Knuth–Morris–Pratt algorithm

The **Knuth–Morris–Pratt string searching algorithm** (or **KMP algorithm**) searches for occurrences of a "word" W within a main "text string" S by employing the observation that when a mismatch occurs, the word itself embodies sufficient information to determine where the next match could begin, thus bypassing re-examination of previously matched characters.

The algorithm was conceived by Donald Knuth and Vaughan Pratt and independently by James H. Morris in 1977, but the three published it jointly.

NOTE: *This article uses zero-based arrays to represent strings; thus the 'C' in S={'A','B','C'} is denoted S[2].*

KMP algorithm

Worked example of the search algorithm

To illustrate the algorithm's details, we work through a (relatively artificial) run of the algorithm. At any given time, the algorithm is in a state determined by two integers, m and i, which denote respectively the position within S which is the beginning of a prospective *match* for W, and the *index* in W denoting the character currently under consideration. This is depicted, at the start of the run, like 1 2 m: 01234567890123456789012 S: ABC ABCDAB ABCDABCDABDE W: ABCDABD i: 0123456 We proceed by comparing successive characters of W to "parallel" characters of S, moving from one to the next if they match. However, in the fourth step, we get S[3] is a space and W[3] = 'D', a mismatch. Rather than beginning to search again at S[1], we note that no 'A' occurs between positions 0 and 3 in S except at 0; hence, having checked all those characters previously, we know there is no chance of finding the beginning of a match if we check them again. Therefore we move on to the next character, setting m = 4 and i = 0. (m will first become 3 since m + i - T[i] = 0 + 3 - 0 = 3 and then become 4 since T[0] = -1) 1 2 m: 01234567890123456789012 S: ABC ABCDAB ABCDABCDABDE W: ABCDABD i: 0123456 We quickly obtain a nearly complete match "ABCDAB" when, at W[6] (S[10]), we again have a discrepancy. However, just prior to the end of the current partial match, we passed an "AB" which could be the beginning of a new match, so we must take this into consideration. As we already know that these characters match the two characters prior to the current position, we need not check them again; we simply reset m = 8, i = 2 and continue matching the current character. Thus, not only do we omit previously matched characters of S, but also previously matched characters of W. 1 2 m: 01234567890123456789012 S: ABC ABCDAB ABCDABCDABDE W: ABCDABD i: 0123456 This search fails immediately, however, as the pattern still does not contain a space, so as in the first trial, we return to the beginning of W and begin searching at the next character of S: m = 11, reset i = 0. 1 2 m: 01234567890123456789012 S: ABC ABCDAB ABCDABCDABDE W: ABCDABD i: 0123456 Once again we immediately hit upon a match "ABCDAB" but the next character, 'C', does not match the final character 'D' of the word W. Reasoning as before, we set m = 15, to start at the two-character string "AB" leading up to the current position, set i = 2, and continue matching from the current position. 1 2 m: 01234567890123456789012 S: ABC ABCDAB ABCDABCDABDE W: ABCDABD i: 0123456 This time we are able to complete the match, whose first character is S[15].

Description of and pseudocode for the search algorithm

The above example contains all the elements of the algorithm. For the moment, we assume the existence of a "partial match" table T, described below, which indicates where we need to look for the start of a new match in the event that the current one ends in a mismatch. The entries of T are constructed so that if we have a match starting at S[m] that fails when comparing S[m + i] to W[i], then the next possible match will start at index m + i - T[i] in S (that is, T[i] is the amount of "backtracking" we need to do after a mismatch). This has two implications: first, T[0] = -1, which indicates that if W[0] is a mismatch, we cannot backtrack and must simply check the next character; and second, although the next possible match will *begin* at index m + i - T[i], as in the example above, we need not actually

check any of the T[i] characters after that, so that we continue searching from W[T[i]]. The following is a sample pseudocode implementation of the KMP search algorithm.

```
algorithm kmp_search:
    input:
        an array of characters, S (the text to be searched)
        an array of characters, W (the word sought)
    output:
        an integer (the zero-based position in S at which W is found)

    define variables:
        an integer, m ← 0 (the beginning of the current match in S)
        an integer, i ← 0 (the position of the current character in W)
        an array of integers, T (the table, computed elsewhere)

    while m+i is less than the length of S, do:
        if W[i] = S[m + i],
            if i equals the (length of W)-1,
                return m
            let i ← i + 1
        otherwise,
            let m ← m + i - T[i],
            if T[i] is greater than -1,
                let i ← T[i]
            else
                let i ← 0

    (if we reach here, we have searched all of S unsuccessfully)
    return the length of S
```

Efficiency of the search algorithm

Assuming the prior existence of the table T, the search portion of the Knuth–Morris–Pratt algorithm has complexity O(k), where k is the length of S and the O is big-O notation. As except for the fixed overhead incurred in entering and exiting the function, all the computations are performed in the **while** loop, we will calculate a bound on the number of iterations of this loop; in order to do this we first make a key observation about the nature of T. By definition it is constructed so that if a match which had begun at S[m] fails while comparing S[m + i] to W[i], then the next possible match must begin at S[m + (i - T[i])]. In particular the next possible match must occur at a higher index than m, so that $T[i] < i$.

