

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	"i"
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 i -ésima substring 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 eficiente

Exemplos de permutações e classes de equivalência

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

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

Implementação do *counting sort*

```
4 using vi = vector<int>;
5 using ii = pair<int, int>;
6
7 template<typename T> void
8 counting_sort(vi& ps, const T& xs, size_t alphabet_size)
9 {
10     // Gera o histograma dos elementos distintos
11     vector<int> hs(alphabet_size, 0);
12
13     for (auto x : xs)
14         ++hs[x];
15
16     // Faz a soma prefixada para estabelecer a ordem
17     for (size_t i = 1; i < alphabet_size; ++i)
18         hs[i] += hs[i - 1];
19
20     // Preenche a permutação referente à ordenação
21     for (int i = ps.size() - 1; i >= 0; --i)
22         ps[--hs[xs[i]]] = i;
23 }
```


Preenchimento das classes de equivalência

```
25 template<typename T> int
26 update_equivalence_classes(vi& cs, const vi& ps, const T& xs)
27 {
28     int c = 0;
29     cs[ps[0]] = c;
30
31     // Processa os elementos de s na ordem indicada pela permutação
32     for (size_t i = 1; i < ps.size(); ++i)
33     {
34         // Elementos distintos pertencem a classes distintas
35         if (xs[ps[i - 1]] != xs[ps[i]])
36             ++c;
37
38         cs[ps[i]] = c;
39     }
40
41     // Retorna o número de classes distintas
42     return c + 1;
43 }
```

Complexidade da construção do *suffix array*

- A transição consiste em computar os valores ps e cs para substrings de tamanho 2^k , se conhecidos estes vetores para strings de tamanho 2^{k-1}
- Se esta transição for feita em $O(N)$, a complexidade do algoritmo terá complexidade $O(N \log N)$, pois esta atualização deverá ser feita $O(\log N)$ vezes
- Esta transição pode ser feita em $O(N \log N)$, o que aumenta a complexidade assintótica do algoritmo para $O(N \log^2 N)$
- Esta piora na complexidade é compensada por uma codificação mais curta em termos de linhas de código

Transições

- Observe que a substring de tamanho 2^k que inicia na posição i é formada pela concatenação das strings de tamanho 2^{k-1} que começam nas posições i e $i + 2^{k-1} \pmod{N}$, respectivamente
- Assim, na ordenação das substrings de tamanho 2^k , a comparação entre as strings com início em i e j equivale à comparação dos pares ordenados

$$(cs[i], cs[i + 2^{k-1} \pmod{N}]) \text{ e } (cs[j], cs[j + 2^{k-1} \pmod{N}])$$

- Estas transições devem ser feitas para todos os valores $2^{k-1} < N$
- Após a última iteração é preciso remover do vetor de permutações ps o índice corresponde ao caractere sentinela que fora adicionado à string original antes do caso base e das transições
- O índice deste caractere ocupará a primeira posição de ps
- As demais posições corresponderão ao vetor de sufixos da strings original

Implementação $O(N \log^2 N)$ do *suffix array*

```
1 #include <bits/stdc++.h>
2
3 using namespace std;
4 using iii = tuple<int, int, int>; // (cs[i], cs[i + 2^k], i)
5
6 void update_permutations(vector<int>& ps, vector<iii>& pps)
7 {
8     // Ordena os pares
9     sort(pps.begin(), pps.end());
10
11     // Atualiza as permutações e remove a referência
12     for (size_t i = 0; i < ps.size(); ++i)
13     {
14         ps[i] = get<2>(pps[i]);
15         get<2>(pps[i]) = 0;
16     }
17 }
18
19 void update_equivalence_classes(vector<int>& cs,
20     const vector<int>& ps, const vector<iii>& pps)
21 {
```

Implementação $O(N \log^2 N)$ do *suffix array*

```
22     int c = 0;
23     cs[ps[0]] = c;
24
25     // O vetor pps está ordenado
26     for (size_t i = 1; i < ps.size(); ++i)
27     {
28         // Elementos distintos pertencem a classes distintas
29         if (pps[i - 1] != pps[i])
30             ++c;
31
32         cs[ps[i]] = c;
33     }
34 }
35
36 vector<int> suffix_array(const string& S)
37 {
38     auto s = S + "$";
39     auto N = s.size();
40
41     vector<int> ps(N), cs(N);
42     vector<iii> pps(N);
```

Implementação $O(N \log^2 N)$ do *suffix array*

```
44 // Caso base
45 for (size_t i = 0; i < N; ++i)
46     pps[i] = iii(s[i], s[i], i);
47
48 update_permutations(ps, pps);
49 update_equivalence_classes(cs, ps, pps);
50
51 // Transições: mask = 2^(k - 1)
52 for (size_t mask = 1; mask < N; mask <= 1)
53 {
54     for (size_t i = 0; i < N; ++i)
55         pps[i] = iii(cs[i], cs[(i + mask) % N], i);
56
57     update_permutations(ps, pps);
58     update_equivalence_classes(cs, ps, pps);
59 }
60
61 ps.erase(ps.begin());
62
63 return ps;
64 }
```

