

KMP e Suffix Trie/Tree/Array

Laboratório de Programação Competitiva - 2020

Pedro Henrique Paiola

Busca em strings

- O problema substring search ou pattern search consiste em encontrar uma dada string dentro de outra.
- Exemplo:

S = “Que a Força esteja com você”

P = “Força”

A string P pode ser encontrada dentro de S, a partir da posição 6

Busca em strings

- Algoritmo ingênuo

```
int search(string S, string P) {  
    for(int i = 0; i <= S.size() - P.size(); i++) {  
        for(int j = 0; j < P.size(); j++)  
            if (S[i+j] != P[j])  
                break;  
        if (j == P.size())  
            return i;  
    }  
    return -1;  
}
```

Busca em strings

- Esse algoritmo, no pior caso, tem complexidade $O(m.n)$, fazendo $m.n$ comparações. Porém, em geral, ele não chega a realizar tantas comparações.
- Usar esse algoritmo é bastante razoável para vários casos, principalmente quando as strings não são muito grandes.
- Mas, existem algoritmos de busca de substrings mais eficientes, que podem ser necessários em alguns casos, como exemplo temos o KMP.

KMP

- Knuth Morrit Pratt
- Complexidade: $O(n)$ no pior caso
- No algoritmo ingênuo, sempre que detectamos caracteres diferentes, avançávamos um caracter na string principal ($i++$) e testamos toda a substring, desde o começo (começando sempre com $j = 0$).
- O KMP, porém, aproveita as comparações que foram feitas antes de encontrar dois caracteres diferentes, evitando comparar novamente caracteres que já sabemos que são compatíveis.

KMP

- A principal ideia deste algoritmo é pré-processar o padrão P, de modo a obter um vetor de inteiros *lps*, que conta o número de caracteres que podem ser “ignorados” em uma nova comparação.
- O nome *lps* refere-se à “*longest proper prefix and suffix*”, ou seja, o maior prefixo próprio (não pode ser a própria palavra) que também é sufixo.

KMP

P = "ABABAC"

lps = {}

P = "ABABAC"

lps = {0}

P = "ABABAC"

lps = {0, 0}

P = "ABABAC"

lps = {0, 0, 1}

P = "ABABAC"

lps = {0, 0, 1, 2}

P = "ABABAC"

lps = {0, 0, 1, 2, 3}

P = "ABABAC"

lps = {0, 0, 1, 2, 3, 0}

KMP

- E como isto ajuda? Isso permite pular comparações desnecessárias, por exemplo:
- Pelo algoritmo ingênuo:

S = ABABABCABABABCABABABC

P = ABABAC

KMP

- E como isto ajuda? Isso permite pular comparações desnecessárias, por exemplo:
- Pelo algoritmo ingênuo:

S = ABABAB CABABAB CABABAB CAB

P = ABABAC

KMP

- E como isto ajuda? Isso permite pular comparações desnecessárias, por exemplo:
- Pelo algoritmo ingênuo:

S = ABABABCABABABCABABABC

P = ABABAC

KMP

- E como isto ajuda? Isso permite pular comparações desnecessárias, por exemplo:
- Pelo algoritmo ingênuo:

S = ABABABCABABABCABABABC

P = ABABAC

KMP

- E como isto ajuda? Isso permite pular comparações desnecessárias, por exemplo:
- Pelo algoritmo ingênuo:

S = ABABABCABABABCABABABC

P = ABABAC

KMP

- E como isto ajuda? Isso permite pular comparações desnecessárias, por exemplo:
- Pelo algoritmo ingênuo:

S = ABABABCABABABCABABABC

P = ABABAC

KMP

- E como isto ajuda? Isso permite pular comparações desnecessárias, por exemplo:
- Pelo algoritmo ingênuo:

S = ABABABCABABABCABABABC

P = ABABAC

KMP

- E como isto ajuda? Isso permite pular comparações desnecessárias, por exemplo:
- Pelo algoritmo ingênuo:

S = ABABABCABABABCABABABC

P = ABABAC

KMP

- E como isto ajuda? Isso permite pular comparações desnecessárias, por exemplo:
- Pelo KMP:

$S = \text{ABABABCABABABCABABABC}$

$P = \text{ABABAC}$

$\text{lps} = \{0, 0, 1, 2, 3, 0\}$

KMP

- E como isto ajuda? Isso permite pular comparações desnecessárias, por exemplo:
- Pelo KMP:

S = ABABAB CABABAB CABABAB CAB

P = ABABAC

lps = {0, 0, 1, 2, 3, 0}

E agora?

mantemos o valor de i (ponteiro para posição de S)

$j = \text{lps}[j - 1] = 3$

KMP

- E como isto ajuda? Isso permite pular comparações desnecessárias, por exemplo:
- Pelo KMP:

S = AB**ABA**BCABABABCBABABABC

P = **ABA**BAC

lps = {0, 0, 1, 2, 3, 0}

KMP

```
int a[MAX], n, m;
char S[MAX], P[MAX];

void calculatePrefix(){
    int i = 0, j = -1;
    a[0] = -1;
    while(i < m){
        while(j >= 0 && P[i] != P[j])
            j = a[j];
        i++; j++;
        a[i] = j;
    }
}
```

