Text Data
Algorithms and Data Structures

MohammadTaghi Hajiaghayi and Eleni Drinea

MIT and Harvard

hajiagha@theory.lcs.mit.edu and *edrinea@deas.harvard.edu*

Outline

- Motivation for text data
- Some definitions
- Inverted index
- Suffix array
- Suffix tree

Why string data are interesting?

Many applications:

- Digital libraries and product catalogues
- Electronic white and yellow pages
- Specialized information sources (e.g. genomic or patent dbs)
- Web pages repositories
- Private information dbs
- ...

String collections are growing at a staggering rate:

- ... more than 10Tb of textual data on the web
- ... more than 15Gb of base pairs in the genomic dbs

The need for an index

Brute-force scanning is not a viable approach

- We need very fast searches, e.g., in Google
- American Heritage Dictionary: Index is anything that serves to guide, point out or otherwise facilitate reference, as: alphabetized listing, file or catalogue.

> In computer science an index is a persistent data structure that allows to focus the search for a *query string* (or a set of them) on a *provably small portion* of the data collection.

Two families of indexes

Types of data:

- Linguistic or tokenizable text, e.g., dbs files
- Raw sequence of characters or bytes, e.g., DNA sequences, Audio-video or Executable files

Types of query:

- Word-based query
- Character-based query

Types of indexing approaches:

- Word-based indexes: concept of *word* must be defined, e.g., inverted files
- Full-text indexes, no constraint on text and queries, e.g., suffix array, suffix tree

Inverted files or indexes

We create an index file for tokenizable documents contains:

- Term (or word)
- Number of documents in which a word has been appeared
- Total frequency of each word
- Pointers to places of appearances

. . .			
is	1	1	(1,5)
. . .			
time	2	2	(1,12),(2,5)
. . .			

Doc #1

Now is the time

The time was past

Doc #2

Index Construction

We can create an Index as follows:

- Create doc-place-term pairs $< d, p, t >$ sorted by increasing $< d, p >$
- Merge-sort on the third component t;
- Build the index file from adjacent triples with equal t;

Basic Notation

- Text $T = t_1 t_2 \ldots t_N$
- T_i: the suffix that starts at position i in T.
- Pattern P, $|P| = p$, occurs in T at position i *iff* P is a p-prefix of T_i (denote by $P =_p T_i$)

$$\text{E.g.,} \quad \begin{array}{rcr} T & = & \text{bananas} \\ P & = & \text{ana} \end{array} \Longrightarrow P =_3 T_2 =_3 T_4 \Longrightarrow P \text{ occurs at 2,4}$$

- Occurrences of P in T: all suffixes having P as prefix.
- SUF(T): ordered set of suffixes of T

Suffix Array SA (Manber, Myers '90)

Property: All suffixes in SUF(T) having prefix P are contiguous.

Intuitively, SA "holds" the suffixes in lexicographic order.

Formally, SA is an array of N integers, s.t.

$SA[i]$ = starting position of i-th smallest suffix

Then $T_{SA[i]}$ is the i-th smallest suffix in T.

Space: $\Theta(N)$ instead of $\Theta(N^2)$.

Suffix array SA

E.g., T = bananam

$$
\begin{array}{ccc}
\text{Suffixes} & & \text{Sorted Suffixes} & & \text{Suffix Array} \\
\begin{aligned}
T_1 &= \text{bananam} \\
T_2 &= \text{ananam} \\
T_3 &= \text{nanam} \\
T_4 &= \text{anam} \\
T_5 &= \text{nam} \\
T_6 &= \text{am} \\
T_7 &= \text{m}
\end{aligned}
& \Longrightarrow &
\begin{aligned}
\text{am} &= T_6 \\
\text{anam} &= T_4 \\
\text{ananam} &= T_2 \\
\text{bananam} &= T_1 \\
\text{m} &= T_7 \\
\text{nam} &= T_5 \\
\text{nanam} &= T_3
\end{aligned}
& \Longrightarrow &
\text{SA} = \begin{bmatrix} 6 \\ 4 \\ 2 \\ 1 \\ 7 \\ 5 \\ 3 \end{bmatrix}
\end{array}
$$

Searching in Suffix Array

Problem: Find all occurrences of P in T, $|P| = p$.

