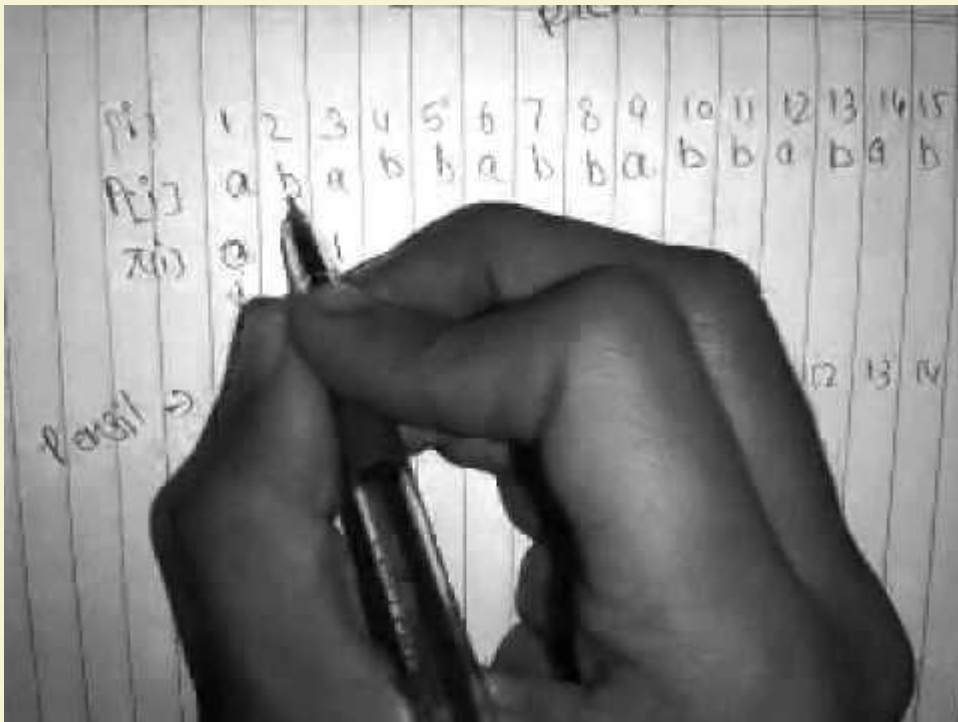


字符串匹配的 KMP 算法

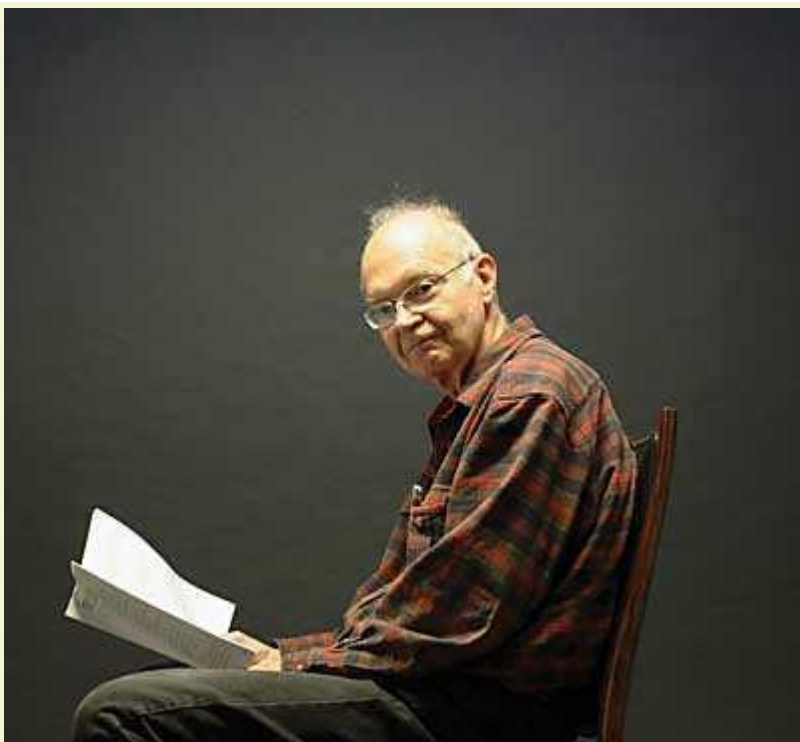
作者：阮一峰

字符串匹配是计算机的基本任务之一。

举例来说，有一个字符串"BBC ABCDAB ABCDABCDABDE"，我想知道，里面是否包含另一个字符串"ABCDABD"？



许多算法可以完成这个任务，Knuth-Morris-Pratt 算法（简称 KMP）是最常用的之一。它以三个发明者命名，起头的那个 K 就是著名科学家 Donald Knuth。



这种算法不太容易理解，网上有很多解释，但读起来都很费劲。直到读到 Jake Boxer 的文章，我才真正理解这种算法。下面，我用自己的语言，试图写一篇比较好懂的 KMP 算法解释。

1.