KMP

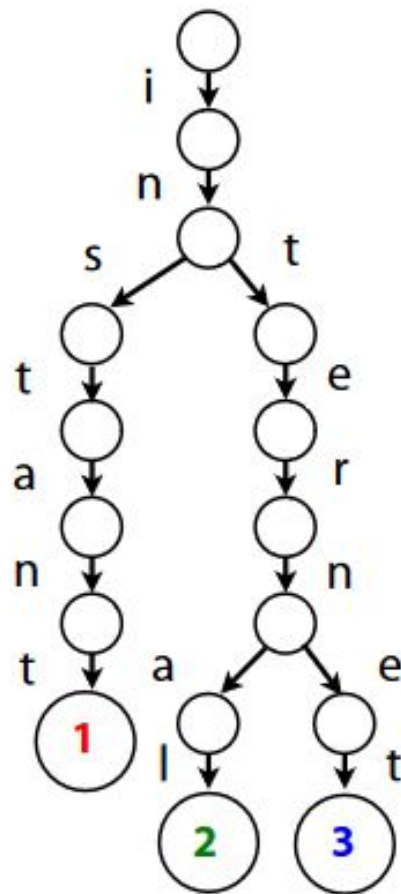
```
vector<int> KMP2(){ //retorna todas as ocorrências da substring
    vector<int> resp;
    int i = 0, j = 0;
    calculatePrefix();
    while(i < n){
        while(j >= 0 && S[i] != P[j]) j = a[j];
        i++; j++;
        if (j == m){
            resp.push_back(i - m);
            j = a[j];
        }
    }
    return resp;
}
```

Trie

- Uma **trie** é uma árvore que representa uma coleção de strings com nós únicos prefixos comuns.
- As arestas são rotuladas com letras, e um caminho da raiz até uma folha determina uma string armazenada na **trie**.
- Comumente usada para implementar um map em que as chaves são strings.

Trie

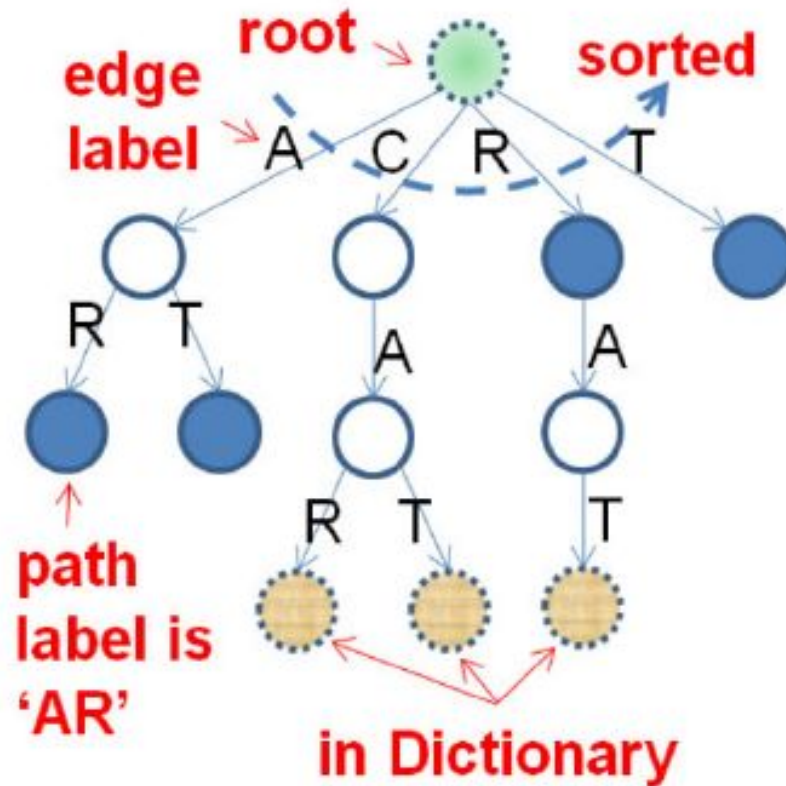
Key	Value
instant	1
internal	2
internet	3



Suffix Trie

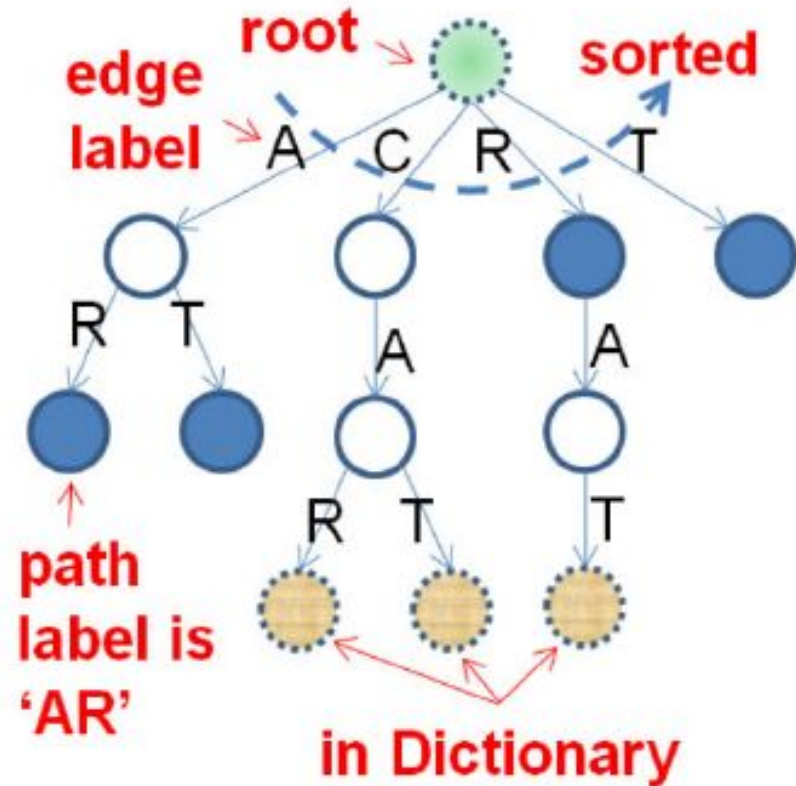
- Porém, para nós é mais interessante uma variação desta estrutura, que armazena todos os sufixos de uma palavra (ou um conjunto de palavras).
- $S = \{\text{CAR, CAT, RAT}\}$
- Sufixos de $S = \{\text{CAR, AR, R, CAT, AT, RAT}\}$
- Sufixos ordenados de $S = \{\text{AR, AT, CAR, CAT, R, RAT, T}\}$

