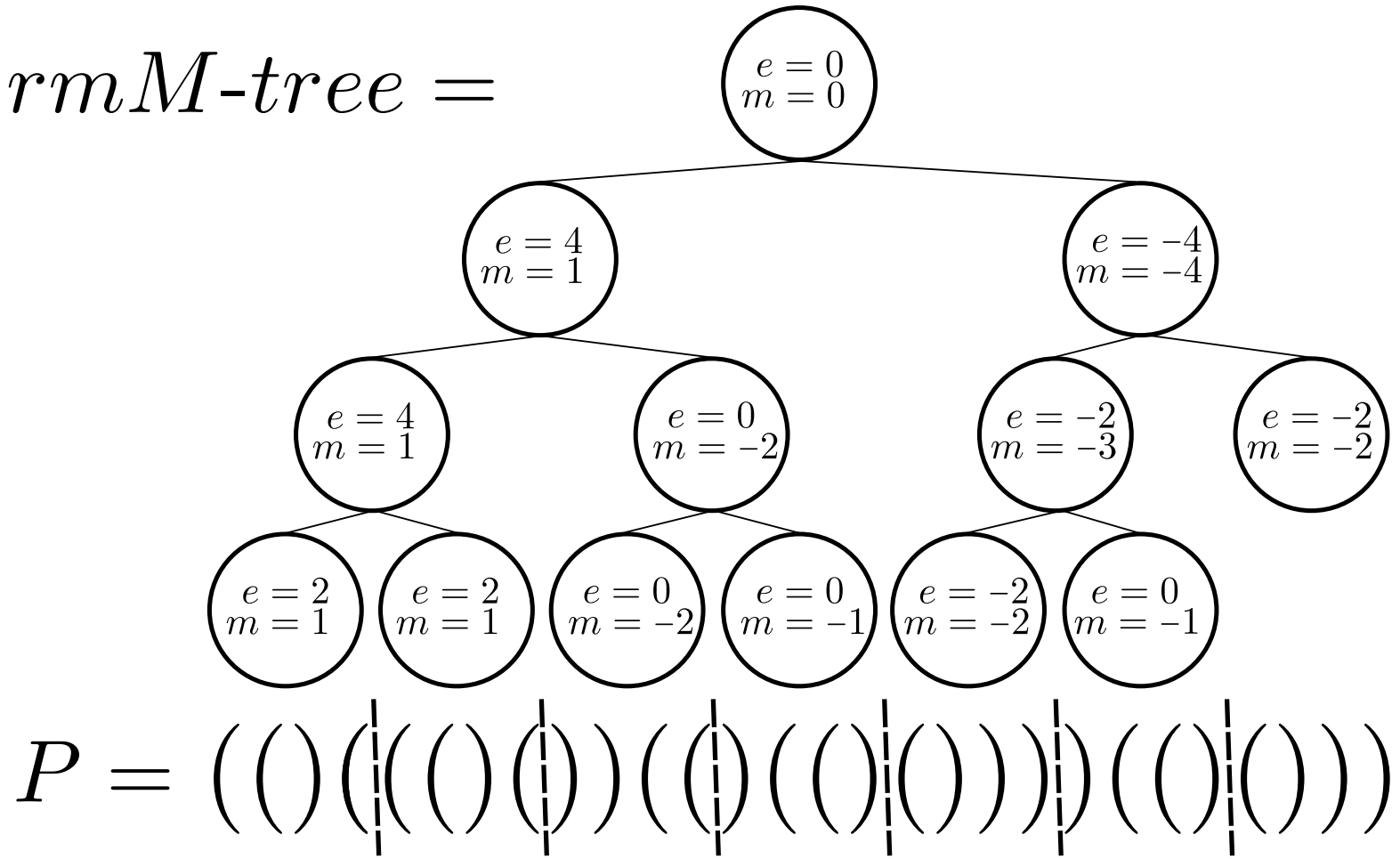

$excess(i,j) = rank_( - rank_)$
$i$
$j$


$min\_excess(i,j)$
$i$
$j$

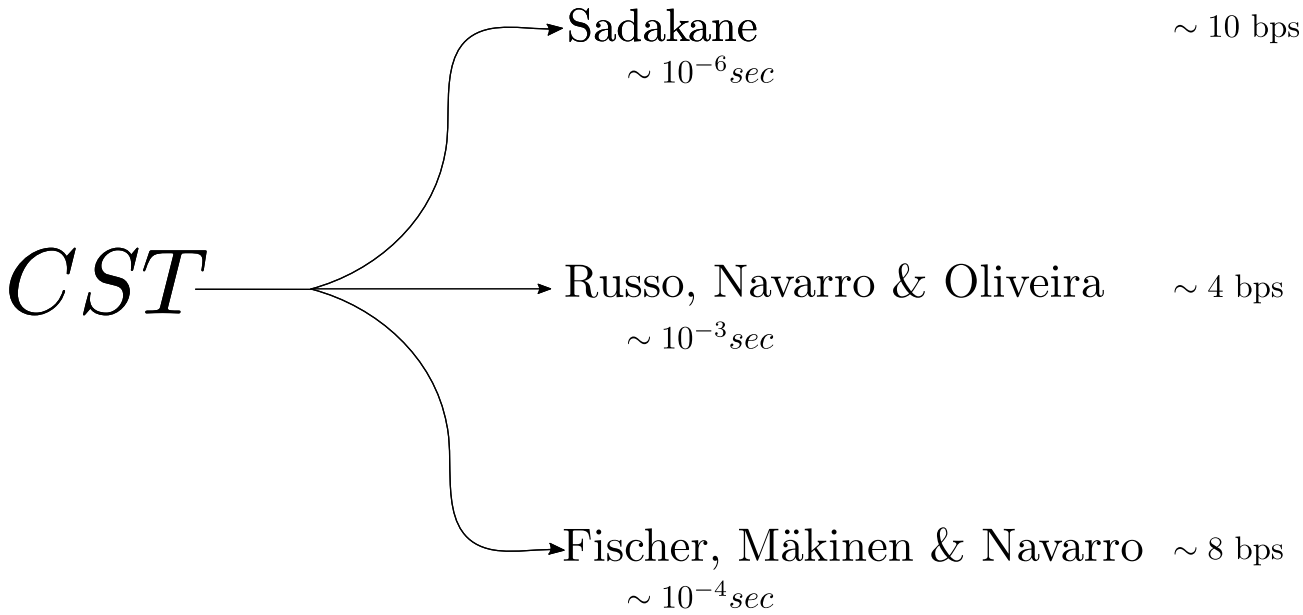

$fwdsearch(i,d)$
$d$
$i$


$bwdsearch(i,d)$
$d$
$i$

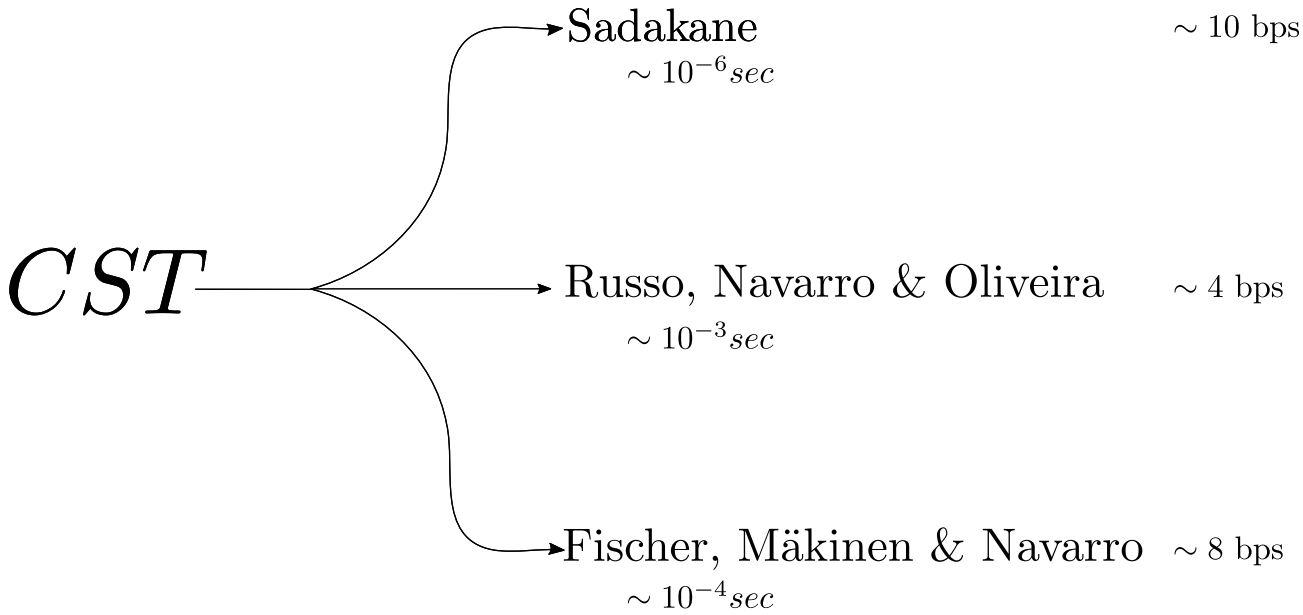
# Range min-Max Tree



# State of the Art

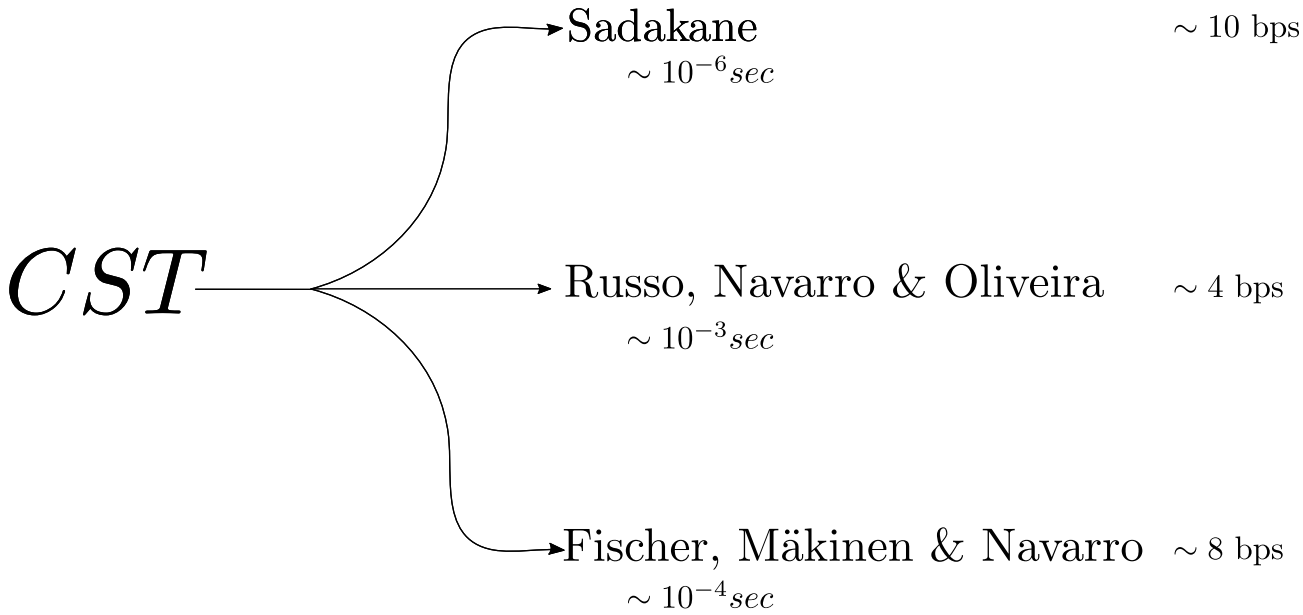


