Compact Text Indexing for Advanced Pattern Matching Problems: Parameterized, Order-isomorphic, 2D, etc.

Sharma Thankachan

University of Central Florida, Orlando

CPM 2022: 33rd Annual Symposium on Combinatorial Pattern Matching
Prague, Czech Republic, June 27–29, 2022

Basic Text/String Indexing Problem

Find all occurrences of a pattern P[1,m] in a text T[1,n]

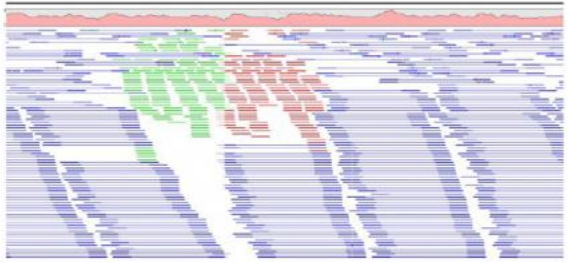
T = a c a t t b c a t t c

P = a t t

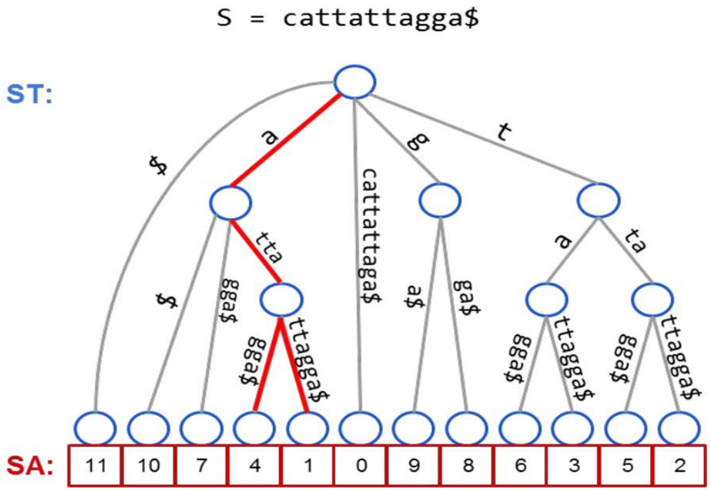
Occurrences = {3, 8}

Index T in space propotional to “n” and answer queries in time propotional to “m”

Mapping “reads” to a reference genome



String Indexing using Suffix Tree

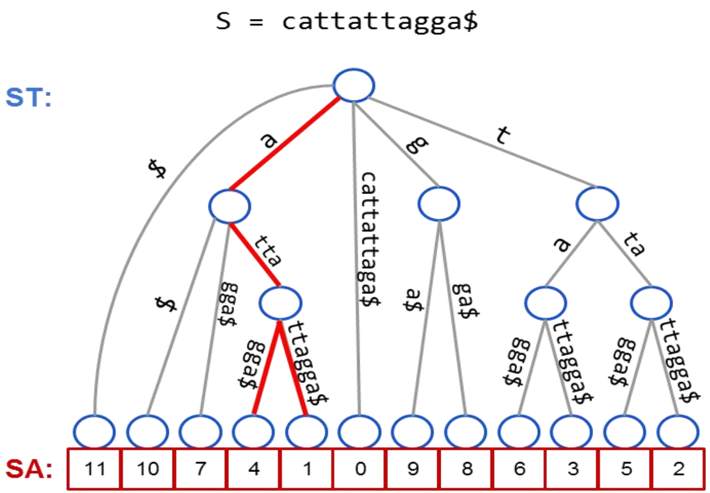


Suffix Tree = Suffix Array + Tree Structure + …

Space = O(n log n) bits

Text's space = n log |**Σ**| bits

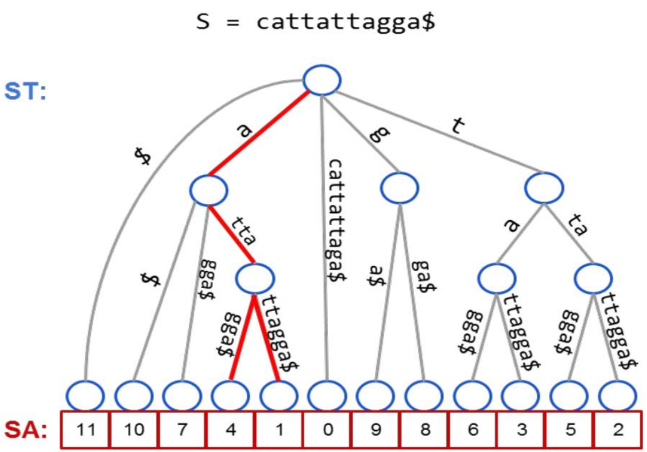
Suffix Trees & its (1000++) Applications



- longest repeat finding
- longest common substring
- Indexed pattern/substring matching
- Matching statistics
- Shortest unique substring
- Longest palindromic substring
- ....
- ....
- MEMs/MUMs
- All pairs Suffix prefix overlaps
- Clustering
- Compression Algorithms
- ....