Suffix Trie



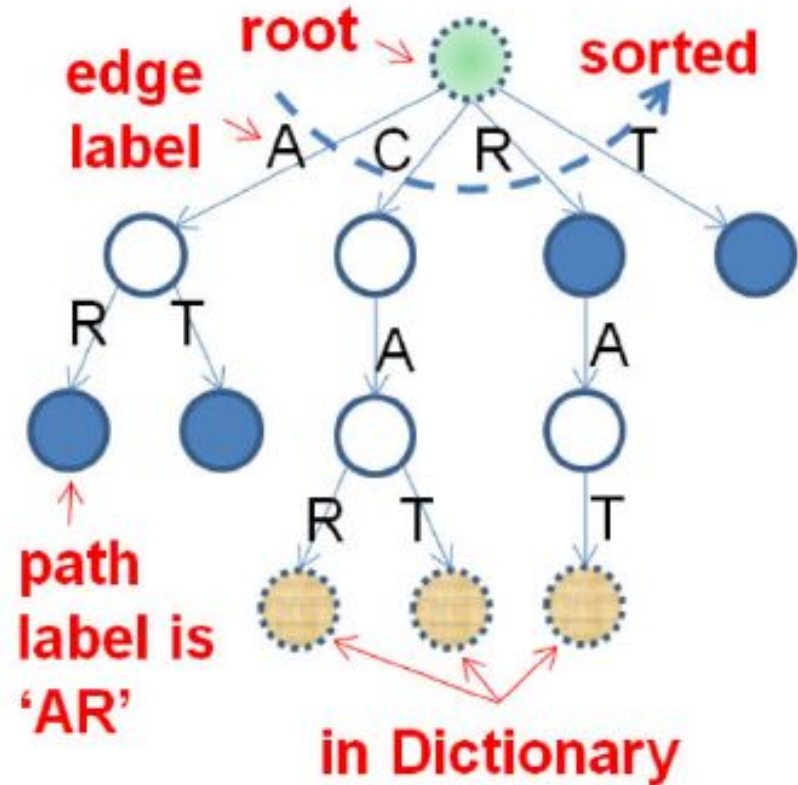
Suffix Trie

- Cada vértice possui duas flags booleanas: isSuffix e isWord
- A suffix trie é construída inicialmente contendo apenas o nó raiz. Em seguida, cada sufixo $S[i...n-1]$ é inserido na árvore caractere a caractere
- Ao inserir um sufixo na árvore, se um caractere do sufixo não estiver presente na árvore, cria-se uma nova ramificação



Suffix Trie

- A suffix trie é uma estrutura de dados eficiente para operações em dicionário.
- Uma vez que ela é construída para um conjunto de strings, é possível determinar se um padrão P está presente no dicionário em tempo $O(|P|)$



Suffix Tree

- A Suffix Trie, porém, apresenta uma certa ineficiência de memória. Para cada palavra, muitos vértices podem ser criados.
- A Suffix Tree surge como uma melhoria da Suffix Trie, aplicando uma compressão de caminhos, de forma que vértice com um único filho são contraídos.

