String Algorihtms I

Rabin-Karp, Trie

Section 1

Introduction

String

A *String* is a sequence of elements from alphabet set Σ. For example, if the alphabet set is the lower case letters in English, then abcccda is a string. In this lecture, we assume the alphabet set is the lower case letters plus the empty letter. We number the elements in Σ from 0 to $|\Sigma| - 1$ which allows the elements participating arithmetic operations. (Hence the lower case letters are numbered from 0 to 25, and the empty letter is 26)

For a string S, a *substring* of S is a contiguous sub seuqnce of S. For example, if S=abcde, then abc is a substring of S, but abd is not.

Overview

String matching is the central of study. By matching, we mean the exact sub-string matching. For example, a classical problem is given strings S and T, check if T matches some substring of S.

We will introduce the following kinds algorithms.

- Hashing based: Rabin-Karp
- Automata based: Trie and AC automata
- Suffix structure based: Suffix tree (array), Suffix automata

Section 2

Rabin-Karp

Rabin-Karp: Hash Function

A hash function maps a string to an integer. The hash function of RK algorithm for a string S is defined to be $r(S) = \sum_{i=0}^{|S|-1} S_i \cdot p^{|S|-1-i} \mod N$, for some constant p, N.

Observe that if we set $p = |\Sigma|$ then the hash function is actually the representation of the string in base p, except that we have to mod N.

We note that if for string S and T, $r(S) \neq r(T)$, then we know $S \neq T$ for sure; however, if $r(S) = r(T)$, we actually have to check whether S really equals T. This issue is usually called *collision*.

Rabin-Karp: Implementation Notes

In reality the integer data type usually has a limited range. For example, in C++/Java, "int" is only 32 bit, and the longest native data type is only 64 bit. Therefore, we modify $r(S)$ to be $r(S)$ mod N, where N is a large integer. The recommended way is set N to be 2^{32} if "int" is used, and set it to be 2^{64} if long long is used (in C++). This way, we may then avoid using the module operator and just let the number overflow.

The guideline for setting p is to set it to be the least prime number that is at least $|\Sigma|$.

To accelerate the calculation p^i, we can pre-calculate all the p^i mod N for all possible i.

Rabin-Karp: String Matching

We introduce the application of RK on the classical string matching algorithm.
Suppose we are given S and T. Our goal is to test whether T is a substring of S, and if it is, return the first occurence in S. Wlog, assume $|S| \geq |T|$.

```
Data: String $S$, $T$
begin
    Let $s = r(S_0 S_1 \dots S_{|T|-1})$, and $t = r(T)$
    for $i = 0 \to |S| - |T|$ do
        if $s = t$ and $S_i S_{i+1} \dots S_{i+|T|-1} = T$ then
            Report matching at $i$ and terminate
        end
        $s := p \cdot (s - S_i \cdot p^{|T|-1}) + S_{i+|T|}$
    end
end
```

Algorithm 1: RK Matching

Analysis

The worst case complexity is $O(|S| \cdot |T|)$, since it is possible that the hash value for each substring of S equals that of T but the substrings are different.

However, in reality, the worst case is very hard to encounter. It is believed that the algorithm usually performes in nearly linear time. In particular, we may assume that the number of collision is at most some constant. We can assume so even in ACM/ICPC.

Rabin-Karp: Recursion

A very important propoerty of RK hash function is used in the matching algorithm: we can caclculate $r(S[i+1, i+|T|])$ in constant time given $r(S[i, i+|T|-1])$.

Another important property is for calculating the hash for the concatenation of two strings S and T. Suppose we know $r(S)$ and $r(T)$, then $r(ST) = p^{|T|} \cdot r(S) + r(T)$.

Rabin-Karp: Dynamic String Matching

We consider a dynamic string matching problem[1].

We are given n ($n \leq 10^4$) keywords $w_1, w_2, \ldots w_n$, where the total length of w_i's is at most $2 \cdot 10^6$. Initially, we are also given a string S with length at most 10^5. Then we have m ($m \leq 10^5$) operations each in one of the following forms:

- Query(l, r). Check whether $S[l, r]$ is a keyword, and return "true" if it is and "false" otherwise.
- Change(x, c). Change $S[x]$ to be c.

[1]From `http://acm.fzu.edu.cn/problem.php?pid=2045`

Rabin-Karp: Dynamic String Matching

Native approach: just maintain strings in a naive way (in arrays). If the operation is "Change", then it takes $O(1)$ time; otherwise, we first fetch $S[l, r]$ and then scan w_i's once in $O(S \cdot \sum_i |w_i|)$ time.

We can use RK to improve it. Observe that the keyword set is static, we may calculate the hash value for them once, and use the hash value to represent them. If we then have the hash value of $S[l, r]$ by some way, we can perform a binary search to test if it is contained in the keyword set.

