Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching

Roberto Grossi

Università di Pisa

Jeff Vitter

Duke University

Center for Geometric Computing

String Matching: The Problem

Motivation: Huge collections of textual data.

Input:

- ☆ Text T of length n.
- ☆ Pattern P of length $m \leq n$.
- ☆ Binary alphabet.

Types of Queries:

- ☆ *Existential:* Does P occurs in T?
- ☆ *Counting:* Give number occ of occurrences of P in T.
- ☆ *Enumerative:* List all positions where P occurs in T.

String Matching: Revisited Tour on the Bounds

Search time	*Space (words)*	*Authors*
$O(m+n)$	$O(m)$	Knuth-Morris-Pratt'77, . . . long list.
$O(m+n)$	$O(1)$	Galil-Seiferas'83, Crochemore-Perrin.
$O(m)$	$O(n)$	Morrison'68, Weiner'73, . . . long list.
$o(m)$?	$o(n)$?	*Yes!!! The topic of this talk.*

- ☆ Typically pattern length $m \ll$ text length n.
- ☆ Binary alphabet and standard RAM with wordsize $O(\log n)$.

String Matching: Text Indexing

- Basic idea from Morrison'68, Weiner'73, . . . :
    1. Scan the text only initially.
    2. Build an index: e.g., suffix array or suffix tree (Patricia trie storing all suffixes of the text).
    3. Pattern search: index lookup.
- Search time drops from $O(m+n)$ to $O(m)$ (for counting and existence queries).
- Additional output sensitive cost $O(occ)$ (for enumerative queries).
- Space increases from $O(1)$ to $O(n)$ words, i.e., $O(n \log n)$ bits!

String Matching: Space

☆ *Criticism on greediness of space:* **$\boldsymbol{\Omega(n \log n)}$** *bits.*

- Need to store $\Omega(n)$ positions explicitly:

  Pattern $P \longrightarrow \boxed{\text{Suffix tree index}} \longrightarrow$ P occurs at i

- Each position requires at least $\log n$ bits.
- Index at least **$\boldsymbol{\log n}$** times **larger** than the text!

☆ *Inverted lists take less space (but less functionality).*

Space Reduction Issues

- ⋆ *Analysis of constants in* $O(n \log n)$ *bit space:*

  Manber-Myers'93, Andersson-Nilsson'95, Kärkkäinen'95, Clark'96, Clark-Munro'96, Kurtz'98 (many refs), Giegerich-Kurtz-Stoye'99,

- ⋆ *Making suffix trees sparse:*

  Morrison'68, Gonnet-BaezaYates-Snider'92, Manber-Wu'94, Colussi-DeCol'96, Kärkkäinen-Ukkonen'96ab, Andersson-Larsson-Swanson'99

- ⋆ *LZ and BW compression:*

  Kärkkäinen-Sutinen'98, Ferragina-Manzini'00

- ⋆ *Succinct representation*:

  Jacobson'89, Clark-Munro'96, Munro-Raman'97, Munro-Raman-SrinivasaRao'98

Our Results (1)

- ⋆ Break through both the time barrier of $\boldsymbol{O(m)}$ time and the space barrier of $\boldsymbol{O(n \log n)}$ bits.
- ⋆ Compressed suffix arrays:
  - *compress* in $\boldsymbol{O(n)}$ bits and $\boldsymbol{O(n)}$ time. [$O(n \log\log n)$ bits]
  - *lookup* in $\boldsymbol{O(\log^{\epsilon} n)}$ time, $\epsilon < 1$. [$O(\log\log n)$ time]
- ⋆ Provably as good as inverted lists in space usage and more functionality on arbitrary substrings.

Our Results (2)

- ☆ Compressed suffix trees in $\boldsymbol{O(n)}$ bits: *same* space as that of text.
- ☆ Text indexing on T: only $\boldsymbol{O(n)}$ bits.
  - Existential & Counting in $\boldsymbol{o(m)}$ time, specifically

$$\begin{cases} \boldsymbol{O(1)} & \text{for } m < \epsilon \log n; \\ \boldsymbol{O(m/\log n + \log^{\epsilon} n)} & \text{otherwise.} \end{cases}$$

  - Enumerative for *occ* occurrences in additional time

$$\begin{cases} \boldsymbol{O(occ)} & \text{for } m = \Omega\big((\log^3 n) \log\log n\big) \text{ or } occ = \Omega(n^{\epsilon}); \\ \boldsymbol{O(occ \log^{\epsilon} n)} & \text{otherwise.} \end{cases}$$

Vanilla Suffix Arrays SA

Definition of *suffix array* SA:

$SA[i]$ = starting position of ith lexicographically smallest suffix.

