String Match Algorithms

Anna Budkina

3 мая 2023 г.

Glossary

- Alphabet $\{a_1, a_2, \ldots, a_A\}$
- String S: $s_1 s_2 \ldots s_k$
- Substring $S[i : j]$: $s_i s_{i+1} \ldots s_{j-1}$
- Prefix $S[: j]$: $s_1 s_2 \ldots s_j$
- Suffix $S[j :]$ $s_j s_{j+1} \ldots s_k$

Pattern Matching Task

Input:

- String T (target): $t_1 t_2 \ldots t_n$
- String P (pattern): $p_1 p_2 \ldots p_m$, $m \leq n$

Output:

- Substring $T[i : j]$: $T[i, j] = P$.

Pattern Matching Algorithms: Simple Approaches

- Naive Algorithm
- Boyer-Moore Algorithm
- Knuth-Morris-Prath (prefix function)
- Aho-Corasick algorithm
- Rabin-Karp algorithm (hashing)

Naive Algorithm

Idea

Try to check every substring for matching!

T (target): *acbcdcbaabcacb*, P (pattern): *bcacb*

T:	a	c	b	c	d	c	b	a	a	b	c	a	c	b
P:	b	c	a	c	b									

Naive Algorithm

Idea

Try to check every substring for matching!

T (target): *acbcdcbaabcacb*, P (pattern): *bcacb*

T:	a	c	b	c	d	c	b	a	a	b	c	a	c	b
P:		b	c	a	c	b								

Naive Algorithm

Idea

Try to check every substring for matching!

T (target): *acbcdcbaabcacb*, P (pattern): *bcacb*

T:	a	c	b	c	d	c	b	a	a	b	c	a	c	b
P:			b	c	a	c	b							

Naive Algorithm

Idea

Try to check every substring for matching!

T (target): *acbcdcbaabcacb*, P (pattern): *bcacb*

T:	a	c	b	c	d	c	b	a	a	b	c	a	c	b
P:			b	c	a	c	b							

Complexity: $O(T \cdot P)$

Boyer-Moore Algorithm

Idea

Learn from character comparisons to skip pointless alignments

T:	a	c	b	c	d	c	b	c	a	b	c	a	c	b
P:			b	c	a	c	b							

Alignments - left-to-right order

Comparisons - right-to-left order

Boyer-Moore Algorithm: Bad character rule

Upon mismatch, skip alignments until

a. Mismatch becomes a match

T:	a	c	b	c	d	c	b	a	a	b	c	a	c	b
P:			d	c	a	c	b							

Boyer-Moore Algorithm: Bad character rule

Upon mismatch, skip alignments until

a. Mismatch becomes a match

T:	a	c	b	c	**d**	c	b	a	a	b	c	a	c	b
P:					**d**	c	a	c	b					

Boyer-Moore Algorithm: Bad character rule

Upon mismatch, skip alignments until

b. P moves past mismatched character

T:	a	c	b	c	d	c	b	a	a	b	c	a	c	b
P:			b	c	a	c	b							

Boyer-Moore Algorithm: Bad character rule

Upon mismatch, skip alignments until

b. P moves past mismatched character

T:	a	c	b	c	d	c	b	a	a	b	c	a	c	b
P:						b	c	a	c	b				

Boyer-Moore Algorithm: Good suffix rule

t - substring matched by the inner loop. Skip alignments until

a. There are no mismatches between P and t

T:	a	c	b	c	d	**c**	**b**	a	a	b	c	a	c	b
P:			**c**	**b**	a	c	b							

Boyer-Moore Algorithm: Good suffix rule

t - substring matched by the inner loop. Skip alignments until

a.There are no mismatches between P and t

T:	a	c	b	c	d	c	b	a	a	b	c	a	c	b
P:						c	b	a	c	b				

Boyer-Moore Algorithm: Good suffix rule

t - substring matched by the inner loop. Skip alignments until

b. P moves past t

T:	a	c	b	c	d	c	b	a	a	b	c	a	c	b
P:			b	c	a	c	b							

Boyer-Moore Algorithm: Good suffix rule

t - substring matched by the inner loop. Skip alignments until
b. P moves past t

T:	a	c	b	c	d	c	b	a	a	b	c	a	c	b
P:							b	c	a	c	b			

Boyer-Moore Algorithm

We can precalculate the number of alignments we can skip:

P:	b	b	a	d	c
a	1	2	-	1	2
b	-	-	1	2	3
c	1	2	3	4	-
d	1	2	3	-	1

Complexity (worst-case): $O(T \cdot P)$
Complexity (best-case): $O(T/P)$

Knuth-Morris-Pratt Algorithm

Definition

Prefix function for string S: max length n ($n < |S|$) of prefix $S[: n]$ that equals to suffix $S[-n :]$.

