CS612 Algorithm Design and Analysis

Lecture 17. String matching [1]

Dongbo Bu

Institute of Computing Technology
Chinese Academy of Sciences, Beijing, China

[1]The slides are made based on Randomized Algorithm by R. Motwani and P. Raghavan, http://www-igm.univ-mlv.fr/ lecroq/string/, and a lecture by T. Chan.

Outline I

- Introduction
- DFA and KMP methods
- A Monte Carlo method
- Rabin-Karp randomized algrorithm;

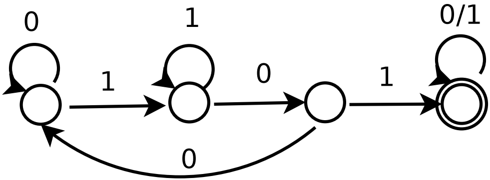
INPUT:
Given strings $t = t_1t_2...t_n$,$(t_i \in \{0, 1\}, i = 1, 2, ..., n)$ and $p = p_1p_2...p_m$, $(p_j \in \{0, 1\}, j = 1, 2, ..., m)$, $m \leq n$;
OUTPUT:
Is p a substring of t?

t is called "text" and p is called "pattern".

DFA-based methods

- Brute-force method: checking every possible occurence of p in t. Time-complexity: $O(nm)$ (when searching for $a^{m-1}b$ in a^n for instance.)
- DFA-based method:
  1. building a DFA for all string containing p;
  2. running this DFA on t;
  3. Time-complexity: $O(f(m) + n)$. Here, $f(m)$ denotes the time to build a DFA. It is possible that $f(m) = O(m)$.

e.g.: $p = 101$



KMP and BM algorithms

1. Karp-Morris-Parrat method:
   - Similar to DFA but with a "compressed" representation of DFA.

i	0	1	2	3	4	5	6	7	8
$x[i]$	G	C	A	G	A	G	A	G	
$kmpNext[i]$	-1	0	0	-1	1	-1	1	-1	1

2. Boyer-Moore method:
   - The Boyer-Moore algorithm is considered as the most efficient string-matching algorithm in usual applications.
   - The algorithm scans the characters of the pattern from right to left beginning with the rightmost one. In case of a mismatch (or a complete match of the whole pattern) it uses two precomputed functions to shift the window to the right. These two shift functions are called the good-suffix shift (also called matching shift and the bad-character shift (also called the occurrence shift).

An interesting sub-problem I

Problem:

- Alice has a string u, and Bob has a string v, $u, v \in \{0, 1\}^*$, $|u| = |v| = n$.
- They want to see whether $u = v$.

Possible ways:

1. transmit n bits;
2. transmit a "fingerprint" with $O(d \log n)$ bits;

A randomized finger-print:

- Let $x = \{0, 1, ..., P-1\}$, where P is a large prime number;
- Define a finger-print $F_x : \{0,1\}^* \rightarrow \{0, 1, 2, ..., P-1\}$ as follows:
  $F_x(a_{n-1}a_{n-2}...a_0) = \sum_i a_i x^i \mod P$.

Monte-Carlo algo:

1. $x = random(0, P-1)$, where P is a large prime number;
2. Alice transfers $F_x(u)$ to Bob;
3. Bob calculate $F_x(v)$ first, and reports "Yes" if $F_x(u) = F_x(v)$; Otherwise reports "No" (definitely "No").

Error analysis:

1. Case 1: $(u = v)$. Correct.

2. Case 2: $(u \neq v)$ Error: Bob reports "Yes", i.e., $F_x(u) = F_x(v)$ when $u \neq v$.

$$\begin{aligned} & \Pr(F_x(u) = F_x(v) | u \neq v) \\ = & \Pr(\sum_{i=0}^{n-1} u_i x^i = \sum_{i=0}^{n-1} v_i x^i \mod P) \\ = & \Pr(\sum_{i=0}^{n-1} (u_i - v_i) x^i = 0) \\ \leq & \frac{n-1}{P} \quad \text{by Fact 1.} \end{aligned}$$

3. $\Pr(F_x(u) = F_x(v)) \leq \frac{1}{n^d}$ when setting $P = n^{d+1}$.

Fact 1:
A polynomial of degree $\leq n-1$ has at most $n-1$ roots $\mod P$.

Rabin-Karp randomized algorithm for string matching. I

- Rabin-Karp Algo (s, t):
  1. $x = random(0, P-1)$;
  2. $A = F_x(s_0 s_1 ... s_m)$, $B = F_x(t_0 t_1 ... t_m)$;
  3. for $i = 0$ to $n - m$
  4. //compare $s_{i+1} s_{i+2} ... s_{i+m}$ with $t_1 t_2 ... t_m$;
  5. $A = (A - a_{i+1} x^m) x + a_{i+m+1} \mod P$;
  6. if ($A == B$) return "possibly match";
  7. return "No";
- Time-complexity: $O(n)$.
- Error probability:
  1. Let E_i denote the event: algo errs at the i-th iteration. We have:
  2. $\Pr(E_i) \leq \frac{m-1}{P}$
  3. $\Pr(Error) = \Pr(\cup_{i=0}^{n-m} E_i) \leq \sum_i \Pr(E_i) \leq \frac{nm}{P} \leq \frac{n^2}{P}$
  4. $\Pr(Error) \leq \frac{1}{n^d}$ by setting $P = n^{d+1}$.

A Las Vegas version

Algo:

1. Run Karp-Rabin;
2. if it returns "No", returns "No";
3. else
4. Verify $s = t$;
5. if so, return "Yes"; else goto Step 1;

Analysis:

- If Karp-Rabin is correct: $O(n)$ time is enough;
- otherwise, the execution of brute-force algo costs $O(mn)$ time.

Expected running time: $E(T) = O(n)(1 - \frac{1}{n^d}) + O(mn)\frac{1}{n^d} = O(n)$. (setting $d = 1$)