Suffix Tree

i	Suffix
0	GATAGACA
1	ATAGACA
2	TAGACA
3	AGACA
4	GACA
5	ACA
6	CA
7	A

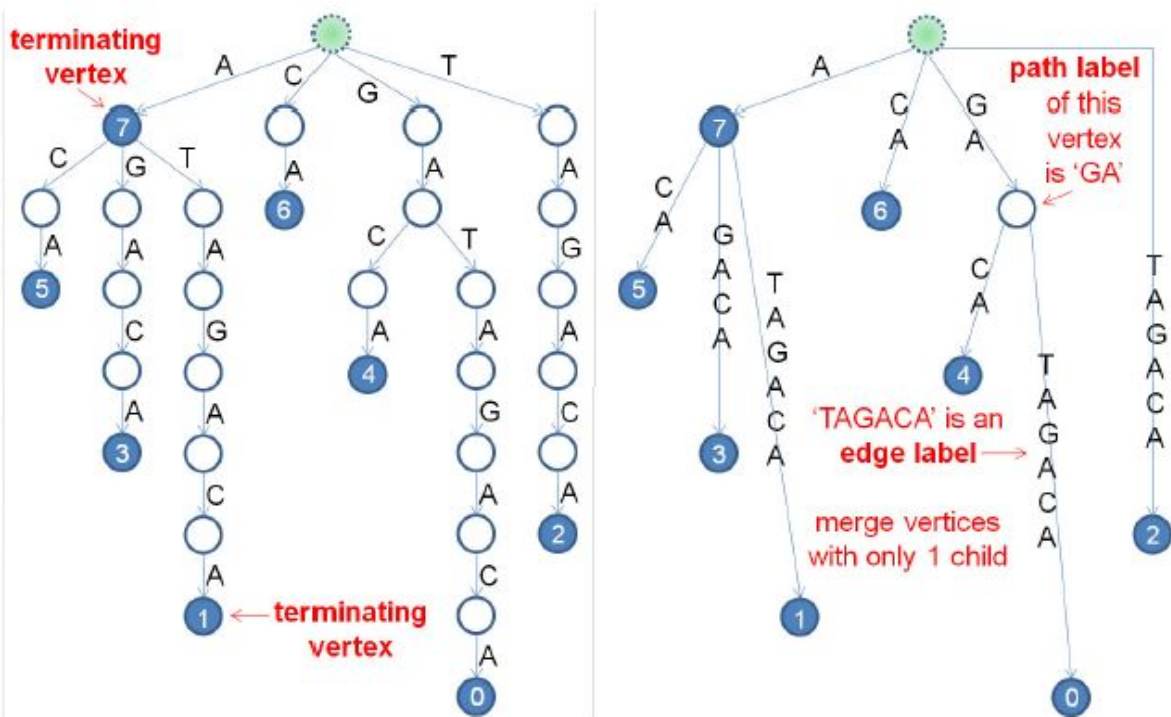


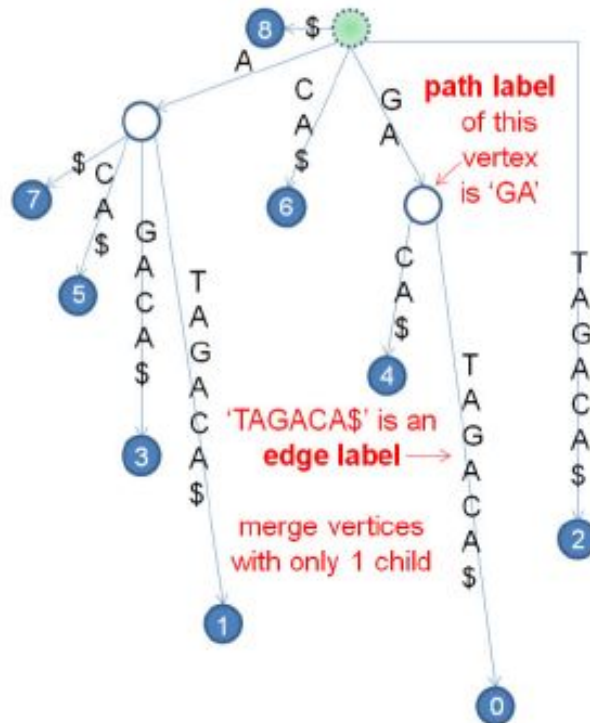
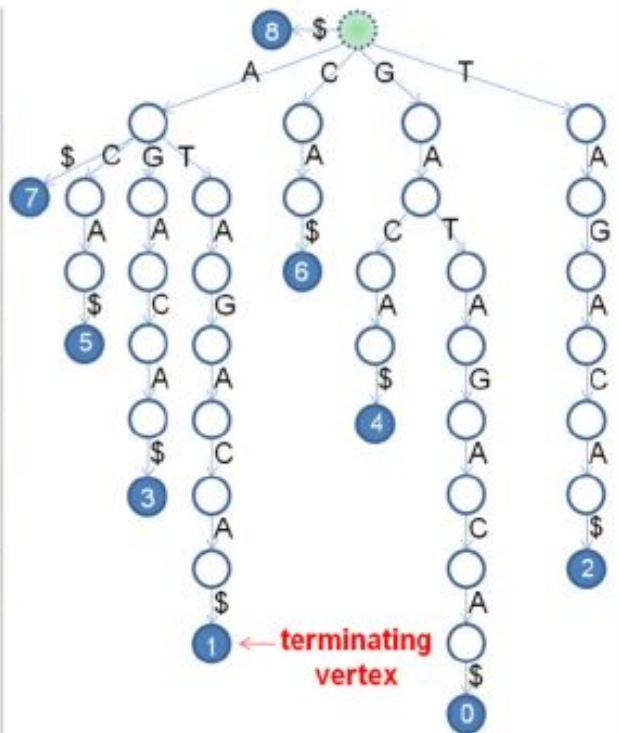
Figure 6.3: Suffixes, Suffix Trie, and Suffix Tree of $T = \text{'GATAGACA'}$

Suffix Tree

- Além disso, é comum adicionarmos o caractere especial **\$** ao final de uma string, com o objetivo de garantir que todos os sufixos sejam terminados por ele e que todos os sufixos sejam folhas na árvore.
- Isto também pode ser aplicado em uma suffix trie