Example: what is prefix function for *abracadabra*?

$PrefixFunction(\underline{abra}cad\underline{abra}) = 4$: *abra* is the prefix and suffix

Knuth-Morris-Pratt Algorithm

We can calculate PrefixFunction for all prefixes:

Letter	a	b	r	a	c	a	d	a	b	r	a
PrefixFunction	1	0	0	1	0	1	0	1	2	3	4

Knuth-Morris-Pratt Algorithm: PrefixFunction calculation

abacab $\rightarrow$ abacaba

$PrefixFunction(\underline{ab}ac\underline{ab}) = 2$

How to calculate $PrefixFunction(abacaba)$?

$PrefixFunction(\underline{aba}c\underline{aba}) = PrefixFunction(abacab) + 1$!

Knuth-Morris-Pratt Algorithm: PrefixFunction calculation

acaacbacaac $\rightarrow$ $acaacbacaac$**a**

$PrefixFunction(\underline{acaac}b\underline{acaac}) = 5.$

$S[6] = b \neq a = S[12]$

But: $PrefixFunction(\underline{aca}acbaca\underline{aca}) = 3 = PrefixFunction(\underline{ac}a\underline{ac}) + 1$

Idea

We can recalculate PrefixFunction recursively from prefixes!

Knuth-Morris-Pratt Algorithm

- Precalculate prefix function for prefixes of P.
- Calculate $PrefixFunction(P\$T)$ ($ is separator) iteratively for each character
- Find prefix U: $PrefixFunction(U) = Length(P)$

Example: $P = banana$, $T = ana$

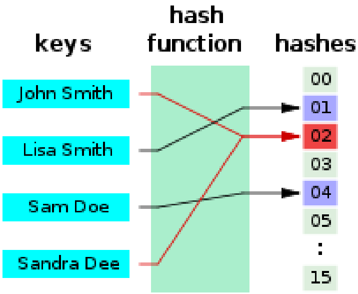
Letter	a	n	a	$	b	a	n	a	n	a
PrefixFunction	0	0	1	0	0	1	2	3	2	3

Complexity - $O(T + P)$.

Rabin-Karp algorithm: hashing

Idea

Translate string (k-mer) to hash!



https://en.wikipedia.org/wiki/Hash_function

Rabin-Karp algorithm: glossary

- Alphabet A: $\{a_1, \dots a_A\}$.
- Number presentation: $\{\alpha_1, \dots, \alpha_A\}$.
- String T: $s_1 \dots s_k$, its number presentation: $\sigma_1 \dots \sigma_k$
- String polynom $N(S) = (\sigma_1 A^{k-1} + \sigma_2 A^{k-2} + \dots + \sigma_k)$
- String hash $H(i) = N(i) \mod M$, where A and M are coprime.

Rabin-Karp algorithm: pattern search

$$T : \ldots \overbrace{\sigma_i \, \sigma_{i+1} \ldots \sigma_{i+m}}^{H(T[i:i+m])} \! \underbrace{\sigma_{i+1} \ldots \sigma_{i+m} \sigma_{i+m+1}}_{H(T[i+1:i+m+1])} \ldots$$

Searching pattern P in T, $length(P) = m$

Pattern is found when $H(T[i : i + m]) = H(P)$

Rabin-Karp algorithm: pattern search

$$T : \ldots \overbrace{\sigma_i \, \underbrace{\sigma_{i+1} \ldots \sigma_{i+m}}^{H(T[i:i+m])} \sigma_{i+m+1}}_{H(T[i+1:i+m+1])} \ldots$$

Searching pattern P in T, $length(P) = m$
Pattern is found when $H(T[i : i + m]) = H(P)$

- Complexity: $O(T + P)$ - best case, $O(T + P + kP)$ for k matches.

