Exact Matching : Techniques, Data structures, Algorithms & Applications

Motivation:
Inexact/approximate match computing is time consuming, esp. for large scale data sets.

Exact Match detection can be used as a filter (pre-condition check)

Types of exact matches — Fixed-length
                        — variable-length

Data Structures for Exact Matches:

- Lookup table
- PATRICIA tree
- Suffix Tree, suffix array + LCP array
- Compacted-trie

Lookup Tables

Good for detection of small, fixed-length exact matches

e.g.; find all substrings of length 10 shared in common among a collection of strings.

Definition:-

Let Σ ← alphabet
s ← input string of length n
w ← user-defined parameter for fixed-length match

Given, $\{s, \Sigma, w\}$ as input, a lookup table for s is ~~def~~ a table/record of all positions, that map to in s all possible substrings of length w.

eg; $w = 2$ ⇒ There are $4^2 = 16$ possible strings, assuming $\Sigma = \{a, c, g, t\}$

Index: Table T

Index		
0	AA	
1	AC	→ 1 → ⊥
2	AG	→ 4 → 7 → ⊥
3	AT	
4	CA	→ 3 → ⊥
5	CC	→ 2 → ⊥
6	CG	
7	CT	
8	GA	
9	GC	
10	GG	
11	GT	→ 5 → 8 → ⊥
12	TA	→ 6 → ⊥
13	TC	→ 9 → ⊥
14	TG	
15	TT	

Let

1	2	3	4	5	6	7	8	9	10
a	c	c	a	g	t	a	g	t	c

List of w-mers in S:

	Pos #
ac	→ 1
cc	→ 2
ca	→ 3
ag	→ 4
gt	→ 5
ta	→ 6
ag	→ 7
gt	→ 8
tc	→ 9

Purpose of a lookup table:

Locations in S that store the same substring of length w, will be grouped together in the table.

Lookup Table Construction Algorithm:

0) Initialize lookup table T with null pointers
1) Slide a window of length w on S
2) for each window, length S_i be the substring i.e.; $S_i = S[i \ldots i+w-1]$
3) index ← hash(S_i)
4) T[index].insert(i);

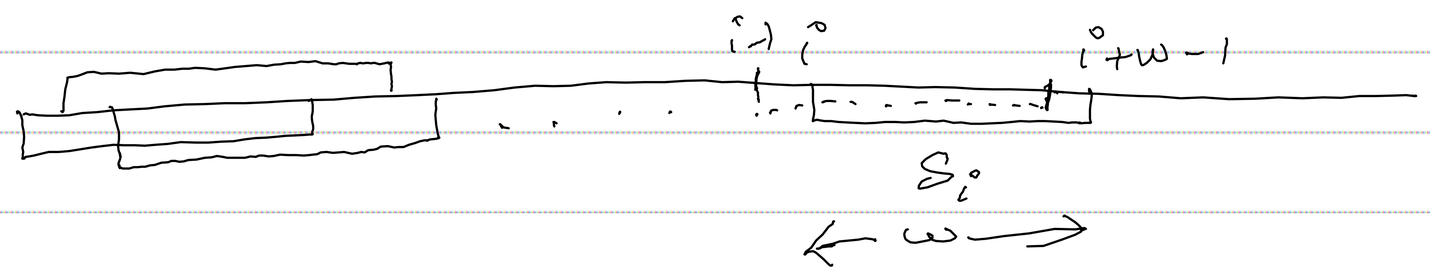
Key Properties of lookup Table:

1) space = $O(|\Sigma|^w)$ entries or pointers
   + $O(n)$ linked list entries

2) LT construction time = $O(|\Sigma|^w + n)$
   - $|\Sigma|^w$ → step(0)
   - n → steps(1–4)

(assuming hash(S_i) can be computed in constant time)

How to compute $hash(s_i)$? (for all $s_i \in S$)