Implementação eficiente do *suffix array*

- A ordenação dos pares ordenados pode ser feita em $O(N)$, através da combinação do *counting sort* com uma técnica semelhante ao *radix sort*
- A ideia é ordenar as substrings de tamanho 2^k inicialmente pela segunda metade, e em seguida, pela primeira metade
- Porém as substrings de tamanho 2^{k-1} já foram ordenadas na iteração anterior
- Assim, a ordenação pela segunda metade pode ser realizada substraindo 2^{k-1} de todos os elementos do vetor ps
- Isto porque se a menor substring de tamanho 2^{k-1} começa no índice i , a substring de tamanho 2^k com menor segunda metade começa no índice $i - 2^{k-1}$
- Esta subtração deve ser feita com aritmética modular, de modo que o vetor ps permanece sendo uma permutação dos índices da string S

Implementação eficiente do *suffix array*

- Uma vez que ps está ordenado pela segunda metade das substrings de tamanho 2^k , basta ordená-lo pela primeira metade, usando um algoritmo de ordenação estável e as classes de equivalência já estabelecidas
- Para tal, basta usar o *counting sort*
- Para evitar duplicação de código, a implementação do *counting sort* deve retornar a permutação dos índices do vetor a ser ordenado, sem alterar tal vetor
- Assim, após a geração da permutação que ordena rs , basta utilizá-la para gerar o novo vetor ps
- Esta abordagem leva a um algoritmo $O(N \log N)$ para a geração do *suffix array*

Implementação $O(N \log N)$ do *suffix array*

```
1 #include <bits/stdc++.h>
2
3 using namespace std;
4 using vi = vector<int>;
5 using ii = pair<int, int>;
6
7 template<typename T> void
8 counting_sort(vi& ps, const T& xs, size_t alphabet_size)
9 {
10     // Gera o histograma dos elementos distintos
11     vector<int> hs(alphabet_size, 0);
12
13     for (auto x : xs)
14         ++hs[x];
15
16     // Faz a soma prefixada para estabelecer a ordem
17     for (size_t i = 1; i < alphabet_size; ++i)
18         hs[i] += hs[i - 1];
19 }
```

Implementação $O(N \log N)$ do *suffix array*

```
20 // Preenche a permutação referente à ordenação
21 for (int i = ps.size() - 1; i >= 0; --i)
22     ps[--hs[xs[i]]] = i;
23 }
24
25 template<typename T> int
26 update_equivalence_classes(vi& cs, const vi& ps, const T& xs)
27 {
28     int c = 0;
29     cs[ps[0]] = c;
30
31     // Processa os elementos de s na ordem indicada pela permutação
32     for (size_t i = 1; i < ps.size(); ++i)
33     {
34         // Elementos distintos pertencem a classes distintas
35         if (xs[ps[i - 1]] != xs[ps[i]])
36             ++c;
37
38         cs[ps[i]] = c;
39     }
40 }
```

Implementação $O(N \log N)$ do *suffix array*

```
41 // Retorna o número de classes distintas
42 return c + 1;
43 }
44
45 vector<int> suffix_array(const string& S)
46 {
47     auto s = S + "$";
48     auto N = s.size();
49
50     vector<int> ps(N), cs(N), rs(N), xs(N);
51     vector<ii> ys(N);
52
53     // Caso base
54     counting_sort(ps, s, 256);
55     int c = update_equivalence_classes(cs, ps, s);
56 }
```

Implementação $O(N \log N)$ do *suffix array*

```
57 // Transições: mask = 2^(k - 1)
58 for (size_t mask = 1; mask < N; mask <= 1)
59 {
60     // Atualiza as permutações e gera os pares
61     for (size_t i = 0; i < N; ++i)
62     {
63         rs[i] = (ps[i] + N - mask) % N;
64         xs[i] = cs[rs[i]];
65         ys[i] = ii(cs[i], cs[(i + mask) % N]);
66     }
67
68     // Gera a permutação que ordena rs, usando as classes xs
69     counting_sort(ps, xs, c);
70
71     // Atualiza ps a partir da ordenação de rs
72     for (size_t i = 0; i < N; ++i)
73         ps[i] = rs[ps[i]];
74
75     // Atualiza cs a partir dos pares de classes de equivalência
76     c = update_equivalence_classes(cs, ps, ys);
77 }
```

Implementação $O(N \log N)$ do *suffix array*

```
79     ps.erase(ps.begin());
80
81     return ps;
82 }
83
84 int main()
85 {
86     string s;
87     cin >> s;
88
89     int N = s.size();
90     auto sa = suffix_array(s);
91
92     for (int i = 0; i < N; ++i)
93         cout << sa[i] << '\t' << s.substr(sa[i]) << '\n';
94
95     return 0;
96 }
```

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.