Optimal/LINEAR time solutions for EXACT matching version

The Challenge is Space



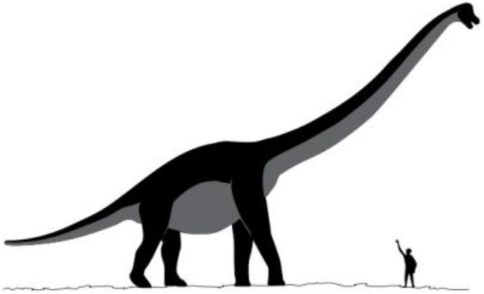
Consider indexing a human genome
(3.2 billion symbols from {A,C,G,T})

text's space ≅ 0.8 GB
Index space ≅ 30 to 40 GB

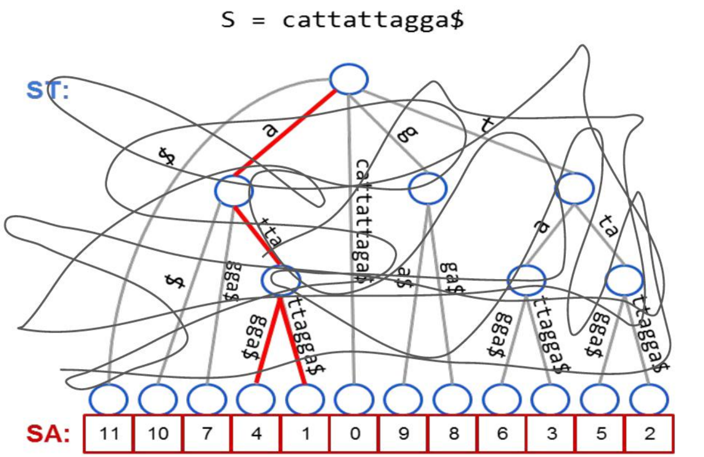
Suffix Tree = Suffix Array + Tree Structure + …

Space = O(n log n) bits

Text's space = n log |**Σ**| bits



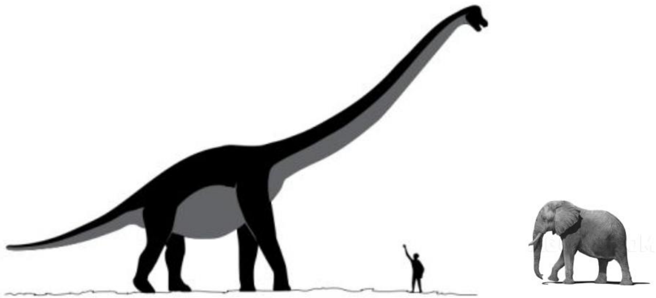
String Indexing using ~~Suffix Tree~~ Suffix Array



Consider indexing a human genome
(3.2 billion symbols from {A,C,G,T})

text's space ≅ 0.8 GB
Index space ≅ 12 GB (15 times)

Pattern matching via Binary Search in time O(P log n +occ)
P = a t t



~~Suffix Tree =~~ **Suffix Array** ~~+ Tree Structure~~

Space = ~~O(n log n)~~ n log n bits +text

Text's space = n log |**Σ**| bits

A Big Open Problem in Late 90's

Indexing in Space close to text's space?

A Big Open Problem in Late 90's

Indexing in Space close to text's space?

The answer is “YES”

We can simultaniously encode the text and the suffix array/tree in text's space

The Compressed Suffix Array [Grossi and Vitter, STOC 2000]
FM-index [Ferragina and Manzini, FOCS 2000]
LCP Array Compression (Compressed ST) [Sadakane, SODA 2002]
....
.....
....
r-index and Suffix Trees in Space propotional to BWT-runs
[Gagie, Prezza, Navarro, SODA 2018]
.....

(Compressed)
Text Indexing
for
Problems Beyond Exact Matching
?

Problems Beyond Exact Matching

- Approximate PM (under mismatches/edits/gaps, etc)
- Jumbled PM
- Parameterized PM
- Order-isomorphic PM
- 2D (multi-dimensional) PM
- Cartesian Tree Matching
- Structural PM
- Circular PM
- Episode Matching
- Functional PM
- Blocked PM
- Permuterm PM
- ....
- ....

Suffix Trees with Missing Suffix Links

- Parameterized pattern matching
- Order-isomorphic pattern matching
- 2D (multi-dimensional) pattern matching

Suffix Tree Like Solutions exist

BUT

Compressed Space Solutions?