- SA is lexicographically sorted $\Rightarrow$ need to find the first suffix T_i in SA whose p-prefix equals P.

- Binary search for T_i.

E.g., P = ana

SA = [6 4 2 1 7 5 3]

[6	4	2	1	7	5	3]
a	a	a	b	m	n	n
m	n	n	a		a	a
	a	a	n		m	n
	m	n	a			a
		a	n			m
		m	a			
			m			

Searching in Suffix Array (ctd)

Algorithm: (sketch)

- Binary search for $L_P = \min\ i$ s.t. $P \leq_p T_{SA[i]}$
- Brute Force compare for $R_P = \max\ i$ s.t. $T_{SA[i]} \leq_p P$
- Report all $i \in [L_P, R_P]$.

➡ Time: $O(p(\log N + occ))$ single-symbol comparisons

➡ Can be improved to: $O(p + \log N + occ)$

An $O(p \log N)$ search for L_P

if $P \leq T_{SA[1]}$ then $L_P \leftarrow 1$
else if $P > T_{SA[N]}$ then $L_P \leftarrow N + 1$
else
    $(L, R) \leftarrow (1, N)$
    while $R - L > 1$ do $\{$
        $M \leftarrow (R + L)/2$
        if $P \leq T_{SA[M]}$ then
            $R \leftarrow M$
        else
            $L \rightarrow M$
    $\}$
    $L_P \leftarrow R$
$\}$

Improved searching

Main Idea: use precomputed information about longest common prefixes of the suffixes of T

- Let lcp(v, w) = length of longest common prefix of v, w

- lcp(v, w) is a byproduct of comparing v, w

- While binary searching, exactly $N-2$ intervals may arise; for each interval, precompute the lcp's between left endpoint and middle, middle and right endpoint; using this information the total number of single-symbol comparisons to compute L_P goes down to $p + \lceil \log_2 (N-1) \rceil$

Sorting the Suffix Array

- Sorting in stages
- Stage 1: bucket sort according to first symbol
- End of Stage 1: $N_1 = 4$

Bucket 1	Bucket 2	Bucket 3	Bucket 4
ananam	bananam	m	nanam
anam			nam
am			

$\Rightarrow$ SA $= [2\ 4\ 6\ 1\ 7\ 3\ 5]$

- Time: $O(N)$

Sorting a Suffix Array (ctd)

- Stage H: double the number of symbols defining the buckets; sort; repartition the buckets

- End of Stage H ($H = 1, 2, 4, , \ldots$):
  - suffixes partitioned into N_H buckets
  - each bucket holds suffixes with same first H symbols
  - buckets sorted according to their first H symbols

- Sorting complete after $\lceil \log_2(N+1) \rceil$ stages

- Need to show how to sort elements inside each H-bucket to produce the $2H$ ordering in $O(N)$ time

Sorting according to $2H$ symbols

➡ Let T_i, T_j be in the same bucket after H-th stage

Key Idea: Comparing T_i, T_j according to their first $2H$ symbols $\Longleftrightarrow$ comparing T_{i+H}, T_{j+H} according to their first H symbols.

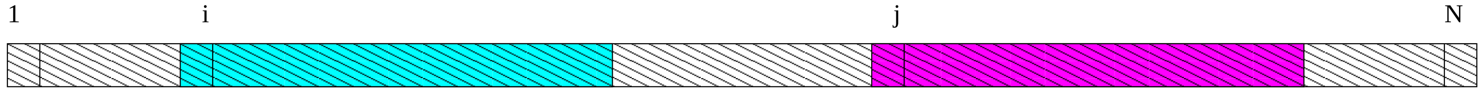
➡ Let T_i be first element in first bucket $\Rightarrow$ then T_{i-H} must be first in its $2H$ bucket:

- move T_{i-H} to the top of its H-bucket and mark this fact.
- repeat for every suffix in the order it appears in the SA.

