

Processamento de Strings

SCC 211 - LAb. Alg. Avançados II

Veja o código abaixo

```
11 int main(){
12     char text[1000], pattern[100];
13
14     fgets(text, 1000, stdin);
15     fgets(pattern, 100, stdin);
16
17     text[strlen(text)-1] = '\0';
18     pattern[strlen(pattern)-1] = '\0';
19
20     int tam_t = strlen(text);
21     int tam_pat = strlen(pattern);
22
23     int i = 0;
24     char *p = text;
25
26     while (tam_t-i >= tam_pat && strncmp(p, pattern, tam_pat) ){
27         //cout << p << endl;
28         i++; p++;
29     }
30
31     if (tam_t-i >= tam_pat)
32         cout << "A sub string foi encontrada na posicao = " << i << endl;
33
34     return 1;
35 }
```

Qual a complexidade da busca?

$O(n)$???
talvez, em textos normais...

texto =
AAAAAAAAAAB

Pattern =
AAAAB

$O(nm)$

Este código é o mesmo, só que sem strncmp

```
28     int tam_t = strlen(text);
29     int tam_pat = strlen(pattern);
30
31
32     for (int i=0; i<tam_t; i++){
33         bool achou = true;
34         for (int j=0; j<tam_pat; j++){
35             if (i+j >= tam_t || text[i+j] != pattern[j])
36                 achou = false;
37         }
38         if (achou)
39             cout << "A sub string foi encontrada na posicao = " << i << endl;
40
41     }
42
43     return 1;
```

O algoritmo KMP (Knuth - Morris - Pratt) 1977

- Veja o caso de “AAAAAAAAAABB” e “AAAAB”
 - o algoritmo anterior vai, no início, sempre comparar todos os A's para falhar miseravelmente, e sempre, quando chegar no primeiro B
- É possível “pular” comparações, ao invés de deslocar o padrão caracter a caracter no texto?
- Ou seja, é possível evitar “recomparar” um caracter em T (texto) que já é o igual ao mesmo caracter em P (pattern) ?
- O algoritmo KMP faz exatamente isso.
- Vejamos..

KMP

	1	2	3	4	5
	0	1	2	3	4
T =	I	D	O	N	O
P =	S	E	V	E	N

012345678901234567890123456789012345678901234567890
T = I DO NOT LIKE SEVENTY SEV BUT SEVENTY SEVENTY SEVEN
P = SEVENTY SEVEN
0123456789012

- O 1º caracter de P não casa com T[i], para i = 0 até 13
 - Neste caso, KPM se comporta como o algoritmo força bruta anterior, incrementando i em uma unidade...
- i = 14 e j = 0 (sempre associaremos o índice j ao padrão P), temos:

KPM: $i = 14$, $j = 0$...

	1	2	3	4	5
	0	1	2	3	4
T =	I	D	O	N	O
P =					

012345678901234567890123456789012345678901234567890

SEVENTY SEV BUT SEVENTY SEVENTY SEVEN

SEVENTY SEVEN

0123456789012

- Temos 11 acertos em $i = 14$ até 24, e um erro em $i = 25$ ($j = 11$).
- O algoritmo força bruta continuaria em $i = 15$, mas KPM pode retomar em $i=25$
- A parte do padrão até antes do erro é: “SEVENTY SEV”
- SEV (tam=3) aparece no início (prefixo) e no fim (sufixo) da string.
 - é chamada de borda
- Não podemos simplesmente começar em $i = 25$ e $j = 0$, mas podemos voltar $j = 3$, pulando $11 - 3 = 8$ caracteres (‘SEVENTY ‘), enquanto i permanece em 25.

KPM

```

          1         2         3         4         5
012345678901234567890123456789012345678901234567890
T = I DO NOT LIKE SEVENTY SEV BUT SEVENTY SEVENTY SEVEN
P =                      SEVENTY SEVEN
                          0123456789012
                            1
                        ^ then immediate mismatch at index i = 25, j = 3
```

- Neste ponto, a string antes do mismatch é “SEV”. Não há borda.
- Portanto, $j = 0 \rightarrow$ voltamos a comparar P a partir de seu início.
- Há erros de $i = 25$ até 29.

KPM

T = I DO NOT LIKE SEVENTY SEV BUT SEVENTY SEVENTY SEVEN
P = SEVENTY SEVEN

- Então temos um casamento entre $i = 30$ até $i = 42$
- temos uma borda de $\text{tam} = 5$ ("SEVEN"), então $j = 5$, pulando $13 - 5 = 8$ caracteres., recomeçando a busca em $i = 43$, quando temos um novo casamento!
- Este é um algoritmo eficiente!