Indexing for Parameterized Matching (P-matching)

Text = a b b a c a t b b a b c a

Pattern = x y x

Two strings are p-match IFF there exists a one-to-one function that renames the characters in one to another

E.g., x → a, y → c in x y z gives a c a

Indexing for Parameterized Matching (P-matching)

Text = a b b a c a t b b a b c a

Pattern = x y x

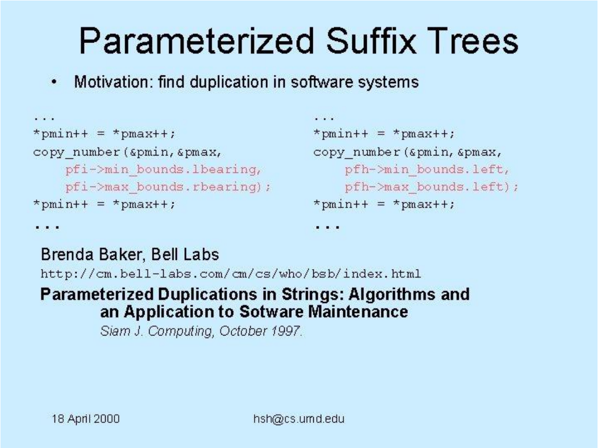
Two strings are p-match IFF there exists a one-to-one function that renames the characters in one to another

E.g., x → a, y → c in x y z gives a c a

Main idea: Prev encoding !!!
Prev(ababb) = 00221

Two strings are p-match IFF they have the same Prev encoding

Prev(x y x) = 0 0 1 = Prev(a c a)



Parameterized Suffix Trees

- Motivation: find duplication in software systems

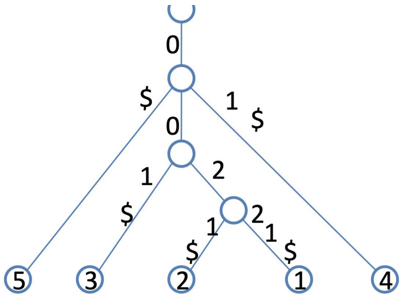
```
...                                   ...
*pmin++ = *pmax++;                    *pmin++ = *pmax++;
copy_number(&pmin,&pmax,              copy_number(&pmin,&pmax,
    pfi->min_bounds.lbearing,             pfh->min_bounds.left,
    pfi->max_bounds.rbearing);            pfh->max_bounds.left);
*pmin++ = *pmax++;                    *pmin++ = *pmax++;
...                                   ...
```

Brenda Baker, Bell Labs
http://cm.bell-labs.com/cm/cs/who/bsb/index.html
Parameterized Duplications in Strings: Algorithms and an Application to Sotware Maintenance
Siam J. Computing, October 1997.

18 April 2000 hsh@cs.umd.edu

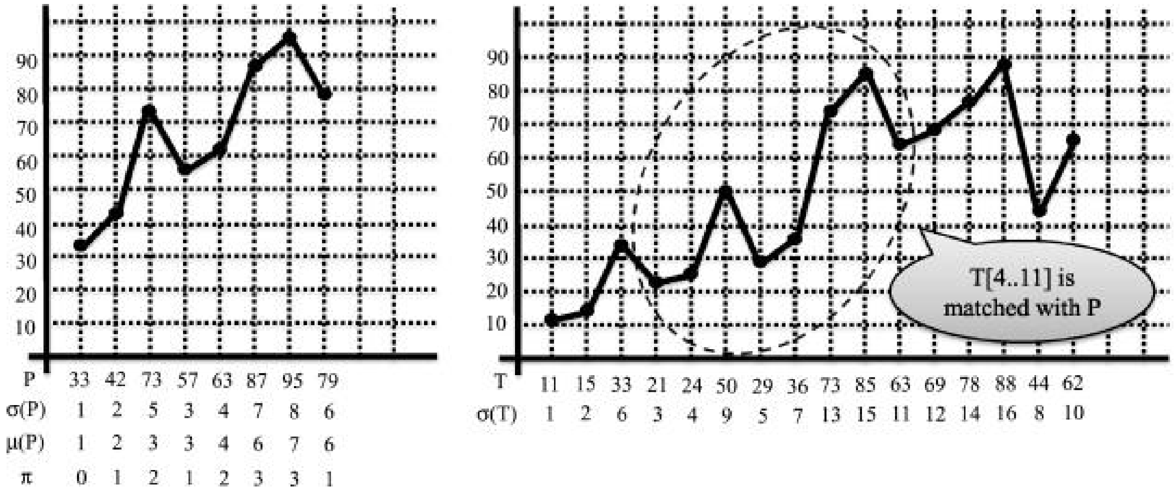
1. ababb$ → 00221$
2. babb$ → 0021$
3. abb$ → 001$
4. bb$ → 01$
5. b$ → 0$

pv() of each suffix



Compressed Index via **Parameterized BWT** --- Efficient and Elegant [SODA 17, ICALP 2022]

(Compact) **Indexing for Order-isomorphic Matching**
(notoriously difficult)



Picture Courtesy: Kim et al. [TCS 2014]

Order Isomorphic Suffix Trees [Crochemore, SPIRE 2013]

Compressed version [LF-successor, ICALP 2021] - Messy and Somewhat Slow :(

Indexing for 2D matching (2D Suffix Array)

- There are m+n-1 diagonals in m x n array
- For each diagonal form a square array
- For each square array, decomposing in a "┘"shapes,
- Each "┘" is mapped to a number (Giancarlo[7]), and a square is a string num(s), forming quasi-suffix collection (each with different ending symbol)
- since m+n-1 diagonals, m+n-1 square for a multiple quasi-suffix collection

COMPACT indexing?