Suffix Tree

i	Suffix
0	GATAGACA\$
1	ATAGACA\$
2	TAGACA\$
3	AGACA\$
4	GACA\$
5	ACA\$
6	CA\$
7	A\$
8	\$



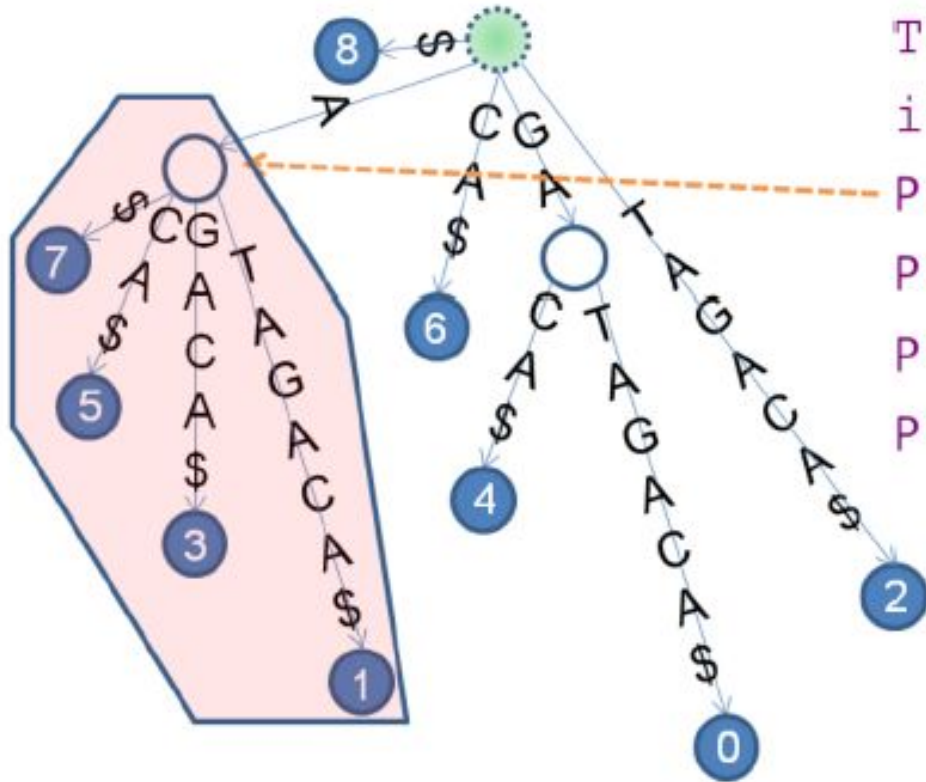
Suffix Tree

- Assumindo que a árvore de sufixos para uma string S foi construída, é possível utilizá-la para diversas aplicações, dentre elas:
 - Busca de string
 - Máxima substring repetida
 - Máxima substring comum

Suffix Tree

- Busca de string:
 - Com a árvore de sufixos é possível encontrar todas as ocorrências de um padrão P em uma string S em tempo $O(m + c)$, em que $m = |P|$ e c é o número total de ocorrências de P em S .
 - Uma vez que a árvore de sufixos já foi construída, a busca torna-se independente do tamanho da string S , o que pode ser muito mais eficiente do que o algoritmo KMP, em especial quando se quer realizar várias buscas.

Suffix Tree



T = 'GATAGACA\$'

```
i = '012345678'
```

P = 'A' → Occurrences: 7, 5, 3, 1

P = 'GA' → Occurrences: 4, 0

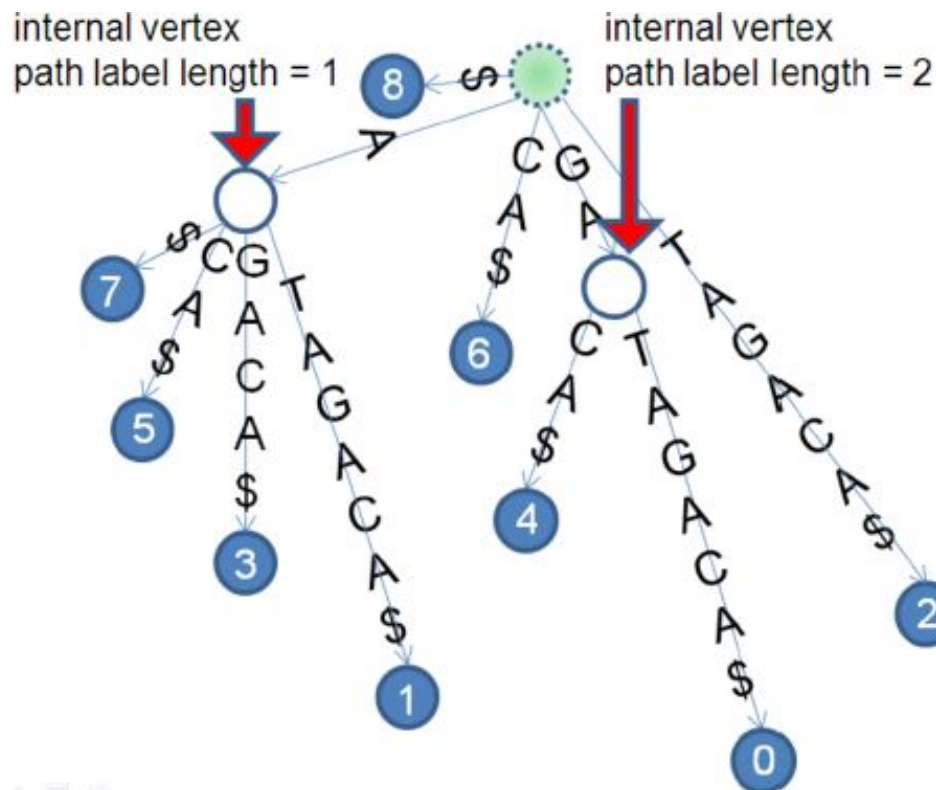
$P = 'T' \rightarrow$ Occurrences: 2

P = 'Z' → Not Found

Suffix Tree

- Máxima substring repetida:
 - Objetivo: encontrar a maior substring de S que tenha pelo menos duas ocorrências em S (LRS - Longest Repeated Substring)
 - O caminho correspondente ao **nó interno mais profundo x** da árvore é a solução do LRS
 - O fato de x ser um nó interno implica que ele representa mais de um sufixo (do contrário ele entraria na compressão de caminhos)
 - O fato de x ser o nó interno mais profundo implica que seu caminho corresponderá a uma substring de maior comprimento