Rabin-Karp algorithm: pattern search

$$T : \ldots \overbrace{\sigma_i \, \underbrace{\sigma_{i+1} \ldots \sigma_{i+m}}^{H(T[i:i+m])}\sigma_{i+m+1}}_{H(T[i+1:i+m+1])} \ldots$$

Searching pattern P in T, $length(P) = m$
Pattern is found when $H(T[i : i + m]) = H(P)$

- Complexity: $O(T + P)$ - best case, $O(T + P + kP)$ for k matches.
- Beware of hash collisions: for $T[i : i + m] \neq T[j : j + m]$ $H(T[i : i + m]) = H(T[j : j + m])$.

Rabin-Karp algorithm: hashing

$$T : \dots \mathrlap{\overbrace{\phantom{\sigma_i\,\sigma_{i+1}\dots\sigma_{i+m}}}^{N(T[i:i+m])}}\sigma_i\,\underbrace{\sigma_{i+1}\dots\sigma_{i+m}\sigma_{i+m+1}}_{\mathbf{N(T[i+1:i+m+1])}]}\dots$$

$$N(T[i:i+m]) = (\sigma_i \cdot A^m + N(T[i+1:i+m]))$$

$$N(T[i+1:i+m]) = N(T[i:i+m]) - \sigma_i \cdot A^m$$

$$\mathbf{N(T[i+1:i+m+1])} = A \cdot N(T[i+1:i+m]) + \sigma_{i+m+1}$$

$$\mathbf{N(T[i+1:i+m+1])} = A \cdot (N(T[i:i+m]) - \sigma_i \cdot A^m) + \sigma_{i+m+1}$$

Rabin-Karp algorithm: example

Alphabet: a, b, c, d
Number presentation: $0, 1, 2, 3$
Pattern P: *adcca*
$H(P) = (\sigma_0 A^4 + \sigma_1 A^3 + \sigma_2 A^2 + \sigma_3 A^1 + \sigma_4) = 0 \cdot 4^4 + 3 \cdot 4^3 + 2 \cdot 4^2 + 2 \cdot 4^1 + 0 = 232$

Rabin-Karp algorithm: example

$H(P) = 232$

T:	b	c	a	d	c	c	a	a	a	b	c	a	c	b
P:	a	d	c	c	a									

$$T(0) = T(\mathbf{bcadc}) = 1 \cdot 4^4 + 2 \cdot 4^3 + 0 \cdot 4^2 + 3 \cdot 4^1 + 2 = 398 \neq 232$$

Rabin-Karp algorithm: example

$H(P) = 232$

T:	b	c	a	d	c	c	a	a	a	b	c	a	c	b
P:		a	d	c	c	a								

$T(0) = T(\text{bcadc}) = 1 \cdot 4^4 + 2 \cdot 4^3 + 0 \cdot 4^2 + 3 \cdot 4^1 + 2 = 398 \neq 232$

$T(1) = T(\text{cadcc}) = (T(0) - 1 \cdot 4^4) \cdot 4 + 2 = (398 - 1 \cdot 4^4) \cdot 4 + 2 = 570 \neq 232$

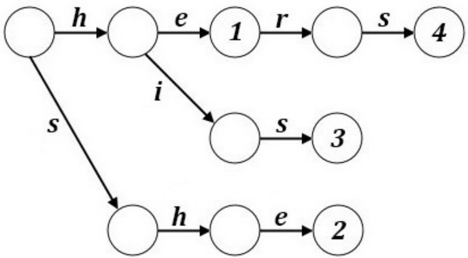
Rabin-Karp algorithm: example

$H(P) = 232$

T:	b	c	a	d	c	c	a	a	a	b	c	a	c	b
P:			a	d	c	c	a							

$$T(0) = T(\textbf{bcadc}) = 1\cdot 4^4 + 2\cdot 4^3 + 0\cdot 4^2 + 3\cdot 4^1 + 2 = 398 \neq 232$$
$$T(1) = T(\textbf{cadcc}) = (T(0) - 1\cdot 4^4)\cdot 4 + 2 = (398 - 1\cdot 4^4)\cdot 4 + 2 = 570 \neq 232$$
$$T(2) = T(\textbf{adcca}) = (T(1) - 2\cdot 4^4)\cdot 4 + 0 = (570 - 2\cdot 4^4)\cdot 4 + 0 = 232 = 232$$

Trie



https://neerc.ifmo.ru/wiki/index.php?title=Алгоритм_

Words: he, she, his, hers

Adding a word, searching a word - $O(n)$

Aho-Corasick algorithm

Input:

- T - target
- m pattern strings: $P_1, \ldots P_m$

Output:

- All occurrences of $P_1, \ldots P_m$ in T.