	1	2	3	4	5				
	012345678901234567890123456789012345678901234567890								
T =	I DO NOT LIKE SEVENTY SEV BUT SEVENTY SEVENTY SEVEN								
P =	SEVENTY SEVEN 0123456789012								

KPM = como calcular o reset j no padrão

	0	1	2	3	4	5	6	7	8	9	0	1	2	3
P =	S	E	V	E	N	T	Y		S	E	V	E	N	
b =	-1	0	0	0	0	0	0	0	0	1	2	3	4	5

- KPM pré-processa o padrão no início, de forma a calcular o tamanho de todas as possíveis bordas, criando o vetor b. Observe acima...
- Se o “erro” ocorrer em $j = 11$ (após achar casamento para “SEVENTY SEV”), devemos recomençar P em $j = b[11] = 3$
- O algoritmo tem complexidade $O(n+m)$

KPM

```
11 void kmpPreProcess(int *b, char *pat, int tam_pat){
12     int i = 0, j = -1; b[0] = -1;
13
14     while (i < tam_pat){
15         while (j >= 0 && pat[i] != pat[j]) // eh diferente... Reseta j, usando b .
16             j = b[j];
17         i++; j++;
18         b[i] = j;      // observe que b[i] so deixa de ser zero quando o padrao se repetir em pattern!
19     }
20 }
21
22 void kpmSearch(int *b, char *pat, char *text, int tam_pat, int tam_t){
23     int i = 0, j = 0;
24     while (i < tam_t){
25         while (j >= 0 && text[i] != pat[j])
26             j = b[j];
27         i++; j++;
28         if (j == tam_pat) { // opa... teve matching !!!
29             cout << "A sub string foi encontrada na posicao = " << i - j << endl;
30             j = b[j];      // prepara j para um provavel matching novo....
31         }
32     }
33 }
```

Suffix Trie, Suffix Tree e Suffix Array

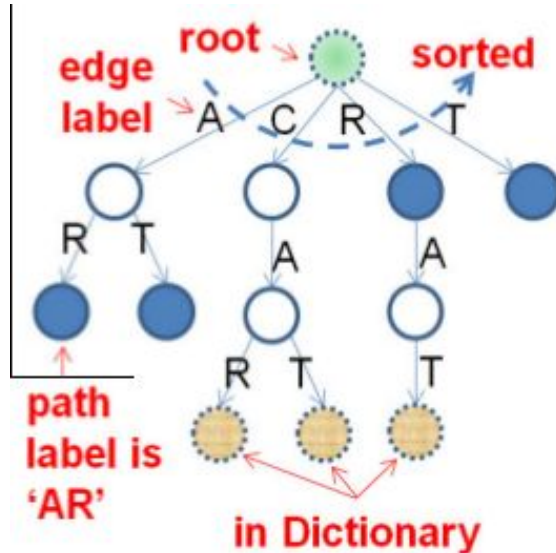
- Estas são estruturas de dados muito eficientes para manipulação de strings
- A motivação cresceu bastante com a evolução dos trabalhos em sequenciamento genético onde a busca por padrões em cadeias enormes é fundamental para o entendimento do processo.

Suffix Trie

- Não é um “typo”. Trie vem de “information retrieval”
- Um sufixo i (ou i^{th} sufixo) de uma string é uma substring que vai do i^{th} caracter da string até último caracter da string
 - Seja a string STEVEN: o 2o sufixo é “EVEN”; o 4o sufixo é “EN” (indexação em 0)
- Uma sufix trie de um conjunto de strings S é uma árvore com todos os possíveis sufixos das strings em S
 - o rótulo de uma aresta representa um caracter
 - o nó (vértice) representa um sufixo indicado pela trilha de rótulos: sequência de arestas da raiz até o vértice.
 - cada vértice é conectado a possíveis outros 26 vértices (assumindo que usamos letras maiúsculas apenas)
 - cada vértice tem 2 flags booleanos para indicar: palavra ou sufixo.

Suffix Trie

- Seja $S = \{\text{"CAR"}, \text{"CAT"}, \text{"RAT"}\}$
- Os sufixos = $\{\text{"CAR"}, \text{"AR"}, \text{"R"}, \text{"CAT"}, \text{"AT"}, \text{"T"}, \text{"RAT"}, \text{"AT"}, \text{"T"}\}$;
- Ordenando e eliminando repetições = $\{\text{"AR"}, \text{"AT"}, \text{"CAR"}, \text{"CAT"}, \text{"R"}, \text{"RAT"}, \text{"T"}\}$;



- Bastante usado em dicionários.
- Uma vez construída, uma busca pode ser feita em $O(m)$, onde m é o comprimento do padrão.
- Basta visitar os nós a partir da raiz