Suffix Tree



e.g. $T = \text{'GATAGACA\$'}$

The longest repeated substring is 'GA' with path label length = 2

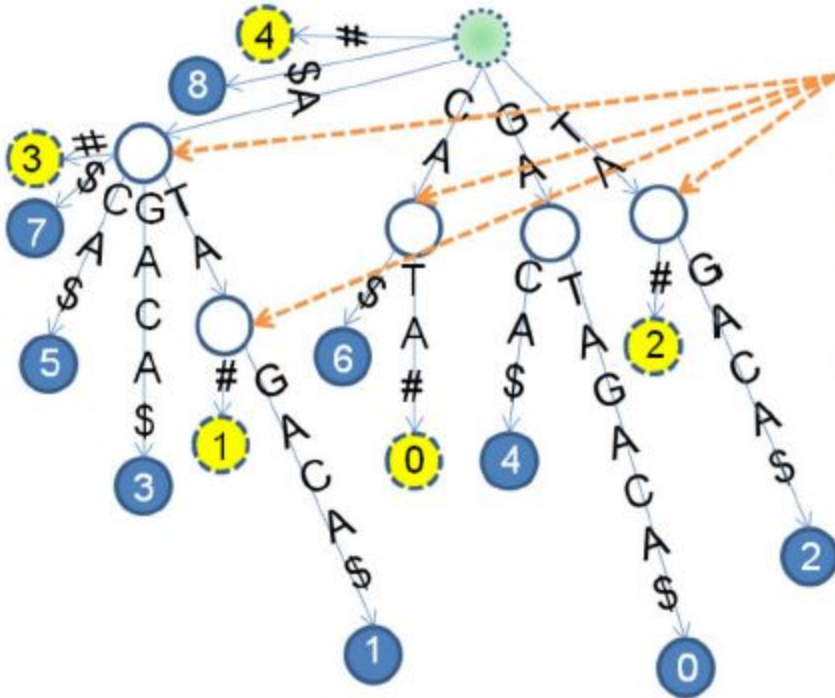
The other repeated substring is 'A', but its path label length = 1

Suffix Tree

- Máxima substring comum:
 - Objetivo: determinar a máxima substring comum (LCS - Longest Common Substring) de duas ou mais strings
 - Nesse caso, a árvore de sufixos deve ser construída para o conjunto das strings, que devem ser marcadas com caracteres especiais diferentes como terminais de strings
 - O caminho correspondente ao nó interno mais profundo que pertença a todas as strings representadas é a solução do LCS

Suffix Tree

$S_1 = \text{GATAGACA\$}$ e $S_2 = \text{CATA\#}$



These are the internal vertices representing suffixes from both strings

The deepest one has path label 'ATA'

Suffix Tree

- Existem algoritmos de tempo linear para a formação de uma árvore de sufixos, como por exemplo, o **algoritmo de Ukkonen**.
- Este algoritmo é “relativamente” complicado.

Suffix Array

- Um vetor de sufixos (suffix array) possui funcionalidade similar à árvore de sufixo, porém com uma construção mais simples e mais eficiente em termos de uso de memória.
- Porém, algumas operações que eram realizadas em uma suffix tree podem ser mais custosas em um suffix array.

Suffix Array

- Um suffix array é um vetor que armazena os sufixos de uma string ordenados lexicograficamente
 - Na prática, ele armazena os índices dos sufixos