Aho-Corasick algorithm

Input:

- T - target
- m pattern strings: $P_1, \dots P_m$

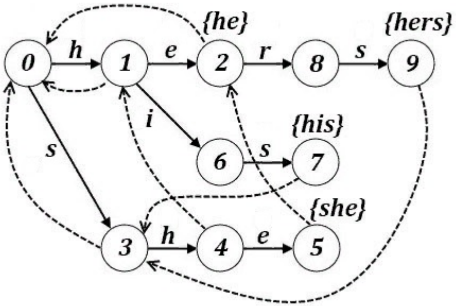
Output:

- All occurrences of $P_1, \dots P_m$ in T.

Idea

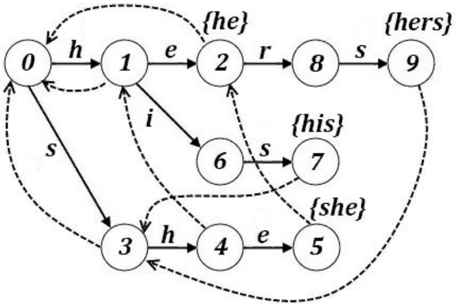
Build trie from $P_1, \dots P_m$ with suffix links.
Extend *PrefixFunction* via links in tree.

Aho-Corasick algorithm: Suffix links



Adding **suffix links** to the trie:
Each node is assigned a pointer to the string that is the longest string suffix at that node.

Aho-Corasick algorithm



The main idea is to immediately display all patterns that end in the processed text character. All such patterns are suffixes of each other.

T:	s	h	e	r	s

Path for search :

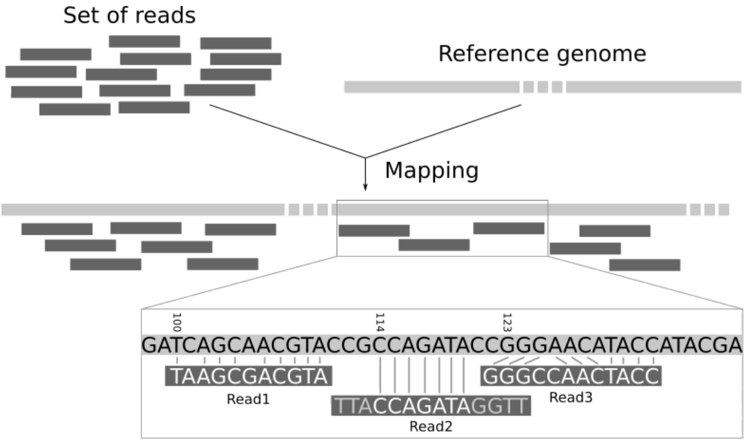
$\rightarrow s \rightarrow h \rightarrow e \rightarrow suffixlink \rightarrow r \rightarrow s$

Aho-Corasick algorithm Complexity

- $Length(T) = m$
- $\sum(Length(P_i)) = n$
- z - number of occurrences of patterns

Complexity: $O(n + m + z)$

Reads alignment



https://training.galaxyproject.org/training-material/topics/sequence-analysis/tutorials/mapping/tutorial.html

Pattern Matching Algorithms: Advanced Approaches

Idea

Time consuming text T preprocessing for fast pattern search

- Suffix Array
- Suffix Tree

Suffix Array

$S : ABAABA\$$ ($ - end of string)

Suffix Array

$S : ABAABA\$$ ($ - end of string)

A	B	A	A	B	A	$
0	1	2	3	4	5	6

Suffix Array

$S : ABAABA\$$ (\$ - end of string)

A	B	A	A	B	A	\$
0	1	2	3	4	5	6

index	suffix
0	$ABAABA\$$

Suffix Array

S : $ABAABA\$$ ($ - end of string)

