維基百科

KMP算法

维基百科，自由的百科全书

> 在本文中，将使用始于零的数组来表示字符串。比如，若字符串S = "ABC"，则S[2]表示字符'C'。这种表示方法与C语言一致。

在计算机科学中，**Knuth-Morris-Pratt字符串查找算法**（简称为**KMP算法**）可在一个字符串S内查找一个词W的出现位置。一个词在不匹配时本身就包含足够的信息来确定下一个匹配可能的开始位置，此算法利用这一特性以避免重新检查先前匹配的字符。

这个算法由高德纳和沃恩·普拉特在1974年构思，同年詹姆斯·H·莫里斯也独立地设计出该算法，最终三人于1977年联合发表。

目录

查找过程

以W="ABCDABD"，S="ABC ABCDAB ABCDABCDABDE"为例说明查找过程。查找过程同时使用两个循环变量m和i：

- m代表主文字符串S内匹配字符串W的当前查找位置，
- i代表匹配字符串W当前做比较的字符位置。

图示如下：

```
           1         2
m: 012345678901234567890123
S: ABC ABCDAB ABCDABCDABDE
W: ABCDABD
i: 0123456
```

從W與S的開頭比較起。比對到S[3](=' ')時，發現W[3](='D')與之不符。接著並不是從S[1]比較下去。已經知道S[1]~S[3]不與W[0]相合。因此，略過這些字元，令m = 4以及i = 0。

```
           1         2
m: 012345678901234567890123
S: ABC ABCDAB ABCDABCDABDE
W:     ABCDABD
i:     0123456
```

如上所示，檢核了"ABCDAB"這個字串。然而，下一字符便不相合。可以注意到，"AB"在"ABCDAB"的頭尾處均有出現。這意味著尾端的"AB"可以作為下次比較的起始點。因此，令`m = 8, i = 2`，繼續比較。圖示如下：

```
              1         2
m: 012345678901234567890 12
S: ABC ABCDAB ABCDABCDABDE
W:         ABCDABD
i:         0123456
```

於`m = 10`的地方，又出現不相符的情況。類似地，令`m = 11, i = 0`繼續比較：

```
              1         2
m: 0123456789012345678901 2
S: ABC ABCDAB ABCDABCDABDE
W:            ABCDABD
i:            0123456
```

這時，`S[17](='C')`不與`W[6]`相同，但是已匹配部分"ABCDAB"亦為首尾均有"AB"，採取一貫的作法，令`m = 15`和`i = 2`，繼續搜尋。

```
              1         2
m: 0123456789012345 6789012
S: ABC ABCDAB ABCDABCDABDE
W:                ABCDABD
i:                0123456
```

找到完全匹配的字串了，其起始位置於`S[15]`的地方。

部分匹配表

部分匹配表，又称为**失配函数**，作用是让算法无需多次匹配S中的任何字符。能够实现线性时间搜索的关键是在主串的一些字段中检查模式串的*初始字段*，可以确切地知道在当前位置之前的一个潜在匹配的位置。换句话说，在不错过任何潜在匹配的情况下，"预搜索"这个模式串本身并将其译成一个包含所有可能失配的位置对应可以绕过最多无效字符的列表。

对于`W`中的任何位置，都希望能够查询那个位置前（不包括那个位置）有可能的`W`的最长初始字段的长度，而不是从`W[0]`开始失配的整个字段，这长度就是查找下一个匹配时回退的距离。因此`T[i]`是`W`的可能的*适当*初始字段同时也是结束于`W[i - 1]`的子串的最大长度。使空串长度是0。当一个失配出现在模式串的最开始，这是特殊情况（无法回退），设置`T[0] = -1`，在下面讨论。

创建表算法示例

以`W = "ABCDABD"`为例。以下将看到，部分匹配表的生成过程与前述查找过程大同小异，且出于类似原因是高效的。

首先，设定`T[0] = -1`。为求出`T[1]`，必须找到一个"A"的真后缀（真后缀指不等于原串的后缀）兼`W`的前缀。但"A"没有真后缀，所以设定`T[1] = 0`。类似地，`T[2] = 0`。

继续到`T[3]`，注意到检查**所有**后缀有一个捷径：假设存在符合条件的前后缀，两者分别为`W[0..1] = W[1..2]`，则必有`W[0..0] = W[1..1]`。由于`W[0..0]`亦是`W`的真前缀，上一步必然已经得到`T[2] = 1`（而有`T[2] = 0`，说明假设不成立）。一般地，遍历到每个字符时，只有上一步已经发现一个长为m的有效后缀，才需要判断有无长为m+1的后缀，而毋需考虑长为m+2、m+3等的后缀。

从而，不必考虑长为2的后缀，而唯独需要考虑的长度1亦不可行，故得到`T[3]=0`。

接下来是`W[4] = 'A'`。基于同样的理由，需要考虑的最大长度为1，并且在`'A'`这个情况中有效，回退到寻找的当前字符*之前*的字段，因此`T[4] = 0`。