Compressed **Inverse** Suffix Array [Patel and Shah, ISAAC 2021]

Next:

Parameterized BWT and
Parameterized Suffix Array Compression

Lets start with a quick review of BWT, LF mapping, etc

Encoding the Suffix Array

LF mapping:
j = LF[i] iff SA[j] = SA[i] -1

Text = m i s s i s s i p p i $

LF[9]=11

LF[3] = 9

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

Encoding the Suffix Array

LF mapping:
j = LF[i] iff SA[j] = SA[i] -1

Text = m i s s i s s i p p i $

LF[9]=11

LF[3] = 9

	1	2	3	4	5	6	7	8	9	10	11	12
Suffix Array	**12**	**11**	**8**	**5**	**2**	**1**	**10**	**9**	**7**	**4**	**6**	**3**
Sampled SA (sampling factor D = 3)	**12**							**9**			**6**	**3**

Computing SA[3]

- LF[3] = 9
- LF[9] = 11
- SA[11] = 6 is stored
- SA[3] = 6+2 = 8

Suffix Array = LF mapping + sampled suffix array

how to encode?

(n/D) log n bits

Encoding the Suffix Array

Text = m i s s i s s i p p i $

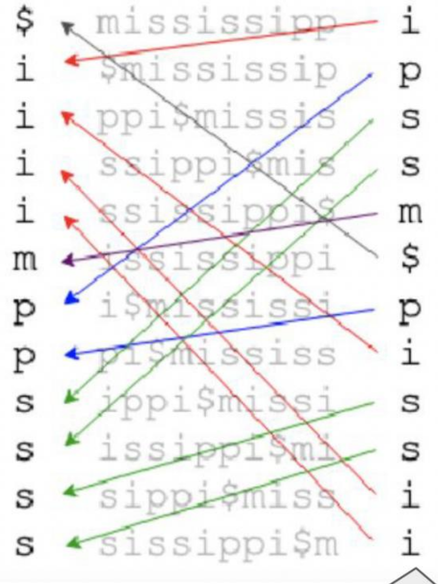
12	11	8	5	2	1	10	9	7	4	6	3

BWT = i p s s m $ p i s s i i

1	mississippi$
2	ississippi$m
3	ssissippi$mi
4	sissippi$mis
5	issippi$miss
6	ssippi$missi
7	sippi$missis
8	ippi$mississ
9	ppi$mississi
10	pi$mississip
11	i$mississipp
12	$mississippi

sort ⇒

12	$mississippi
11	i$mississipp
8	ippi$mississ
5	issippi$miss
2	ississippi$m
1	mississippi$
10	pi$mississip
9	ppi$mississi
7	sippi$missis
4	sissippi$mis
6	ssippi$missi
3	ssissippi$mi



Burrows Wheeler Transform (BWT)

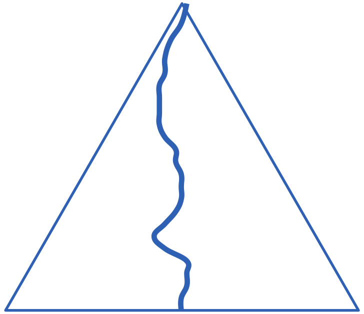
If BWT[i] = c & # of c in BWT[1,i] = k
then LF[i] = kth position in c's range

Another way of looking at LF Mapping and BWT

- Let i and j, where i < j are two leaves in the suffix tree.

  We say (i, j) is an **inversion** if L[i] > L[j] and **non-inversion** otherwise.

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



n1 = # of non-inversions on left of i

i

n2 = # of inversions on the right of i

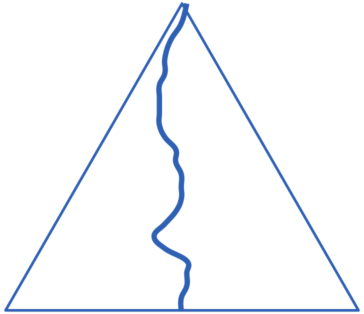
LF[i] = n1+n2+1

LF can be implemented via **inversion counting**

Another way of looking at LF Mapping and BWT

- Let i and j, where i < j are two leaves in the suffix tree.
  We say (i, j) is an **inversion** if L[i] > L[j] and **non-inversion** otherwise.

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



n1 = # of non-inversions on left of i

i

n2 = # of inversions on the right of i

LF[i] = n1+n2+1

- Can we store some (succinct) information with each leaf, so that given an (i, j) pair, we can quickly decide if it is an inversion.

- This is exactly BWT, inversion IFF BWT[i] > BWT[j]