Suffix Array

i	Suffix
0	GATAGACA\$
1	ATAGACA\$
2	TAGACA\$
3	AGACA\$
4	GACA\$
5	ACA\$
6	CA\$
7	A\$
8	\$

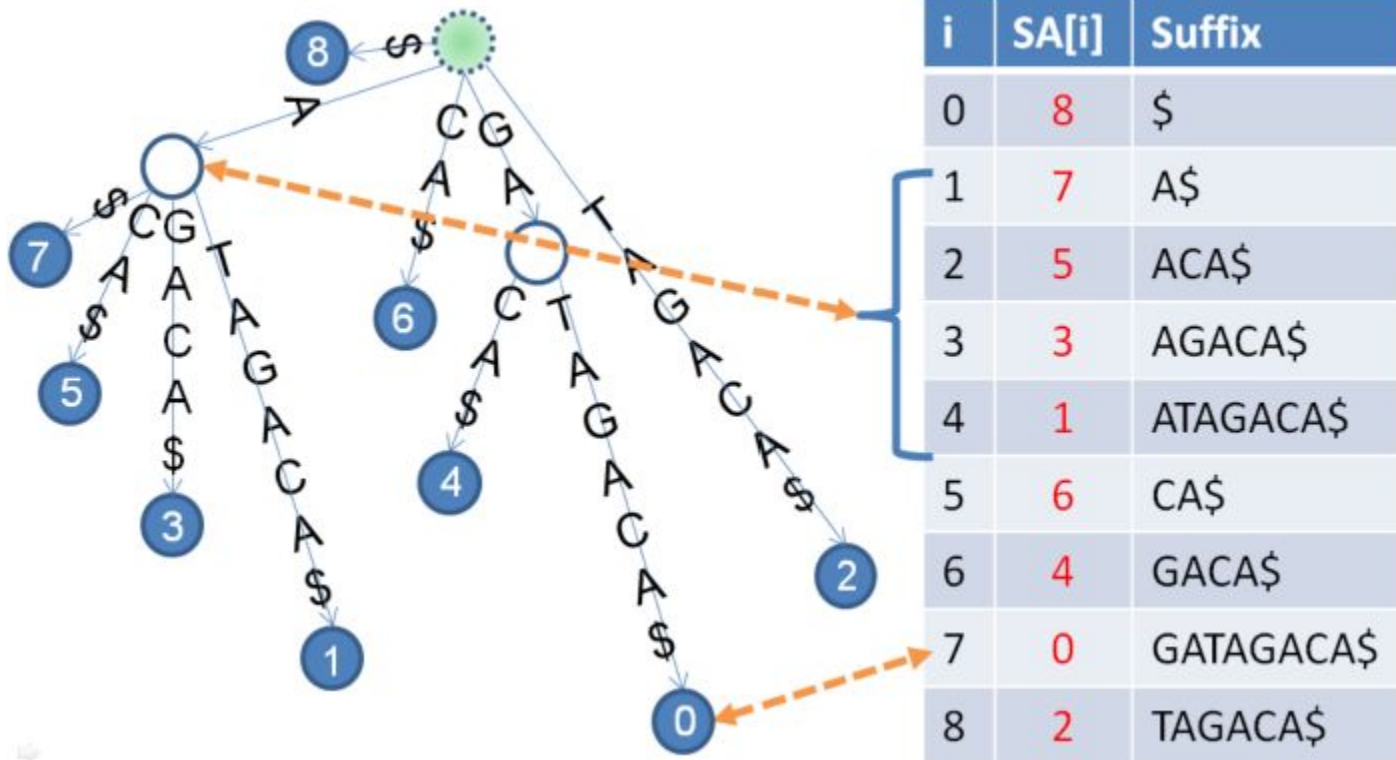
Sort →

i	SA[i]	Suffix
0	8	\$
1	7	A\$
2	5	ACA\$
3	3	AGACA\$
4	1	ATAGACA\$
5	6	CA\$
6	4	GACA\$
7	0	GATAGACA\$
8	2	TAGACA\$

Suffix Array

- Um vetor de sufixos continua fortemente atrelado a uma árvore de sufixos.
- Uma busca em profundidade na árvore de sufixos visita os nós terminais (folhas) na ordem definida pelo vetor de sufixos.
- Um **nó interno** da árvore de sufixos corresponde a um **intervalo** do vetor de sufixos (uma coleção de sufixos ordenados com um prefixo em comum).
- Um **nó folha** da árvore de sufixos corresponde a um único **elemento** do vetor de sufixos.

Suffix Array



Suffix Array

- Todas as operações que podíamos fazer em uma suffix tree, podemos fazer também utilizando um suffix array.
- Vamos analisar os exemplos de aplicações que vimos anteriormente

Suffix Array

- Busca de string
 - Com a árvore de sufixos, percorríamos a árvore considerando os caracteres do nosso padrão P. Se terminássemos em um nó interno, o número de filhos do nó era o número de ocorrências de P em S. Complexidade: $O(m + c)$
 - Um nó interno é representado por um intervalo em um suffix array, dessa forma, temos que encontrar o limitante inferior e o limitante superior.
 - Para isso, aplica-se duas buscas binárias. Complexidade: $O(m \cdot \log(n))$

Suffix Array


- Exemplo: $S = \text{BANANA}$ e $P = \text{NA}$

i	VS[i]	Sufixo
0	5	A
1	3	ANA
2	1	ANANA
3	0	BANANA
4	4	NA
5	2	NANA

- Encontrando o limitante inferior

Suffix Array

- Exemplo: $S = \text{BANANA}$ e $P = \text{NA}$



i	VS[i]	Sufixo
0	5	A
1	3	ANA
2	1	ANANA
3	0	BANANA
4	4	NA
5	2	NANA

- Encontrando o limitante superior

Suffix Array

- Exemplo: $S = \text{BANANA}$ e $P = \text{NA}$



i	VS[i]	Sufixo
0	5	A
1	3	ANA
2	1	ANANA
3	0	BANANA
4	4	NA
5	2	NANA

- Ocorrências em 4 a 2.

Suffix Array

- Máxima substring repetida:
 - Precisamos incluir o campo “Maior Prefixo Comum” (mpc) no vetor de sufixos
 - Calculado a cada par de sufixos consecutivos no vetor
 - A posição de maior MPC é a maior substring repetida

Suffix Array

i	VS[i]	MPC[i]	Sufixo
0	5	0	A
1	3	1	<u>A</u> NA
2	1	3	<u>AN</u> ANA
3	0	0	BANANA
4	4	0	NA
5	2	2	<u>NA</u> NA

- T = "BANANA";
- A substring mais comprida repetida é "ANA";
- Outras strings repetidas são "A" e "NA".