Rabin-Karp: Dynamic String Matching

To calculate $S[l, r]$ efficiently, we may use a segment tree. The segement tree essentially maintains the string (array) S, and we also maintain the hash value for each interval.

This way, by the recursion $r(ST) = p^{|T|} \cdot r(S) + r(T)$, we can implement the segment tree, and both the update and query can be performed in $O(\log |S|)$ time.

Rabin-Karp: Collision

However, in the above approach, we have no way to test whether a "match" is a collision or not, since it is too time consuming (recalling that $|w_i| \leq 10^4$). Luckily, since the bad input is not easy to generate, even if we do not check, we may still get accepted.

But is there a really reliable way to solve the problem? Actually the original RK algorithm is randomized, and we can really guarantee that the collision does not happen for all the operations with very high probability.

Rabin-Karp: Randomized Testing

The key is a randomized matching testing. Suppose we have a string S and T (of the equal length), such that $r(S) = r(T)$. To test wether $S = T$, the safest way is to scan through S and T and compare. However, an observation is that if $S \neq T$ but $r(S) = r(T)$ happens then it must be because we choose the bad N, since if we do not module any number then there is no such case. So the idea is to randomly choose k modules N_i for $i = 1, 2, \ldots, k$ in advance, and define $r_{N_i}(S)$ to be the hash function that mods N_i.

In the original RK paper "Efficient randomized pattern-matching algorithms" `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.86.9502`, the k random numbers are drawn uniformly from the prime numbers at most M, where M and k are to be picked later.

Rabin-Karp: Randomized Testing

In Corollary 4(b) of the paper, if the sub-string to be match is of length at most n, and the number of matchings to be performed is t, then the failure probability is upper bounded by $t \cdot (\frac{\pi(n)}{\pi(M)})^k$, for $n \geq 29$ and $\pi(x)$ is the number of prime numbers at msot x. A classical result shows that $\pi(x) = \Theta(\frac{x}{\ln x})$. Here by "failure", we mean *at least one* of the t tests fails.

Therefore, in our question, we set $n = 10^5$ (the length of S), and $t = 10^4 \cdot 10^5 = 10^9$ (since each query operation tries to match all the keywords). By picking $k = 4$, $M = 10^9$, we have that the probability that all the t tests are successful is at least $1 - 10^{-7}$.

Rabin-Karp: Ranomized Testing

In a nutshell, we pick $k = 4$ and $M = 10^9$, and sample N_1, N_2, N_3, N_4 at the very beginning of the program. Then whenever we have $r(S) = r(T)$ for some S and T, we test for $k = 1 \to 4$ whether all the $r_{N_i}(S) = r_{N_i}(T)$. We report a matching only if all the N_i's indicates an equality.

Rabin-Karp: Matrix Matching

The power of RK is not limited to 1-dimensional matching. In fact, we can generalize the hash function into high dimension.

Consider the maximum sub-matrix problem[2]. We are given a lower case letter matrix of n rows m columns ($m, n \leq 500$). We say two sub-matrix are equal, if they are of the same dimension and all the corresponding letters are the same. Our goal is to find the largest K such that there exists two distinct equal $K \times K$ sub-matrix. By distinct, we mean they have different upper left coner coordinate.

[2] `http://acm.timus.ru/problem.aspx?space=1&num=1486`

2-dimensional RK

We generalize the RK to 2-dimensional. For a matrix M, we define $r(M)$ to be the hash value of the string obtained by concatenating all the row strings.

We claim that by a $O(mn)$ pre-calculation, we can calculate $r(M)$ for any submatrix M in $O(1)$ time. Please try to derive the recursion on your own.

Matrix Matching

Then we go back to the matrix matching problem. We present a $O(mn\log^2 mn)$ algorithm.

First of all, we can binary search K. That is, if some K is feasible, then the smaller K's are feasbile as well. Observe that $K \le \max\{m, n\}$, we know that it is sufficient to enumerate $O(\log m + n)$ number of K's.

It remains to check for the feasibility for a given K. We take $O(mn)$ time to calculate and record all the hash values of submatrix of dimension $K \times K$. Then we sort and search if there are two of them have the same hash value, and test if they are really equal or not. This step takes $O(mn\log mn)$.

Since we are affordable to check for the hash collision, we do not need randomness (although we may still use the randomness for safety).

Compressed String Matching[3]

First we look at the naive approach. Since the field is not large, we may enumerate the starting position. For a starting position and a given length L, the string that is read from the field can actually be represented by a compressed string. Since the input is also a compressed string and we can calculate the length after the decompression, we are essentially asked to test whether the two compressed strings are equal.

[3] `http://acm.sgu.ru/problem.php?contest=0&problem=392`

Compressed String Matching

If we have the hash value of the two strings, then we may use RK. However, the two strings are very long and we cannot test the hash collision.