# State of the Art



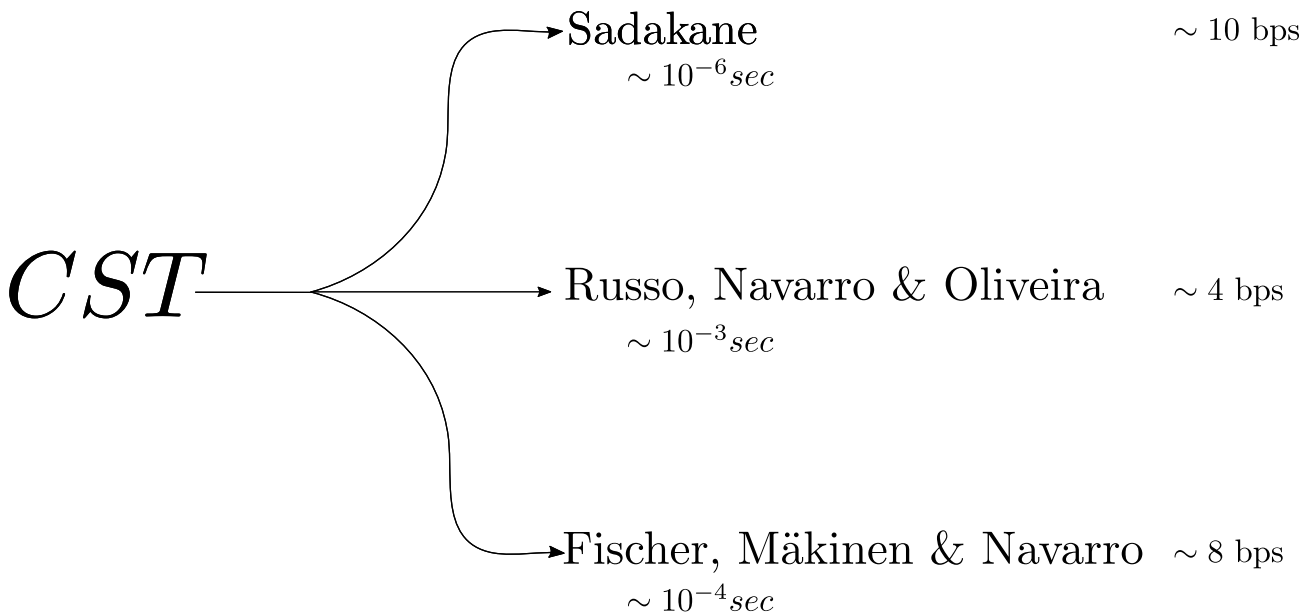

- Still a lot of space

# 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)
  - 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)
  - 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)

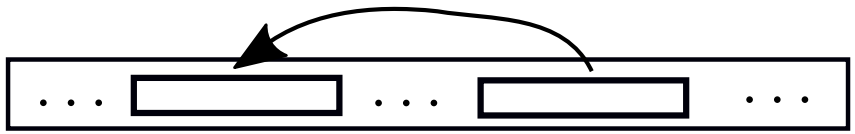
**Performance**

It uses $\sim 2$ bps and operates in $10^{-5}$ sec.