BBC ABCDAB ABCDABCDABDE
ABCDABD

首先，字符串"BBC ABCDAB ABCDABCDABDE"的第一个字符与搜索词"ABCDABD"的第一个字符，进行比较。因为 B 与 A 不匹配，所以搜索词后移一位。

2.

BBC ABCDAB ABCDABCDABDE
ABCDABD

因为 B 与 A 不匹配，搜索词再往后移。

3.

BBC ABCDAB ABCDABCDABDE
ABCDABD

就这样，直到字符串有一个字符，与搜索词的第一个字符相同为止。

4.

BBC ABCDAB ABCDABCDABDE
ABCDABD

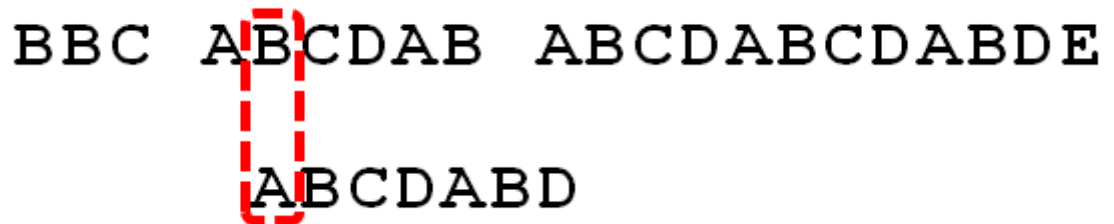
接着比较字符串和搜索词的下一个字符，还是相同。

5.

BBC ABCDAB ABCDABCDABDE
ABCDABD

直到字符串有一个字符，与搜索词对应的字符不相同为止。

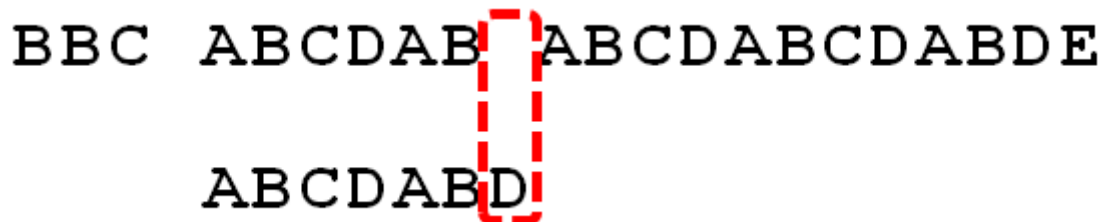
6.



BBC ABCDAB ABCDABCDABDE
ABCDABD

这时，最自然的反应是，将搜索词整个后移一位，再从头逐个比较。这样做虽然可行，但是效率很差，因为你要把"搜索位置"移到已经比较过的位置，重比一遍。

7.



BBC ABCDAB ABCDABCDABDE
ABCDABD

一个基本事实是，当空格与 D 不匹配时，你其实知道前面六个字符是 "ABCDAB"。KMP 算法的想法是，设法利用这个已知信息，不要把"搜索位置"移回已经比较过的位置，继续把它向后移，这样就提高了效率。


8.

搜索词	A	B	C	D	A	B	D
部分匹配值	0	0	0	0	1	2	0

怎么做到这一点呢？可以针对搜索词，算出一张《部分匹配表》（Partial Match Table）。这张表是如何产生的，后面再介绍，这里只要会用就可以了。

9.

BBC ABCDAB ABCDABCDABDE
 ABCDABD



已知空格与 D 不匹配时，前面六个字符"ABCDAB"是匹配的。查表可知，最后一个匹配字符 B 对应的"部分匹配值"为 2，因此按照下面的公式算出向后移动的位数：

$$\text{移动位数} = \text{已匹配的字符数} - \text{对应的部分匹配值}$$

因为 $6 - 2$ 等于 4，所以将搜索词向后移动 4 位。

10.

BBC ABCDAB ABCDABCDABDE
ABCDABD

因为空格与 C 不匹配，搜索词还要继续往后移。这时，已匹配的字符数为 2 ("AB")，对应的"部分匹配值"为 0。所以，移动位数 = 2 - 0，结果为 2，于是将搜索词向后移 2 位。

11.

BBC ABCDAB ABCDABCDABDE
ABCDABD

因为空格与 A 不匹配，继续后移一位。

12.

BBC ABCDAB ABCDABCDABDE
ABCDABD

逐位比较，直到发现 C 与 D 不匹配。于是，移动位数 = 6 - 2，继续将搜索词向后移动 4 位。

13.