Suffix Tree

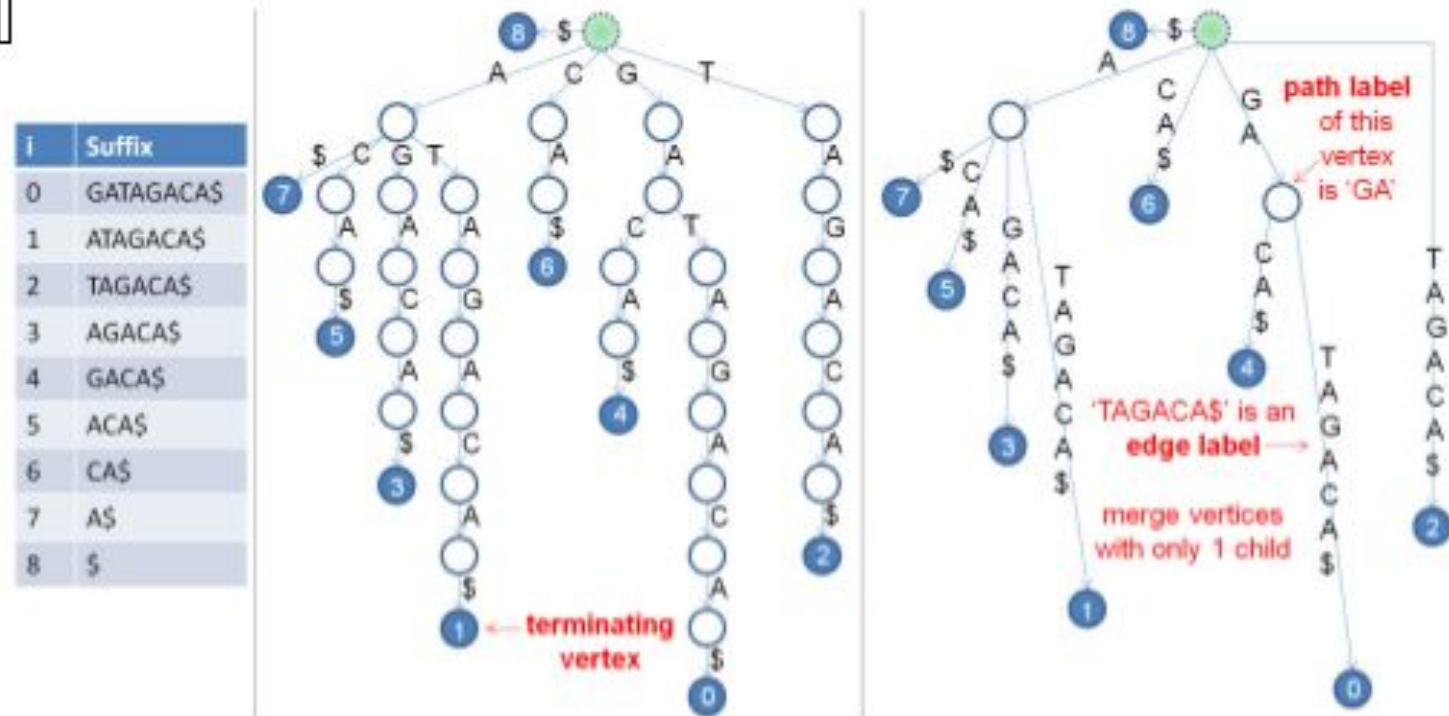


Figure 6.3: Suffixes, Suffix Trie, and Suffix Tree of $T = \text{'GATAGACAS'}$

Suffix Tree

- Em uma suffix trie, quanto maior a string mais vértices duplicados teremos
- Em uma suffix Tree, todos os vértices com apenas um filho são combinados (compressão de trilha!), reduzindo o número de nós.
- Obviamente, os rótulos agora serão maiores e representam um sufixo.
- a representação da árvores de sufixos é bem mais compacta que a da suffix trie, com no máximo $2n$ vértices
 - há no máximo n folhas para os n sufixos. Todos nós internos não raiz sempre se ramificam. Portanto, deve haver no máximo $n-1$ nós internos $\rightarrow$ Total = n folhas + $n-1$ nós internos + 1 raiz = $2n$.

Suffix Tree

- A implementação de árvores de sufixo não é trivial, especialmente se quisermos fazer com que a montagem da árvore seja de $O(n)$, onde n é o tamanho do texto (dicionário).
- No entanto, é extremamente eficiente para diversas aplicações. Seja m o comprimento da string de busca. Seja z o nro de ocorrências de uma substring no texto.
 - Busca de substring (verifica se o padrão existe ou não): $O(m)$
 - Busca de todas ocorrências de uma substring: $O(m + z)$
 - Busca da substring mais longa: $O(n)$
 - Busca da substring comum mais longa em 2 substrings $s1$ e $s2$: construção da árvore $O(\text{tam}(s1) + \text{tam}(s2))$; busca: $O(\text{tam}(s1) + \text{tam}(s2))$

Suffix Tree

- Sugiro um excelente material na web:
 - <http://www.geeksforgeeks.org/ukkonens-suffix-tree-construction-part-1/>
 - conceitos
 - disponibilização do código fonte
 - exemplos que podem ser executados no site.
- Vamos acompanhar toda a discussão neste material.