# Block Tree

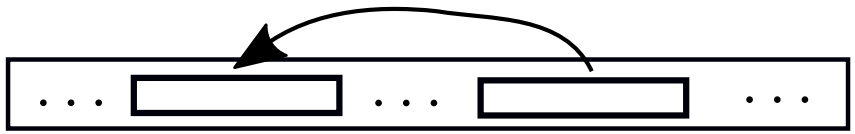
# Block Tree

- *Lempel-Ziv* bounded structure

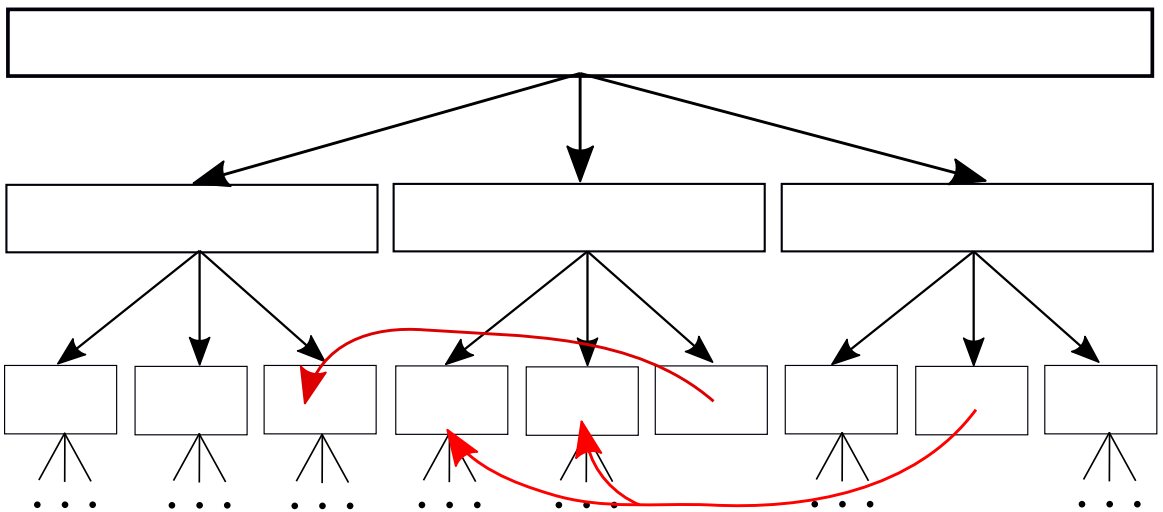
$T =$ 

# Block Tree

- *Lempel-Ziv* bounded structure

$T =$ 

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

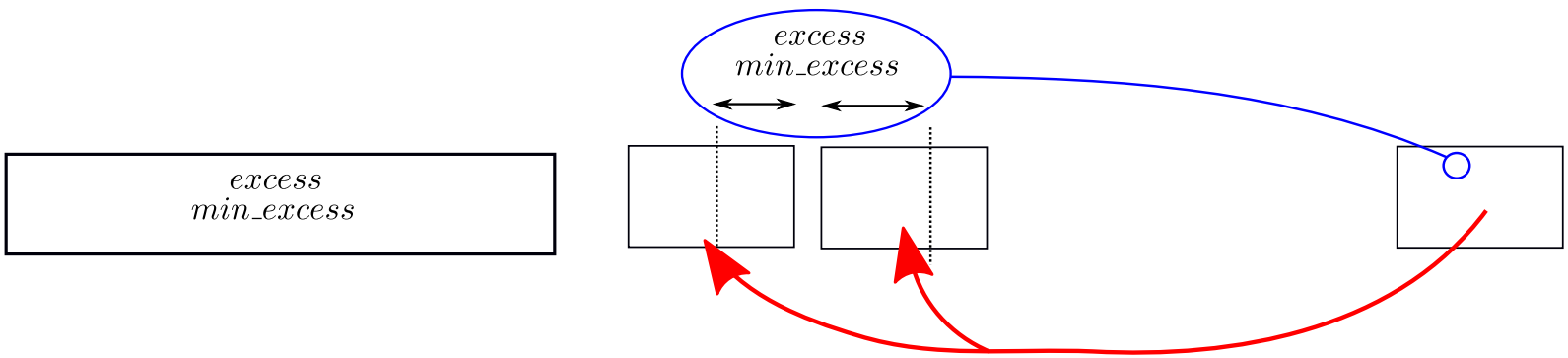
$T =$ 

# Main Idea

"Represent the balanced parentheses topology with a Block Tree and enhance its nodes with *excess* and *min_excess* information to answer the primitives efficiently "

# Block Tree Compressed Topology (BT-CT)

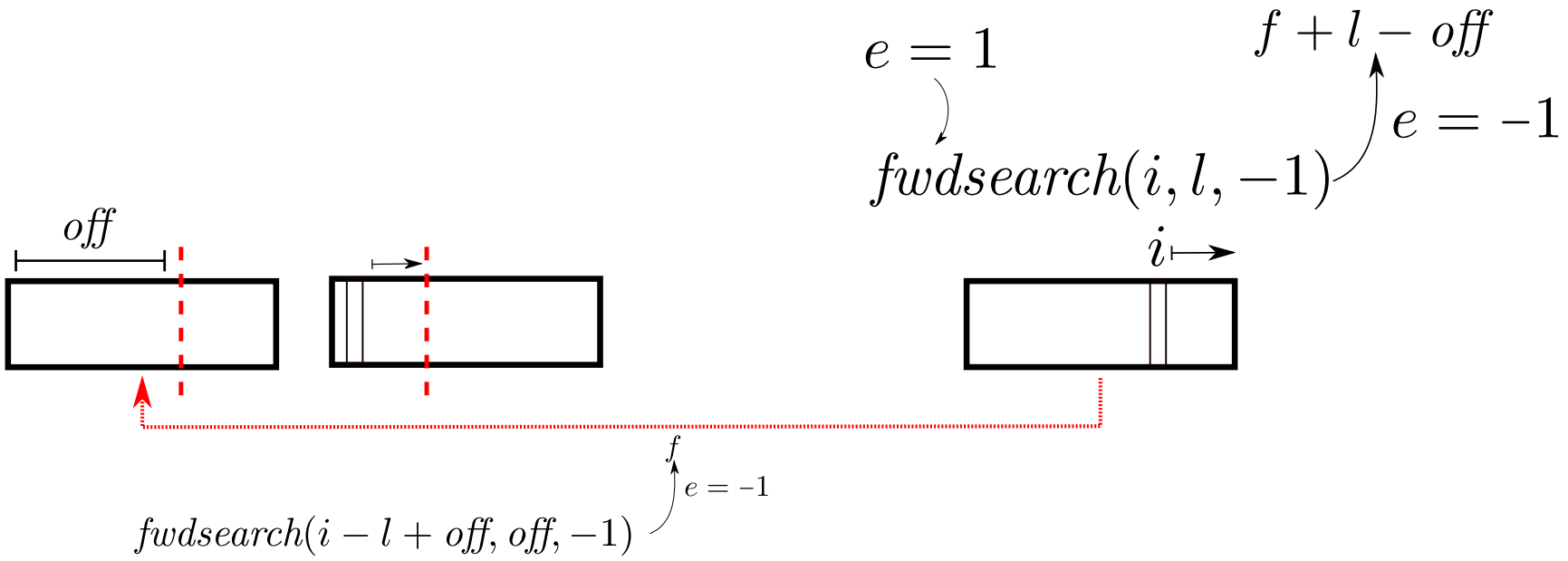
# Augmentation



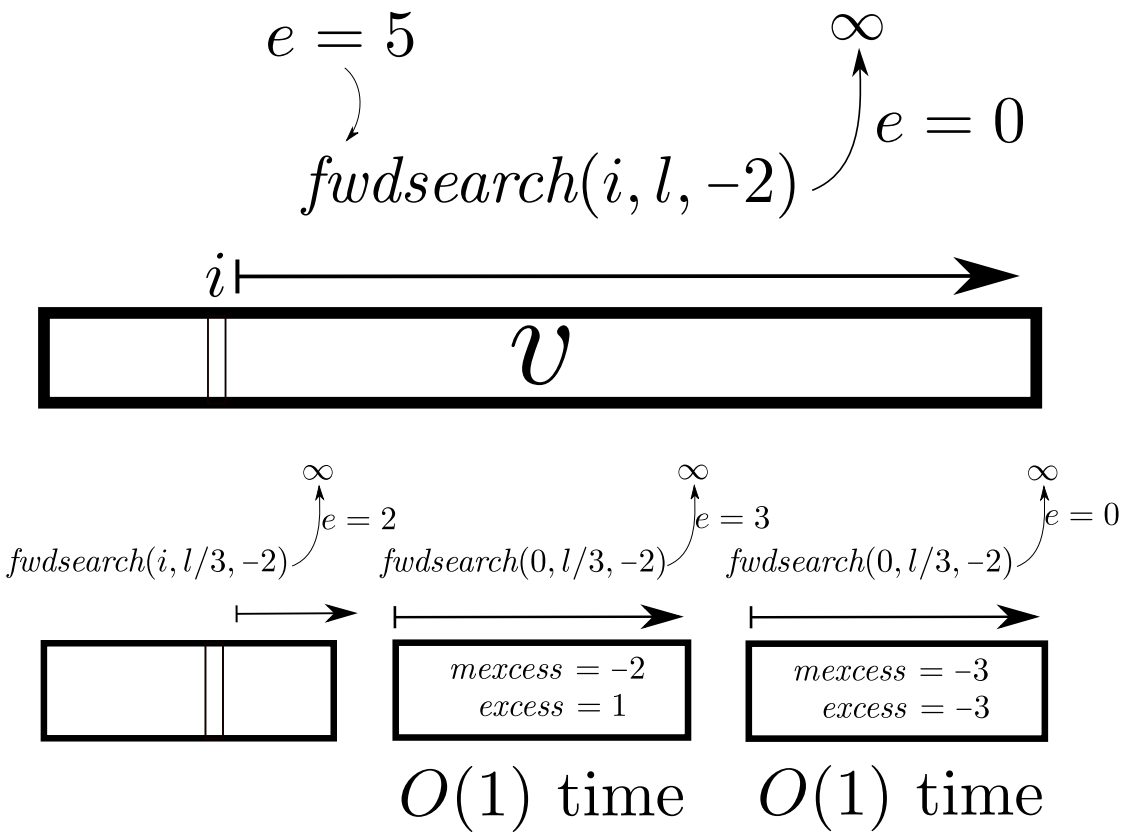
# Operations

"The basic principle to solve the operations is to compute them recursively and used the stored fields to skip recursive computation"