$\Sigma \Rightarrow$ integers

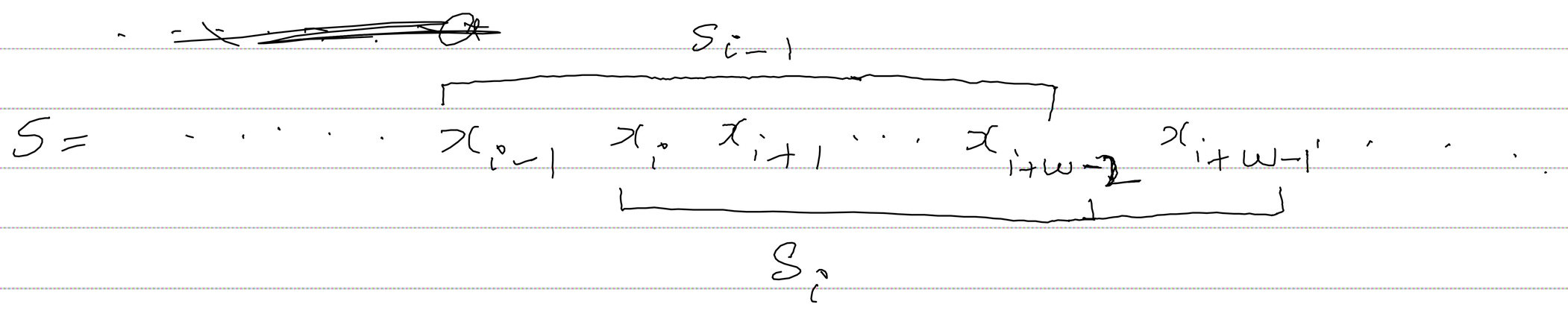
lexicographic
- a ⇒ 0
- c ⇒ 1
- g ⇒ 2
- t ⇒ 3

Interpret each w-mer as a w-bit number with base $|\Sigma|$

i.e.,

aa = 0	ca = 4	ga = 8	ta = 12
ac = 1	cc = 5	gc = 9	tc = 13
ag = 2	cg = 6	gg = 10	tg = 14
at = 3	ct = 7	gt = 11	tt = 15

We can compute $hash(s_i)$ using the value $hash(s_{i-1})$



$$hash(s_i) = \left[hash(s_{i-1}) - h(x_{i-1}) \times |\Sigma|^{w-1} \right] \times |\Sigma| + h(x_{i+w-1})$$

Lookup Tables used heavily in BLAST and a lot other applications.

Practical limitations:

→) Cannot scale for large values of w

e.g. ~~w=~~ for DNA alphabet,

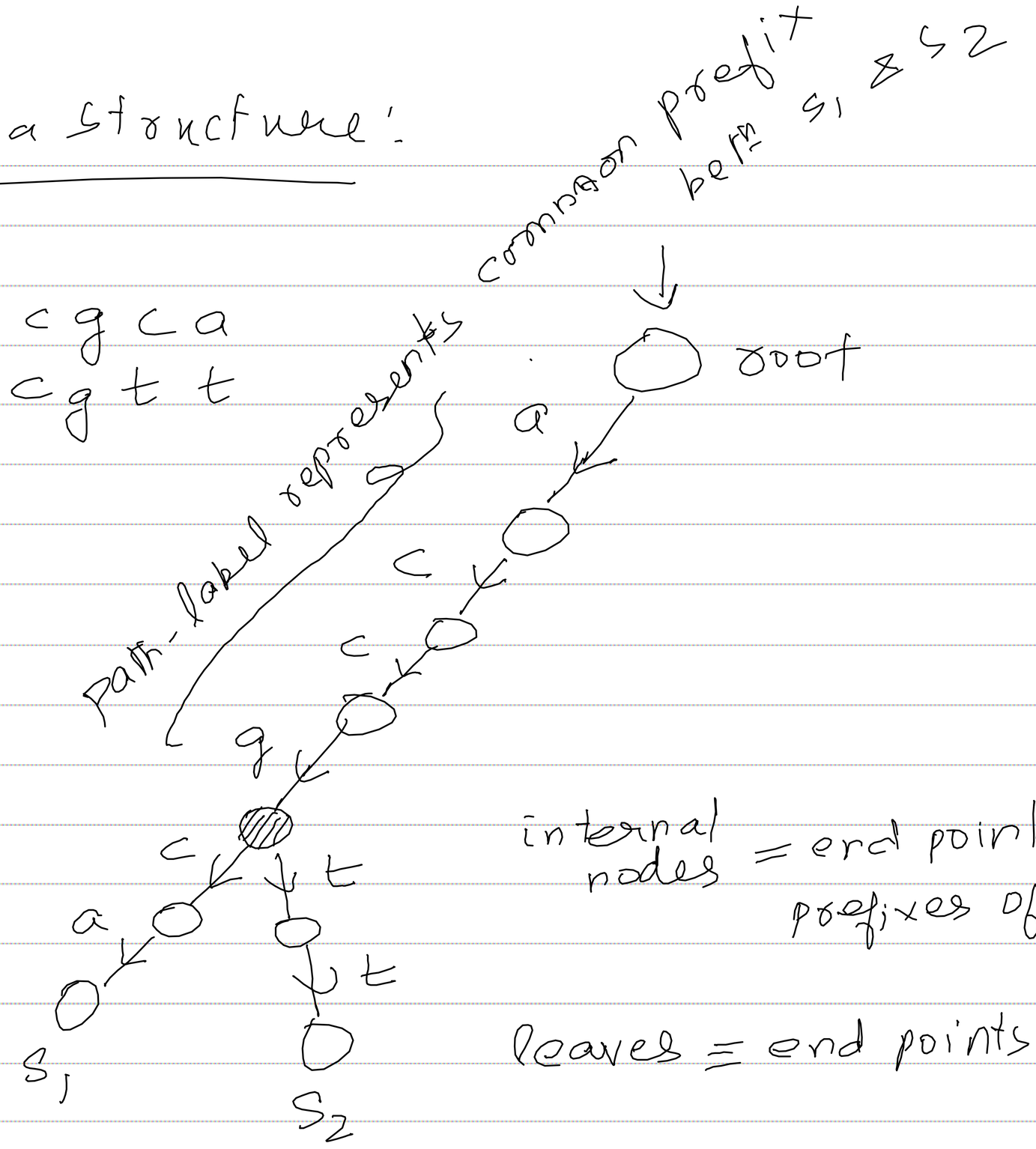
$w = 10 \Rightarrow 4^{10} \Rightarrow \approx 10^6$ entries