Suffix Array - vetores de sufixos.

- bem mais fáceis de implementar que suffix trees
- embora não tão eficientes: $O(n \log n)$ na construção.
- buscas por padrões podem ser feitas em $O(m \log n)$

Suffix Array

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

Figure 6.7: Sorting the Suffixes of $T = \text{'GATAGACA$'}$

vetor que armazena uma permutação de n índices de sufixos ordenados.

suffix arrays X suffix trees

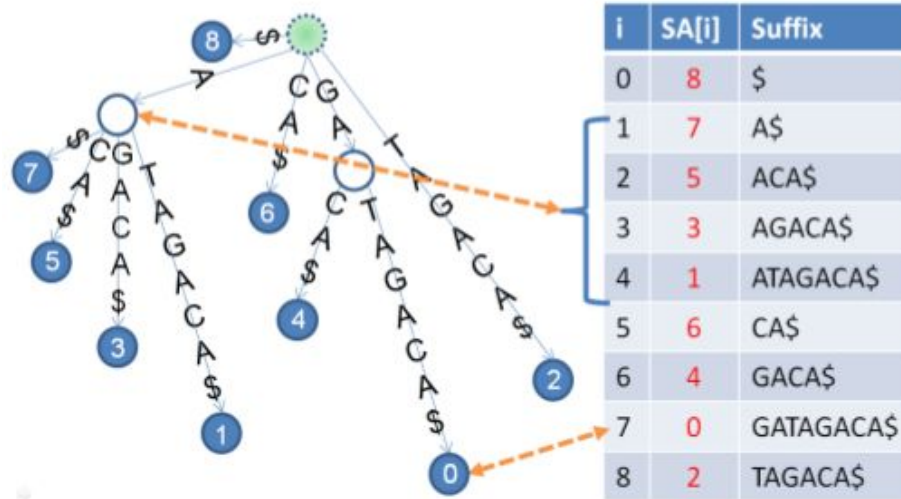


Figure 6.8: Suffix Tree and Suffix Array of $T = \text{'GATAGACA\$'}$

suffix arrays - construção

```
#include <algorithm>
#include <cstdio>
#include <cstring>
using namespace std;

#define MAX_N 1010 // first approach:  $O(n^2 \log n)$ 
char T[MAX_N]; // this naive SA construction cannot go beyond 1000 chars
int SA[MAX_N], i, n; // in programming contest settings

bool cmp(int a, int b) { return strcmp(T + a, T + b) < 0; } //  $O(n)$ 

int main() {
    n = (int)strlen(gets(T)); // read line and immediately compute its length
    for (int i = 0; i < n; i++) SA[i] = i; // initial SA: {0, 1, 2, ..., n-1}
    sort(SA, SA + n, cmp); // sort:  $O(n \log n)$  * cmp:  $O(n)$  =  $O(n^2 \log n)$ 
    for (i = 0; i < n; i++) printf("%2d\t%s\n", SA[i], T + SA[i]);
} // return 0;
```

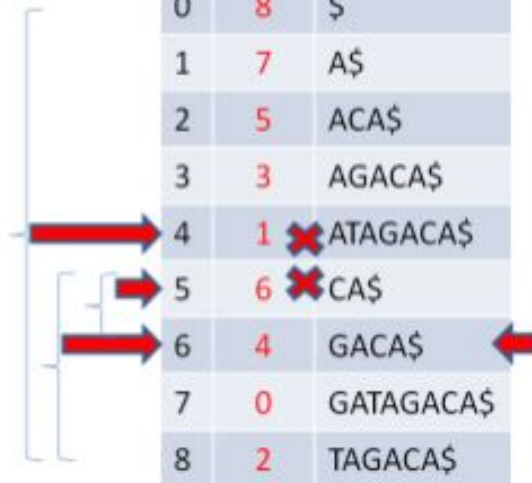
suffix arrays - Exemplo em string matching

- a procura por um padrão de tam= m em T de tam= n é $O(m \log n)$.
- Isso é $\log n$ vezes mais lento que na árvore de sufixos, mas é aceitável.
 - precisamos de 2 buscas binárias $O(\log n)$ nos $O(m)$ sufixos ordenados
 - as duas buscas binárias
 - 1a $\rightarrow$ busca pelo menor i , tal que o prefixo do sufixo $SA[i]$ casa com o padrão P
 - 2a $\rightarrow$ busca pelo maior i , tal que o prefixo do sufixo $SA[i]$ casa com o padrão P

suffix arrays - Exemplo em string matching

Finding lower bound

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\$



Finding upper bound

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\$