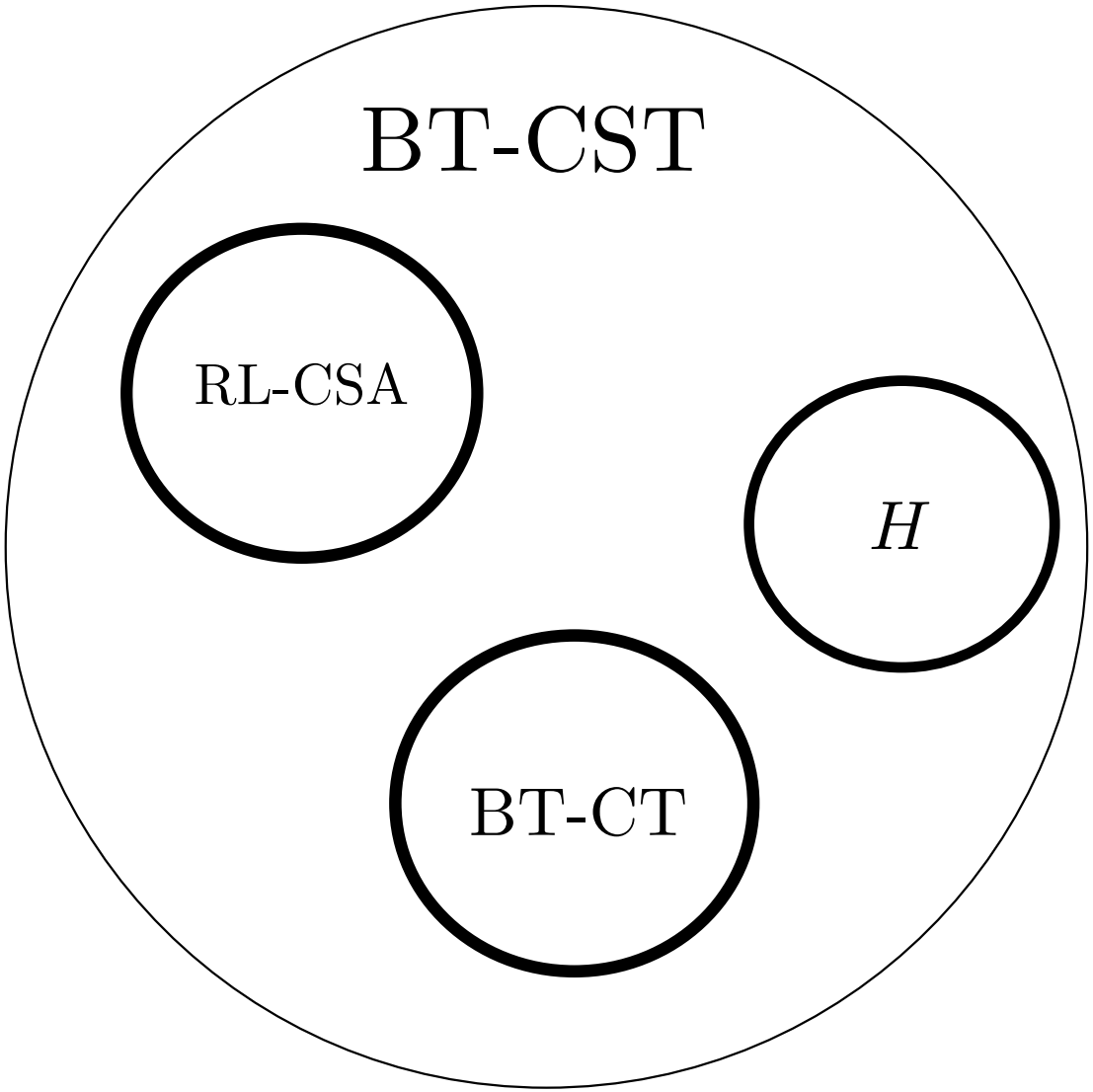
# $fwd\text{-}search(i, d \leq 0)$



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

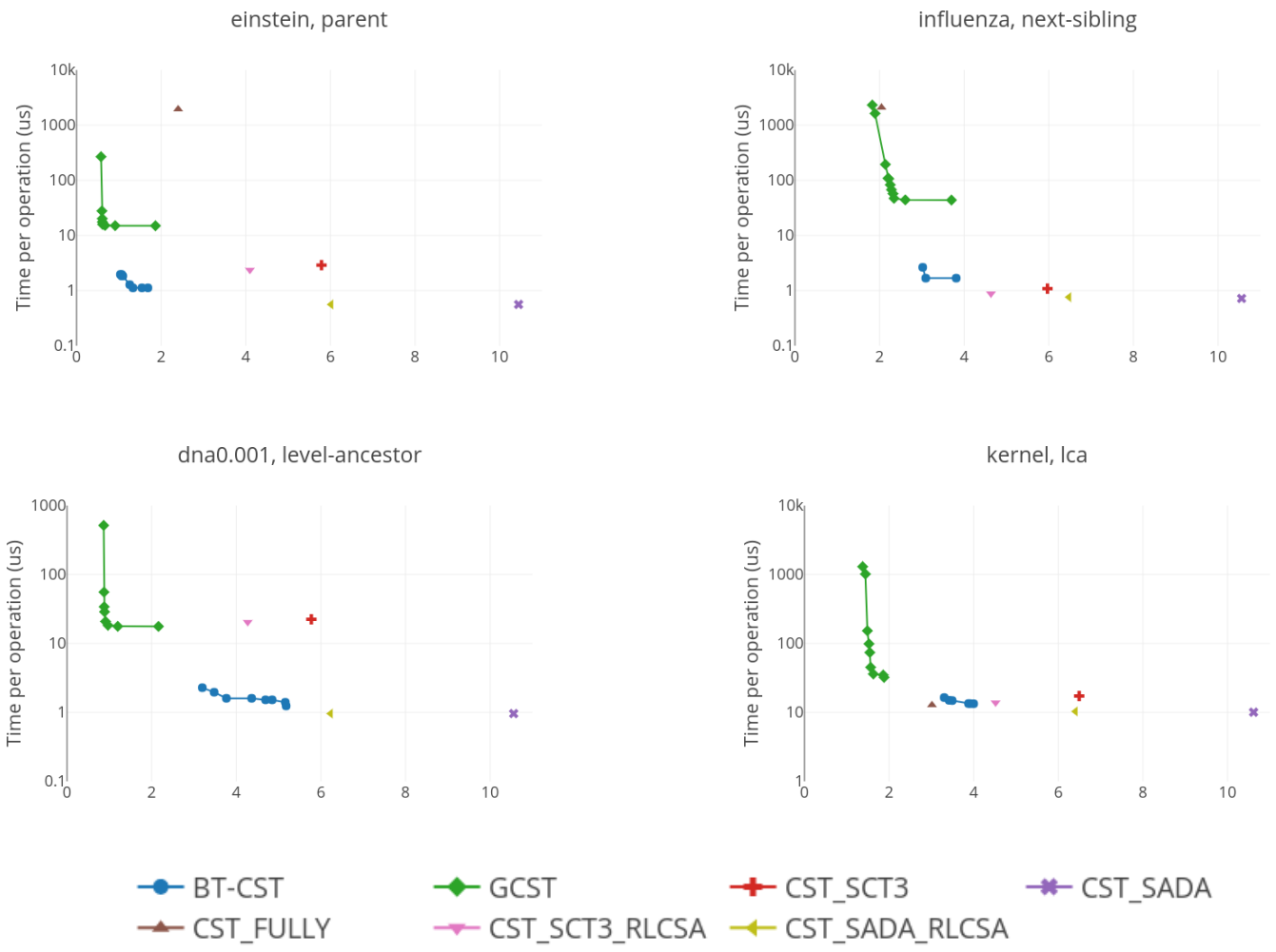


# BT-CST

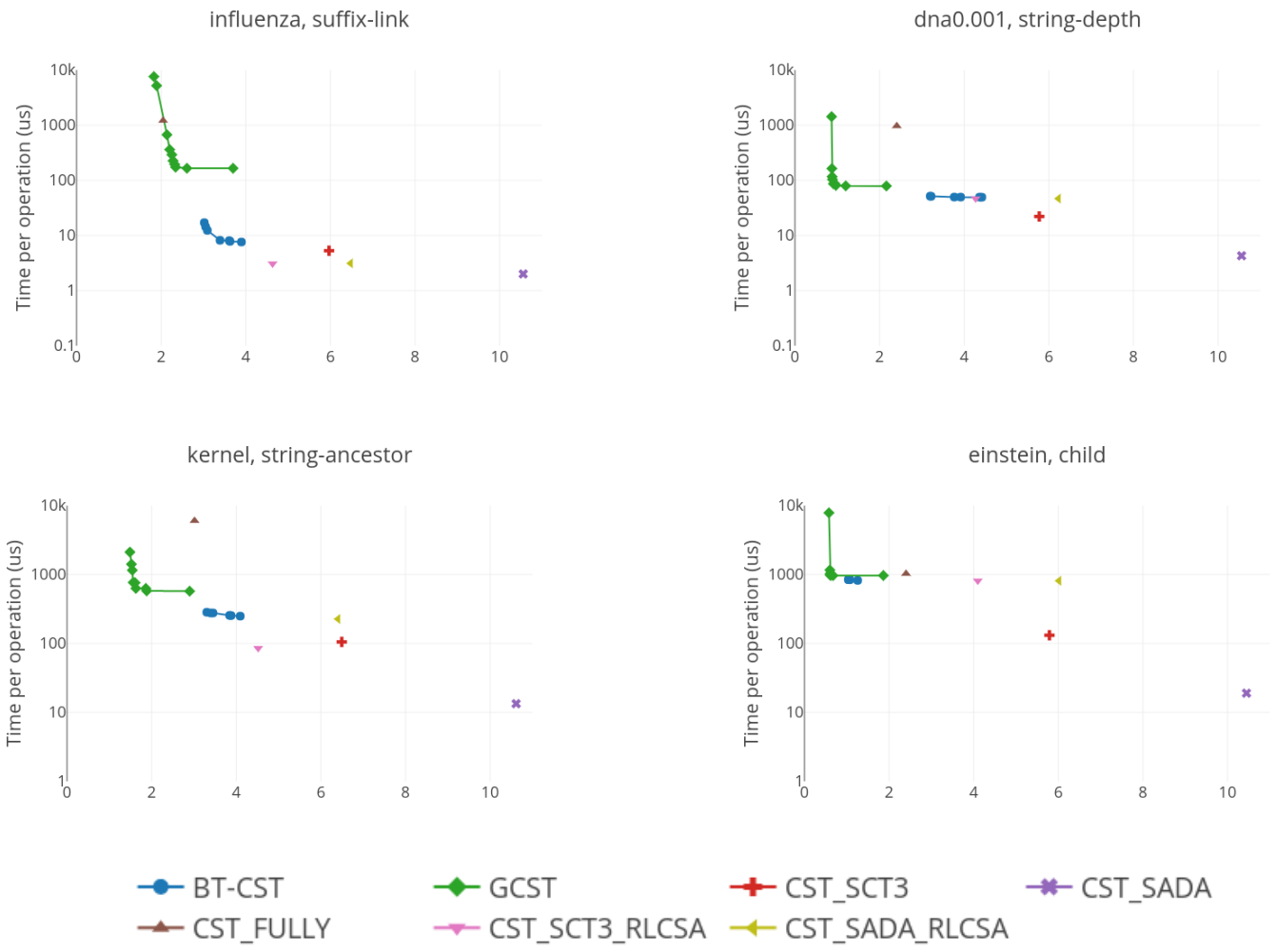


# Results

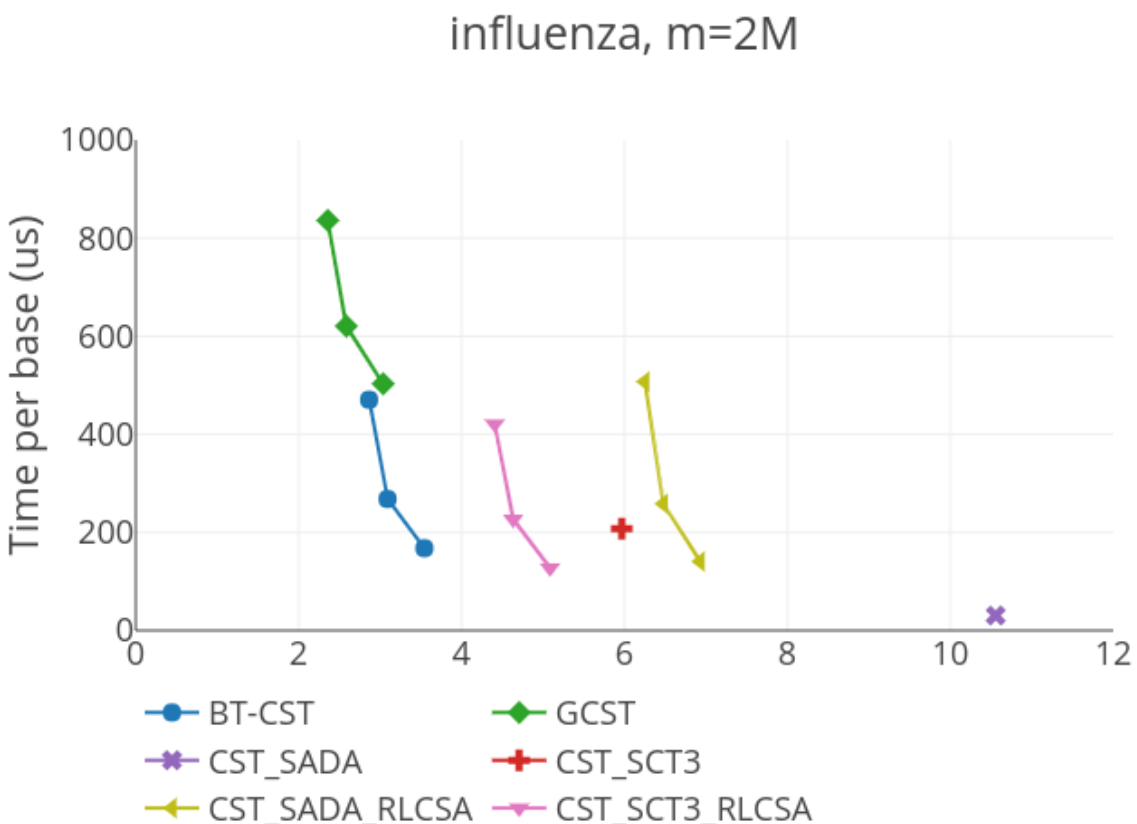
# Results – Topology Operations



# Results – CSA Based Operations



# Results – Maximal Substrings



# Conclusions/Future Work

# Conclusions/Future Work

- Fastest repetition-aware parenthesis topology

# Conclusions/Future Work

- Fastest repetition-aware parenthesis topology
- Fastest repetition-aware compressed suffix tree

# Conclusions/Future Work

- Fastest repetition-aware parenthesis topology
- Fastest repetition-aware compressed suffix tree
