Knuth–Morris–Pratt algorithm - Wikipedia, the free encyclopedia

In computer science, 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 in 1970 by Donald Knuth and Vaughan Pratt, and independently by James H. Morris. The three published it jointly in 1977.[1]

Background[edit]

A string matching algorithm wants to find the starting index `m` in string `S[]` that matches the search word `W[]`.

The most straightforward algorithm is to look for a character match at successive values of the index `m`, the position in the string being searched, i.e. `S[m]`. If the index `m` reaches the end of the string then there is no match, in which case the search is said to "fail". At each position `m` the algorithm first checks for equality of the first character in the word being searched, i.e. `S[m] =? W[0]`. If a match is found, the algorithm tests the other characters in the word being searched by checking successive values of the word position index, `i`. The algorithm retrieves the character `W[i]` in the word being searched and checks for equality of the expression `S[m+i] =? W[i]`. If all successive characters match in `W` at position `m`, then a match is found at that position in the search string.

Usually, the trial check will quickly reject the trial match. If the strings are uniformly distributed random letters, then the chance that characters match is 1 in 26. In most cases, the trial check will reject the match at the initial letter. The chance that the first two letters will match is 1 in 26^2 (1 in 676). So if the characters are random, then the expected complexity of searching string `S[]` of length k is on the order of k comparisons or $O(k)$. The expected performance is very good. If `S[]` is 1 billion characters and `W[]` is 1000 characters, then the string search should complete after about one billion character comparisons.

That expected performance is not guaranteed. If the strings are not random, then checking a trial `m` may take many character comparisons. The worst case is if the two strings match in all but the last letter. Imagine that the string `S[]` consists of 1 billion characters that are all A, and that the word `W[]` is 999 A characters terminating in a final B character. The simple string matching algorithm will now examine 1000 characters at each trial position before rejecting the match and advancing the trial position. The simple string search example would now take about 1000 character comparisons times 1 billion positions for 1 trillion character comparisons. If the length of `W[]` is n, then the worst-case performance is $O(k \cdot n)$.

The KMP algorithm has a better worst-case performance than the straightforward algorithm. KMP spends a little time precomputing a table (on the order of the size of `W[]`, $O(n)$), and then it uses that table to do an efficient search of the string in $O(k)$.

The difference is that KMP makes use of previous match information that the straightforward algorithm does not. In the example above, when KMP sees a trial match fail on the 1000th character (`i` = 999) because `S[m+999] ≠ W[999]`, it will increment `m` by 1, but it will know that the first 998 characters at the new position already match. KMP matched 999 *A* characters before discovering a mismatch at the 1000th character (position 999). Advancing the trial match position `m` by one throws away the first *A*, so KMP knows there are 998 *A* characters that match `W[]` and does not retest them; that is, KMP sets `i` to 998. KMP maintains its knowledge in the precomputed table and two state variables. When KMP discovers a mismatch, the table determines how much KMP will increase (variable `m`) and where it will resume testing (variable `i`).

KMP algorithm[edit]

Worked example of the search algorithm[edit]

To illustrate the algorithm's details, consider a (relatively artificial) run of the algorithm, where `W` = "ABCDABD" and `S` = "ABC ABCDAB ABCDABCDABDE". At any given time, the algorithm is in a state determined by two integers:

- `m`, denoting the position within `S` where the prospective match for `W` begins,
- `i`, denoting the index of the currently considered character in `W`.

In each step the algorithm compares `S[m+i]` with `W[i]` and advances `i` if they are equal. This is depicted, at the start of the run, like

```
            1         2
m: 01234567890123456789012
S: ABC ABCDAB ABCDABCDABDE
W: ABCDABD
i: 0123456
```

The algorithm compares successive characters of `W` to "parallel" characters of `S`, moving from one to the next by incrementing `i` if they match. However, in the fourth step `S[3] = ' '` does not match `W[3] = 'D'`. Rather than beginning to search again at `S[1]`, we note that no `'A'` occurs between positions 1 and 2 in `W`; hence, having checked all those characters previously (and knowing they matched the corresponding characters in `S`), there is no chance of finding the beginning of a match. Therefore, the algorithm sets `m = 3` and `i = 0`.

```
            1         2
m: 01234567890123456789012
```

```
S: ABC ABCDAB ABCDABCDABDE
W:    ABCDABD
i:    0123456
```

This match fails at the initial character, so the algorithm sets `m = 4` and `i = 0`

```
            1         2
m: 01234567890123456789012
S: ABC ABCDAB ABCDABCDABDE
W:     ABCDABD
i:     0123456
```

