Algorithms: COMP3121/3821/9101/9801

Rabin – Karp Algorithm ：

https://www.cnblogs.com/golove/p/3234673.html

String matching with finite automata（字符串匹配之有限自动机）：

https://www.cnblogs.com/jolin123/p/3443543.html

Aleks Ignjatović

School of Computer Science and Engineering
University of New South Wales

9. STRING MATCHING ALGORITHMS

String Matching algorithms

- Assume that you want to find out if a string $B = b_0 b_1 \dots b_{m-1}$ appears as a (contiguous) substring of a much longer string $A = a_0 a_1 \dots a_{n-1}$.
- The "naive" string matching algorithm does not work well if B is much longer than what can fit in a single register; we need something cleverer.
- We now show how hashing can be combined with recursion to produce an efficient string matching algorithm.

Rabin - Karp Algorithm

- We compute a hash value for the string $B = b_0 b_1 b_2 \ldots b_m$ in the following way.
- Assume that strings A and B are in an alphabet $\mathcal{A}$ with d many symbols in total.
- Thus, we can identify each string with a sequence of integers by mapping each symbol s_i into a corresponding integer i:

$$\mathcal{A} = \{s_0, s_1, s_2, \ldots, s_{d-1}\} \longrightarrow \{0, 1, 2, \ldots, d-1\}$$

- To any string $B = b_0 b_1 \ldots b_{m-1}$ we can now associate an integer whose digits in base d are integers corresponding to each symbol in B:

  变成相应的d进制

$$h(B) = h(b_0 b_1 b_2 \ldots b_m) = d^{m-1} b_0 + d^{m-2} b_1 + \ldots + d \cdot b_{m-2} + b_{m-1}$$

- This can be done efficiently using the Horner's rule:

$$h(B) = b_{m-1} + d(b_{m-2} + d(b_{m-3} + d(b_{m-4} + \ldots + d(b_1 + d \cdot b_0))) \ldots)$$

- Next we choose a large prime number p such that $(d+1)\,p$ fits in a single register and define the hash value of B as $H(B) = h(B) \bmod p$.

Rabin - Karp Algorithm

- Recall that $A = a_0 a_1 a_2 a_3 \ldots\ldots a_s a_{s+1} \ldots a_{s+m-1} \ldots\ldots a_{N-1}$ where $N >> m$.
- We want to find efficiently all s such that the string of length m of the form $a_s a_{s+1} \ldots a_{s+m-1}$ and string $b_0 b_1 \ldots b_{m-1}$ are equal.
- For each contiguous substring $A_s = a_s a_{s+1} \ldots a_{s+m-1}$ of string A we also compute its hash value as

$$H(A_s) = (d^{m-1} a_s + d^{m-2} a_{s+1} + \ldots + d^1 a_{s+m-2} + a_{s+m-1}) \bmod p$$

- We can now compare the hash values $H(B)$ and $H(A_s)$ and do a symbol-by-symbol matching only if $H(B) = H(A_s)$.
- Clearly, such an algorithm would be faster than the naive symbol-by-symbol comparison only if we can compute the hash values of substrings A_s faster than what it takes to compare strings B and A_s character by character.
- This is where recursion comes into play: we do not have compute the hash value $H(A_{s+1})$ of $A_{s+1} = a_{s+1} a_{s+2} \ldots a_{s+m}$ "from scratch", but we can compute it efficiently from the hash value $H(A_s)$ of $A_s = a_s a_{s+2} \ldots a_{s+m-1}$ as follows.

Rabin - Karp Algorithm

Since

$$H(A_s) = (d^{m-1} a_s + d^{m-2} a_{s+1} + \ldots d^1 a_{s+m-2} + a_{s+m-1}) \bmod p$$

by multiplying both sides by d we obtain

$$(d \cdot H(A_s)) \bmod p =$$

$$= (d^m a_s + d^{m-1} a_{s+1} + \ldots d \cdot a_{s+m-1}) \bmod p$$

$$= ((d^m a_s) \bmod p + (d^{m-1} a_{s+1} + \ldots d^2 a_{s+m-2} + d\, a_{s+m-1} + a_{s+m}) \bmod p - a_{s+m}) \bmod p$$

$$= ((d^m \bmod p)\, a_s + H(A_{s+1}) - a_{s+m}) \bmod p$$

Consequently, $H(A_{s+1}) = (d \cdot H(A_s) - (d^m \bmod p) a_s + a_{s+m}) \bmod p$.

- Note that in the expression

  $$H(A_{s+1}) = (d \cdot H(A_s) - (d^m \bmod p) a_s + a_{s+m}) \bmod p$$

  the value $d^m \bmod p$ can be precomputed and is smaller than p, while $a_s, a_{s+m} < d < p$.