- Public available code for researchers and practitioners

# Conclusions/Future Work

- Fastest repetition-aware parenthesis topology
- Fastest repetition-aware compressed suffix tree
- Public available code for researchers and practitioners
- Lack of worst-case time analysis or new algorithms for primitives

# Conclusions/Future Work

- Fastest repetition-aware parenthesis topology
- Fastest repetition-aware compressed suffix tree
- Public available code for researchers and practitioners
- Lack of worst-case time analysis or new algorithms for primitives
- Time improvement on CSA based operations (WCTA'19)

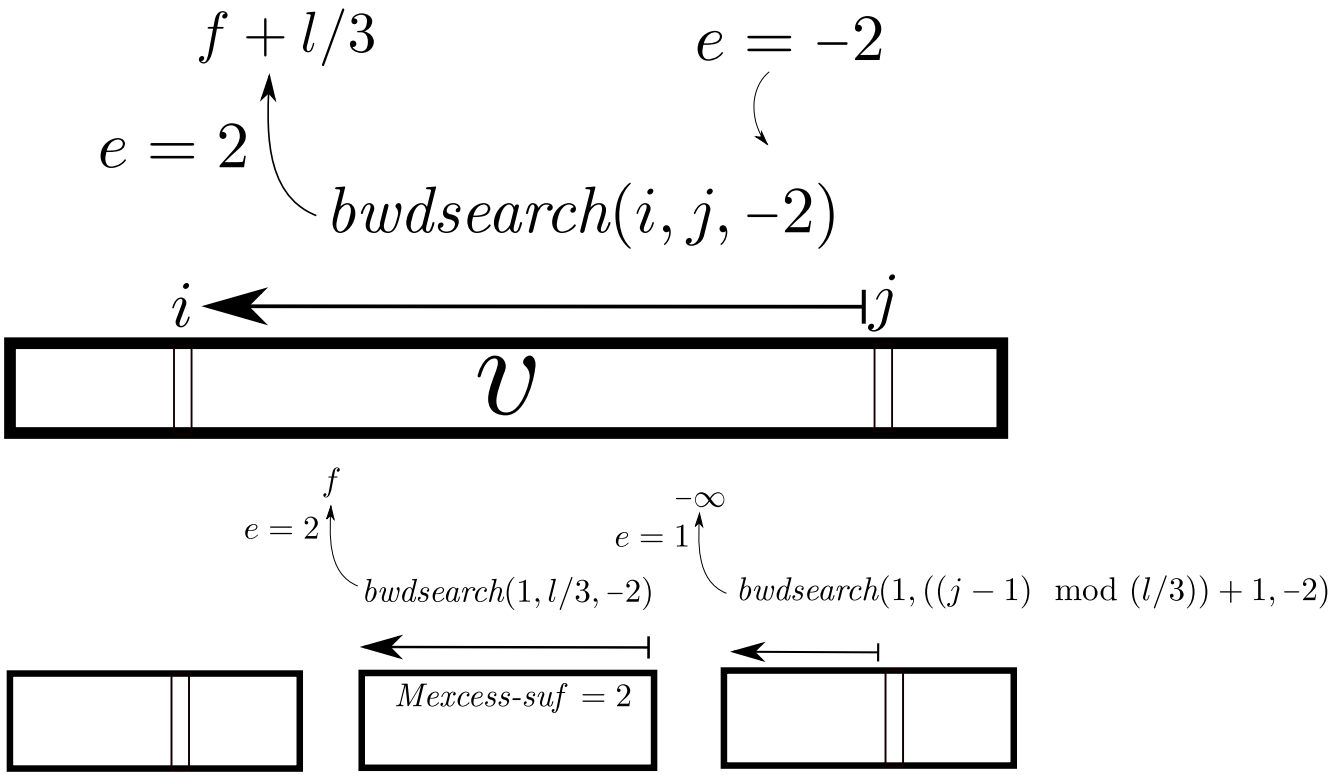
# Faster Repetition-Aware Compressed Suffix Trees based on Block Trees