Linear Time Expected-case Variations

Assume that all N-symbol strings are equally likely.

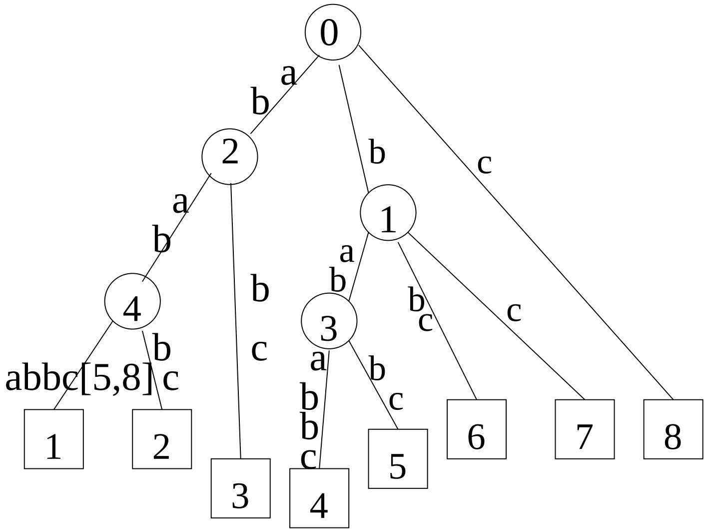
- Expected length of longest repeated substring is $2 \log_{|\Sigma|} N + O(1)$



- Sorting stages: $O(\log \log N) \Rightarrow$ expected sorting time $O(N \log \log N)$ (can be reduced to $O(N)$)

- Suffix tree: A compact data structure built on all text suffixes in which search of a string P, can be done in $O(|P|)$

Searching for $P = babc$



T=abababbc#

Properties of Text and Suffix tree

- T1: The final character of T is unique, e.g., $\#$.

Thus, no suffix of T is a prefix of a different suffix of T.

- S1: Each arc has a non-empty string as a label
- S2: Each node has at least two children
- S3: Sibling arcs begin with different character
- S4: For each node, we can keep a hash table, in which, we query the first characters of the arcs