Example for $n = 7$ (text length 6):

(Assume that `a` $<$ `#` $<$ `b`)

Input text:

`b a b a b a #`

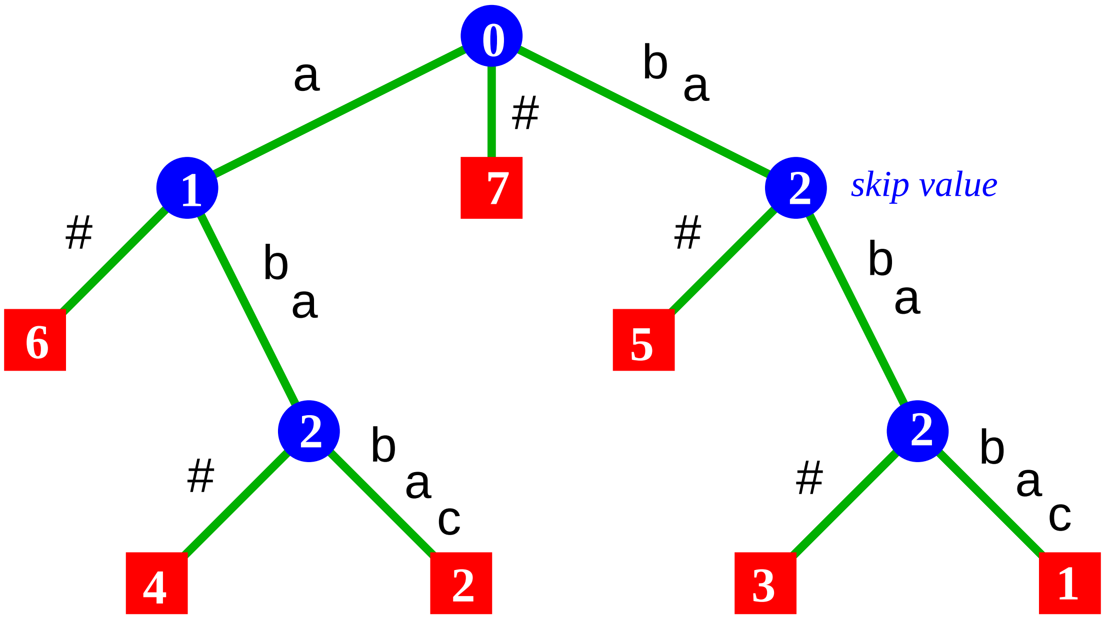
Suffix array:

6 4 2 7 5 3 1

Sorted list of suffixes

6 : `a #`
4 : `a b a #`
2 : `a b a b a #`
7 : `#`
5 : `b a #`
3 : `b a b a #`
1 : `b a b a b a #`

Suffix Trees (Patricias)



State of the Art on Suffix Trees

Patricia tries compress a chain of unary nodes, so the total number of tree nodes reduces from $\boldsymbol{n^2}$ to $\boldsymbol{n}$.

[Morrison'68]

Munro et al.'s 3 remarks on further space saving:

1. **The Patricia trie topology**: $\boldsymbol{O(n)}$ bits

   [Jacobson'89, Munro-Raman'97]

2. **The skip values**:
   $\boldsymbol{\widetilde{O}(\log\log n)}$ bits each and $\boldsymbol{O(1)}$ retrieval time

   [Clark-Munro'96]

   $\boldsymbol{O(1)}$ bits each and $\boldsymbol{O(\mathit{skip\_value})}$ retrieval time

   [Munro-Raman-SrinivasaRao'98]

3. **The suffix pointers** (i.e., suffix array): $\boldsymbol{n\log n}$ bits

   [previously $\boldsymbol{o(n\log n)}$ unknown]

Abstract Data Type Optimization [Jacobson'89]

This talk: **suffix arrays + operations *compress* & *lookup*.**

- DATA STRUCTURE $\longrightarrow$ $\boxed{\text{Optimize}}$ $\longrightarrow$ data structure
- # distinct DATA STRUCTURES $= C(n)$
  $\Longrightarrow$ Each data structure occupies $O(\log C(n))$ bits.
- Same set of supported operations.
- Guarantee on the time complexity?

