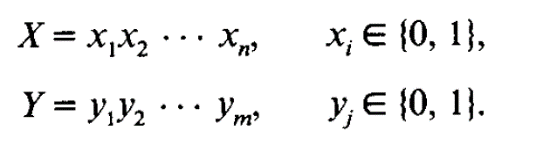
|  |
| --- |
| *\\samba1\vpandey\dcengr\Desktop\School_of_Engineering_Red_Logo.jpgCOEN 379: Advanced DesIGN & ANALYSIS OF ALGORITHM* |
| *Project* |
| *Rabin - Karp Fingerprint: A randomized approach to Pattern Matching problem* |
|  |
| ***Aaraadhya Nara, Himanshu Gupta & Varun Pandey*** |
| ***5/10/2015*** |

|  |
| --- |
| ***The project documentation contains following sub-sections: -*** |
| 1. C:\Users\Varun\Downloads\qr_code(1)\qr_code_without_logo.jpg*Universe outside Rabin Karp* 2. *Details of Rabin-Karp Algorithms* 3. *Mathematical Analysis* 4. *References* |

**Some Important Notations**



n <= m



**Overview: -**

We started with the analysis of other string matching techniques. String matching can be done in linear time by many algorithms. So what’s great about Rabin Karp algorithm? A beautiful analysis about Rabin Karp algorithm (aka RK) is that we convert the pattern matching problem to an arithmetic problem. This helps us to use the full capacity of processing unit and ALU operations. Use of limited hardware registers is an important benefits of RK.

We went on to read the actual RK paper and studied three different algorithm proposed by RK to have a heuristic/definitive match. We went on to analyze the algorithms to find the probability of false hit. We also implemented the algorithm. As a novelty, we implemented a parallel algorithm that uses RK (implementation in CUDA).

**Exact String Matching: The Fundamental String Problem**

**What is Exact Matching?**

Given a string *X* called the *pattern* and a longer string *Y* called the *text,* the exact matching problem is to find all occurrences, if any, of pattern *X* in text Y*.*

For example, if X = aba and text Y = bbabaxababay then X occurs in Y starting at positions 3, 7, and 9.Note that two occurrences of X may overlap as, illustrated by the occurrences of X at positions 7 and 9.

**Importance of exact matching problem:**

The practical importance of exact matching problem is very obvious. For example, the problem arises in a wide variety of applications like word processors, utilities like *grep*  on Unix, in textual information retrieval programs such as Medline, Lexis, or Nexis to name a few. With this we understand the practical importance of exact string matching problem, however one might ask hasn't exact matching been so well solved that it can be put in a black box and taken for granted? The answer is that for typical word-processing applications there is probably very little left to do. The exact matching problem is solved for these applications. But for other applications, the story changes. For example, users of Melvyl, the on-line catalog of University of California library system, often experiences long, frustrating delays even for fairly simple matching requests. Even *grepping* through a large directory can demonstrate that exact matching is not trivial. But perhaps the most important reason to study exact matching in detail is to understand the various ideas developed for it. The entire field of string algorithms remains vital and open and the education one gets from studying exact matching may be crucial for solving less understood problems.

Before explaining the approaches currently in use for exact string matching, it necessary to introduce some of the basic definitions used through this report.

***Definition:***  A *string S* is an ordered list of characters written contiguously from left to right. For any string *S*, *S*[i..j] is the contiguous *substring* of S that starts at position i and ends at position j of S. In particular, S[1..i] is the prefix of string S that ends at position i, and S[i..|S|] is the *suffix* of string S that begins at position i, where |S| denotes the number of characters in string S.

***Definition:*** S[i..j] is the empty string if i > j.

***Definition:*** A proper prefix, suffix, or substring of S is, respectively, a prefix, suffix, or substring that is not the entire string S, nor the empty string.

***Definition:***  For any string S, S(i) denotes the ith character of S.

***Definition:*** Whencomparing two characters, we say that the characters match if they are equal, otherwise we say that they mismatch.