Using this fact, we will show that the loop can execute at most 2k times. For in each iteration, it executes one of the two branches in the loop. The first branch invariably increases i and does not change m, so that the index m + i of the currently scrutinized character of S is increased. The second branch adds i - T[i] to m, and as we have seen, this is always a positive number. Thus the location m of the beginning of the current potential match is increased. Now, the loop ends if m + i = k; therefore each branch of the loop can be reached at most k times, since they respectively increase either m + i or m, and $m \leq m + i$: if m = k, then certainly $m + i \geq k$, so that since it increases by unit increments at most, we must have had m + i = k at some point in the past, and therefore either way we would be done.

Thus the loop executes at most 2k times, showing that the time complexity of the search algorithm is O(k).

Here is another way to think about the runtime: Let us say we begin to match W and S at position i and p, if W exists as a substring of S at p, then W[0 through m] == S[p through p+m]. On succeed, that is, the word and the text matched at the positions(W[i] == S[p+i]), we increase i by 1 (i++). On fail, that is, the word and the text does not match at the positions(W[i] != S[p+i]), the text pointer is kept still, while the word pointer roll-back a certain amount(i = T[i], where T is the jump table) And we attempt to match W[T[i]] with S[p+i]. The maximum number of roll-back of i is bounded by i, that is to say, for any failure, we can only roll-back as much as we have progressed up to the failure. Then it is clear the runtime is 2k.

"Partial match" table (also known as "failure function")

The goal of the table is to allow the algorithm not to match any character of S more than once. The key observation about the nature of a linear search that allows this to happen is that in having checked some segment of the main string against an *initial segment* of the pattern, we know exactly at which places a new potential match which could continue to the current position could begin prior to the current position. In other words, we "pre-search" the pattern itself and compile a list of all possible fallback positions that bypass a maximum of hopeless characters while not sacrificing any potential matches in doing so.

We want to be able to look up, for each position in W, the length of the longest possible initial segment of W leading up to (but not including) that position, other than the full segment starting at W[0] that just failed to match; this is how far we have to backtrack in finding the next match. Hence T[i] is exactly the length of the longest possible *proper* initial segment of W which is also a segment of the substring ending at W[i - 1]. We use the convention that the empty string has length 0. Since a mismatch at the very start of the pattern is a special case (there is no possibility of backtracking), we set T[0] = -1, as discussed above.

Worked example of the table-building algorithm

We consider the example of W = "ABCDABD" first. We will see that it follows much the same pattern as the main search, and is efficient for similar reasons. We set T[0] = -1. To find T[1], we must discover a proper suffix of "A" which is also a prefix of W. But there are no proper suffixes of "A", so we set T[1] = 0. Likewise, T[2] = 0.

Continuing to T[3], we note that there is a shortcut to checking *all* suffixes: let us say that we discovered a proper suffix which is a proper prefix and ending at W[2] with length 2 (the maximum possible); then its first character is a proper prefix of a proper prefix of W, hence a proper prefix itself, and it ends at W[1], which we already determined cannot occur in case T[2]. Hence at each stage, the shortcut rule is that one needs to consider checking suffixes of a given size m+1 only if a valid suffix of size m-1 was found at the previous stage (e.g. T[x]=m-1).

Therefore we need not even concern ourselves with substrings having length 2, and as in the previous case the sole one with length 1 fails, so T[3] = 0.

We pass to the subsequent W[4], 'A'. The same logic shows that the longest substring we need consider has length 1, and although in this case 'A' *does* work, recall that we are looking for segments ending *before* the current character; hence T[4] = 0 as well.

Considering now the next character, W[5], which is 'B', we exercise the following logic: if we were to find a subpattern beginning before the previous character W[4], yet continuing to the current one W[5], then in particular it would itself have a proper initial segment ending at W[4] yet beginning before it, which contradicts the fact that we already found that 'A' itself is the earliest occurrence of a proper segment ending at W[4]. Therefore we need not look before W[4] to find a terminal string for W[5]. Therefore T[5] = 1.

Finally, we see that the next character in the ongoing segment starting at W[4] = 'A' would be 'B', and indeed this is also W[5]. Furthermore, the same argument as above shows that we need not look before W[4] to find a segment for W[6], so that this is it, and we take T[6] = 2.

