始终

字符串匹配：KMP 算法

发表于 2016 年 12 月 20 日 | 分类于 Algorithm and Computer Science | 本文共被围观 3012 次

所谓字符串匹配，就是拿着一个字符串（也称为模式串），去到另一个字符串（母串）里去查找完全相同的子串的过程。显然，只要能定义相等关系，那么字符串匹配算法可以扩展到任意的序列匹配算法。因此，这会是一类用途很广的算法。

解决字符串匹配问题，最朴素的办法就是拿着模式串逐字符地沿着待匹配的串去比对，每次向前移动一个字符，直到完全匹配或者找不到匹配。显然，这个算法的复杂度是 $O(n \cdot m)$（n 表示母串的长度，m 表示模式串的长度），是比较高的。

这里介绍的 KMP 算法，能够在 $O(n)$ 时间内完成任务，它是由 Donald Knuth/James H. Morris/Vaughan Pratt 发明的。当然，你也可以称之为「看毛片算法」——你高兴就好。

从朴素算法开始

为了体现 KMP 算法的优势，也为了更容易地说明问题，我们先从最朴素的算法开始。

假设有

- 母串 `S`：`ababaababc`
- 模式串 `P`：`ababc`

现在我们的任务是在母串中找到与模式串完全相同的子串。朴素地算法是这样的：

1. 将母串与模式串从头对齐；
2. 从模式串的头部开始与母串对比
   a. 若字符相同，则继续对比；
   b. 若字符相同，且当前字符是模式串的最后一个字符，则匹配成功；
   c. 若字符不同，则将模式串沿着母串向后移动一位，再从头开始匹配；
   d. 若字符不同，且当前字符是母串的最后一个字符，则匹配失败。

在我们的示例里，具体的操作流程是这样的。

```
-> # 从头开始匹配
abab|aababc
abab|c
-> # 匹配失败，移动一位，继续尝试匹配
a|babaababc
 |ababc
-> # 匹配失败，移动一位，继续尝试匹配
ababa|ababc
  aba|bc
-> # 匹配失败，移动一位，继续尝试匹配
aba|baababc
   |ababc
-> # 匹配失败，移动一位，继续尝试匹配
ababa|ababc
    a|babc
-> # 匹配失败，移动一位，继续尝试匹配
ababaababc|
     ababc|
-> # 匹配成功，返回子串在母串中的位置
```

对多余工作的分析

优化算法一个很重要的方法，就是寻找重复/多余的工作，然后用合适的方法去除它们。因此，我们应该试着分析上述朴素算法，看看有哪些工作是多余的，或者是重复的。

```
-> # 从头开始匹配
abab|aababc
abab|c
-> # 匹配失败，移动一位，继续尝试匹配
a|babaababc
 |ababc
```

我们来观察第一次匹配失败，沿着母串移动模式串的位置的过程。匹配失败，是因为 `S[0:4] == P[0:4]` 但是 `S[4] != P[4]` 。于是我们将 `P[0]` 对齐 `S[1]` ，继续尝试匹配。但是，实际上在验证 `S[0:4] == P[0:4]` 的过程中，我们已经知道了 `S[0] == P[0] and S[0] != S[1]` 。因此，如果将 `P[0]` 与 `S[1]` 对齐，那么必然是匹配失败的。

既然在匹配的过程中，我们获得的信息，已经足够说明仅仅移动一位，必然匹配失败。那么这就是多余的工作，我们应该想办法规避掉这些多余的工作。那么，我们应该怎么办呢？

注意，在第一次尝试匹配的过程中，我们确定了 `S[0:4] == P[0:4]` ，又容易观察，对于模式串 `P` 来说，有 `P[0:2] == P[4 - 2:4]` ；即模式串 `P` 成功匹配的部分中，它的首两个字符与末两个字符完全相同。于是，因为 `S[0:4] == P[0:4]` ，所以我们有 `S[4 - 2:4] == P[4 - 2:4] == P[0:2]` 。这也就是说，如果将模式串对齐 `S[4 - 2]` 位置，我们天然就能确认两个位置的匹配，只需要接着向后尝试匹配就可以了——KMP 算法就是这样做的。

总结起来就是：

```
if S[i:i + j] == P[0:j] and S[i + j] != P[j]: # i + j < n and j < m
    k = argmax(k){P[0:k] == P[j - k:j]} # 0 <= k < j
    align P[0] to S[i + j - k]
```

