String Searching

Larry Wang and Charles Zhao
presented by Mihir Patel

October 27, 2017

1 Introduction

The problem of string searching is as follows: find an occurrence (or all occurrences) of a pattern string of length N within a text string of length M. A naive brute-force solution would be to consider every character in the text as the starting character of the pattern. Although this solution does not require any precomputation, it has a time complexity of $O(NM)$, which is undesirable in many cases.

2 Knuth-Morris-Pratt

The problem with the brute-force solution is that whenever we find a mismatch, we have to backtrack our position in the text string. This is similar to "forgetting" the last few characters we matched and simply starting the search over again. KMP allows us to effectively avoid backtracking in the text string to improve the running time.

To see how this works, supposed we found a mismatch at index j in the pattern string. Now consider the substring *substr* := `pattern[0, j]`. We know that the last j characters in the text match *substr* exactly. If *substr* has a proper suffix of length l that is also a prefix of *substr*, then we can then align the last l characters in the text with the first l characters in the pattern and continue our search without backtracking. If *substr* does not have a suffix that is also a prefix, then we can just restart our search beginning at the character of the mismatch.

To actually implement this, we must compute the partial match table (or failure function) `T`, where `T[i]` is the length of the longest suffix that is also a prefix of `pattern[0, i]`. We also define `T[0]` to be -1. To compute `T` efficiently, we note that the longest suffix of `pattern[0, i]` must be a suffix of `pattern[0, i-1]` with `pattern[i]` appended. Furthermore, we have already computed all of the suffixes that are also prefixes up to index i. The following psuedocode accomplishes the table construction algorithm. Notice that the maximum iterations of the loop is $2M$ (j is only incremented when i is incremented, so the number of times the second case occurs is bounded by M), so the table construction is $O(M)$.

Algorithm 1 KMP Table Construction

```
function TABLE($pattern$, $T$)
    $i \leftarrow 1$                                  ▷ $i$ is the current position in the pattern
    $j \leftarrow 0$                                  ▷ $j$ is the last character of a prefix that is also a suffix
    $T[0] \leftarrow -1$
    while $i < len(pattern)$ do
        if $pattern[i-1] = pattern[j]$ then
            $T[i] \leftarrow j + 1$
            $j \leftarrow j + 1$
            $i \leftarrow i + 1$
        else if $j > 0$ then
            $j \leftarrow T[j]$
            $T[j] \leftarrow 0$
        else
            $T[i] \leftarrow 0$
            $i \leftarrow i + 1$
```

To actual find occurrences of the pattern in the text, we store positions in the text and pattern. If a match occurs, then we increment both pointers. Otherwise, we reset the pattern pointer to its longest suffix that is also a prefix using the partial match table. The following pseudocode describes this process. We return the position that the match was found, or -1 if no match was found. The maximum number of iterations of the loop is $2N$ (by a similar argument as above), so the complexity is $O(N)$.

Algorithm 2 KMP Search

```
function SEARCH($pattern$, $text$, $T$)
    $i \leftarrow 0$                                  ▷ $i$ is the position of the beginning of the match in the text
    $j \leftarrow 0$                                  ▷ $j$ is the position in the pattern
    while $i + j < len(text)$ do
        if $pattern[j] = text[i+j]$ then
            if $j = len(pattern) - 1$ then
                return $i$
            else
                $j \leftarrow j + 1$
        else
            if $T[j] > -1$ then
                $i \leftarrow i + j - T[j]$
                $j \leftarrow T[j]$
            else
                $i \leftarrow i + 1$
                $j \leftarrow 0$
    return $-1$
```

It's also worth nothing that constructing the partial math table is equivalent to constructing a deterministic finite state automata, which we can think of as a collection of states and links between these states. A link pointing "forward" represents a successful match and a link falling "backwards" represents which state to go to after encountering a mismatch. The algorithm to search the text is equivalent to simulating the DFA .

Putting this all together, the final complexity of KMP is $O(N + M)$, which is clearly much better than the brute-force solution.

3 Rabin-Karp

The Rabin-Karp algorithm uses hashing to efficiently search for one or more strings in a text simultaneously. Whereas each string comparison in the naive approach runs in linear time, Rabin-Karp uses hashing to bring this down to constant time. To do this, the hashing algorithm must be able to efficiently (i.e. in constant time) compute the hash for position $i + 1$ in the text string given the hash for position i. We define the hash of the substring starting at position i as

$$x_i = t_i R^{N-1} + t_{i+1} R^{N-2} + \ldots + t_{i+N-1} R^0$$

where t_i represents the character at position i within the text string and R is some prime number. This operation runs in $O(N)$, which is linear. However, we only need to perform this operation once. Because of the way we have defined the hash function, we know that

$$x_{i+1} = (x_i - t_i R^{N-1})R + t_{i+N}$$

which runs in constant time. This operation subtracts the first character's contribution to the hash, multiplies by R, and then adds the new character. To account for hash collisions, we still must perform a linear-time character-by-character comparison when we find a hash match, but this occurs much less frequently than in the naive approach. Although Rabin-Karp has a worst case time complexity of $O(NM)$, which would occur if there were a hash collision at every substring, in practice, it runs in $O(N + M)$. Rabin-Karp is best when searching for multiple pattern strings of the same length, whereas KMP is generally faster when searching for a single pattern string. In the pseudocode below, since the hashes may be quite large, we mod the results by a large prime number Q.

Algorithm 3 Rabin-Karp

```
function HASH($p$, $N$)
    $h \leftarrow 0$
    for $i = 0 \ldots N-1$ do
        $h \leftarrow (R \times h + p[i]) \% Q$
    return $h$
function RABINKARP($s$, $patterns$, $N$)
    $hashedPatterns \leftarrow emptySet$
    for each $p \in patterns$ do
        ADD($p$, $hashedPatterns$)
    $x \leftarrow$ HASH($s[0 \ldots N-1]$, $N$)
    if $x \in hashedPatterns$ and $s[0 \ldots N-1] \in patterns$ then
        return 0
    for $i = N \ldots M-1$ do
        $x \leftarrow (x - R^{N-1} \times s[i-N] \% Q) \% Q$        ▷ subtract contribution from first character
        $x \leftarrow (x \times R + s[i]) \% Q$                       ▷ add next character
        if $x \in hashedPatterns$ and $s[i \ldots i+N-1] \in patterns$ then
            return $i$
```