chapter 9: String matching

You Me

date: Yesterday

目录

1 Let us begin !!!

这里我并没有非常考虑读者的感受...

这里我们说, 哎, 真的, 我没什么时间了, 总之我想快速地将这个东西飞完.

我们说, 字符串匹配有什么要讲的呢? 实际上就是面对这个 string matching 问题, 提出各种各样的算法. 我们可以 list a list:

1. 朴素匹配算法
2. Rabin-Karp 算法
3. finite automata
4. KMP
5. BST

我们这里简单过一下, 这里面的几个算法其实都比较有趣. 比如说 finite automata. 这个状态机的概念是非常重要的. 有助于我们理解什么是计算机.

OK, 我们先是假设我们已经知道什么是字符串匹配问题. 这里简单描述一下:

Definition 1 (string matching problem)**.** *We are given a text, denoted as T. Moreover, we are given a pattern, denoted as P. They are strings. We need to know that if there is pattern hidden in the text.*

这里我们还没说清楚, 我们说一个优先字符串是什么? 实际上是给定一个字母表 $\sum$, 而有限字符串就是这个一个序列 $\langle a_0 \cdots a_n \rangle, a_i \in \sum, i = 0, 1, \cdots, n$ 这就是一个字符串了.

Definition 2 (shift)**.** *我们这样表示字符串的一部分. 我们从 0 开始计数, 我们说 $T[0]$ 就是 T 的初始的字母. $T[0,1,2,3]$ 就代表这个 T 中编号 $0,1,2,3$ 组成的 sub 字符串.*

而后我们说, string matching 问题就是要找出一个 shift s, 有

$$T[s, \cdots, s+m-1] = P[0, 1, 2, \cdots, m]$$

其中 m 是这个东西的长度. 噢, 并且我们要找出这个所有的 s

好的, 说明完了, 我们可以开始一个概述, 我们从朴素方法开始.

1.1 朴素方法

朴素方法 朴素方法没什么好说的, 这是因为, 额, 就是因为太简单了吧. 其实就是噢, 我, 啊, 面对每一个 shift, 我都恭敬地将什么东西都比较了一遍.

这里列举出时间复杂度.

$$T(n) = \Theta((n-m+1)m)$$

只能说是明显.

1.2 Rabin-Karp algorithm

Rabin-Karp algorithm 这个是什么?

就是介绍了一个指纹方法: 我们给定一个进制 d 然后给出一个 string 的指纹.

$$f(P) = \sum_{i=0}^{m-1} d^{m-i-1} \times p[i]$$

就是将这个字母表映射到了一个正整数数组上面, 我们比较两个数字的速度要快得多, 利用这点来加快比较过程.

并且有一个递归方法进行指纹的计算, 我们设 t_s 是 shift 为 s 的时候的 T 的对应的长度为 m 的 sub 字符串的指纹值.

$$t_{s+1} = \left[t_s - T[s] \times d^{m-1}\right] \times d + T[s+m]$$

Moreover, Rabin-Karp 算法还提倡使用 hash 方法进行一个优化, 比如说当 m 稍微长一点的时候, 就装不下了, 溢出了. 这个时候使用 hash 方法, 我也不是知道是不是这个名字, 总之就是选取一个大于 $|\Sigma|$ 的质数. 取模, 而后以模值作为指纹值. q 为质数

同样的, 我们也有一个递归方法计算:

$$t_{s+1} = \Big[[t_s - T[s] \cdot c] \times d + T[s+m]\Big] \mod q$$

其中 c 是 $d^{m-1} \mod q$. 这个公式非常重要!!!!! 至于这是哪里来的, 我超, 我哪里知道.

```
Rabin-Karp (T,P,d,p){
    int n = T.length;
    int m = P.length;
    int h = d^{m-1} mod q;
    p = 0;
    t_0 = 0;
    for (i=1 to m){
        p = (dp + P[i]) mod q;
        t_0 = (dt_0 + T[i]) mod q;
    }// this for loop is for preprocessing
    for (s=0 to n-m){
        if p==t_s {
            if P[0,...,m-1] == T[s,...,s+m-1]
                print s;
        }
        if (s < n-m)
            t_s+1 = blahblah;
    }// this for loop is for matching
}
```

我们说, s 有 $n-m+1$ 个取值, 如果说每一个 s 都好巧不巧, tm 的都有指纹相等的话, 这个东西就相当于朴素匹配方法了.