LF can be implemented via **inversion counting**

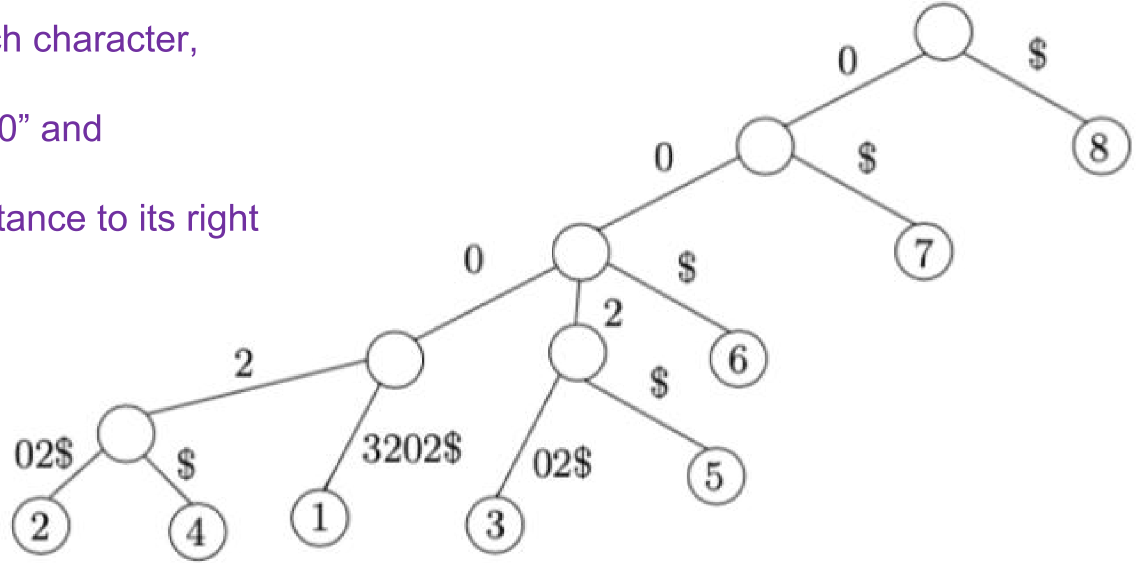
Lets move on to **parameterized suffix array**

Parameterized Suffix Tree/Array

Prev Encoding of a string S: for each character,

- replace the first occurrence by “0” and
- any other occurrence by the distance to its right most previous occurrence

Example: Prev(ababb) = 00221



i	T_i	prev(T_i)	prev($T_{\mathsf{PSA}[i]}$)	$T_{\mathsf{PSA}[i]}$	PSA[i]
1	xyzxzwz$	0003202$	000202$5	yzxzwz$x	2
2	yzxzwz$x	000202$5	0002$504	xzwz$xyz	4
3	zxzwz$xy	00202$50	0003202$	xyzxzwz$	1
4	xzwz$xyz	0002$504	00202$50	zxzwz$xy	3
5	zwz$xyzx	002$0043	002$0043	zwz$xyzx	5
6	wz$xyzxz	00$00432	00$00432	wz$xyzxz	6
7	z$xyzxzw	0$004320	0$004320	z$xyzxzw	7
8	$xyzxzwz	$0003202	$0003202	$xyzxzwz	8

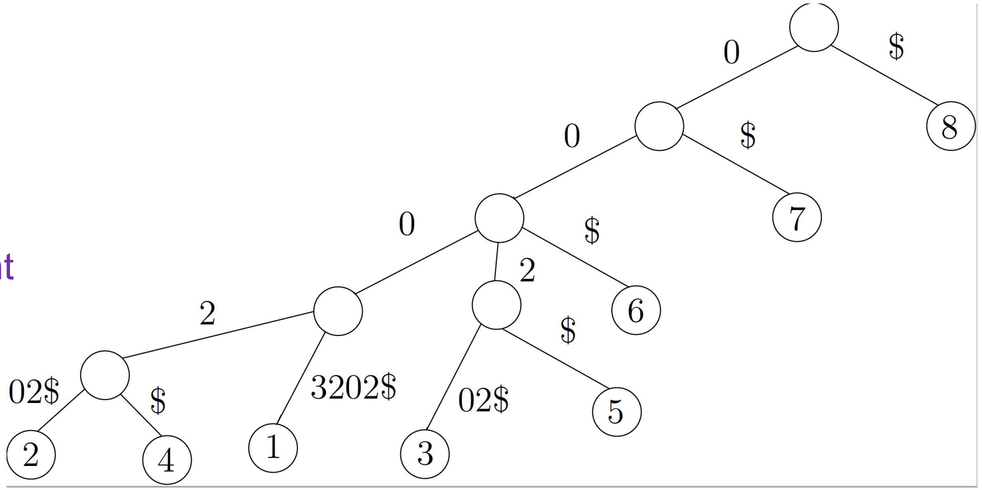
Figure 1 The text is $T[1,8] = xyzxzwz\$$, where $\Sigma = \{w, x, y, z, \$\}$

Parameterized Suffix Tree

Prev Encoding of a string S: for each character,

- replace the first occurrence by “0” and
- any other occurrence by the distance to its right most previous occurrence