Manuel Cáceres / Gonzalo Navarro

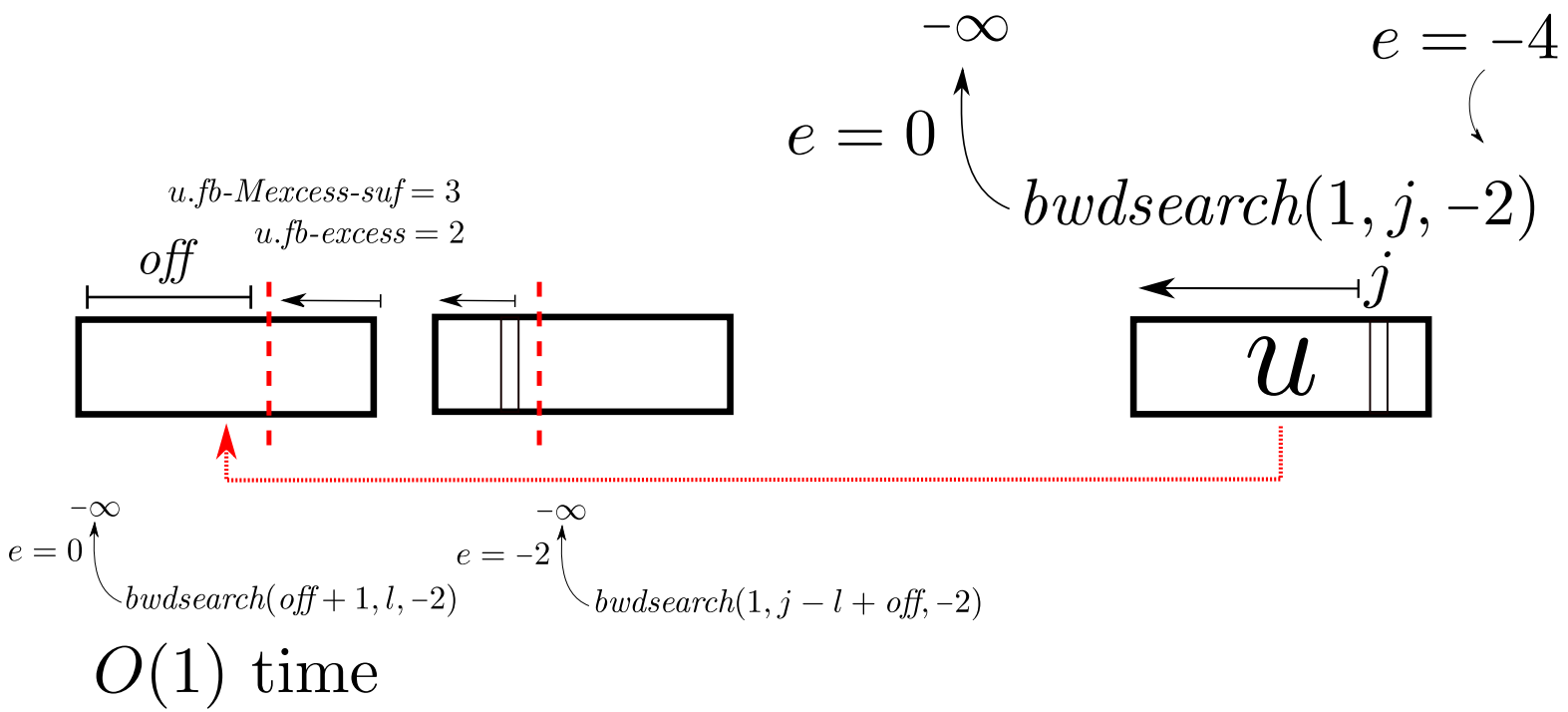
SPIRE

09/09/2019

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

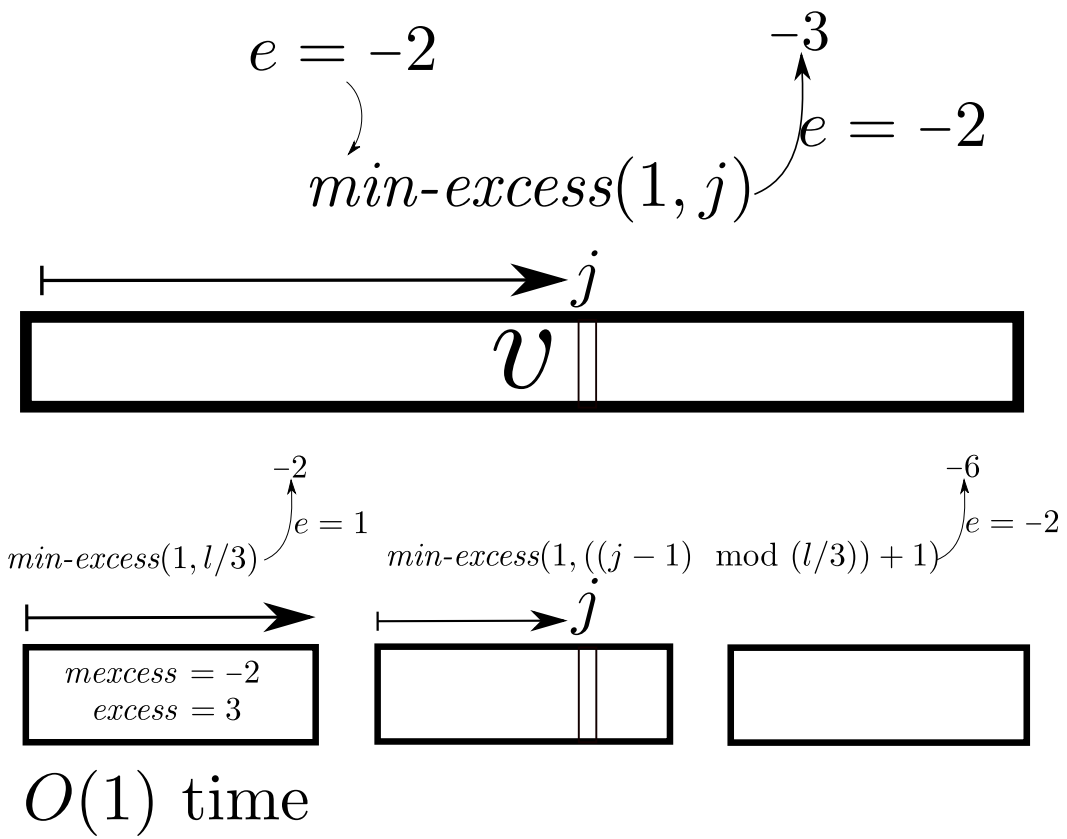


# $bwd\text{-}search(i, d \leq 0)$

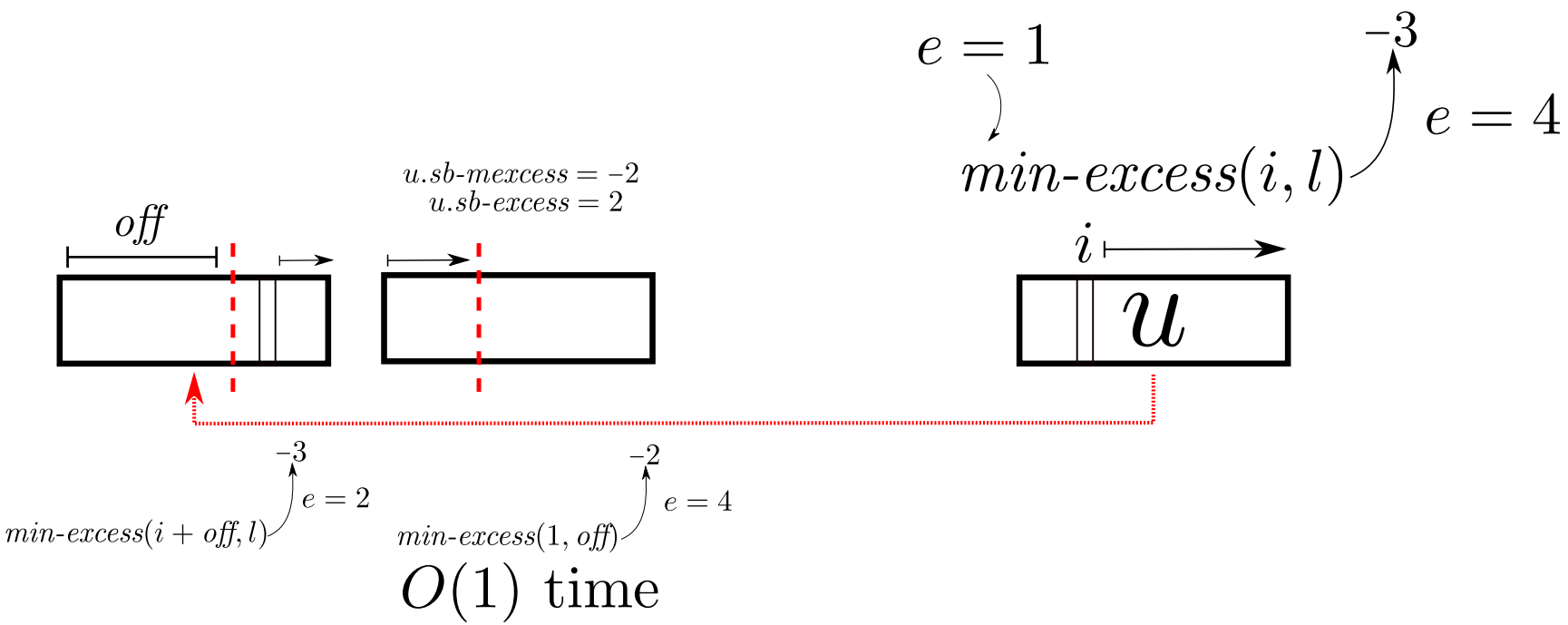



$O(1)$ time

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