BBC ABCDAB ABCDABCDABDE
ABCDABD

逐位比较，直到搜索词的最后一位，发现完全匹配，于是搜索完成。

如果还要继续搜索（即找出全部匹配），移动位数 = $7 - 0$ ，再将搜索词向后移动 7 位，这里就不再重复了。

14.

字符串： "bread"

前缀： b , br , bre , brea

后缀： read , ead , ad , d

下面介绍《部分匹配表》是如何产生的。

首先，要了解两个概念："前缀"和"后缀"。 "前缀"指除了最后一个字符以外，一个字符串的全部头部组合；"后缀"指除了第一个字符以外，一个字符串的全部尾部组合。

15.

搜索词	A	B	C	D	A	B	D
部分匹配值	0	0	0	0	1	2	0

"部分匹配值"就是"前缀"和"后缀"的最长的共有元素的长度。以
"ABCDABD"为例，

- "A"的前缀和后缀都为空集，共有元素的长度为 0；
- "AB"的前缀为[A]，后缀为[B]，共有元素的长度为 0；
- "ABC"的前缀为[A, AB]，后缀为[BC, C]，共有元素的长度 0；
- "ABCD"的前缀为[A, AB, ABC]，后缀为[BCD, CD, D]，共有元素的长度为 0；
- "ABCDA"的前缀为[A, AB, ABC, ABCD]，后缀为[BCDA, CDA, DA, A]，共有元素为"A"，长度为 1；
- "ABCDAB"的前缀为[A, AB, ABC, ABCD, ABCDA]，后缀为[BCDAB, CDAB, DAB, AB, B]，共有元素为"AB"，长度为 2；

— "ABCDABD"的前缀为[A, AB, ABC, ABCD, ABCDA, ABCDAB], 后缀为[BCDABD, CDABD, DABD, ABD, BD, D], 共有元素的长度为 0。

16.

BBC ABCDAB ABCDABCDABDE
ABCDABD

"部分匹配"的实质是, 有时候, 字符串头部和尾部会有重复。比如, "ABCDAB"之中有两个"AB", 那么它的"部分匹配值"就是 2 ("AB"的长度)。搜索词移动的时候, 第一个"AB"向后移动 4 位 (字符串长度-部分匹配值), 就可以来到第二个"AB"的位置。

(完)

留言 (82 条)

Shadowmourne 说:

回想起了高中 NOIP 的日子.....

2013 年 5 月 1 日
19:05 | 档案 | 引用

Mr.king 说:

Robert Sedgewick <https://class.coursera.org/algs4partII-001/lecture/40> 用自动机解释 kmp, 感觉更妙, 就是实现的时候空间复杂度高点。个人感觉 Sedgewick 这个老头讲东西确实牛逼, 很多看书很难理

解的算法，他一讲就明白了，有兴趣的可以去 [part1](#) 看看他讲的红黑树，特别牛逼。

[2013 年 5 月 1 日](#)
[19:34](#) | [档案](#) | [引用](#)

test 说:

有时候，字符串内部会有重复。比如，"ABCDAB"之中有两个"AB"，那么它的"部分匹配值"就是 2 ("AB"的长度)

这个应该说是，字符串头部和尾部有重复？

如果是 ABCABD 这样即使有两个 AB 也不能部分移动吧

[2013 年 5 月 1 日](#)
[19:42](#) | [档案](#) | [引用](#)

hoot 说:

的确简单易懂，可以来一本适用指南啊，想当初看这个算法 话了我好几天去理解 各种数学符号，各种证明 头都大了

[2013 年 5 月 1 日](#)
[19:51](#) | [档案](#) | [引用](#)

阮一峰 说:

引用 test 的发言:

这个应该说是，字符串头部和尾部有重复？

如果是 ABCABD 这样即使有两个 AB 也不能部分移动吧

谢谢指出，我确实没想到这一点，已经更正了。

[2013 年 5 月 1 日](#)
[20:12](#) | [档案](#) | [引用](#)

pi1ot 说:

那就是说需要搜索的词如果内部完全没有重复，那这个算法就退化成了遍历？

[2013 年 5 月 1 日](#)
[20:31](#) | [档案](#) | [引用](#)

Chris 说:

因为空格与 A 不匹配，继续后移一位。

但是这个移动位数 = 已匹配的字符数 - 对应的部分匹配值 = $0 - 0 = 0$

矛盾啊？

[2013 年 5 月 1 日](#)

[20:48](#) | [档案](#) | [引用](#)

hejianchao 说：

原来如此。解释得真清晰！

[2013 年 5 月 1 日](#)

[20:51](#) | [档案](#) | [引用](#)