Here `i` increments through a nearly complete match `"ABCDAB"` until `i = 6` giving a mismatch at `W[6]` and `S[10]`. However, just prior to the end of the current partial match, there was that substring `"AB"` that could be the beginning of a new match, so the algorithm must take this into consideration. As these characters match the two characters prior to the current position, those characters need not be checked again; the algorithm sets `m = 8` (the start of the initial prefix) and `i = 2` (signaling the first two characters match) and continues matching. Thus the algorithm not only omits previously matched characters of `S` (the `"BCD"`), but also previously matched characters of `W` (the prefix `"AB"`).

```
            1         2
m: 01234567890123456789012
S: ABC ABCDAB ABCDABCDABDE
W:          ABCDABD
i:          0123456
```

This search fails immediately, however, as `W` does not contain another `"A"`, so as in the first trial, the algorithm returns to the beginning of `W` and begins searching at the mismatched character position of `S`: `m = 10`, reset `i = 0`.

```
            1         2
m: 01234567890123456789012
S: ABC ABCDAB ABCDABCDABDE
W:           ABCDABD
i:           0123456
```

The match at `m=10` fails immediately, so the algorithm next tries `m = 11` and `i = 0`.

```
            1         2
m: 01234567890123456789012
S: ABC ABCDAB ABCDABCDABDE
W:            ABCDABD
i:            0123456
```

Once again, the algorithm matches `"ABCDAB"`, but the next character, `'C'`, does not match the final character `'D'` of the word `W`. Reasoning as before, the algorithm sets `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: 012345678901234567890 12
S: ABC ABCDAB ABCDABCDABDE
W:                ABCDABD
i:                0123456
```

This time the match is complete, and the first character of the match is `S[15]`.

Description of pseudocode for the search algorithm[edit]

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 < length(S) do
        if W[i] = S[m + i] then
            if i = length(W) - 1 then
                return m
            let i ← i + 1
```

```
        else
            if T[i] > -1 then
                let m ← m + i - T[i], i ← T[i]
            else
                let i ← 0, m ← m + 1

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

Efficiency of the search algorithm[edit]

Assuming the prior existence of the table `T`, the search portion of the Knuth–Morris–Pratt algorithm has complexity $O(n)$, where n is the length of `S` and the O is big-O notation. Except for the fixed overhead incurred in entering and exiting the function, all the computations are performed in the **`while`** loop. To bound the number of iterations of this loop; observe that `T` 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`.

This fact implies that the loop can execute at most $2n$ times, since at 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. At the same time, the second branch leaves `m + i` unchanged, for `m` gets `i - T[i]` added to it, and immediately after `T[i]` gets assigned as the new value of `i`, hence `new_m + new_i = old_m + old_i - T[old_i] + T[old_i] = old_m + old_i`. Now, the loop ends if `m + i` = n; therefore, each branch of the loop can be reached at most n times, since they respectively increase either `m + i` or `m`, and `m ≤ m + i`: if `m` = n, then certainly `m + i` ≥ n, so that since it increases by unit increments at most, we must have had `m + i` = n at some point in the past, and therefore either way we would be done.

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

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..m] = S[p..p+m]`. Upon success, that is, the word and the text matched at the positions (`W[i] = S[p+i]`), we increase `i` by 1. Upon failure, 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 is rolled 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 $2n$.

"Partial match" table (also known as "failure function")[edit]

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 below.

Worked example of the table-building algorithm[edit]

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 `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 was found at the previous stage (e.g. `T[x] = m`).

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	D
T[i]	-1	0	0	0	0	1	2

Another example, more interesting and complicated:

i	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18
W[i]	P	A	R	T	I	C	I	P	A	T	E		I	N		P	A	R	A
T[i]	-1	0	0	0	0	0	0	0	1	2	0	0	0	0	0	0	1	2	3

Description of pseudocode for the table-building algorithm[edit]

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

Efficiency of the table-building algorithm[edit]

The complexity of the table algorithm is `O(k)`, where `k` 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(k)` 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 $\texttt{pos} \geq \texttt{pos - cnd}$, this means that at each stage either `pos` or a lower bound for `pos` increases; therefore since the algorithm terminates once `pos = k`, it must terminate after at most `2k` iterations of the loop, since `pos - cnd` begins at `1`. Therefore the complexity of the table algorithm is `O(k)`.

Efficiency of the KMP algorithm[edit]

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`.

Variants[edit]

A real-time version of KMP can be implemented using a separate failure function table for each character in the alphabet. If a mismatch occurs on character

x
in the text, the failure function table for character

x
is consulted for the index

i
in the pattern at which the mismatch took place. This will return the length of the longest substring ending at

i
matching a prefix of the pattern, with the added condition that the character after the prefix is

x
. With this restriction, character

x
in the text need not be checked again in the next phase, and so only a constant number of operations are executed between the processing of each index of the text. This satisfies the real-time computing restriction.

The Booth algorithm uses a modified version of the KMP preprocessing function to find the lexicographically minimal string rotation. The failure function is progressively calculated as the string is rotated.