Observe that the string is repetitive, so the collison is actually small. We may either just ignore the collision or use the randomized testing. (Exercise: how do we pick M and k in this case?)

The only remaining problem is to calculate the hash value. We still use the recursion of the concatenation. The recursion is similar as in calculating a^n in $O(\log(n))$ time. Let $f(n)$ denote the hash value for A^n, where A is a sring and A^n means the concatenation of A for n times.

Then,

- $f(n) = f(n-1) \cdot p^{|A|} + r(A)$, when $n > 1$ and n is odd.
- $f(n) = f(\frac{n}{2}) \cdot (p^{|A| \cdot \frac{n}{2}} + 1)$, when n is even and $n > 1$.
- $f(1) = r(A)$.

Note that we module N for arithmetic operations. This recursion can be solved in $O(\log(n))$ time.
Combine the techniques together to solve the problem.

Section 3

Trie

Trie: Definition

Trie is a *dictionary* structure for strings. In particular, it supports

- Test if a given string S is inside Trie, in $O(|S|)$ time.
- Add a string S into Trie, in $O(|S|)$ time.

We may implement it in an automata-like structure.

Trie: Implementation

We initialize the Trie by creating a single initial state $q_0$with transmit function $\delta(q_0, c) = \text{NULL}, \forall c \in \Sigma$, where NULL is a special state. We define the two operations as follows, and the exact structure of Trie is implied by the procedures.

```
Data: String $S$
begin
    Let $q = q_0$
    for $i = 0 \to |S| - 1$ do
        if $\delta(q, S_i) = NULL$ then
            Create state $q'$ with $\delta(q', c) = \text{NULL}, \forall c \in \Sigma$
            Let $\delta(q, S_i) = q'$
        end
        Let $q = \delta(q, S_i)$
    end
    Mark $q$ as an accepting state
end
```

Algorithm 2: Insert

Trie: Find

```
Data: String $S$
begin
    Let $q = q_0$
    for $i = 0 \to |S| - 1$ do
        if $\delta(q, S_i) = NULL$ then
            Return false
        end
        Let $q = \delta(q, S_i)$
    end
    if $q$ is an accepting state then
        Return true
    end
    else
        Return false
    end
end
```

Algorithm 3: Find

Trie: Properties

From the above procedure, we have the following properties:

- Trie is acyclic (and therefore it is a tree), and each intersted string corresponds to a path from the root (q_0) to an accpecting state.
- For any state q in Trie, the path from q_0 to q corresponds to a string P. Moreover, an inserted string S has prefix P, if and only if the accepting state of S is in the subtree rooted at q. (That's why Trie is sometimes called "Prefix Tree")

Trie and Exclusive Or (Xor)

We are given an edge weighted tree T with n nodes ($n \leq 10^5$), where the weights are within the range of int. The distance of two nodes is define by the exclusive or of the weights of the edges in the path connecting them. Our goal is to calculate the maximum distance between all pairs of nodes.[4]

Again, naive approach would take $O(n^2)$ time. The naive approach is first fix node 1 as the root, and then do a DFS to calculate the distances between the root 1 and other nodes. Denote $d(x)$ to be the distance from x to the root. Then the distance between x and y is $d(x) \oplus d(y)$ (please check by yourself).

[4] POJ 3764

Accelarating the Naive Approach

By using Trie, we may accelerate the native approach. In particular, we still use DFS to pre-calculate $d(x)$, and then for each node x, we can calculate $\max_{y \in T} d(x) \oplus d(y)$ in constant time.

Since the weights can all be represented in 32 binary bits, we can treat all the $d(x)$'s as binary strings. Then we build a Trie using those binary strings. We then suppose x is fixed, and we show how to find a y such that $d(x) \oplus d(y)$ is maximized.

We can try to guess the answer (optimal $d(x) \oplus d(y)$) bit by bit. That is, from the highest bit (1st bit), we test if it can be 1. By "can be", we mean there exists y such that $d(x) \oplus d(y)$ has highest bit 1. If it can be 1, then we know that the optimal has highest bit 1; otherwise even the optimal cannot have highest bit 1 and must be 0.

This way, we can determine the highest bit uniquely. Then we use the same procedure for the second bit, third bit, ..., until we fix all the 32 bits.

The following is an implementation of the idea.

```
Data: Fixed string $S$ denoting $d(x)$
begin
    Let $q = q_0$
    for $i = 0 \to 31$ do
        Let $b = S_0 \oplus 1$ denoting the desired bit of $d(y)$ in order for
        the $i$-th bit of $S \oplus d(y)$ being 1
        if $\delta(q, b) \neq NULL$ then
            Set the $i$-th bit of the result to be 1
            $q = \delta(q, b)$
        end
        else
            Set the $i$-th bit of the result to be 0
            $q = \delta(q, b \oplus 1)$
        end
    end
end
```