Example: Prev(ababb) = 00221



PLF mapping

PLF[2] = 4

PSA

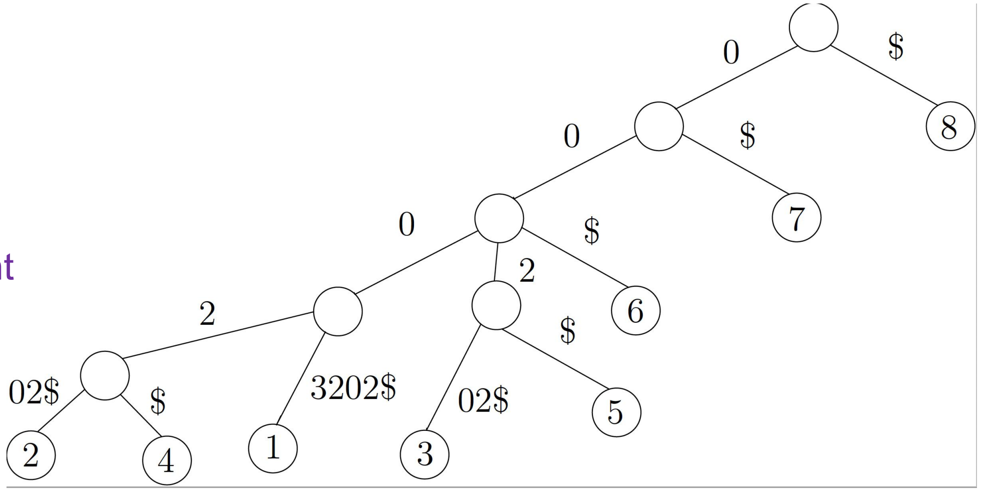
1	2	3	4	5	6	7	8
2	4	1	3	5	6	7	8

Parameterized Suffix Tree

Prev Encoding of a string S: for each character,

- replace the first occurrence by "0" and
- any other occurrence by the distance to its right most previous occurrence

Example: Prev(ababb) = 00221



PLF mapping

PLF[2] = 4

PSA	1	2	3	4	5	6	7	8
	2	4	1	3	5	6	7	8

What (succinct) information can be associated with the leaves, so that order inversion (and hence PLF) can be computed? .... pBWT and tree topology, etc

BWT vs pBWT

<u>BWT</u>

Let T[x, n] be the string corresponding to i-th leaf in (standard) suffix tree, then BWT[i] records edits between strings corresponding to i-th and LF[i]-th leaves. i.e., BWT[i] = T[x-1].

BWT vs pBWT

BWT

Let T[x, n] be the string corresponding to i-th leaf in (standard) suffix tree, then BWT[i] records edits between strings corresponding to i-th and LF[i]-th leaves. i.e., BWT[i] = T[x-1].

pBWT

Let Prev(T[x, n]) be the string corresponding to i-th leaf in parameterized suffix tree, then pBWT[i] records edits between strings corresponding to i-th and pLF[i]-th leaves.

Example: Let T[x, n] = a a b a b c a d b ... and T[x-1] = c, then

Prev(T[x, n]) = Prev(a a b a b c a d b ...) = 0 1 0 2 2 0 3 0 4 ...
Prev(T[x-1, n]) = Prev(c a a b a b c a d b ...) = 0 0 1 0 2 2 6 3 0 4 ...

pBWT[i] = 3, since the 3rd “0” is changed ---- to the largest value possible at its location

BWT vs pBWT

BWT
Let T[x, n] be the string corresponding to i-th leaf in (standard) suffix tree, then BWT[i] records edits between strings corresponding to i-th and LF[i]-th leaves. i.e., BWT[i] = T[x-1].

pBWT
Let Prev(T[x, n]) be the string corresponding to i-th leaf in parameterized suffix tree, then pBWT[i] records edits between strings corresponding to i-th and pLF[i]-th leaves.

Example: Let T[x, n] = a a b a b c a d b ... and T[x-1] = c, then

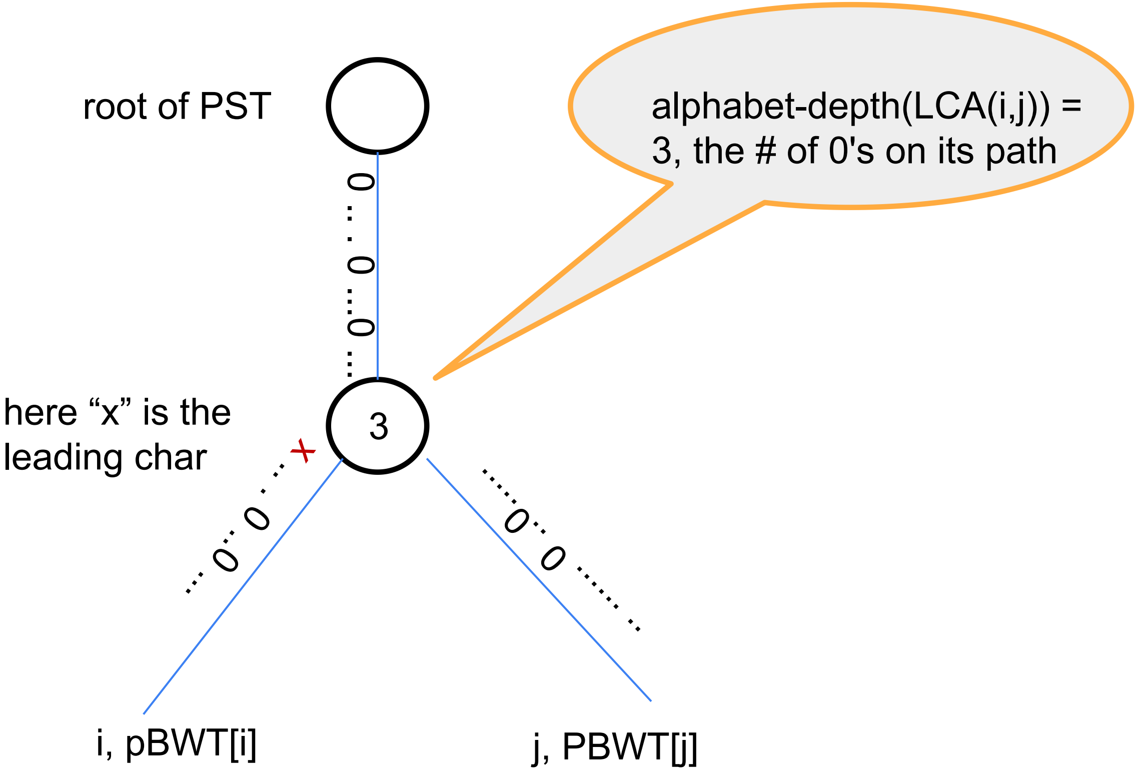
Prev(T[x, n]) = Prev(a a b a b c a d b ...) = 0 1 0 2 2 0 3 0 4 ...
Prev(T[x-1, n]) = Prev(c a a b a b c a d b ...) = 0 0 1 0 2 2 6 3 0 4 ...