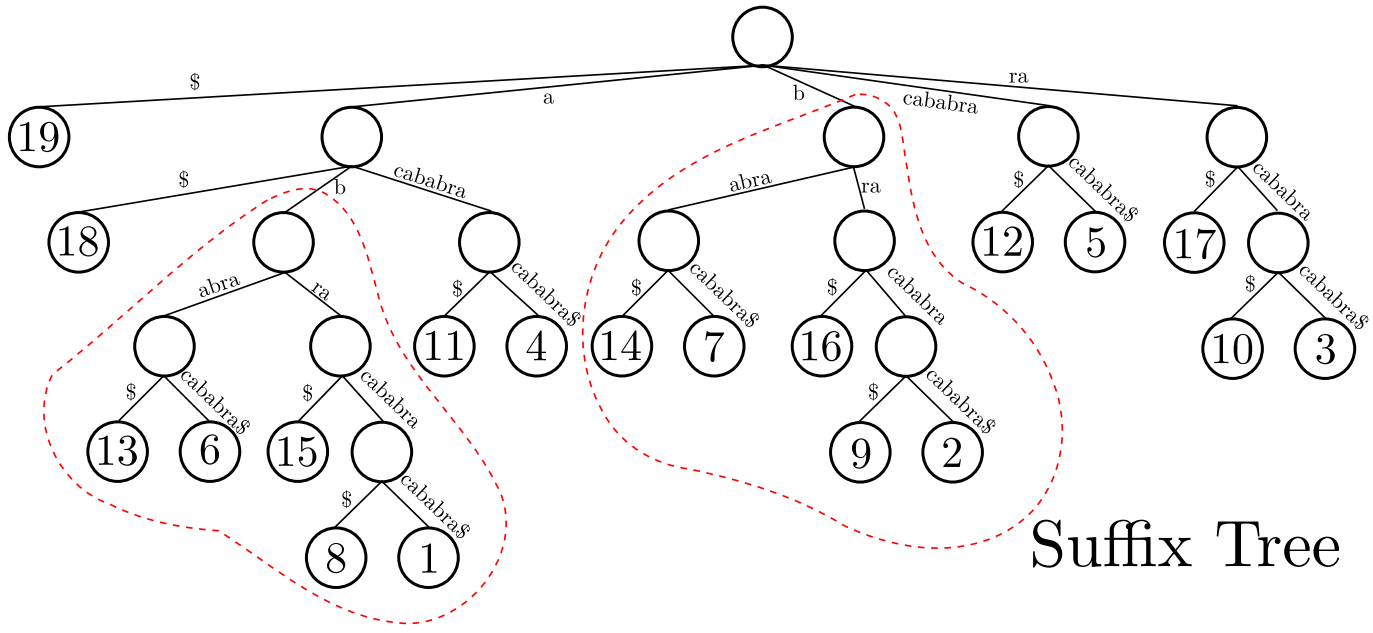


# $min\text{-}excess(i, j)$



# Repetitiveness

$T = abracadabracadabra\$$




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

# Block Tree definition

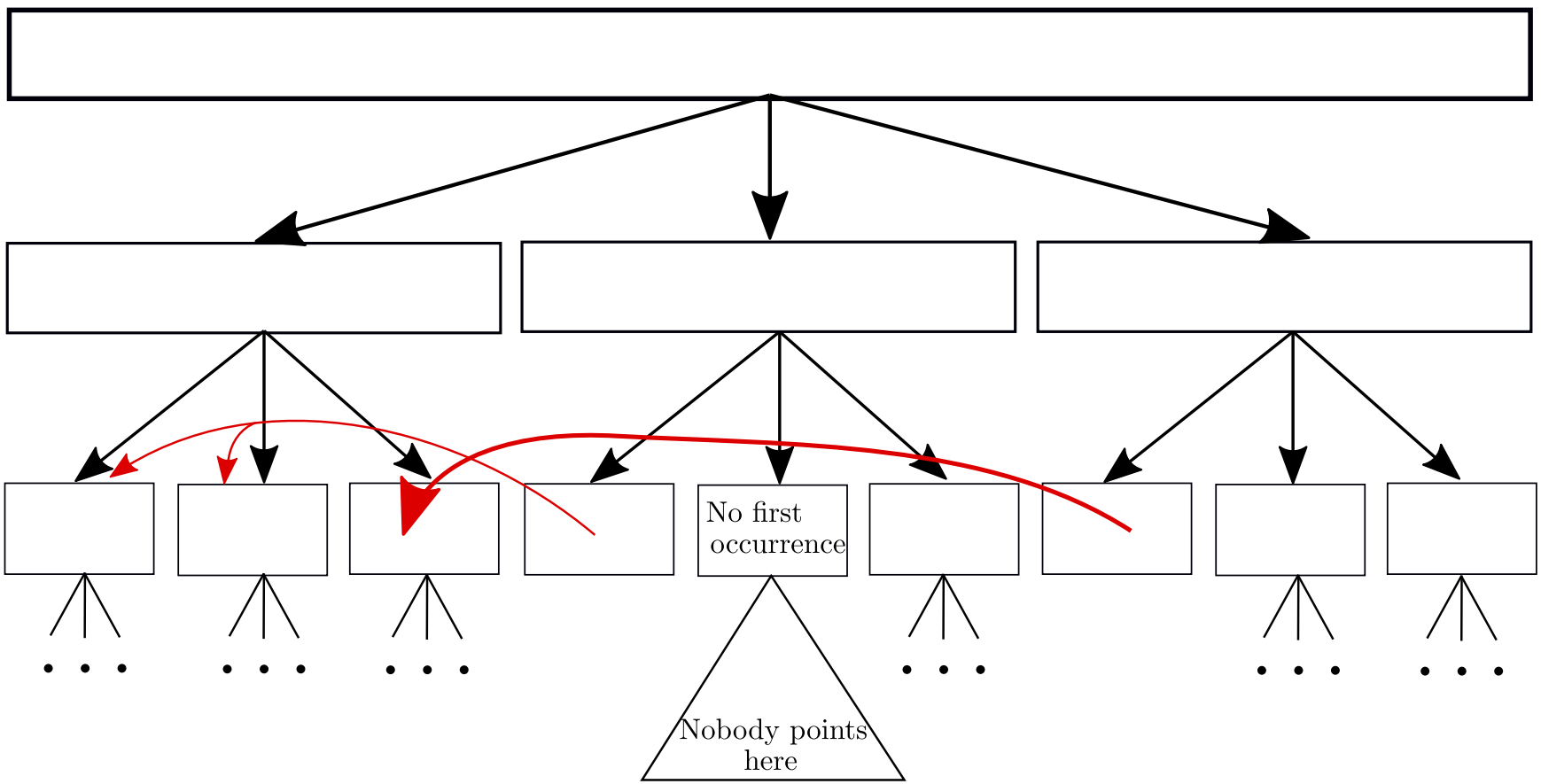
A node $v$, representing $v.blk = T[i, i + b - 1]$ can be of three types:

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

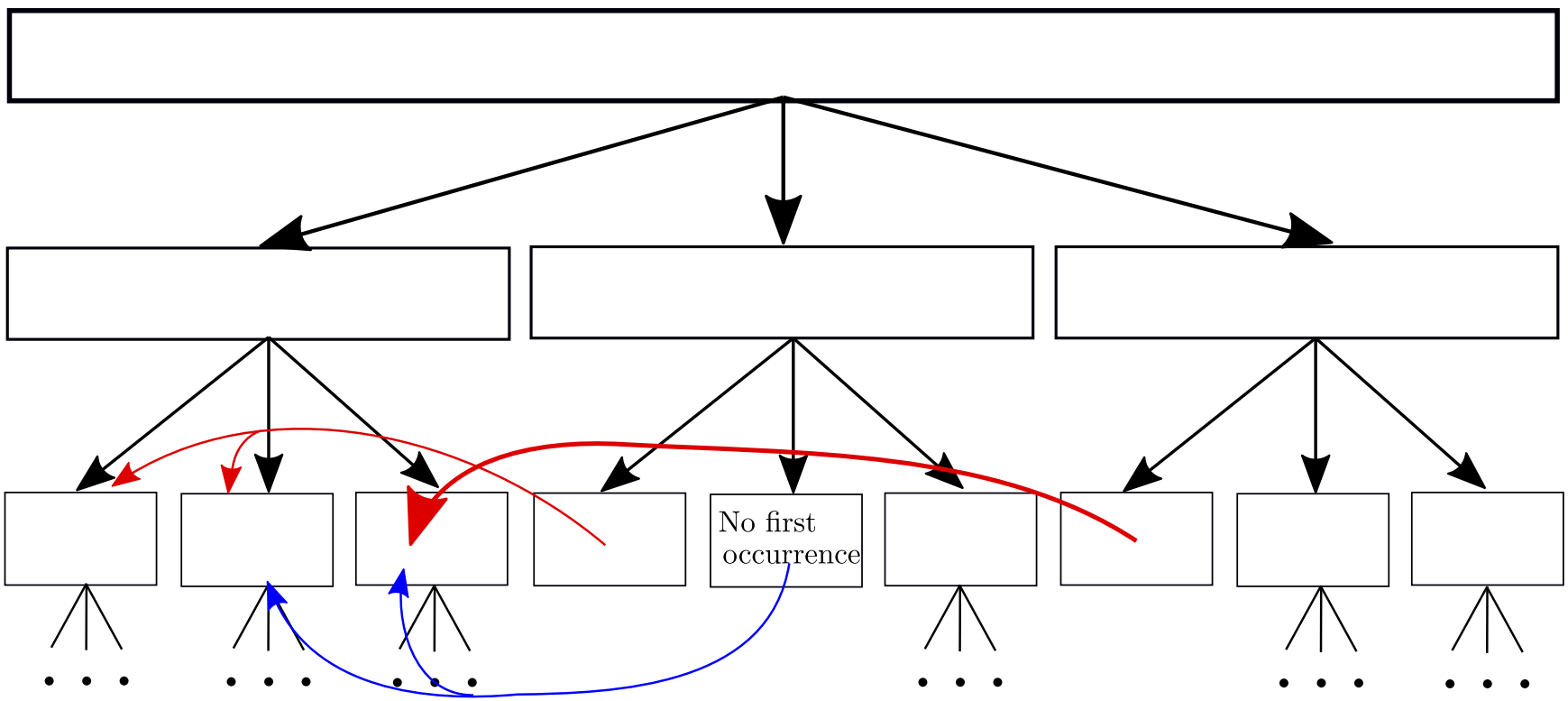
**BackBlock:** Otherwise, if $T[i - b, i + b - 1]$ and $T[i, i + 2b - 1]$ are not their leftmost occurrences in $T$, then the block is replaced by its leftmost occurrence in $T$

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

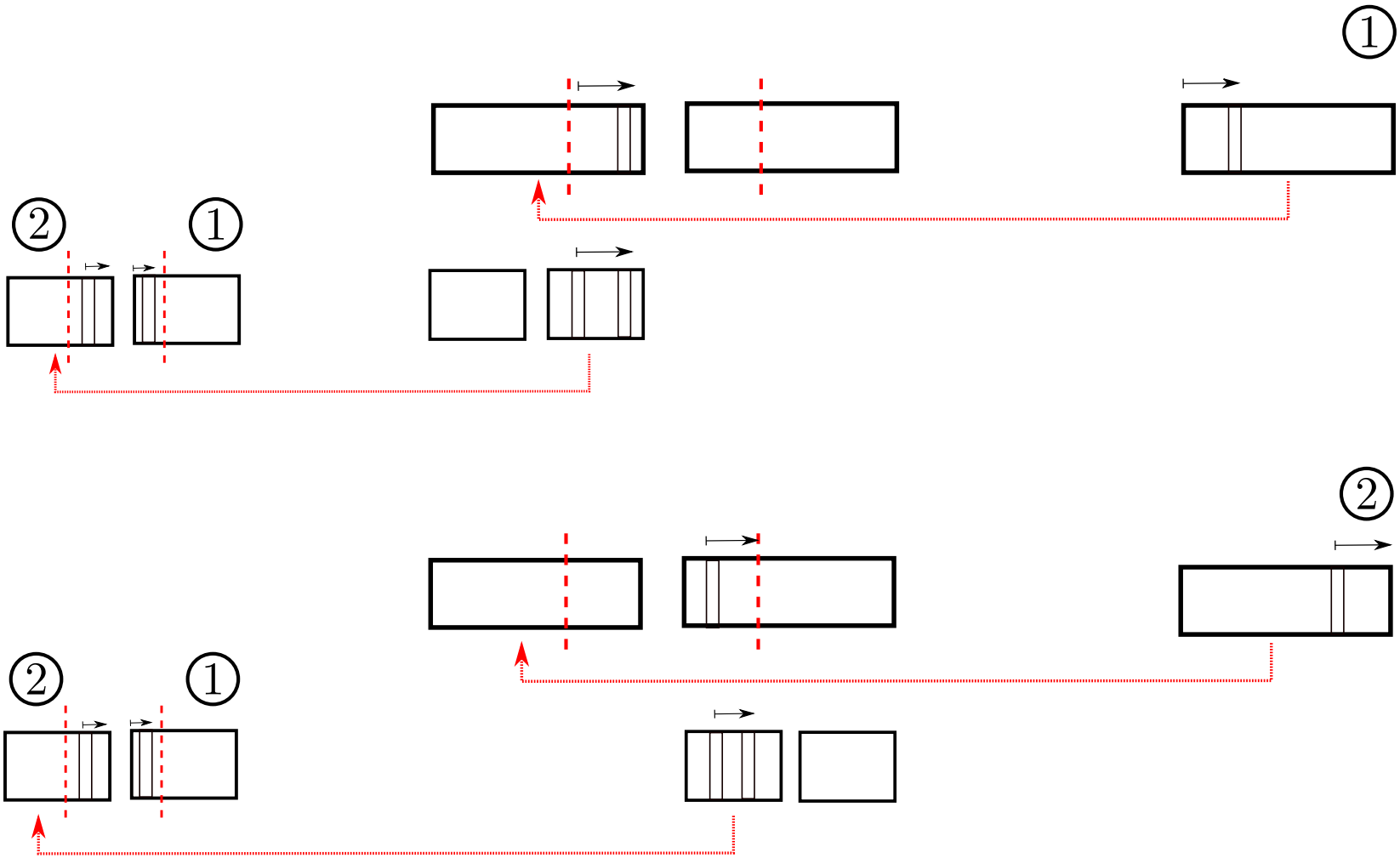
# Pruning



# Pruning



# Bad Instance



# Construction Time/Max Space