A	B	A	A	B	A	$
0	1	2	3	4	5	6

index	suffix
0	$ABAABA\$$
1	$BAABA\$$

Suffix Array

S : $ABAABA\$$ (\$ - end of string)

A	B	A	A	B	A	\$
0	1	2	3	4	5	6

index	suffix
0	$ABAABA\$$
1	$BAABA\$$
2	$AABA\$$

Suffix Array

S : $ABAABA\$$ (\$ - end of string)

A	B	A	A	B	A	\$
0	1	2	3	4	5	6

index	suffix
0	$ABAABA\$$
1	$BAABA\$$
2	$AABA\$$
3	$ABA\$$

Suffix Array

S : $ABAABA\$$ (\$ - end of string)

A	B	A	A	B	A	\$
0	1	2	3	4	5	6

index	suffix
0	$ABAABA\$$
1	$BAABA\$$
2	$AABA\$$
3	$ABA\$$
4	$BA\$$

Suffix Array

$S : ABAABA\$$ (\$ - end of string)

A	B	A	A	B	A	\$
0	1	2	3	4	5	6

index	suffix
0	$ABAABA\$$
1	$BAABA\$$
2	$AABA\$$
3	$ABA\$$
4	$BA\$$
5	$A\$$

Suffix Array

$S : ABAABA\$$ (\$ - end of string)

A	B	A	A	B	A	\$
0	1	2	3	4	5	6

index	suffix
0	$ABAABA\$$
1	$BAABA\$$
2	$AABA\$$
3	$ABA\$$
4	$BA\$$
5	$A\$$
6	\$

Suffix Array

S : $ABAABA\$$ (\$ - end of string)

A	B	A	A	B	A	\$
0	1	2	3	4	5	6

index	suffix
0	$ABAABA\$$
1	$BAABA\$$
2	$AABA\$$
3	$ABA\$$
4	$BA\$$
5	$A\$$
6	$\$$

Sort →

index	suffix
6	$\$$
5	$A\$$
2	$AABA\$$
3	$ABA\$$
0	$ABAABA\$$
4	$BA\$$
1	$BAABA\$$

Suffix Array

$S : ABAABA\$$

A	B	A	A	B	A	$
0	1	2	3	4	5	6

$T = ABA$

index	suffix
6	$\$$
5	$A\$$
2	$AABA\$$
3	$ABA\$$
0	$ABAABA\$$
4	$BA\$$
1	$BAABA\$$

Suffix Array

$S : ABAABA\$$

A	B	A	A	B	A	\$
0	1	2	3	4	5	6

index	suffix
6	$\$$
5	$A\$$
2	$AABA\$$
3	$ABA\$$
0	$ABAABA\$$
4	$BA\$$
1	$BAABA\$$

$T = ABA$
Binary search for lower bound!

Suffix Array

$S : ABAABA\$$

A	B	A	A	B	A	$
0	1	2	3	4	5	6

$T = ABA$
Binary search for lower bound!

index	suffix
6	$\$$
5	$A\$$
2	$AABA\$$
3	$ABA\$$
0	$ABAABA\$$
4	$BA\$$
1	$BAABA\$$

Suffix Array

$S : ABAABA\$$

A	B	A	A	B	A	$
0	1	2	3	4	5	6

index	suffix
6	$\$$
5	$A\$$
2	$AABA\$$
3	$ABA\$$
0	$ABAABA\$$
4	$BA\$$
1	$BAABA\$$

$T = ABA$

Binary search for lower bound!

Suffix Array

$S : ABAABA\$$

A	B	A	A	B	A	$
0	1	2	3	4	5	6

$T = ABA$
Binary search for lower bound!

index	suffix
6	$\$$
5	$A\$$
2	$AABA\$$
3	$ABA\$$
0	$ABAABA\$$
4	$BA\$$
1	$BAABA\$$

Suffix Array

$S : ABAABA\$$

A	B	A	A	B	A	\$
0	1	2	3	4	5	6

index	suffix
6	$\$$
5	$A\$$
2	$AABA\$$
3	$ABA\$$
0	$ABAABA\$$
4	$BA\$$
1	$BAABA\$$

$T = ABA$

Binary search for lower bound!