pBWT[i] = 3, since the 3rd "0" is changed ---- to the largest value possible at its location

Given (i, j, pBWT[i], pBWT[j]), where i < j, can we quickly decide if pLF[i] > pLF[j] (i.e., inversion)?

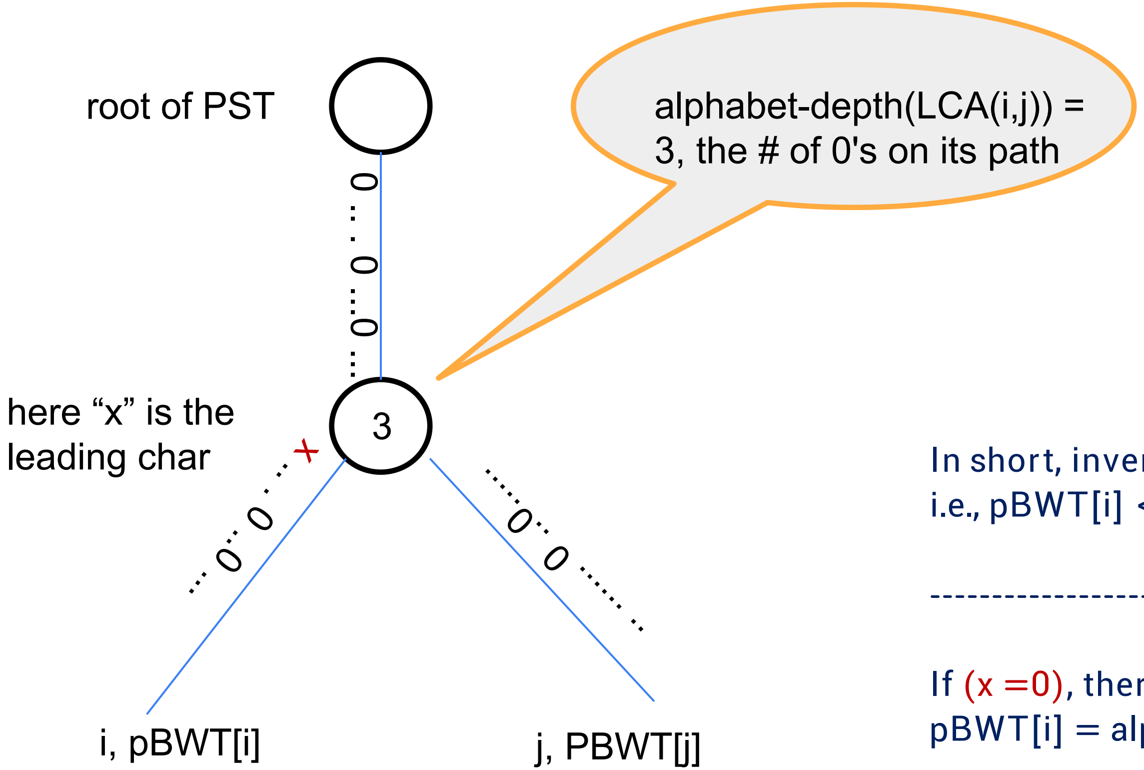
Inversion or NOT?

Given (i, j, pBWT[i], pBWT[j]), where i < j, can we quickly decide if pLF[i] > pLF[j] (i.e., inversion)?



Inversion or NOT?

Given (i, j, pBWT[i], pBWT[j]), where i < j, can we quickly decide if pLF[i] > pLF[j] (i.e., inversion)?



