String Matching

Sreenath Are

November 1st, 2013

The Problem

Given two strings, $S[0...s-1]$ and $W[0...w-1]$, find every occurrence of W in S.

Naïve Approach

The most straightforward method would be to simply check every position of S for a match with W. Here's an example with S ='abcababcababababca' and W ='ababc' (the asterisks indicate that a match has been found):

```
abcababcababababca
aba
 a
  a
   ababc*
    a
     aba
      a
       a
        ababc
         a
          ababc*
           a
```

As you can see, this approach will have fairly good runtime for most strings, because you can stop checking for a match once a single letter differs. In fact, the expected value of the runtime of this approach is $O(s)$. However, its worst-case runtime is horrendous: consider the string S ='AAA...A' consisting of one billion 'A's, and the search term W ='AAA...AB' consisting of 999 'A's followed by one 'B'. The match at each index will only fail at the 1000th character, meaning that every character of W will be checked for each character of S. In this example, this result in one trillion comparison operations. In general, the worst-case runtime is $O(sw)$.

The Rabin-Karp Algorithm

Our previous approach was slow because every check took $O(w)$ comparisons (in the worst case). The Rabin-Karp algorithm improves the checking operation by reducing its runtime to $O(1)$ (with some preprocessing). We define a hash of the string $S[0...s-1]$ by

$$H(S) = \left(\sum_{k=0}^{s-1} S_k \cdot p_1^k\right) \%p_2 = \left(S_0 + S_1 \cdot p_1 + S_2 \cdot p_1^2 + \cdots + S_{n-1} \cdot p_1^{n-1}\right) \%p_2$$

By doing this, we reduce the comparison of two strings to the comparison of two numbers, because we know that if $H(A) = H(B)$, then $A = B$ with high probability.

To solve the matching problem, we iterate through the string S as in the naïve approach. Instead of comparing $S[i...w + i - 1]$ to W, we compare $H(S[i...w + i - 1])$ to $H(W)$. If the hashes are equal, we then have to check all w characters (because of the possibility of a hash collision – two different strings getting mapped to the same hash). Note that the worst case runtime is still $O(sw)$, but this algorithm will outperform the naïve algorithm on most real (not contrived) datasets.

The Knuth-Morris-Pratt Algorithm

To avoid the high worst-case runtime of these algorithms, we can use the fact that when a character match fails, it is not necessary to go back to the starting of the search term, because the search term itself contains information about where the next occurrence could begin. We build the array $m[0...w - 1]$ where $m[i]$ is the length of the longest proper prefix of $W[0...i]$ that is equal to a proper suffix of the same substring. For example, consider the string $W =$'abababca'.

```
i     0 1 2 3 4 5 6 7
W[i]  a b a b a b c a
m[i]  0 0 1 2 3 4 0 1
```

And for the earlier example of $W =$'ababc':

```
i     0 1 2 3 4
W[i]  a b a b c
m[i]  0 0 1 2 0
```

When a mismatch occurs, we know that the previous $m[i]$ characters match the first $m[i]$ characters of the string, so we don't need to check them again. We use two variables i and j (initially 0) to store the indices in S and W, respectively, we're currently comparing. If they match, increment i and j. Otherwise, since the previous $m[j]$ characters of S match the first $m[j]$ characters of S, we can skip checking those and set j to $m[j]$ and continue. Matches occur when $j = w$. Since this is a single loop over all values of i, the runtime of this step is $O(s)$. The same technique can be used to compute $m[i + 1]$ given $m[i]$ and W, computing m in a total of $O(w)$ operations. Thus the overall runtime is $O(s + w)$ in the worst case.