阮一峰 说：

引用 Chris 的发言：

因为空格与 A 不匹配，继续后移一位。

但是这个移动位数 = 已匹配的字符数 - 对应的部分匹配值 = $0 - 0 = 0$ 矛盾啊？

不是。如果第一个字符就不匹配，搜索词直接比较下一个字符，不用考虑《部分匹配表》。

表中第一列的那个“A”，是指有一个 A 匹配。

[2013 年 5 月 1 日](#)

[21:15](#) | [档案](#) | [引用](#)

t.k. 说：

是我最喜欢的算法之一。

[2013 年 5 月 1 日](#)

[21:16](#) | [档案](#) | [引用](#)

RedNax 说：

我一直以为 KMP 是 string search 最优的算法（时间复杂度是 $O(n+k)$ ），直到我知道了 Boyer-Moore 算法.....才发现原来还有更加巧妙的方法.....

[2013 年 5 月 1 日](#)

[21:40](#) | [档案](#) | [引用](#)

sokoban 说：

引用 pilot 的发言：

那就是说需要搜索的词如果内部完全没有重复，那这个算法就退化成了遍历？

没错，这个算法本质上就是在遍历基础上的一个改进。在最坏的情况下，和没有改进之前一样，甚至严格说还差了一点点，因为每一步都多了比较的开销。

2013 年 5 月 1 日

21:42 | [档案](#) | [引用](#)

Chris 说：

引用 sokoban 的发言：

没错，这个算法本质上就是在遍历基础上的一个改进。在最坏的情况下，和没有改进之前一样，甚至严格说还差了一点点，因为每一步都多了比较的开销。

引用 pilot 的发言：

那就是说需要搜索的词如果内部完全没有重复，那这个算法就退化成了遍历？

应该不是的吧，最坏情况下也是 $o(m+n)$ 的，而遍历是 $o(m*n)$

2013 年 5 月 1 日

21:56 | [档案](#) | [引用](#)

sokoban 说：

引用 Chris 的发言：

应该不是的吧，最坏情况下也是 $o(m+n)$ 的，而遍历是 $o(m*n)$

你说得对。

但是最坏个例的确可能一样。如在

AAAAAAAAAAAAAAAAAAAAAAAAAAAAA 里面找 AB

2013 年 5 月 1 日

22:04 | [档案](#) | [引用](#)

sokoban 说:

引用 sokoban 的发言:

你说得对。

但是最坏个例的确可能一样。如在

AAAAAAAAAAAAAAAAAAAAAAAAAAAAA 里面找 AB

我还是错的。KMP 的确似乎任何时候都要快一些。

[2013 年 5 月 1 日](#)

[22:06](#) | [档案](#) | [引用](#)

sokoban 说:

极端例子: 在 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA 里面找 BB。

这个例子总算对了吧?

[2013 年 5 月 1 日](#)

[22:09](#) | [档案](#) | [引用](#)

sokoban 说:

引用 RedNax 的发言:

我一直以为 KMP 是 string search 最优的算法 (时间复杂度是 $O(n+k)$), 直到我知道了 Boyer-Moore 算法.....才发现原来还有更加巧妙的方法.....

搜索了一下, Boyer-Moore 算法每次从要找的 Pattern 末尾开始比较, 并且一般用两条规则来控制移位 (KMP 只有一条移位规则), 比 KMP 还好。

[2013 年 5 月 1 日](#)

[22:14](#) | [档案](#) | [引用](#)

ballcat 说:

引用 sokoban 的发言:

极端例子: 在 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA 里面找 BB。

这个例子总算对了吧?

不会

尝试匹配第一个,发现是 A,所以要滑移。BB 里面根本没有 A,所以会滑移一个位置。

这是一个经典的势能分析例子。原串的指针只会往后移,目标串的指针只在原串指针往后移时往后移相同的步数,显然目标串指针往前移的步数不会多于往后移的步数。所以加起来所有指针移动的步数不会超过三倍的原串长度。

后缀匹配的平均时间可以是亚线性因为可以根据目标串的特点忽略掉原串的某些部分。但是最坏情况下还是线性的.....

2013 年 5 月 1 日
23:15 | [档案](#) | [引用](#)

zt 说:

看《算法导论》的图 32-10,就比较好理解了。

2013 年 5 月 1 日
23:16 | [档案](#) | [引用](#)

kanaz 说:

最简单解释就是把搜索词自相关一下

2013 年 5 月 2 日
01:04 | [档案](#) | [引用](#)

Henry 说:

引用 kanaz 的发言:

最简单解释就是把搜索词自相关一下

能详细点么

2013 年 5 月 2 日
08:59 | [档案](#) | [引用](#)

王大德 说:

我以前也写了一篇博文,理解 kmp,核心在于 next 数组.我认为不要拘泥于过程,而要用抓住其递归的本质,摘录一部分如下:

理解 next 数组生成的算法花费了一些力气.该算法用了数学归纳法,读起来还有一些递归的意味.