现在考虑下一个字符W[5] = 'B'，使用这样的逻辑：如果曾发现一个子模式在上一个字符W[4]之前出现，继续到当前字符W[5]，那么在它之前它本身会拥有一个结束于W[4]合适的初始段，与事实相反的是已经找到'A'是最早出现在结束于W[4]的合适字段。因此为了找到W[5]的终止串，不需要查看W[4]。因此T[5] = 1。

最后到W[4] = 'A'。下一个字符是'B'，并且这也确实是W[5]。此外，上面的相同参数说明为了查找W[6]的字段，不需要向前查看W[4]，所以得出T[6] = 2。

于是得到下面的表：

i	0	1	2	3	4	5	6
W[i]	A	B	C	D	A	B	D
T[i]	-1	0	0	0	0	1	2

另一个更复杂和有趣的例子：

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
W[i]	P	A	R	T	I	C	I	P	A	T	E		I	N		P	A	R	A	C	H	U	T	E
T[i]	-1	0	0	0	0	0	0	0	1	2	0	0	0	0	0	0	1	2	3	0	0	0	0	0

建立表算法的伪代码的解释

上面的例子以最少的复杂步骤展示了组织这个表格的一般性方法。这么做的原理是对整体的搜索：大多数工作已经在检测到当前位置的时候做完了，剩下需要做的很少。略微复杂的一点是找到一个共同前后缀。这就需要有一些初始化的代码。

```
algorithm kmp_table:
    input:
        an array of characters, W (the word to be analyzed)
        an array of integers, T (the table to be filled)
    output:
        nothing (but during operation, it populates the table)

    define variables:
        an integer, pos ← 2 (the current position we are computing in T)
        an integer, cnd ← 0 (the zero-based index in W of the next
character of the current candidate substring)

    (the first few values are fixed but different from what the algorithm
might suggest)
    let T[0] ← -1, T[1] ← 0

    while pos < length(W) do
        (first case: the substring continues)
        if W[pos - 1] = W[cnd] then
            let cnd ← cnd + 1, T[pos] ← cnd, pos ← pos + 1

        (second case: it doesn't, but we can fall back)
        else if cnd > 0 then
            let cnd ← T[cnd]

        (third case: we have run out of candidates.  Note cnd = 0)
        else
            let T[pos] ← 0, pos ← pos + 1
```

建立表的算法的效率

建立表的算法的复杂度是$O(n)$，其中n是W的长度。

除去一些初始化的工作，所有工作都在循环中完成。为说明循环执行用了$O(n)$的时间，考虑pos和pos - cnd的大小。

- 在第一个分支里，pos - cnd不变，而pos与cnd同时自增，自然，pos增加了。

- 在第二个分支里，cnd被更小的T[cnd]所替代，从而增加了pos - cnd。
- 在第三个分支里，pos增加了，而cnd不变，所以pos和pos - cnd都增加了。

因为pos ≥ pos - cnd，即在每一个阶段要么pos增加，要么pos的一个下界增加。故既然算法在pos = n时终止，此循环必然在最多$2n$次迭代后终止。因此建立表的算法的复杂度是$O(n)$。

另见

- Boyer-Moore字符串搜索算法

外部連結

- （英文）An explanation of the algorithm (http://www.ics.uci.edu/~eppstein/161/960227.html)（页面存档备份 (https://web.archive.org/web/20210116080759/http://www.ics.uci.edu/~eppstein/161/960227.html)，存于互联网档案馆） and sample C++ code (http://www.ics.uci.edu/~eppstein/161/kmp/)（页面存档备份 (https://web.archive.org/web/20201004125500/http://www.ics.uci.edu/~eppstein/161/kmp/)，存于互联网档案馆） by David Eppstein
- （英文）Knuth-Morris-Pratt algorithm (http://www-igm.univ-mlv.fr/~lecroq/string/node8.html)（页面存档备份 (https://web.archive.org/web/20210125110301/http://www-igm.univ-mlv.fr/~lecroq/string/node8.html)，存于互联网档案馆） description and C code by Christian Charras and Thierry Lecroq
- （英文）Interactive animation for Knuth-Morris-Pratt algorithm (https://web.archive.org/web/20090310031611/http://www.ics.uci.edu/~goodrich/dsa/11strings/demos/pattern/) by Mike Goodrich
- （英文）Explanation of the algorithm from scratch (http://www.inf.fh-flensburg.de/lang/algorithmen/pattern/kmpen.htm)（页面存档备份 (https://web.archive.org/web/20200121185531/http://www.inf.fh-flensburg.de/lang/algorithmen/pattern/kmpen.htm)，存于互联网档案馆） by FH Flensburg.

引用

- 高德纳; James H. Morris, Jr, Vaughan Pratt. Fast pattern matching in strings. SIAM Journal on Computing. 1977, **6** (2): 323–350 [2006-07-27].（原始内容存档于2010-01-04）.
- Thomas H. Cormen; Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. Section 32.4: The Knuth-Morris-Pratt algorithm. Introduction to Algorithms Second edition. MIT Press and McGraw-Hill. 2001: 923–931. ISBN 978-0-262-03293-3.