String Matching Algorithms

Hariank Muthakana   Corwin de Boor

October 3, 2014

1 Introduction

Suppose you want to find all occurrences of some string N in another string M. This is known as the *string matching* problem.

One way we could do this is iterating through all possible starting locations. However, this is not very efficient; this naive solution is $O(nm)$ in the worst case for strings of length n and m. Two better string matching methods are the Rabin-Karp hash algorithm and the Knuth-Morris-Pratt algorithm.

2 Rabin-Karp

The main issue with our naive approach is that we have to check every character of N for each starting point in M. We want improve this comparison between from linear to constant time. This is where the Rabin-Karp hash comes in.

2.1 Definition

A *hash* is essentially an operation to convert between data types. Here, we aim to hash strings to integers, because it is much faster to compare numbers than a set of characters. We use the following *hash function*:

$$H(S, j) = \sum_{i=0}^{j-1} S_i \cdot p^i = S_0 + S_1 \cdot p + S_2 \cdot p^2 + \ldots + S_{j-1} \cdot p^{j-1}$$

Here, p is a prime number and S_i is the integer value of character i in the string. The hash values could be very large, so one common modification is taking the result mod some other (large) prime number.

2.2 Comparing Strings

The definition accounts for every string beginning at the first character, but how do we compare strings in the middle? We can represent any range within string M using a prefix-sum approach:

$$H(S, j, k) = \sum_{i=j}^{k-1} S_i \cdot p^i = \frac{1}{p^k}(H(S, j+1) - H(S, k))]$$

Therefore, if we simply precompute $H(S, j)$ for all indexes j, we can compare hash values of any substring in constant time. Note that for a very good hash function, $H(A) = H(B)$ will imply $A = B$. However, when we find hash matches, we do need to check character-by-character in case of a *hash collision* - where two different strings have the same hash. Of course, the number of times we would have to do this is still much less than in the naive approach.

Rabin-Karp is $O(n + m)$ and $O(nm)$ in the worst case. Because of the precomputation involved, Rabin-Karp is most useful for multiple string matching - searching for many patterns in the same string.

3 Knuth-Morris-Pratt

Another other important string matching algorithm is the Knuth-Morris-Pratt algorithm. KMP takes advantage of the way we traverse the string. When a character-character match fails, we don't have to start over with string N that we are searching for, the needle. Essentially, we learn information about the needle, and we use it when we find a mismatch.

3.1 Definition

We first create a *partial match table* `T[i]` by iterating through the needle N. We set `T[0] = -1`. All other elements `T[i]` hold the length of the *longest prefix equal to the suffix* of `N[0..i-1]`, the substring ending with the current index. Equivalently, this can be thought of as maximizing `T[i] = j` such that `N[0..j-1] = N[i-j..i-1]`.

The partial match table provides insight into overlaps in the needle. Then, when a mismatch is found, the overlapping portions provide a place to start searching in the needle once again. Using this, we don't have to backtrack all the way to the beginning of N. Instead, we jump back to `T[i]`, the index of the previous overlap.

We iterate through strings N and M with `i` and `j`, respectively (initially 0). If `N[i] = M[j]`, increment both variables. If not, first check if `T[i] = -1`. If it is, then we are at the beginning of N and we can't jump back further, so we just move on to the next character in M. If it is not, then we jump back by setting `i` to `T[i]` and continue iterating. When we get to the end of N, a match has been found.

For example, with the haystack $M =$ `cabababcaa` and the needle $N =$ `ababc`:

```
M       c  a  b  a  b  a  b  c  a  a

j       0  1  2  3  4  5
                       6  7  8 MATCH

N       a  b  a  b  c

T      -1  0  0  1  2

i       0
        1  2  3  4  5
           6  7  8 MATCH
```

The variables `i` and `j` here denote where they point at the numbered iteration step. Notice that both pointers move forward when the corresponding characters in N and M match and only one pointer moves when they do not. This can equivalently be written out in the following iteration table.

	j	$M[j]$	i	$N[i]$	$T[i]$	Notes
0	0	c	0	a	-1	no match
1	1	a	**0**	a	-1	
2	2	b	1	b	0	
3	3	a	2	a	0	
4	4	b	3	b	1	
5	5	a	4	c	**2**	backtrack to index 2
6	5	a	**2**	a	0	
7	6	b	3	b	1	
8	7	c	4	c	2	DONE

Note the bolded cells that highlight the interesting changes in the pointers.

3.2 Pseudocode

We use one loop to fill array T and a second one to iterate through M:

```
cur = 2, ind = 0
while cur < n:
    if N[cur - 1] = N[ind]:
        ind++
        T[cur] = ind
        cur++
    else if ind > 0:
        ind = T[ind]
    else
        T[cur] = 0
        cur++

while j < m:
    if N[i] == M[j]:
        if i == n - 1:
            return j - i
        i++, j++
    else:
        if T[i] == -1:
            j++
        else
            i = T[i]
```

If we want to find multiple matches, every time we find a match we first record the position. Then, we treat it as a mismatch and continue rather than returning.

Knuth-Morris-Pratt is $O(n + m)$, $O(n)$ to compute array T and $O(m)$ to iterate through M. This approach avoids the worst-case inefficiency of Rabin-Karp, so KMP is more useful for single string matching.

3.3 References

- SCT String Matching 2013
- SCT Cool String Tricks 2011
- Wikipedia