这里再复述一下,求 $\text{next}[i+1]$,就是看 $P[1,i]$ 的最大子串(姑且叫这个吧),如果匹配好说,如果不匹配则继续看这个最大子串的最大子串,重复这个过程,直到算出结果.

2013 年 5 月 2 日

11:06 | [档案](#) | [引用](#)

rian 说:

总结出了两种情况:

1: 移动一位, 如果搜索词第一个字符与目标字符串不匹配就移动一位, 如果搜索词第一个字符与目标字符串匹配但是第二个字符与目标字符串不匹配也是移动一位 (因为可以理解如果搜索词值匹配了第一个字符或者第一个字符都不匹配, 那么展现不出特征无法推理搜索词移动一位是否是没有意义的, 所以只能移动一位)

2: 移动大于一位, 如果搜索词第一个字符和第二个字符都与目标字符串匹配, 那么就展现出了特征, 可以推理, 移动一位是没有意义的, 所以可以尝试移动 2 位甚至更多

给博主一个建议:

觉得应该可以在部分匹配表再加一行移动位数的数据, 以后直接查询移动位数就可以了, 不要临时再算

2013 年 5 月 2 日

12:26 | [档案](#) | [引用](#)

V 客小站 说:

各位有没有在实际的项目中用过 KMP 算法呢?

2013 年 5 月 2 日

22:18 | [档案](#) | [引用](#)

pilot 说:

引用 V 客小站的发言:

各位有没有在实际的项目中用过 KMP 算法呢?

没有，我遇到过的实际需求更多是对一段长本文进行词表匹配或者替换，词表本身很大，但是每个词条比较短，类似于 editor keyword highlight 场景

2013 年 5 月 3 日

15:52 | [档案](#) | [引用](#)

nklike 说:

楼主的这个算法感觉和这里面的例子第二步不能够对应上去

<http://www.cs.utexas.edu/~moore/best-ideas/string-searching/kpm-example.html#step02>

按理说

example 的表应该是

0000001

ER 和 EX 不匹配，按照楼主的说法应该是移动一位，但是例子里面直接移动了 2 位

应该是已经匹配的个数 2 减去 EX 对应的部分匹配值 0，而不是 1-0

2013 年 5 月 3 日

16:18 | [档案](#) | [引用](#)

阮一峰 说:

@nklike:

是跟他不一样。

那里的理由是 R 不出现在 EX 里，所以可以移 2 位。我觉得那不是 KMP 算法，而是 Boyer-Moore 算法的“坏字符规则”。

2013 年 5 月 3 日

18:47 | [档案](#) | [引用](#)

biaobiaoqi 说:

最近时常从各个信息源获取到 ruanyifeng 的博客。小弟我也郑在自己学着积累思考，写博客。从博主这里学习了很多。

2013 年 5 月 3 日

21:09 | [档案](#) | [引用](#)

hbprotoss 说:

感谢 LZ，这篇文章确实比较容易理解

2013 年 5 月 4 日
09:04 | [档案](#) | [引用](#)

周浩楠 说:

有醍醐灌顶的感觉。

2013 年 5 月 4 日
09:44 | [档案](#) | [引用](#)

longsail 说:

部分匹配值"就是"前缀"和"后缀"的最长的共有元素的长度，这里以 AACDAA 为例，

"AACDAA"的前缀为[A, AA, AAC, AACD, AACDA]，后缀为 [A,AA,DAA,CDAA,ACDAA]，共有元素为"A","AA"，这时取最长字符串的长度？

2013 年 5 月 4 日
10:07 | [档案](#) | [引用](#)

Melo618 说:

移动位数 = 已匹配的字符数 - 最后一个匹配字符对应的部分匹配值
这样更准确

2013 年 5 月 4 日
12:03 | [档案](#) | [引用](#)

阮一峰 说:

@longsail:

最长字符串的长度是 2。

2013 年 5 月 4 日
12:14 | [档案](#) | [引用](#)

YangZhixuan 说:

初学 KMP 的时候，一直不得要领。后来学习 AC 自动机的时候，一下子明白了 KMP 实际上是 AC 自动机的特殊情况。

2013 年 5 月 6 日
16:37 | 档案 | 引用

Daniel 说:

难道是我的错觉，第 12 步:

12.