**Universe Outside Rabin Karp Algorithm: -**

Three Algorithms: -

1. Naïve Algorithm
2. KMP
3. Boyerset. al.

**The Naive Method:**

Almost all discussions of exact matching begin with the naive method. The naive method aligns the left end of X and left end of Y and then compares the characters of X and Y left to right until either two unequal characters are found or until X is exhausted, in which case an occurrence of X is reported. In either case, X is then shifted one place to the right and the comparison is restarted from the left end of X. This process repeats until the right end of X shifts past the right end of Y. Using n to denote the length of X and m to denote the length of Y, the worst case time number of comparisons made by this method is  The naive method is easy to understand and implement, but its worst-case running time of  is unsatisfactory and can be improved.

Some ideas for speeding up the naive method try to shift X by more than one character when a mismatch occurs, but never shift it so far as to miss an occurrence of X in Y. One such approach is known as Boyer Moore algorithm.

**The Boyer-Moore Algorithm:**

As in Naive algorithm, Boyer-Moore algorithm successively aligns X and Y and then checks whether X matches the opposing characters of Y. Further, after the check is complete, X is shifted right relative to Y just as in the naive algorithm. However, Boyer-Moore algorithm contains three clever ideas not present in the naive algorithm: the right to left scan, the bad character shift rule, and the good suffix shift rule. The Right to Left Matching heuristic compares X with a subsequence of Y moving backwards. In the Bad Character Shift Rule, when a mismatch occurs at Y[i] = C and if X contains C, X is shifted so as to align the last occurrence of C in X with Y[i] otherwise, X is shifted such that X[0] aligns with Y[i+1]. Together, these ideas lead to a method that typically examines fewer than **m+n** characters (an expected sublinear-time method) however it can also have a worst case running time of **O(mn)** in certain situations.

The best known linear time algorithm for exact matching problem is due to Knuth, Morris and Pratt. Although it is rarely the method of choice, and is much inferior in practice to Boyer-Moore method, it easy to explain and its liner time bound can be easily proved.

**The Knuth-Morris-Pratt algorithm:**

KMP's algorithm compares the pattern to the text in the left-to-right manner, but shifts the pattern more intelligently than the naive algorithm. When a mismatch occurs, it shifts the pattern the most so as to avoid redundant comparisons. For a given alignment of X and Y, suppose the naive algorithm matches the first i characters of X against their counterparts in Y and then mismatches on the next comparison. The naive algorithm would shift X by just one place and begin comparing again from the left end of X. But larger shifts may often be possible. The KMP algorithm is based on this kind of reasoning to make larger shifts than the naive and Boyer-Moore algorithms make. For example the KMP algorithm searches for the occurrences of a X in text Y by employing the observation that when a mismatch occurs, the pattern itself embodies sufficient information to determine where the next match could begin, thus bypassing re-examination of previously matched characters. By analyzing the KMP algorithm we find that the KMP algorithm runs in optimal time of **O(m+n)** which is linear.

**Approaches of these algorithms:**

These above mentioned linear time exact matching algorithms (except naive algorithm) require, for fast implementation, O(n) registers to store a table of pointers. This table of pointers is created during the pre-processing step required in Boyer-Moore and KMP algorithms. Also these algorithms cannot be used with a multidimensional array of symbols or even irregularly shaped arrangements of symbols. Another characteristics of these algorithms is that they work by comparing characters between the pattern and the text and are deterministic algorithms. Also these algorithms are categorized as offline algorithms meaning that they require the text and pattern in their entirety, so as to perform pre-processing operations on the text and the pattern. They cannot work on a continuous stream of input.

The following summarizes the features and short-comings of the Naive, Boyer-Moore and KMP algorithms:

* Comparison based deterministic algorithms
* Requires pre-processing except Naive algorithm
* Store the pattern in memory
* Offline Algorithms
* Require O(n) registers to store the table of pointers