$w = 15 \Rightarrow 4^{15} \Rightarrow 10^9$ entries

$w = 20 \Rightarrow 4^{20} \Rightarrow 2^{40}$ or 10^{12} entries!

Trie data structure:

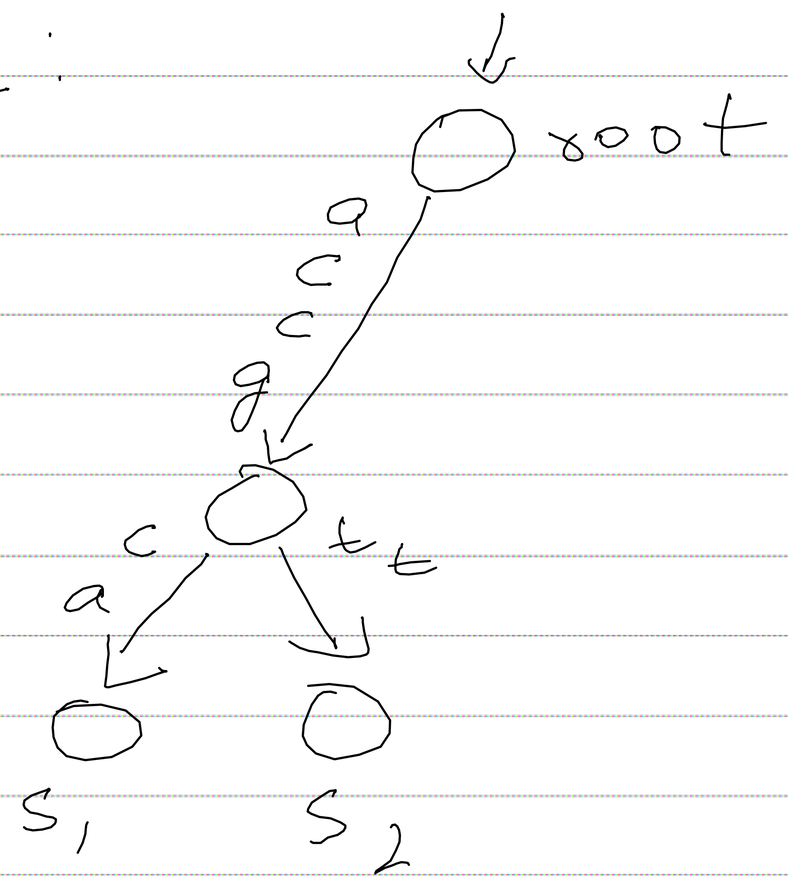
$S_1 =$ a c c g c a
$S_2 =$ a c c g t t



internal nodes = end points of prefixes of strings

leaves = end points of strings

Compacted Trie:



(Morris, 1968)

PATRICIA Tree ≡ compacted trie of an arbitrary number of input strings.

~~PB~~ Practical Appli.

"Practical Algorithm to Retrieve Information Coded in Alphanumeric"