

Strings

Suffix Array – Definição e Construção

Prof. Edson Alves - UnB/FGA

2019

1. Definição
2. Construção do vetor de sufixos em $O(N \log N)$

Definição

- Seja S uma string de tamanho N
- O i -ésimo sufixo de S é a substring que inicia no índice i e termina no índice N , isto é, $S[i..N]$
- Um vetor de sufixos (*suffix array*) $s_A(S)$ de S é um vetor de inteiros que representam os índices iniciais i dos prefixos de S , após a ordenação lexicográfica dos mesmos
- Os vetores de sufixos são usados em problemas que envolvem buscas em strings
- Estes vetores foram propostos por Udi Manber e Gene Myers

Exemplo de *suffix array*

i	$S[i..N]$
1	"abacaxi"
2	"bacaxi"
3	"acaxi"
4	"caxi"
5	"axi"
6	"xi"
7	"i"

j	$s_A[j]$	$S[s_A[j]..N]$
1	1	"abacaxi"
2	3	"acaxi"
3	5	"axi"
4	2	"bacaxi"
5	4	"caxi"
6	7	"xi"
7	6	"xi"

Construção de $s_A(S)$ com complexidade $O(N^2 \log N)$

- A construção de $s_A(S)$ diretamente de sua definição tem complexidade $O(N^2 \log N)$, onde N é o tamanho da string S
- Primeiramente é preciso construir um vetor ps de pares $(S[i..N], i)$
- Em seguida, este vetor deve ser ordenado
- O algoritmo de ordenação tem complexidade $O(N \log N)$, e como as comparações entre as substrings tem complexidade $O(N)$, a complexidade da solução é $O(N^2 \log N)$
- Observe que, após ordenado o vetor ps , o vetor de sufixos $s_A(S)$ é composto apenas pelos índices (segundo elemento de cada par), não sendo necessário armazenar os prefixos explicitamente
- Assim a complexidade de memória é $O(N)$

Construção *naive* de $s_A(S)$

```
5 vector<int> suffix_array(const string& s)
6 {
7     using si = pair<string, int>;
8
9     vector<si> ss(s.size());
10
11     for (size_t i = 0; i < s.size(); ++i)
12         ss[i] = si(s.substr(i), i);
13
14     sort(ss.begin(), ss.end());
15
16     vector<int> sa(s.size());
17
18     for (size_t i = 0; i < s.size(); ++i)
19         sa[i] = ss[i].second;
20
21     return sa;
22 }
```

Observações sobre a construção *naive* de $s_A(S)$

- Embora a construção apresentada seja de fácil entendimento e implementação, ela não é aplicável em strings grandes ($N \geq 10^4$)
- É possível construir $s_A(S)$ com complexidade $O(N \log N)$, porém tanto a implementação é mais sofisticada
- Além disso, a terminologia e os conceitos utilizados para esta redução na complexidade não são triviais
- Estes conceitos e esta construção serão apresentados a seguir

Construção do vetor de sufixos em $O(N \log N)$

Substrings cíclicas

- A notação de substrings de uma string S pode ser estendida para contemplar substrings cíclicas
- A notação padrão $S[i..j]$ pressupõe que $i \leq j$
- Para representar substrings cíclicas, basta fazer

$$S[i..j] = S[i..N] + S[1..j]$$

quando $i > j$

- Por exemplo, para $S = \text{"casado"}$, temos $S[6..2] = \text{"oca"}$ e $S[5..3] = \text{"docas"}$
- Uma rotação cíclica de uma string S é uma substring cíclica de tamanho $|S|$
- Por exemplo, as rotações cíclicas de $S = \text{"abcd"}$ são "abcd" , "bcda" , "cdab" e "dabc"

Ideia central do algoritmo de construção $O(N \log N)$

- A ideia central do algoritmo de construção do vetor de sufixos em $O(N \log N)$ é que é possível ordenar, de forma eficiente, as rotações cíclicas de S
- Para que a ordenação destas rotações cíclicas seja equivalente à ordenação dos sufixos de S , basta concatenar um caractere sentinela ao final de S
- Este sentinela deve ter um valor ASCII inferior a qualquer caractere do alfabeto
- Em geral, este caractere é o caractere '\$', sendo o caractere '#' uma segunda opção viável, caso a string S seja alfanumérica
- Assim, após a ordenação, exceto a primeira rotação, as demais equivalem à ordenação dos prefixos de S

Equivalências entre as rotações cíclicas e os sufixos de S

i	Rotação cíclica	j	$S[j..N]$
0	"\$banana"	-	-
1	"a\$banan"	6	"a"
2	"ana\$ban"	4	"ana"
3	"anana\$b"	2	"anana"
4	"banana\$"	1	"banana"
5	"na\$bana"	5	"na"
6	"nana\$ba"	3	"nana"

Permutações e classes de equivalência

- A cada iteração do algoritmo serão ordenadas todas as substrings de $S[i..j]$ de tamanho 2^k , para $k = 0, 1, \dots, \lceil \log N \rceil$
- Para tal fim, serão mantidos dois vetores auxiliares
- O primeiro deles é o vetor de permutações ps , onde $ps[i]$ é o índice da substring $S[i..j]$ de tamanho k após a ordenação
- O segundo é o vetor cs das classes de equivalência das substrings de tamanho k
- Duas substrings iguais devem estar na mesma classe de equivalência
- Se $S[i..j] < S[r..s]$, então $cs[i] < cs[r]$
- Estas classes de equivalência permitem realizar comparações de forma mais eficiente

Exemplos de permutações e classes de equivalência

k	Substrings cíclicas de tamanho 2^k	ps	cs
0	{ "c", "a", "s", "a" }	{2, 0, 3, 1}	{1, 0, 2, 0}
1	{ "ca", "as", "sa", "ac" }	{2, 1, 3, 0}	{2, 1, 3, 0}
2	{ "casa", "asac", "saca", "acas" }	{2, 1, 3, 0}	{2, 1, 3, 0}

k	Substrings cíclicas de tamanho 2^k	ps	cs
0	{ "a", "b", "b", "a" }	{0, 2, 3, 1}	{0, 1, 1, 0}
1	{ "ab", "bb", "ba", "aa" }	{1, 3, 2, 0}	{1, 3, 2, 0}
2	{ "abba", "bbaa", "baab", "aabb" }	{1, 3, 2, 0}	{1, 3, 2, 0}

Casos base: $k = 0$

- O algoritmo inicia com o caso base, onde $k = 0$, ou seja, ordenado as substrings cíclicas de tamanho 1
- Isto pode ser feito em $O(N)$ usando o *counting sort*
- Após a ordenação e geração do vetor de permutações ps , é necessário atribuir as classes de equivalência a cada substring, gerando o vetor cs
- Vale lembrar que substrings iguais devem pertencer à mesma classe de equivalência
- O vetor ps permite a construção de cs também em $O(N)$, por meio da comparação de caracteres adjacentes

1. CP Algorithms. [Suffix Array](#), acesso em 06/09/2019.
2. **CROCHEMORE**, Maxime; **RYTTER**, Wojciech. *Jewels of Stringology: Text Algorithms*, WSPC, 2002.
3. **HALIM**, Steve; **HALIM**, Felix. *Competitive Programming 3*, Lulu, 2013.