Strings

Suffix Array – Definição e Construção

Prof. Edson Alves - UnB/FGA

2019

Sumário

Definição

Suffix Array

- 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 strings 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 $S =$ "casado", temos $S[6..2] =$ "oca" e $S[5..3] =$ "docas"
- Uma rotação cíclica de uma string S é uma substring cíclica de tamanho $|S|$
- Por exemplo, as rotações cíclicas de $S =$ "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 $k = 0, 1, \ldots, \lceil \log N \rceil$
- 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*

```
5  vector<int> counting_sort(const string& s)
6  {
7      static const int A { 256 };      // Tamanho do alfabeto
8
9      // Gera o histograma dos caracteres
10     vector<int> hs(A, 0);
11
12     for (auto c : s)
13         ++hs[c];
14
15     // Faz a soma prefixada para estabelecer a ordem
16     for (int i = 1; i < A; ++i)
17         hs[i] += hs[i - 1];
18
19     // Preenche a permutação referente à ordenação
20     vector<int> ps(s.size());
21
22     for (size_t i = 0; i < s.size(); ++i)
23         ps[--hs[s[i]]] = i;
24
25     return ps;
26 }
```

Preenchimento das classes de equivalência

```
28 vector<int> equivalence_classes(const string& s, const vector<int>& ps)
29 {
30     int c = 0;
31     vector<int> cs(ps.size());
32 
33     cs[ps[0]] = c;
34 
35     // Processa os elementos de s na ordem indicada pela permutação
36     for (size_t i = 1; i < ps.size(); ++i)
37     {
38         // Elementos distintos pertencem a classes distintas
39         if (s[ps[i]] != s[ps[i - 1]])
40             ++c;
41 
42         cs[ps[i]] = c;
43     }
44 
45     return cs;
46 }
```

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

Referências

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.