Example: Suffix array permutations for $n = 5$

`a a a a #`	`a a a b #`	`a a b a #`	`a a b b #`
1 2 3 4 5	1 2 3 5 4	1 4 2 5 3	1 2 5 4 3
`a b a a #`	`a b a b #`	`a b b a #`	`a b b b #`
3 4 1 5 2	1 3 5 2 4	4 1 5 3 2	1 5 4 3 2
`b a a a #`	`b a a b #`	`b a b a #`	`b a b b #`
2 3 4 5 1	2 3 5 1 4	4 2 5 3 1	2 5 1 4 3
`b b a a #`	`b b a b #`	`b b b a #`	`b b b b #`
3 4 5 2 1	3 5 2 4 1	4 5 3 2 1	5 4 3 2 1

☆ Among the $\boldsymbol{n!}$ permutations, only $\boldsymbol{C(n) = 2^{n-1}}$ are valid.
$\Longrightarrow$ data structure size $\geq$ $\mathbf{log}\, \boldsymbol{C(n) = n - 1}$ bits.

☆ 1-1 correspondence between suffix arrays and strings:
$\Longrightarrow$ naïve $O(n)$-time *compress* & *lookup*. (*TOO SLOW!*)

Our Idea: Recursive Deconstruction (1)

1. Link each valid permutation to the suffixes.
2. Start out with $\boldsymbol{SA_0}$ = suffix array.
3. For $0 \leq k < \log\log n$, perform recursive step k:
   Index only the suffixes starting at text positions
   $2^{k+1}, 2^{k+2}, 2^{k+3}, \ldots, n$.
   This replaces SA_k with a suffix array SA_{k+1} of half its size:
   $$\boldsymbol{SA_k} \longleftrightarrow \{\ \boldsymbol{B_k},\ \boldsymbol{rank_k},\ \boldsymbol{\Psi_k}\ \} \cup \boldsymbol{SA_{k+1}}.$$
4. As a result, $\boldsymbol{|SA_{k+1}|} = \dfrac{\boldsymbol{|SA_k|}}{\boldsymbol{2}} = \dfrac{\boldsymbol{n}}{\boldsymbol{2^{k+1}}}$.
   After **log log** $\boldsymbol{n}$ recursive steps, we have
   $$\boldsymbol{|SA_{\log\log n}|} = \frac{\boldsymbol{n}}{\boldsymbol{\log n}} \quad \Longrightarrow \quad \boldsymbol{O(n)} \text{ bits!}$$

	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*	*20*	*21*	*22*	*23*	*24*	*25*	*26*	*27*	*28*	*29*	*30*	*31*	*32*
T:	`a`	`b`	`b`	`a`	`b`	`b`	`a`	`b`	`b`	`a`	`b`	`b`	`a`	`b`	`a`	`a`	`a`	`b`	`a`	`b`	`a`	`b`	`b`	`a`	`b`	`b`	`b`	`a`	`b`	`b`	`a`	`#`
SA_0:	15	16	31	13	17	19	28	10	7	4	1	21	24	32	14	30	12	18	27	9	6	3	20	23	29	11	26	8	5	2	22	25
B_0:	**0**	**1**	**0**	**0**	**0**	**0**	**1**	**1**	**0**	**1**	**0**	**0**	**1**	**1**	**1**	**1**	**1**	**1**	**0**	**0**	**1**	**0**	**1**	**0**	**0**	**0**	**1**	**1**	**0**	**1**	**1**	**0**
$rank_0$:	0	1	1	1	1	1	2	3	3	4	4	4	5	6	7	8	9	10	10	10	11	11	12	12	12	12	13	14	14	15	16	16
Ψ_0:	2	2	14	15	18	23	7	8	28	10	30	31	13	14	15	16	17	18	7	8	21	10	23	13	16	17	27	28	21	30	31	27

	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*
SA_1:	8	14	5	2	12	16	7	15	6	9	3	10	13	4	1	11
B_1:	**1**	**1**	**0**	**1**	**1**	**1**	**0**	**0**	**1**	**0**	**0**	**1**	**0**	**1**	**0**	**0**
$rank_1$:	1	2	2	3	4	5	5	5	6	6	6	7	7	8	8	8
Ψ_1:	1	2	9	4	5	6	1	6	9	12	14	12	2	14	4	5

	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*
SA_2:	4	7	1	6	8	3	5	2
B_2:	**1**	**0**	**0**	**1**	**1**	**0**	**0**	**1**
$rank_2$:	1	1	1	2	3	3	3	4
Ψ_2:	1	5	8	4	5	1	4	9

	1	*2*	*3*	*4*
SA_3:	2	3	4	1

Recursive Deconstruction (2)

$$\boldsymbol{SA_k} \longleftrightarrow \{\ \boldsymbol{B_k},\ \boldsymbol{rank_k},\ \boldsymbol{\Psi_k}\ \} \cup \boldsymbol{SA_{k+1}}.$$

- ☆ $\boldsymbol{B_k}$ = Bit vector, such that $B_k[i] = \mathbf{1}$ iff $SA_k[i]$ is *even*.
- ☆ $\boldsymbol{rank_k(j)}$ = #**1**s in the first j bits of B_k.
- ☆ Companion items:

$$\boldsymbol{\Psi_k(i)} = \begin{cases} j & \text{if } SA_k[i] \text{ is odd and } SA_k[j] = SA_k[i] + 1; \\ i & \text{otherwise.} \end{cases}$$

- ☆ $\boldsymbol{SA_{k+1}} \leftarrow$ Pack the even values and divide each of them by 2.