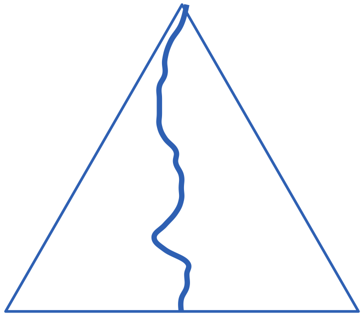
See the following examples for (pBWT[i], pBWT[j]) (assume $x \neq 0$)

- (2, 3) is an inversion
- (2, 4) is an inversion
- (3, 2) is a non-inversion
- (4, 4) is a non-inversion

In short, inversion IFF "i" change before "j" before LCA i.e., pBWT[i] < pBWT[j], alphabet-depth(LCA)

If ($x = 0$), there is an additional CASE: inversion if pBWT[i] = alphabet-depth(LCA) + 1 ≤ pBWT[j]

Putting things together



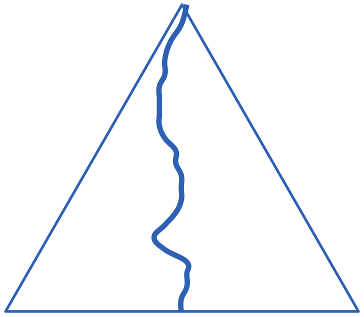
n1 = # of non-inversions on left of i

i

n2 = # of inversions on the right of i

- Inversion or NOT in PST can be Computed Quickly
- Implement inversion counting via batch queries (mostly using standard techniques from succinct data structures)
- LF mapping **LF[i] = n1+n2+1**

Putting things together



- Inversion or NOT in PST can be Computed Quickly
- Implement inversion counting via batch queries (mostly using standard techniques from succinct data structures)
- LF mapping **LF[i] = n1+n2+1**

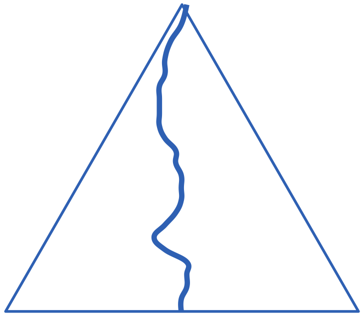
LF mapping in time O(log |Σ|) via
pBWT+tree topology+ etc. in space n log |Σ| + l.o.t (in bits)

Therefore,
Compressed PSA in space n log |Σ|+ O((n log n)/ D)+ l.o.t bits
SA/ISA queries in time O(D log |Σ|) [SODA 17]

i.e., O(n log |Σ|) bits space and O(log n) time by fixing D

Putting things together

- Inversion or NOT in PST can be Computed Quickly
- Implement inversion counting via batch queries (mostly using standard techniques from succinct data structures)
- LF mapping **LF[i] = n1+n2+1**



n1 = # of non-inversions on left of i

i

n2 = # of inversions on the right of i

LF mapping in time $O(\log |\Sigma|)$ via
pBWT+tree topology+ etc. in space $n \log |\Sigma|$ + l.o.t (in bits)

Therefore,
Compressed PSA in space $n \log |\Sigma| + O((n \log n)/D)$ + l.o.t bits
SA/ISA queries in time $O(D \log |\Sigma|)$ [SODA 17]

i.e., $O(n \log |\Sigma|)$ bits space and $O(\log n)$ time by fixing D

More Space-time Trade-offs
and LCP Compression.
i.e., compact PST
[ICALP 2022]

- ✓ $O(n \log |\Sigma|)$ bits space and $O(\log^{\epsilon} n)$ time
- ✓ $O(n \log |\Sigma| \log \log_{|\Sigma|} n)$ bits space and $O(\log \log_{|\Sigma|} n)$ time
- ✓ $O(n \log |\Sigma| \log_{|\Sigma|}^{\epsilon} n)$ bits space and $O(1)$ time

This matches the best space-time trade-offs for suffix trees

Next is

Order-isomorphic Suffix Array/Tree Compression

Order-isomorphic Suffix Tree

<u>Key idea is Pred Encoding.</u>
Pred(S) replace each character by the (closest) distance to its predecessor (and 0 is there is not predecessor)

Order-isomorphic ST is a compact trie of all Pred enoded suffixes of the text.
Linear space and pattern matching can be done efficiently

Compressed Indexing is hard, because of the main changes in Pred encoding. Example,

Let T[x, n] = 7 6 5 4 3 2 ...   and T[x-1] = 1
  Pred(T[x,n]) =   0 0 0 0 0 0 ...
Pred(T[x-1, n]) = 0 1 2 3 4 5 6 ... (i.e., many changes)

Order-isomorphic Suffix Tree

New idea for Compression: LF sccessor,

j = LF-succesor(i) IFF LF[j] = LF[i]+1

- We showed that LF-succesor can be implemented in O(log sigma) time
- Then LF via multiple LF-successors in O(log n) time
- Finally SA/ISA queries via multiple LF queries in O($\log^2$ n) time [ICALP 2021]

Next we have

2D Suffix Array/Tree Compression

Summary and Open Problems

- Compression of
  - Parameterized Suffix Arrays/Trees (well solved)
  - Order-isomorphic Suffix Arrays/trees (we have some solution, but messy and less efficient)
  - 2D Suffix Arrays/Trees (Wide Open for research)

- Repetition-aware Compression
- Efficient Construction
- Indexing for other/newer problems

Thank you for listening

Questions?