在这个优化中，我们让模式串尽可能快地沿着母串向前跳跃；同时尽可能多地保留了已匹配的信息，避免接下来重复匹配。这一优化的关键，就是对每一个 `j`，在模式串中寻找最大的 `k`，使得 `P[0:k] == P[j - k:j]`。显然，对于给定的模式串 `P`，`k` 的取值只与 `j` 有关；我们记作 $k = f(j; P)$，并称之为模式串 `P` 的部分匹配函数。接下来，我们要看看如何快速地得到这个部分匹配函数。

部分匹配函数

根据定义，不难发现

$$f(1) = 0.$$

接下来我们看一个稍微复杂一点的模式串 `P = ababacb`。

```
j:    0 1 2 3 4 5 6
P:    a b a b a c b
f(j):   0 0 1 2 3 ?
```

我们来验证一下

```
j == 2: P[0:0](None) == P[j - 0:j](None) and P[0:1](a)    != P[j - 1:j](b)
j == 3: P[0:1](a)    == P[j - 1:j](a)    and P[0:2](ab)   != P[j - 2:j](ba)
j == 4: P[0:2](ab)   == P[j - 2:j](ab)   and P[0:3](aba)  != P[j - 3:j](bab)
j == 5: P[0:3](aba)  == P[j - 3:j](aba)  and P[0:4](abab) != P[j - 4:j](baba)
```

很好，没有问题。接下来我们看 $f(6)$ 是多少。遇到这样的问题，我们就会想，$f(j)$ 组成的序列，后项是否会与前项有关，存在某种递推关系呢？因此我们会做这样的分析。

首先，因为 $f(5) = 3$，所以我们知道 `P[0:3] == P[5 - 3:5] == P[2:5]`。现在，如果有 `P[3] == P[5]`，也就是 `P[f(5)] == P[5]`，那么 $f(6) = f(5) + 1$。但是现在 `P[3] == b != P[5] == c`，因此 $f(6) = f(5) + 1$ 不成立。

接下来，我们考虑 $f(f(5)) = f(3) = 1$。为什么考虑 $f(3)$ 而不是 $f(4)$ 呢？这是因为，我们已知 $f(5) = 3$，所以有 `P[0:3] == P[2:5]`；同时已知 $f(3) = 1$，就有 `P[0:1] == P[2:3] == P[4:5]`。而 `P[4:5]` 与当前待考虑的字符 `P[5]` 是紧挨着的。于是，如果我们有 `P[f(3)] == P[5]`，那么 $f(6) = f(3) + 1$。但是现在 `P[1] == b != P[5] == c`，因此 $f(6) = f(3) + 1$ 也不成立。

按照同样的分析，我们接下来应该考虑 $f(f(f(5))) = f(1) = 0$。但显然， `P[0] == a != P[5] == c`，因此 $f(6) = f(1) + 1$ 也不成立；于是只能是 $f(6) = 0$ 了。

也就是说，对于已经求得前 k 项部分匹配的模式串 `P` 来说，起第 $k + 1$ 项的部分匹配函数的值可以这样计算：

```
p_table = [-1, 0, ...]
ptr = k
while ptr > 0 and pattern[k] != pattern[res[ptr]]:
    ptr = p_table[ptr]
else:
    p_table.append(p_table[ptr] + 1)
```

这样一来，我们就能快速地计算任意的模式串 `P` 的部分匹配表了。

KMP 算法的实现示例

经过上面的分析，我们很自然地就能得到 KMP 算法（以下是一个用 Python 的实现）。

```python
def getPartialTable(pattern):
    if not pattern:
        return None
    lp = len(pattern)
    res = [-1, 0]
    if lp > 1:
        for curr in xrange(1, lp):
            ptr = curr
            while ptr > 0 and pattern[curr] != pattern[res[ptr]]:
                ptr = res[ptr]
            else:
                res.append(res[ptr] + 1)
    return res

def matchPatternKMP(string, pattern):
    if not string or not pattern:
        return None
    p_table = getPartialTable(pattern)
    start, matched = 0, 0
    ls, lp  = len(string), len(pattern)
    stop    = ls - lp + 1
    res     = list()
    while True:
        while matched == lp or string[start + matched] != pattern[matched]:
            if matched == lp:
                res.append(start)
            start   += matched - p_table[matched]
            matched  = max(0, p_table[matched])
            if not start < stop:
```

```python
                return res
        else:
            matched += 1

if __name__ == '__main__':
    string  = 'abababaababacbababacb'
    pattern = 'aaa'
    print 'string:\t\t%s\npattern:\t%s' % (string, pattern)
    print matchPatternKMP(string, pattern)
```