Suffix Array

- Máxima substring comum
 - Cria-se uma string a partir da concatenação das strings do problema
 - Crie o vetor de sufixos desta nova string
 - Calcule os MPCs
 - Marque qual string é a origem de cada sufixo
 - Retorne o maior valor de MPC cujo sufixo tenha origem em todas as strings do problema

Suffix Array

i	VS[i]	MCP[i]	ORIGEM	Sufixo
0	6	0	2	.MANA
1	10	0	2	A
2	5	1	1	<u>A</u> .MANA
3	8	1	2	<u>A</u> NA
4	3	3	1	<u>ANA</u> .MANA
5	1	3	1	<u>ANANA</u> .MANA
6	0	0	1	BANANA.MANA
7	7	0	2	MANA
8	9	0	2	NA
9	4	2	1	<u>NA</u> .MANA
10	2	2	1	<u>NANA</u> .MANA

Suffix Array

- Para gerar o vetor de sufixos ordenados lexicograficamente, é possível aplicar um algoritmo de ordenação por comparação de complexidade $O(n \cdot \log n)$. Comparar duas strings leva tempo $O(n)$, sendo assim, esta estratégia teria complexidade **$O(n^2 \log n)$**
- Um modo mais eficiente **$O(n \cdot \log n)$** de ordenar os sufixos consiste em uma estratégia iterativa denominada **ordenação por pares de ranks**.

Suffix Array

i	SA[i]	Suffix	RA[SA[i]]	RA[SA[i]+1]
0	0	GATAGACA\$	71 (G)	65 (A)
1	1	ATAGACA\$	65 (A)	84 (T)
2	2	TAGACA\$	84 (T)	65 (A)
3	3	AGACA\$	65 (A)	71 (G)
4	4	GACA\$	71 (G)	65 (A)
5	5	ACA\$	65 (A)	67 (C)
6	6	CA\$	67 (C)	65 (A)
7	7	A\$	65 (A)	36 (\$)
8	8	\$	36 (\$)	00 (-)

Initial ranks $RA[i] = \text{ASCII value of } T[i]$
 $\$ = 36, A = 65, C = 67, G = 71, T = 84$

i	SA[i]	Suffix	RA[SA[i]]	RA[SA[i]+1]
0	8	\$	36 (\$)	00 (-)
1	7	A\$	65 (A)	36 (\$)
2	5	ACA\$	65 (A)	67 (C)
3	3	AGACA\$	65 (A)	71 (G)
4	1	ATAGACA\$	65 (A)	84 (T)
5	6	CA\$	67 (C)	65 (A)
6	0	GATAGACA\$	71 (G)	65 (A)
7	4	GACA\$	71 (G)	65 (A)
8	2	TAGACA\$	84 (T)	65 (A)

If $SA[i] + k \geq n$ (beyond the length of string T),
 we give a default rank 0 with label -

Suffix Array

i	SA[i]	Suffix	RA[SA[i]]	RA[SA[i]+2]
0	8	\$	0 (\$-)	0 (--)
1	7	A\$	1 (A\$)	0 (--)
2	5	ACA\$	2 (AC)	1 (A\$)
3	3	AGACA\$	3 (AG)	2 (AC)
4	1	ATAGACA\$	4 (AT)	3 (AG)
5	6	CA\$	5 (CA)	0 (\$-)
6	0	GATAGACA\$	<u>6 (GA)</u>	7 (TA)
7	4	GACA\$	<u>6 (GA)</u>	5 (CA)
8	2	TAGACA\$	7 (TA)	6 (GA)

\$- (first item) is given rank 0, then for $i = 1$ to $n-1$, compare rank pair of this row with previous row

i	SA[i]	Suffix	RA[SA[i]]	RA[SA[i]+2]
0	8	\$	0 (\$-)	0 (--)
1	7	A\$	1 (A\$)	0 (--)
2	5	ACA\$	2 (AC)	1 (A\$)
3	3	AGACA\$	3 (AG)	2 (AC)
4	1	ATAGACA\$	4 (AT)	3 (AG)
5	6	CA\$	5 (CA)	0 (\$-)
<u>6</u>	<u>4</u>	<u>GACA\$</u>	<u>6 (GA)</u>	<u>5 (CA)</u>
<u>7</u>	<u>0</u>	<u>GATAGACA\$</u>	<u>6 (GA)</u>	<u>7 (TA)</u>
8	2	TAGACA\$	7 (TA)	6 (GA)

If $SA[i] + k \geq n$ (beyond the length of string T), we give a default rank 0 with label -

Suffix Array

i	SA[i]	Suffix	RA[SA[i]]	RA[SA[i]+4]
0	8	\$	0 (\$---	0 (----)
1	7	A\$	1 (A\$--)	0 (----)
2	5	ACA\$	2 (ACA\$)	0 (----)
3	3	AGACA\$	3 (AGAC)	1 (A\$--)
4	1	ATAGACA\$	4 (ATAG)	2 (ACA\$)
5	6	CA\$	5 (CA\$-)	0 (----)
6	4	GACA\$	6 (GACA)	0 (\$---
7	0	GATAGACA\$	7 (GATA)	6 (GACA)
8	2	TAGACA\$	8 (TAGA)	5 (CA\$-)

Now all suffixes have different ranking
We are done

Referências

S. Halim e F. Halim. Competitive Programming 2.

Fábio L. Usberti. Processamento de Cadeias de Caracteres. Summer School 2019

<https://www.geeksforgeeks.org/kmp-algorithm-for-pattern-searching/>

<https://www.ime.usp.br/~pf/estruturas-de-dados/aulas/kmp.html>

<https://www.cs.cmu.edu/~ckingsf/bioinfo-lectures/suffixtrees.pdf>

http://www.cs.jhu.edu/~langmea/resources/lecture_notes/tries_and_suffix_tries.pdf

<https://cp-algorithms-brasil.com/strings/suffixtree.html>

https://web.stanford.edu/~mjkay/suffix_tree.pdf

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

http://jeiks.net/wp-content/uploads/2014/08/TEP_Extra-03.pdf