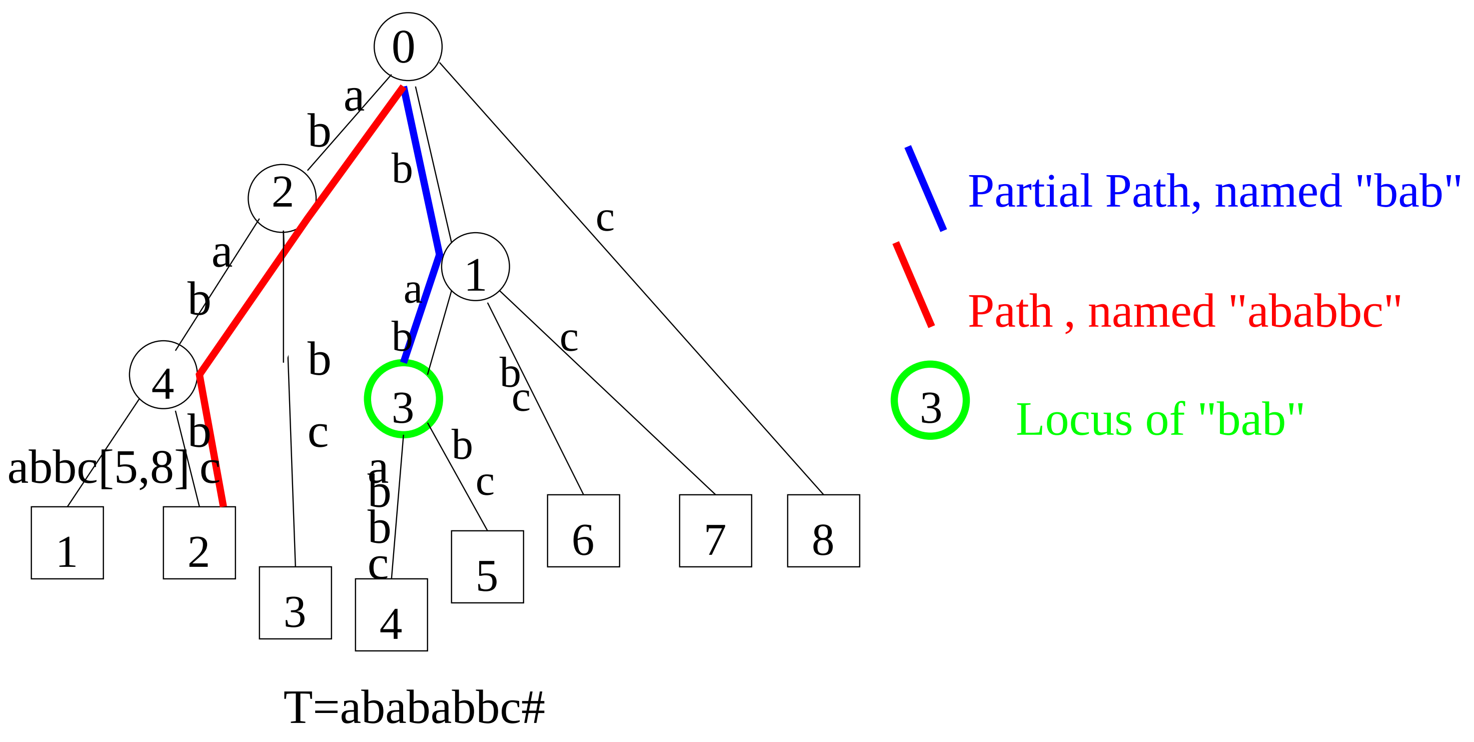
An array can work, but it has many empty pointers

- It takes linear space if we keep edges as with two integers (Intervals)

Some definitions

- Path, Partial path
- Locus of a string, Name of a path
- Extended locus of string α: Locus of the shortest extension of α whose locus is defined.
- *Suf(i)*: The suffix starting from i.
- *Head(i)*: The longest prefix of *Suf(i)* which is also a prefix of *Suf(j)*, for some $j < i$.
- *Tail(i)*: *Suf(i)* $-$ *Head(i)*

Example



0
a
b
2
b
a
b
1
4
b
c
abbc[5,8]
b
c
a
b
3
b
c
c
b
c
a
b
b
c
1
2
3
4
5
6
7
8
T=abababbc#
Partial Path, named "bab"
Path , named "ababbc"
3
Locus of "bab"

Construction of suffix tree

Algorithm: Starting from empty tree, enter paths corresponding to suffixes of T one at a time

1. Start with empty tree S_0;
2. From S_{i-1} to S_i:

2.1. Consider *Suf(i)*;

2.2. Find *Head(i)*;

2.3. Construct a new non-terminal node if necessary;

2.4. Add a terminal node with arc *tail(i)* from the non-terminal node.

Main ideas of the linear time construction

➡ Linear time construction: if we can find the extended locus of *Head(i)*, the rest is constant per each step.

Lemma: If *Head(i-1)* can be written as $x\delta$ for some character x and some (possibly empty) string δ, then δ is a prefix of *Head(i)*. (Proof: clear)

➡ One can obtain *Head(i)*'s in amortized time $O(n)$ (the approach is similar to pattern matching in $O(n + m)$ time).