复杂度分析

好了，现在我们知道为什么 KMP 算法很快，也有了具体的实现。但是，它到底有多快呢？换句话说，它的时间复杂度是怎样的呢？

我们先来看算法的主体部分：

```python
1  while True:
2      while matched == lp or string[start + matched] != pattern[matched]:
3          if matched == lp:
4              res.append(start)
5          start   += matched - p_table[matched]
6          matched  = max(0, p_table[matched])
7          if not start < stop:
8              return res
9      else:
10         matched += 1
```

首先注意到，在 `while` 循环内部，算法执行的操作数目是固定的；同时，每次循环失败，都可能执行最多 $m-1$ 次 `matched += 1` 。因此，整个算法的总体复杂度，就取决于 `while` 循环会被执行多少次。而要确定循环执行的次数，就要观察循环变量的初始值、中间变化和终止条件。

无疑，循环的终止条件与 `start` 有关：它从 `0` 开始，每次进入循环体都会自增，直到 `start < stop` 的条件被破坏。因此，整个循环最多被执行 `stop` 次；整个算法最多有 $(n-m)\cdot m$ 次操作。看起来，这是一个复杂度为 $O(n\cdot m)$ 的算法。然而这是一个足够严格的渐进界限吗？答案是否定的，我们需要使用摊还分析来处理这个算法。

所谓摊还分析，就是抓住某一个变量（或者函数）的性质和行为，对零散、杂乱或不规则的执行进行累计，得到比一般方法更严格的上界。在这里，我们观察 `matched` 变量。它具有这样的性质：

- `0 <= matched <= m`；
- 只在第 10 行增加，每次增加 1；
- 只在第 6 行有可能减少。

考虑到循环的终止条件，第 10 行最多被执行 `stop` 次；也就是 `matched` 最多自增 `stop` 次。考虑到 `matched` 必须保证非负，并且每次执行到第 6 行 `matched` 都会至少减小 `1`，所以第 6 行也最多被执行 `stop` 次。这也就是说，整个部分最多有 $2 \cdot \text{stop}$ 次操作。因此，它的时间复杂度不超过 $O(n)$。

同样的，我们可以用摊还分析的方法，分析部分匹配表的算法。

```
1 for curr in xrange(1, lp):
2     ptr = curr
3     while ptr > 0 and pattern[curr] != pattern[res[ptr]]:
4         ptr = res[ptr]
5     else:
6         res.append(res[ptr] + 1)
```

在这里，我们观察 `res` 这个部分匹配表；它具有这样的性质：

- `0<= res[ptr] <= lp`；
- `res[res[ptr]] < res[ptr]`，只在第 4 行出现 `res[ptr]` 当前值减小的情况；
- `res[ptr] <= res[ptr - 1] + 1`，只在第 6 行有可能成立等号。

很眼熟，对吗？这个分析过程和 KMP 算法的主体几乎一模一样。事实上，求得部分匹配表的过程，就是拿模式串自己匹配自己的过程；无怪乎它和算法主体很相似了。同样，考虑到循环的终止条件，求得部分匹配表的过程，复杂度不超过 $O(m)$。

因此，考虑到总是有 $m < n$，整个 KMP 算法的时间复杂度不超过 $O(n)$。

您的鼓励是我写作最大的动力

俗话说，投资效率是最好的投资。 如果您感觉我的文章质量不错，读后收获很大，预计能为您提高 10% 的工作效率，不妨小额捐助我一下，让我有动力继续写出更多好文章。

支付宝扫码 支付

向Liam（**成）付钱

微信扫一扫 支付

向Liam（**成）付钱

撰写评论

写了这么多年博客，收到的优秀评论少之又少。在这个属于 SNS 的时代也并不缺少向作者反馈的渠道。因此，如果你希望撰写评论，请发邮件至我的邮箱并注明文章标题，我会挑选对读者有价值的评论附加到文章末尾。

#Pattern Match #String

‹ 调度场算法

XGBoost 在计算 NDCG 时的特殊处理 ›