Therefore we compile the following table:

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

Other example more interesting and complex:

i	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3
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

Description of and pseudocode for the table-building algorithm

The example above illustrates the general technique for assembling the table with a minimum of fuss. The principle is that of the overall search: most of the work was already done in getting to the current position, so very little needs to be done in leaving it. The only minor complication is that the logic which is correct late in the string erroneously gives non-proper substrings at the beginning. This necessitates some initialization code.

```
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 is less than the length of W, do:
        (first case: the substring continues)
        if W[pos - 1] = W[cnd],

          let cnd ← cnd + 1, T[pos] ← cnd, pos ← pos + 1

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

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

Efficiency of the table-building algorithm

The complexity of the table algorithm is O(n), where n is the length of W. As except for some initialization all the work is done in the **while** loop, it is sufficient to show that this loop executes in O(n) time, which will be done by simultaneously examining the quantities pos and pos - cnd. In the first branch, pos - cnd is preserved, as both pos and cnd are incremented simultaneously, but naturally, pos is increased. In the second branch, cnd is replaced by T[cnd], which we saw above is always strictly less than cnd, thus increasing pos - cnd. In the third branch, pos is incremented and cnd is not, so both pos and pos - cnd increase. Since pos ≥ pos - cnd, this means that at each stage either pos or a lower bound for pos increases; therefore since the algorithm terminates once pos = n, it must terminate after at most 2n iterations of the loop, since pos - cnd begins at 1. Therefore the complexity of the table algorithm is O(n).

Efficiency of the KMP algorithm

Since the two portions of the algorithm have, respectively, complexities of O(k) and O(n), the complexity of the overall algorithm is O(n + k).

These complexities are the same, no matter how many repetitive patterns are in W or S.

External links

- An explanation of the algorithm [1] and sample C++ code [2] by David Eppstein
- Knuth-Morris-Pratt algorithm [3] description and C code by Christian Charras and Thierry Lecroq
- Explanation of the algorithm from scratch [4] by FH Flensburg.
- Breaking down steps of running KMP [5] by Chu-Cheng Hsieh.
- [6] NPTELHRD youtube lecture video

References

- Donald Knuth; James H. Morris, Jr, Vaughan Pratt (1977). "Fast pattern matching in strings" [7]. *SIAM Journal on Computing* **6** (2): 323–350. doi:10.1137/0206024.
- Thomas H. Cormen; Charles E. Leiserson, Ronald L. Rivest, Clifford Stein (2001). "Section 32.4: The Knuth-Morris-Pratt algorithm". *Introduction to Algorithms* (Second ed.). MIT Press and McGraw-Hill. pp. 923–931. ISBN 978-0-262-03293-3.

References

[1] http://www.ics.uci.edu/~eppstein/161/960227.html
[2] http://www.ics.uci.edu/~eppstein/161/kmp/
[3] http://www-igm.univ-mlv.fr/~lecroq/string/node8.html
[4] http://www.inf.fh-flensburg.de/lang/algorithmen/pattern/kmpen.htm
[5] http://oak.cs.ucla.edu/cs144/examples/KMPSearch.html
[6] http://www.youtube.com/watch?v=Zj_er99KMb8
[7] http://citeseer.ist.psu.edu/context/23820/0

Article Sources and Contributors

Knuth–Morris–Pratt algorithm *Source*: http://en.wikipedia.org/w/index.php?oldid=433780985 *Contributors*: A5b, Acntx, Adityasinghhhh, Adityasinghhhhh, Almi, Amitchaudhary, Antaeus Feldspar, Bikri, Blinken, Bobby.prani, Borgx, Bruguiea, Bryan Derksen, Byronknoll, Chadernook, Chester br, Chucheng, Crescent Moon, David Eppstein, Dcoetzee, Diagonalfish, Ee19921, Elias, Erroneous01, Fibonacci, Haojin, Hariva, J04n, Jagat sastry, Jaredwf, Javy413, Jeremiah Mountain, Johnuniq, Jon Awbrey, KSmrq, Krischik, LOL, Little Mountain 5, LokiClock, MadLex, Madoka, Magicheader, Mark T, Mhss, Michael Hardy, Mikespedia, NeilFraser, Olau, OnePlusTwelve, PACO, Peni, Phe, Pratik.mallya, Quuxplusone, Raknarf44, Rich Farmbrough, Ruud Koot, Ryan Reich, Shell Kinney, Sikuyihsoy, Smallman12q, Spencer4Hire, Swift, Timwi, Tom Alsberg, Tregoweth, Versus, Wahas1234, Wikibob, Ww, Ycl6, Zhaladshar, Znora, 114 anonymous edits

License