逐位比较，直到发现 C 与 D 不匹配。于是，移动位数 = 6 - 2，继续将搜索词向后移动 4 位。

但是

搜索词 A B C D A B D

部分匹配值 0 0 0 0 1 2 0

移动位数 = 已匹配的字符数 - 对应的部分匹配值

难道不是应该 移动位数 = 6 - 0 吗？最后一个 D 的匹配值是 0 啊。

2013 年 5 月 7 日
09:27 | 档案 | 引用

Daniel 说:

Sorry, 我没有看到第 16 步。

2013 年 5 月 7 日
09:32 | 档案 | 引用

Daniel 说:

Sorry, 看东西一直囫圇吞枣，仔细的看了几遍，又码了下代码，终于完全明白了。非常感谢。抱歉前面两次的留言。

2013 年 5 月 7 日
10:40 | 档案 | 引用

红色石头 说:

非常不错，简介易懂，我转载了，并评价为：图文并茂，生动形象易懂。

<http://50vip.com/blog.php?i=228>

2013 年 5 月 9 日

15:03 | 档案 | 引用

callmestring 说:

算法的关键在于 next 数组的生成,用动态规范法生成。

比如: `str = abcabd`, next 初始化为: `[0,0,0,0,0,0]`;

已知第 0 个字符 a 没有任何相同的前后缀, 则 `next[0] = 0`。

加入第 1 个字符, 则前面已知的最长公共前后缀长度为 `next[0]`, 此时如果 `str[next[0]]` 与 `str[1]` 相等, 就可知道 `next[1]=next[0]+1`, 如不相等则可直接判定 `next[1]=0`; 这里 `str[next[0]]!=str[1]`, 故 `next[1]=0`。

..., `next[2]=0`。

..., `next[3]=next[2]+1=1`。

..., `next[4]=next[3]+1=2`。

..., `next[5]=0`。

最后, `next=[0,0,0,1,2,0]`;

不知道 KMP 里面是否是这样生成的?

2013 年 5 月 10 日

21:30 | 档案 | 引用

bibodeng 说:

能把复杂的东西讲清楚, 说明真的理解透彻了。看了这篇真的清晰很多, 之前看《算法导论》, 觉得看过就行了, 其实是很多不懂, 后来又忘了。

2013 年 5 月 11 日

21:22 | 档案 | 引用

Raw 说:

kmp 算法主要是失效函数的这个表如何计算, 其实质就是一个变形的状态机, 用典型的数学归纳法来计算此表, 有一个初始状态 `f(1)=0`, 然后迭代出所有位置上的失效值。

这里只说清楚了如何用失效函数来解决字符串匹配问题, 但并没有给出如何计算失效函数的方法。

另外 kmp 算法对字母类语言理论效率比价高，但是对汉语这种没有太多前缀字符串和后缀字符串重合的语言，其实效率和最普通的匹配没太大差别。

2013 年 5 月 12 日

11:10 | 档案 | 引用

starshine 说:

省略了非常重要的一个问题，

就是部分匹配表的算法问题，如何得到？

只是给出原理是不够的。

另外，对于 kmp 算法正确性的解释，明显不够。

如果真心研究这个问题，还是需要去看看更想尽的解释，不要怕麻烦和困难。

看这个，只能了解表象。

2013 年 5 月 14 日

11:34 | 档案 | 引用

starshine 说:

@callmestring:

这个解释貌似不错。看 next 函数的生成代码，非常简洁，但是理解起来相当的不容易，不明白其推导过程。

2013 年 5 月 14 日

11:38 | 档案 | 引用

吹风 说:

如果是比较中文字符串怎么做呢？

2013 年 5 月 17 日

17:32 | 档案 | 引用

radix 说:

@callmestring:

对于 `str=[A,C,A,A]` 的 `str[3]` 就不正确了。

2013 年 5 月 18 日

16:34 | 档案 | 引用

tom 说:

引用 Mr. king 的发言:

Robert Sedgewick <https://class.coursera.org/algs4partII-001/lecture/40> 用自动机解释 kmp, 感觉更妙, 就是实现的时候空间复杂度高点。个人感觉 Sedgewick 这个老头讲东西确实牛逼, 很多看书很难理解的算法, 他一讲就明白了, 有兴趣的可以去 part1 看看他讲的红黑树, 特别牛逼。

Robert Sedgewick 这个实际上是 string-matching automaton, 需要预算这个 pattern 的整个字符集的自动机, 个人感觉并不是 kmp。

2013 年 5 月 25 日
09:21 | [档案](#) | [引用](#)

于梓峤 说:

部分匹配值的概念为什么要提出来? 为什么又要这么定义? 抱歉看不大懂, 不过对于 KMP 算法大概是明白了

2013 年 5 月 26 日
10:51 | [档案](#) | [引用](#)

DOVE 说:

引用 radix 的发言:

@callmestring:

对于 str=[A,C,A,A]的 str[3]就不正确了。

就是, str = "acaa",照这样算, 在 next[3]就不对了!!!

