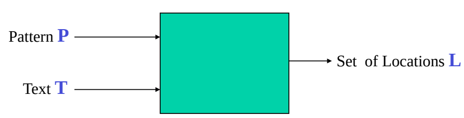
COT 6936: Topics in Algorithms

Giri Narasimhan
ECS 254A / EC 2443; Phone: x3748
giri@cs.fiu.edu

http://www.cs.fiu.edu/~giri/teach/COT6936_S10.html
https://online.cis.fiu.edu/portal/course/view.php?id=427

The String Matching Problem



Approximate String Matching

Input: Text **T**, Pattern **P**

Questions:

Many more variants

- Does **P** occur in **T**?
- Find one/all occurrence of **P** in **T**.
- Count # of occurrences of **P** in **T**.
- Find longest substring of **P** in **T**.
- Find closest substring of **P** in **T**.
- Locate direct repeats of **P** in **T**.

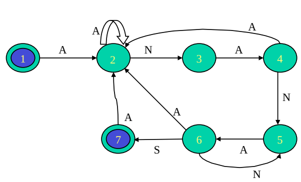
String Matching Algorithms

- Find all occurrences of P in T.
  - Naïve Method
  - Rabin-Karp Method
  - FSA-based method
  - Knuth-Morris-Pratt algorithm
  - Boyer-Moore
  - Suffix Tree method
  - Shift-And method
  - Suffix Arrays
  - Methods based on Burrows-Wheeler transform

Naïve Strategy

Finite State Automata

ANANAS



Finite
State
Automaton

ATAQAANANASPVANAGVERANANESISITALVDANANANANAS

State Transition Diagram

		A	N	S	*
-	0	1	0	0	0
A	1	1	2	0	0
AN	2	3	0	0	0
ANA	3	1	4	0	0
ANAN	4	5	0	0	0
ANANA	5	1	4	6	0
ANANAS	6	1	0	0	0

Sliding Window Strategies

Initialize window on T;

While (window within T) do

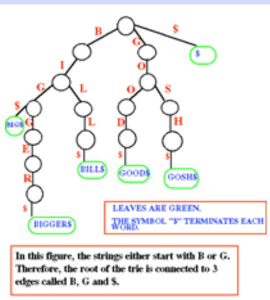
Scan: if (window = P) then report it;

Shift: shift window to right (by ?? positions)

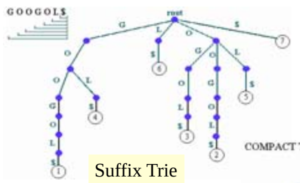
endwhile;

Tries

Storing:
BIG
BIGGER
BILL
GOOD
GOSH

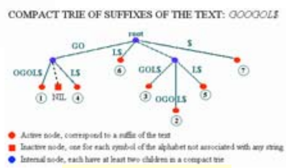


Suffix Tries & Compact Suffix Tries



Store all suffixes of GOOGOL$

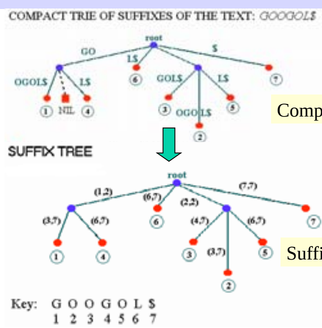
Suffix Trie



Compact Suffix Trie

2/25/10 COT 6936 10

Suffix Tries to Suffix Trees



Compact Suffix Trie

Suffix Tree

2/25/10 COT 6936 11

Suffix Trees

- Linear-time construction!
- String Matching, Substring matching, substring common to k of n strings
- All-pairs prefix-suffix problem
- Repeats & Tandem repeats
- Approximate string matching

1/20/05 CAP5510/CGS5166 (Lec 4) 12

```
123456789012
abracadabra$
12 $
11 a$
 8 abra$
 1 abracadabra$
 4 acadabra$
 6 adabra$
 9 bra$
 2 bracadabra$
 5 cadabra$
 7 dabra$
10 ra$
 3 racadabra$
```