Suffix Array

$S : ABAABA\$$

A	B	A	A	B	A	$
0	1	2	3	4	5	6

$T = ABA$

Binary search for lower bound!

index	suffix
6	$\$$
5	$A\$$
2	$AABA\$$
3	$ABA\$$
0	$ABAABA\$$
4	$BA\$$
1	$BAABA\$$

Suffix Array

$S : ABAABA\$$

A	B	A	A	B	A	\$
0	1	2	3	4	5	6

index	suffix
6	$\$$
5	$A\$$
2	$AABA\$$
3	$ABA\$$
0	$ABAABA\$$
4	$BA\$$
1	$BAABA\$$

$T = ABA$
Binary search for lower bound!

Suffix Array: upper bound

$S : ABAABA\$$

A	B	A	A	B	A	\$
0	1	2	3	4	5	6

$T = ABA$

Binary search for **upper** bound!

Complexity: $O(P\dot{l}ogT)$

index	suffix
6	$\$$
5	$A\$$
2	$AABA\$$
3	$ABA\$$
0	$ABAABA\$$
4	$BA\$$
1	$BAABA\$$

Suffix Array: upper bound

$S : ABAABA\$$

A	B	A	A	B	A	\$
0	1	2	3	4	5	6

$T = ABA$
Binary search for **upper** bound!
Complexity: $O(P\dot{l}ogT)$

index	suffix
6	$\$$
5	$A\$$
2	$AABA\$$
3	$ABA\$$
0	$ABAABA\$$
4	$BA\$$
1	$BAABA\$$

Suffix Array: upper bound

$S : ABAABA\$$

A	B	A	A	B	A	$
0	1	2	3	4	5	6

index	suffix
6	$\$$
5	$A\$$
2	$AABA\$$
3	$ABA\$$
0	$ABAABA\$$
4	$BA\$$
1	$BAABA\$$

$T = ABA$

Binary search for **upper** bound!

Complexity: $O(P\dot{l}ogT)$

Suffix Array: mlr optimization

Searching for P: *bcacedc*

	a	v	d	v	a	c	d
l	b	c	a	a	d	v	w
	b	c	a	b	e	g	a
m	b	c	a	b	d	j	k
	b	c	a	c	a	b	c
r	b	c	a	c	e	d	c
	c	d	e	f	a	b	c

Best case complexity: $O(P + logT)$

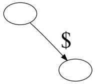
Suffix Tree

index	suffix
6	$\$$
5	$A\$$
2	$AABA\$$
3	$ABA\$$
0	$ABAABA\$$
4	$BA\$$
1	$BAABA\$$

Suffix Tree

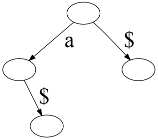
index	suffix
6	$\$$
5	$A\$$
2	$AABA\$$
3	$ABA\$$
0	$ABAABA\$$
4	$BA\$$
1	$BAABA\$$

Suffix Tree



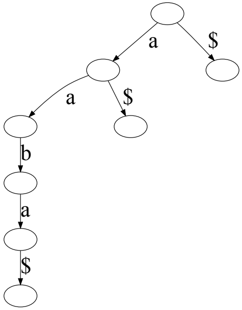
index	suffix
6	$
5	*A*$
2	*AABA*$
3	*ABA*$
0	*ABAABA*$
4	*BA*$
1	*BAABA*$

Suffix Tree



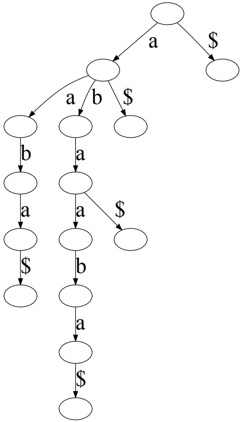
index	suffix
6	$
5	$A$$
2	$AABA$$
3	$ABA$$
0	$ABAABA$$
4	$BA$$
1	$BAABA$$

Suffix Tree



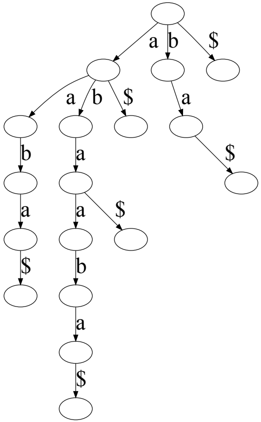
index	suffix
6	$
5	*A$*
2	*AABA$*
3	*ABA$*
0	*ABAABA$*
4	*BA$*
1	*BAABA$*