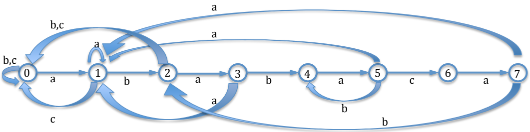
- Since we chose p such that $(d+1)\,p$ fits in a register, all the values and the intermediate results for the above expression also fit in a single register.

- The value of $H(A_s)$ can be computed in constant time independent of the length of the strings A and B.

String matching finite automata

- A string matching finite automaton for a string S with k symbols has $k+1$ many states $0, 1, \ldots k$ which correspond to the number of characters matched thus far and a transition function $\delta(s, c)$ where s is a state and c is a character, given by a table.
- To make things easier to describe, we consider the string $S = ababaca$. The table defining $\delta(s, c)$ would then be

	input			
state	a	b	c	
0	**1**	0	0	a
1	1	**2**	0	b
2	**3**	0	0	a
3	1	**4**	0	b
4	**5**	0	0	a
5	1	4	**6**	c
6	**7**	0	0	a
7	1	2	0	



state transition diagram for string *ababaca*

String matching with finite automata

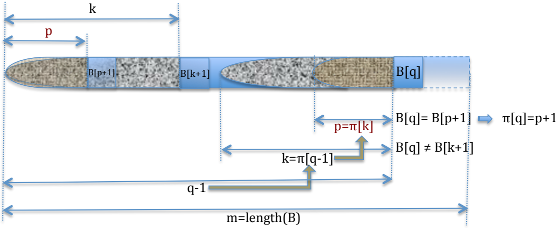
- How do we compute the transition function δ, i.e., how do we fill the table?
- Let B_k denote the prefix of the string B consisting of the first k characters of string B.
- If we are at a state k this means that so far we have matched the prefix B_k; if we now see an input character a, then $\delta(k, a)$ is the largest m such that the prefix B_m of string B is the suffix of the string $B_k a$.
- Thus, if a happens to be $B[k+1]$, then $m = k+1$ and so $\delta(k, a) = k+1$ and $B_k a = B_{k+1}$.

String matching with finite automata

- We can get by without precomputing $\delta(k, a)$ but instead compute it "on the fly".
- We do that by matching the string against itself: we can recursively compute a function $\pi(k)$ which for each k returns the largest integer m such that the prefix B_m of B is a proper suffix of B_k.

The Knuth-Morris-Pratt algorithm

```
function Compute − Prefix − Function(B)
    m ← length[B]
    let π[1..m] be a new array
    π[1] = 0
    k = 0
    for q = 2 to m do
        while k > 0 and B[k+1] ≠ B[q]
            k = π[k]
        if B[k+1] == B[q]
            k = k + 1
        π[q] = k
    end for
    return π
end function
```



Assume that length of B is m and that we have already found that $\pi[q-1] = k$; to compute $\pi[q]$ we check if $B[q] = B[k+1]$; if it is not; then $\pi[q] \neq k+1$ and we find $\pi[k] = p$; if now $B[q] = B[k+1]$ then $\pi[q] = p+1$.

The Knuth-Morris-Pratt algorithm

- We can now do our search for string B in a longer string A:

```
function KMP − Matcher(A, B)
    n ← length[A]
    m ← length[B]
    π = Compute − Prefix − Function(B)
    q = 0
    for i = 2 to n do
        while q > 0 and B[q + 1] ≠ A[i]
        q = π[q]
        if  B[q + 1] == A[i]
            q = q + 1
        if  q == m
        print pattern occurs with shift i − m
        q = π[q]
    end for
end function
```

Looking for imperfect matches

- Sometimes we are not interested in finding just the prefect matches, but also in matches that might have a few errors, such as a few insertions, deletions and replacements.

- So assume that we have a very long string $A = a_0a_1a_2a_3 \ldots \ldots a_s a_{s+1} \ldots a_{s+m-1} \ldots \ldots a_{N-1}$, a shorter string $B = b_0b_1b_2 \ldots b_{m-1}$ where $m << N$ and an integer $k << m$. We are interested in finding all matches for B in A which allow up to k many errors.

- Idea: split B into $k+1$ consecutive subsequences of (approximately) equal length. Then any match in A with at most k errors must contain a subsequence which is a perfect match for a subsequence of B. Thus, we look for all perfect matches for all of $k+1$ subsequences of B and for every hit we test by brute force if the remaining parts of B have sufficient number of matches in the appropriate parts of A.