Search abr (len = m)

- Binary Search in suffix array
- O(log n) string comparisons
- Each comparison may involve comparing O(m) characters
- O(m + logn) search
- O(n log n) space

Suffix Array & BWT Coding

```
12345678901
abracadabra
12 $
11 a$
 8 abra$
 1 abracadabra$
 4 acadabra$
 6 adabra$
 9 bra$
 2 bracadabra$
 5 cadabra$
 7 dabra$
10 ra$
 3 racadabra$
```

Space: n log n + O(n)
Search: O(m + log n)

```
12345678901
abracadabra
                L
12 $abracadabra
11 a$abracadabr
 8 abra$abracad
 1 abracadabra$
 4 acadabra$abr
 6 adabra$abrac
 9 bra$abracada
 2 bracadabra$a
 5 cadabra$abra
 7 dabra$abraca
10 ra$abracadab
 3 racadabra$ab
```

Space: n + O(A) + o(n)
Search: O(m)

- L is the BWT Code
- L is more compressible than the original input T

BWT Decoding: Step 1

Step 1: Compute F from L

```
    F          L
12 $abracadabra
11 a$abracadabr
 8 abra$abracad
 1 abracadabra$
 4 acadabra$abr
 6 adabra$abrac
 9 bra$abracada
 2 bracadabra$a
 5 cadabra$abra
 7 dabra$abraca
10 ra$abracadab
 3 racadabra$ab
```

BWT Decoding: Step 2

```
Step 2: Compute the LF Mapping, i.e., F[LF[i]] = L[i]

    F           L
12  $abracadabra
11  a$abracadabr
 8  abra$abracad
 1  abracadabra$
 4  acadabra$abr
 6  adabra$abrac
 9  bra$abracada
 2  bracadabra$a
 5  cadabra$abra
 7  dabra$abraca
10  ra$abracadab
 3  racadabra$ab
```

16

BWT Decoding: Step 3

```
Step 3: Reconstruct original input

    F           L
12  $abracadabra
11  a$abracadabr
 8  abra$abracad
 1  abracadabra$
 4  acadabra$abr
 6  adabra$abrac
 9  bra$abracada
 2  bracadabra$a
 5  cadabra$abra
 7  dabra$abraca
10  ra$abracadab
 3  racadabra$ab
```

LF mapping gives you previous character for L[i]

17

Backward Search

Backward step 1: search for "a"

i	A[i]	$T_{A[i]-1}$	suffix $T_{A[i],n}$
1:	21	a	$
2:	7	r	_a_la_alabarda$
3:	12	a	_alabarda$
4:	9	a	_la_alabarda$
5:	20	d	a$
6:	11	l	a_alabarda$
7:	8	_	a_la_alabarda$
8:	3	l	abar_a_la_alabarda$
9:	15	l	abarda$
10:	1	$	alabar_a_la_alabarda$
11:	13	_	alabarda$
12:	5	b	ar_a_la_alabarda$
13:	17	b	arda$
14:	4	a	bar_a_la_alabarda$
15:	16	a	barda$
16:	19	r	da$
17:	10	_	la_alabarda$
18:	2	a	labar_a_la_alabarda$
19:	14	a	labarda$
20:	6	a	r_a_la_alabarda$
21:	18	a	rda$

Backward step 2: search for "la"

i	A[i]	$T_{A[i]-1}$	suffix $T_{A[i],n}$
1:	21	a	$
2:	7	r	_a_la_alabarda$
3:	12	a	_alabarda$
4:	9	a	_la_alabarda$
5:	20	d	a$
6:	11	l	a_alabarda$
7:	8	_	a_la_alabarda$
8:	3	l	abar_a_la_alabarda$
9:	15	l	abarda$
10:	1	$	alabar_a_la_alabarda$
11:	13	_	alabarda$
12:	5	b	ar_a_la_alabarda$
13:	17	b	arda$
14:	4	a	bar_a_la_alabarda$
15:	16	a	barda$
16:	19	r	da$
17:	10	_	la_alabarda$
18:	2	a	labar_a_la_alabarda$
19:	14	a	labarda$
20:	6	a	r_a_la_alabarda$
21:	18	a	rda$