2013 年 5 月 30 日
22:40 | [档案](#) | [引用](#)

xxx 说:

next 值求得不对

2013 年 6 月 24 日
10:58 | [档案](#) | [引用](#)

xiaolong 说:

解释的太清晰了, 谢谢啦

2013 年 6 月 29 日

21:24 | [档案](#) | [引用](#)

acmerfight 说:

按照你的做法

ababb 的 next 数组是什么呢？不对吧

2013 年 7 月 2 日

10:59 | [档案](#) | [引用](#)

J 说:

做 OJ 题目的时候恰好用到，现在终于懂了 KMP 是怎么回事了~^ ^

2013 年 7 月 22 日

16:04 | [档案](#) | [引用](#)

刘玉 说:

为什么按照算法导论上的例子“ababababca”，再通过移动位数的公式得不到书上的那个数组呢？

2013 年 7 月 31 日

21:17 | [档案](#) | [引用](#)

楚轩 说:

感谢您的文章，真的写的很好，3Q！

2013 年 8 月 10 日

15:00 | [档案](#) | [引用](#)

serikaoo 说:

看的第三篇关于 KMP 的，直接懂了，写得比其它两篇简单多了。

其实那两篇根本没认真看完，太长了，感觉对于有点计算机基础的人来说，KMP 也不是一个需要那么多文字来描述的算法。

喜欢博主这种简单易懂的说明方式。

2013 年 8 月 12 日

10:27 | [档案](#) | [引用](#)

郭春阳 说:

引用 test 的发言:

有时候，字符串内部会有重复。比如，"ABCDAB"之中有两个"AB"，那么它的"部分匹配值"就是 2（"AB"的长度）
这个应该说是，字符串头部和尾部有重复？
如果是 ABCABD 这样即使有两个 AB 也不能部分移动吧

如果你匹配结束时，此时匹配的是 ABCAB，此时查表肯定是 2 啊。
你仔细理解下后缀。后缀不是整个字符串的后缀，而是已经匹配部分的后缀。

2013 年 8 月 23 日
00:49 | [档案](#) | [引用](#)

郭春阳 说：

最喜欢的博主的文章，通俗易懂，这才是真正的为人民服务。

2013 年 8 月 23 日
00:51 | [档案](#) | [引用](#)

花开献佛堂 说：

解释的不错，终于看懂了，谢谢

2013 年 8 月 25 日
19:26 | [档案](#) | [引用](#)

奔跑的猫 说：

部分匹配值都算出来了 为什么不算出移动值表呢

2013 年 9 月 5 日
14:07 | [档案](#) | [引用](#)

ZYY 说：

真的很清楚易懂啊，谢谢整理

2013 年 9 月 18 日
15:42 | [档案](#) | [引用](#)

gsj 说：

我在网上看了好多，都不怎么懂，看了这个，恍然大悟。

2013 年 10 月 24 日
19:30 | [档案](#) | [引用](#)

charmingcui 说：

之前看了很多博客都不理解，看了楼主的解释忽然间恍然大悟，谢谢楼主！

2013 年 11 月 18 日

15:52 | 档案 | 引用

needle 说:

```
大神,
我刚学三天的 python.
碰巧看到了你的文章后,
我用三个多小时写出 KMP 啦!
会不会很蠢....

origin_str's length is: 23 match_pattern_str'length is: 7
-----origin_Str_index is: 0
-----origin_Str_index is: 1
-----origin_Str_index is: 2
-----origin_Str_index is: 3
-----origin_Str_index is: 4
length_matched is: 1
length_matched is: 2
length_matched is: 3
length_matched is: 4
length_matched is: 5
length_matched is: 6
now, pointer skips 4 index according to KMP
-----origin_Str_index is: 8
length_matched is: 1
length_matched is: 2
now, pointer skips 2 index according to KMP
-----origin_Str_index is: 10
-----origin_Str_index is: 11
length_matched is: 1
length_matched is: 2
```



```
length_matched is: 3
length_matched is: 4
length_matched is: 5
length_matched is: 6
now, pointer skips 4 index according to KMP
-----origin_Str_index is: 15
length_matched is: 1
length_matched is: 2
length_matched is: 3
length_matched is: 4
length_matched is: 5
length_matched is: 6
length_matched is: 7
position is at: 15
True
```

2013 年 11 月 27 日
00:41 | [档案](#) | [引用](#)

KG 说:

很有用，必须要留言，顶楼主！

2013 年 12 月 27 日
15:20 | [档案](#) | [引用](#)

HZ 说:

非常好的文章，受益匪浅。请教一下：作者那个图是怎么画出来的？我指的是图 1 到 16，尤其是上下对齐的红虚线框。谢谢。

2014 年 1 月 16 日
08:58 | [档案](#) | [引用](#)

zhenxing1114 说:

引用 HZ 的发言:

非常好的文章，受益匪浅。请教一下：作者那个图是怎么画出来的？我指的是图 1 到 16，尤其是上下对齐的红虚线框。谢谢。

可以在 ppt、word 里写好，然后截图吧，^_^

2014 年 1 月 22 日
22:59 | [档案](#) | [引用](#)

zhenxing1114 说:

这篇看上去好懂多了，回头再结合其他的文章仔细看看。

2014 年 1 月 22 日
23:01 | [档案](#) | [引用](#)

好人 说:

有空的话把 next 数组的算法也讲讲吧

2014 年 2 月 24 日
14:49 | [档案](#) | [引用](#)

JackieZhu 说:

引用 pilot 的发言:

那就是说需要搜索的词如果内部完全没有重复，那这个算法就退化成了遍历？

完全没有重复也是 $O(n)$ 的啊

2014 年 3 月 17 日
23:19 | [档案](#) | [引用](#)

轻抚麦浪 说:

寥寥数语，便将 KMP 算法的精髓尽数勾勒了出来，很好！

2014 年 3 月 28 日
17:10 | [档案](#) | [引用](#)

jihite 说:

我用 c++ 实现了下，这里贴不开，去我的博客看吧。

<http://www.cnblogs.com/kaituorenseng/p/3633700.html>

2014 年 3 月 30 日
12:13 | [档案](#) | [引用](#)

vivi 单 说:

昨天开始研究这个算法一直不得要领，直到看到这篇博文。不能更赞。接下来结合其他书籍和相关资料深入理解，然后码码代码。

2014 年 4 月 3 日

16:15 | [档案](#) | [引用](#)

Tim 说:

一峰的解释从来都是简明直白，能把复杂的东西清晰简洁的表述出来，而且还方便记忆。

KMP 算法的前缀后缀的匹配理论真是优美。

2014 年 4 月 20 日

22:05 | [档案](#) | [引用](#)

woiavril 说:

引用 test 的发言:

有时候，字符串内部会有重复。比如，"ABCDAB"之中有两个"AB"，那么它的"部分匹配值"就是 2（"AB"的长度）

这个应该说是，字符串头部和尾部有重复？

如果是 ABCABD 这样即使有两个 AB 也不能部分移动吧

你确定你说的对，按博主的理解是前缀是除字符串最后一个字符的所有组合，而后缀是除最前面一个字符的所有组合，如果是 ABCABD,前缀的组合有{ABCAB,ABCA,ABC,AB,A},后缀有{BCABD,CABD,ABD,BD},这两个集合根本没有重复的，所有他的部分匹配值为 0；

2014 年 5 月 14 日

13:27 | [档案](#) | [引用](#)

zuoxinyu 说:

正在练习最简单的遍历匹配，直觉还有更高效的算法，这个虽然看不大懂，但是给了很大启发。

2014 年 5 月 16 日

22:03 | [档案](#) | [引用](#)

Zhenghui 说:

博主太 nb 了！KMP 从头看下来，几分钟，就看懂了！

2014 年 5 月 22 日

03:15 | 档案 | 引用

Alex_Huang 说:

很受用，简单易懂，谢谢楼主分享。

2014 年 6 月 28 日

23:07 | 档案 | 引用

wzz 说:

如果按照这样的规则判断移动位数：移动位数 = 已匹配的字符数 - 对应的部分匹配值，

那第一位 A 如果不匹配应该如何计算移动位数呢？

移动位数 = 0（未匹配任何字符） - 0，结果还是 0，这样的话第一位不匹配就不用移动了，显然是不对的。所以是不是把 A 的部分匹配值变为 -1 更合适？

谢谢

2014 年 6 月 30 日

15:30 | 档案 | 引用

丁松 说:

记得大学时，老师讲过，很麻烦，不过我想计算机执行起来是不会介意

2014 年 7 月 8 日

18:19 | 档案 | 引用

小强 说:

膜拜楼主，楼主很厉害

2014 年 7 月 25 日

14:49 | 档案 | 引用

wkingfly 说:

第 10 步，后移 4 位之后，第一对儿 AB 和第二对 AB 重合，C 对着一个空，那么这是我们还要从第一个 AB 开始比较么，比较和母串对 AB 相等，然后到 C 和空格不相等，是不是应该直接从 C 开始对比呢？

我用 python 实现了下，<https://github.com/wkingfly/kmp> 多多指教

2014 年 8 月 7 日

23:23 | 档案 | 引用

简凡 说:

好像有点漏洞:

比如:

ABCABDABCABC

ABCABC

因为 C 和 D 不匹配, 按照文中的公式, 应右移 $5-2=3$ 位。其实可以直接移动 6 位。

2014 年 8 月 8 日

09:40 | 档案 | 引用