Implementation of Ψ_k's

Level $k = 0$:

`a` list: $\langle 2, 14, 15, 18, 23, 28, 30, 31 \rangle$, $|\texttt{a}\ \text{list}| = 8$

`b` list: $\langle 7, 8, 10, 13, 16, 17, 21, 27 \rangle$, $|\texttt{b}\ \text{list}| = 8$

Level $k = 1$:

`aa` list: $\emptyset$, $|\texttt{aa}\ \text{list}| = 0$

`ab` list: $\langle 9 \rangle$, $|\texttt{ab}\ \text{list}| = 1$

`ba` list: $\langle 1, 6, 12, 14 \rangle$, $|\texttt{ba}\ \text{list}| = 4$

`bb` list: $\langle 2, 4, 5 \rangle$, $|\texttt{bb}\ \text{list}| = 3$

Level $k = 2$: ...

Compressed Suffix Arrays

- ☆ *compress*: Apply $\ell = \Theta(\log\log n)$ recursive steps.
  - Level $k < \ell$ stores only B_k, $rank_k$, Ψ_k in compressed form.
  - Last level ℓ stores only SA_ℓ in $O(n)$ bits.
  - Reconstruct SA_k from SA_{k+1} by the formula:

$$SA_k[i] = 2 \cdot SA_{k+1}\big[rank_k\big(\Psi_k(i)\big)\big] + (B_k[i] - 1).$$

- ☆ $rlookup(i,k)$:

```
if $k = \ell$ then $SA_\ell[i]$
else $2 \times rlookup\big(rank_k(\Psi_k(i)),\ k+1\big) + (B_k[i] - 1)$.
```

Top-level: $k = 0$ to get $SA[i]$.

Center for Geometric Computing

Theorem 1

Bounds for compressed suffix arrays:

i. *compress* in $\boldsymbol{O(n \log \log n)}$ bits and $\boldsymbol{O(n)}$ preprocessing time;
lookup(i) in $\boldsymbol{O(\log \log n)}$ time;

ii. *compress* in $\boldsymbol{O(n)}$ bits and $\boldsymbol{O(n)}$ preprocessing time;
lookup(i) in $\boldsymbol{O(\log^{\epsilon} n)}$ time.

Multi-Level Text Index: Old and New Ingredients

☆ LZ-index for short patterns of length $m < \epsilon \log n$.

[Kärkkäinen-Sutinen'98]

☆ For patterns of length $m \geq \epsilon \log n$:

- *Top level:* Sparse suffix tree (Patricia) with $O(n/\log n)$ nodes.

  [Kärkkäinen-Ukkonen'96ab]

- $O(1)$ *middle levels:* Space efficient Patricias with $O(\log^{\epsilon} n)$ nodes.

  [Munro-Raman-SrinivasaRao'98]

- *Last level:* Our compressed suffix array.
- Trick: Perfect hash to speed up the Patricia traversal.

Theorem 2

Compressed suffix trees and text indexing:

- ☆ Index data structure on text T in $\boldsymbol{O(n)}$ bits.
- ☆ Any pattern string P of m bits packed into $O(m/\log n)$ words:

i. Existential *&* Counting in $\boldsymbol{o(m)}$ time:

$$\begin{cases} \boldsymbol{O(1)} & \text{for } m = o(\log n); \\ \boldsymbol{O(m/\log n + \log^{\epsilon} n)} & \text{otherwise} \end{cases}$$

ii. Enumerative for *occ* occurrences in additional time

$$\begin{cases} \boldsymbol{O(occ)} & \text{for } m = \Omega((\log^3 n)\log\log n) \text{ or } occ = \Omega(n^{\epsilon}); \\ \boldsymbol{O(occ\ \log^{\epsilon} n)} & \text{otherwise} \end{cases}$$

Conclusions

- ⋆ The first index structure to break through both the time barrier of $O(m)$ time and the space barrier of $O(n \log n)$ bits.
  - $O(n)$-bit *compress* and $O(1)$-time *lookup*?
  - Characterize combinatorially the suffix array permutations?
  - Small number of errors in the pattern queries?
- ⋆ Follow-ups: lower bound on the index size; compressed texts.