Worst case:

$$O\left((n-m+1)\, m\right)$$

Computing the transition function

The following procedure computes the transition function δ from a given pattern $P[1\,..\,m]$.

```
COMPUTE-TRANSITION-FUNCTION(P, Σ)
1  m = P.length
2  for q = 0 to m
3      for each character a ∈ Σ
4          k = min(m + 1, q + 2)
5          repeat
6              k = k - 1
7          until P_k ⊐ P_q a
8          δ(q, a) = k
9  return δ
```

图 1: state transition function

平均期望之下, 我们说, 发生 suprious strike 的概率为

$$\frac{1}{q}$$

那么说我们发生匹配的次数为 $\dfrac{n-m+1}{q}$, 乘起来

$$\frac{n-m+1}{q} \times m = O\left(n-m+1\right)$$

finite automata 一个自动机, 是一个 5-tuple $(Q, q_0, A, \Sigma, \delta)$

Definition 3. *定义如下:*

1. *Q 是一个有限的集合, 其中元素称为是 state.*
2. *q_0 是上面 Q 的一个元素, 称为初始状态*
3. *$A \subset Q$ 是一个子集, 称为 accepting state 就是说, 当自动机走到这里的时候自动机就停止 (或者干别的).*
4. *Σ 是一个有限的集合, 其中元素称为字母, Σ 就是字母表.*
5. *δ 是一个函数: $Q \times \Sigma \to Q$ 就是说, 对于每一个 state, 根据当前的 input 是 σ 那么自动机将走到什么 state. 这个函数成为是状态转移函数. (额, 我们可以联想一下马尔可夫链, 虽然差别很大, 但是那边也有一个状态转移函数)*

Figure 1 展示了一个状态转移函数的计算. 我们进行一点点分析.

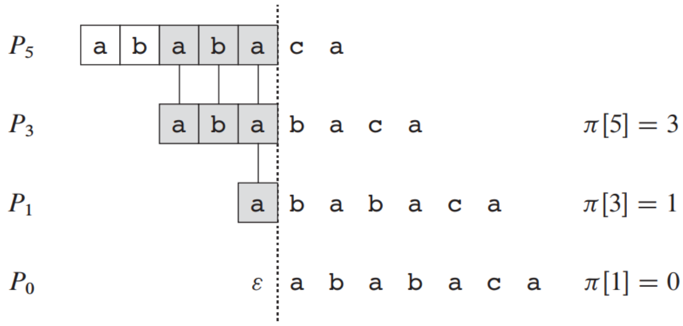
$$\delta(q, x) = \sigma(P[0, q]'x') = \max\{k : P[0, k]\text{是}P[0, q] + \{x\}\text{的後綴}\}$$

- line:2 从 q=0 开始, 相当于从矩阵第一行开始.
- line:3 对于每一个 a
- line:4 $k = \min\{m+1, q+2\}$ 这是因为 state 不会大于这两个前者是因为, 这种情况下匹配已经完成. q+2 是因为这个长度已经超过了当前的长度.
- line:7 直到当前匹配的后缀和 P 的前缀 P_k 相同.

The muching time that finite automata proceed along the Text is pretty good: $O\left(n\right)$. But the running that state transition requires is that $O\left(|\Sigma|\, m\right)$

i	1	2	3	4	5	6	7
$P[i]$	a	b	a	b	a	c	a
$\pi[i]$	0	0	1	2	3	0	1

(a)



(b)

图 2: pi array

```
COMPUTE-PREFIX-FUNCTION(P)
m = P.length
let π[1..m] be a new array
π[1] = 0
k = 0
for q = 2 to m
    while k > 0 and P[k + 1] ≠ P[q]
        k = π[k]
    if P[k + 1] == P[q]
        k = k + 1
    π[q] = k
return π
```

图 3: code

1.3 KMP algorithm

KMP We follow ppt here.

The $\pi[k]$ value is actually the length the longest common prefix and suffix.

The while loop in the algorithm is a way to efficiently lower the value of k. Figure 2 shows a example of how the pi array is worked out. And Figure 3 is the code that calculates the pi array.

And then Figure 4 is the code of how matcher works.

Finally, without any proof, the ppt tells us that the worst case running time is $O(n+m)$. Where we need $O(m)$ time to compute the π array.

```
KMP-MATCHER(T, P)
n = T.length
m = P.length
π = COMPUTE-PREFIX-FUNCTION(P)
q = 0                                         // number of characters matched
for i = 1 to n                                // scan the text from left to right
    while q > 0 and P[q + 1] ≠ T[i]
        q = π[q]                              // next character does not match
    if P[q + 1] == T[i]
        q = q + 1                             // next character matches
    if q == m                                 // is all of P matched?
        print "Pattern occurs with shift" i − m
        q = π[q]                              // look for the next match
```

图 4: code of matcher