Backward step 3: search for "ala"

i	A[i]	suffix $T_{A[i],n}$
1:	21	$
2:	7	_a_la_alabarda$
3:	12	_alabarda$
4:	9	_la_alabarda$
5:	20	a$
6:	11	a_alabarda$
7:	8	a_la_alabarda$
8:	3	abar_a_la_alabarda$
9:	15	abarda$
10:	1	alabar_a_la_alabarda$
11:	13	alabarda$
12:	5	ar_a_la_alabarda$
13:	17	arda$
14:	4	bar_a_la_alabarda$
15:	16	barda$
16:	19	da$
17:	10	la_alabarda$
18:	2	labar_a_la_alabarda$
19:	14	labarda$
20:	6	r_a_la_alabarda$
21:	18	rda$

Fig. 4. Backward search for pattern "ala" on the suffix array of the text "alabar a la alabarda$". Here A is drawn vertically and the suffixes are shown explicitly, compare to Fig. 3.

Backward Search

```
Algorithm count(P[1,p])
i ← p, c ← P[p], First ← 1, Last ← n;
while ((First ≤ Last) and (i ≥ 1)) do
    c ← P[i];
    First ← C[c] + Occ(c, First − 1) + 1;
    Last ← C[c] + Occ(c, Last);
    i ← i − 1;
if (Last < First) then return "no rows prefixed by P[1,p]" else return (First, Last).
```

Fig. 3. Algorithm count for finding the set of rows prefixed by $P[1,p]$, and thus for counting the pattern occurrences $occ = \text{Last} - \text{First} + 1$. Recall that $C[c]$ is the number of text characters which are alphabetically smaller than c, and that $\text{Occ}(c,q)$ denotes the number of occurrences of character c in $T^{bwt}[1,q]$.

Backward Search

Need 2 mappings:

- C: $\sum \rightarrow [1,n]$ EASY
- Occ: $\sum \times [1,n] \rightarrow [1,n]$ NON-TRIVIAL

Rank Query

Need to answer rank queries:

Occ(c, L, i): # of occurrences of c in L[1..i]

- Compute a bit sequence B_c
  - 1, if corresponding character is c
  - 0, otherwise
- Computing Occ is reduced to computing number of 1's in B_c until position i

Rank Query

Need to answer rank queries:

Occ(1, B, i) in O(1) time with the minimal amount of space.

This is sufficient because:

- Occ(1, B, i) = i - Occ(0, B, i)
- Can be easily generalized to non-binary case using Wavelet Trees

Wavelet Trees

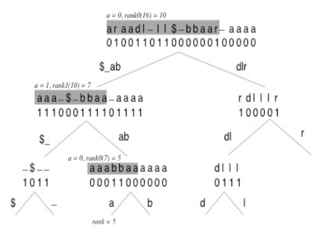


Fig. 5. A binary wavelet tree for the string $L =$ "araadl_ll$_bbaar_aaaa", illustrating the solution of query Occ("a", 15). Only the bit vectors are stored, the texts are shown for clarity.

Huffman-based Wavelet Trees

- Binary tree in the wavelet tree can be replaced with a Huffman-based tree, i.e., more frequent characters are placed at shallower leaves.

Rank query: binary sequences

Use a 2-level dictionary

- Chop into short sequences (of length k = (log n)/2) and store solutions of ranks at regular intervals
- Store global table with answers for every possible short sequence. This table is of size sqrt(n) $\times$ k