Suffix Tree



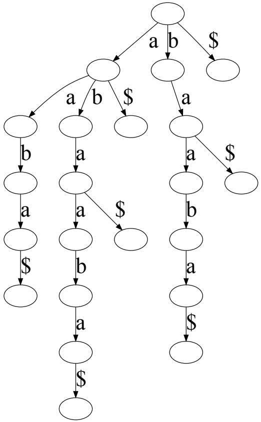
index	suffix
6	$
5	*A$*
2	*AABA$*
3	*ABA$*
0	*ABAABA$*
4	*BA$*
1	*BAABA$*

Suffix Tree



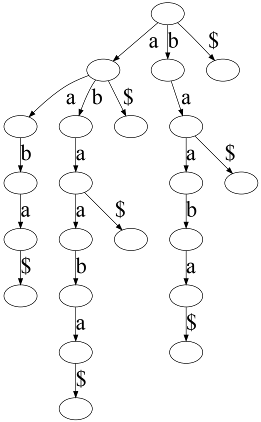
index	suffix
6	$
5	*A*$
2	*AABA*$
3	*ABA*$
0	*ABAABA*$
4	*BA*$
1	*BAABA*$

Suffix Tree



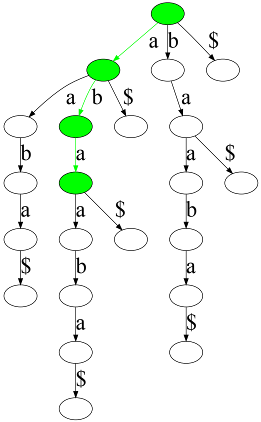
index	suffix
6	$
5	*A$*
2	*AABA$*
3	*ABA$*
0	*ABAABA$*
4	*BA$*
1	*BAABA$*

Suffix Tree: finding pattern



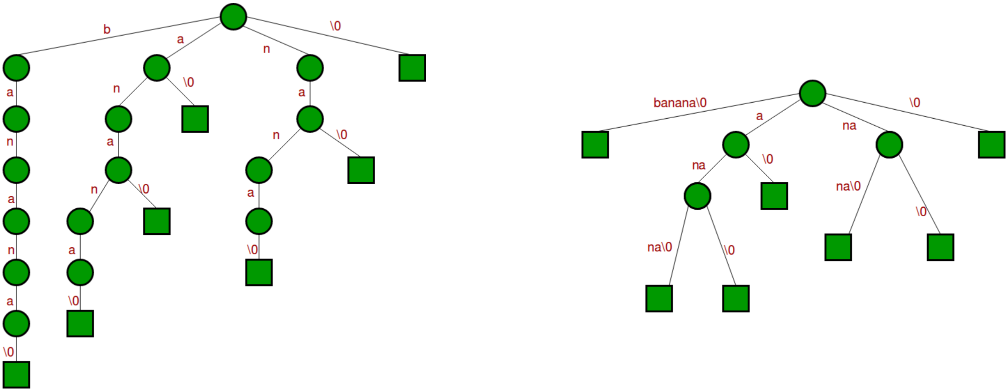
$$P = ABA$$

Suffix Tree: finding pattern



$$P = ABA$$

Suffix Tree: implicit suffix tree



Ukkonen's algorithm for building an implicit suffix tree

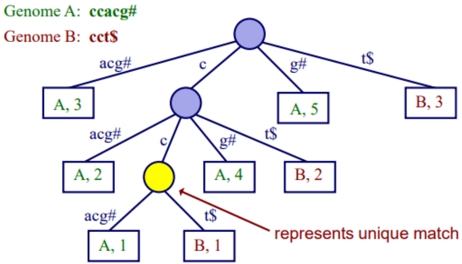
Complexity : O(T)

https://www.geeksforgeeks.org/pattern-searching-using-suffix-tree/

Suffix Tree based algorithms

MUMmer

- Find Maximal Union Matching building united suffix tree from two genomes



https://www.biostat.wisc.edu/bmi776/spring-18/lectures/long-alignment.pdf

Suffix Tree based algorithms

REPuter

- Build Suffix Tree for finding maximal repeated pairs in genome using Gusfield algorithm