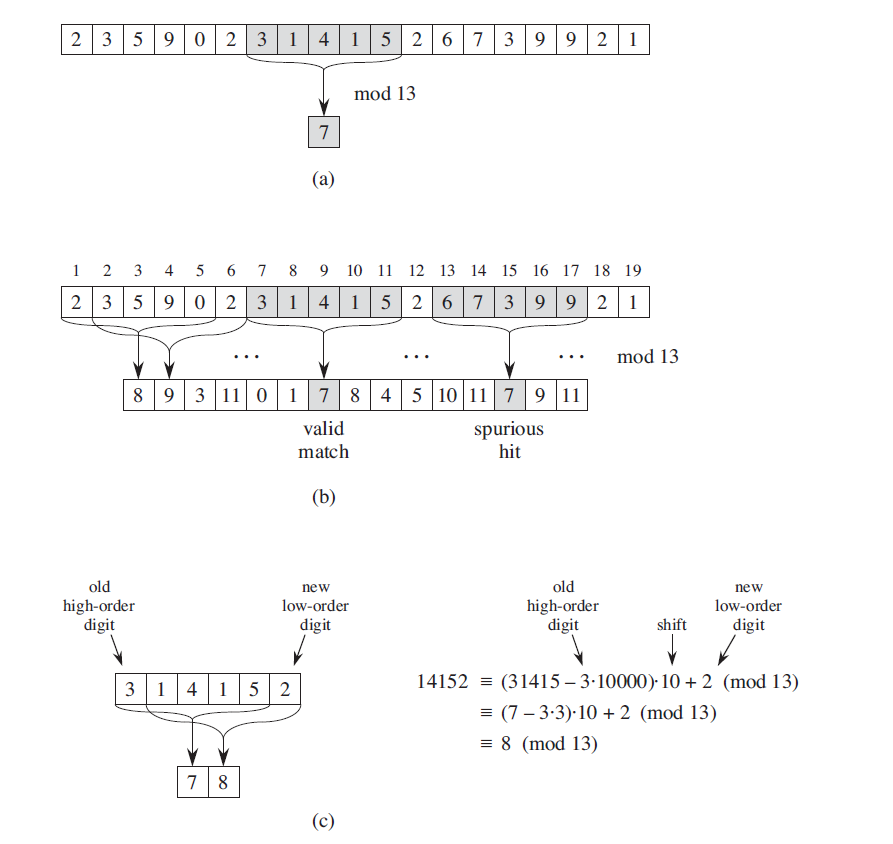
**Features of Rabin-Karp Algorithm:**

* Based on fingerprint function, so runs in real time
* Same theoretical time bounds as deterministic algorithms
* Requires a constant number of registers and requires a substring of length *n* of Text in memory
* Conceptually very simple and easy to implement
* Works well with long patterns as well as multi-dimensional patterns or arrays of symbols
* Faster because randomized algorithm
* No pre-processing required

**Rabin Karp Algorithm: -**

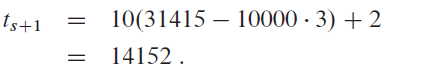
* Convert the String Comparison to an Arithmetic problem.
* Convert the pattern(X) of length n and Text of length(y) into Binary strings.
* Calculate the integer value and then compare the integers.

Stream of Text



31415

Pattern to search



**Finger Prints**

* Finger Print Functions:
* We work with the reduced modulo of a relatively small integer p
* Every fingerpring remains in the range of 0 to p-1
* With modulo arithmetic we can also computer fingerprints of from
* **Lemma 1 :**
* Using this lemma in the calculation of the number never exceeds 2p.

Algorithm 1:

Match :boolean;

r: position from which the substring is considered (always <m)

k: positive integer

for(i = 1 to k){

= ramdomly chosen element of S;

match = false;

r = 0;

while (match =false && r != m -1){

if( {

match =true;

r = r +1;

}

}

}

Algorithm2:

Match :boolean;

r: position from which the substring is considered (always <m)

k: positive integer

= ramdomly chosen element of S;

match = false;

r= 0;

while (match =false && r != m -1){

if( {

if( {

match =true;

}

r = r +1;

}

}

Algorithm 3:

Match :boolean;

r: position from which the substring is considered (always <m)

k: positive integer

= ramdomly chosen element of S;

match = false;

r = 0;

while (match =false && r != m -1){

if( {

if( {

match =true;

}

else{

= ramdomly chosen element of S;

}

r = r + 1;

}

}

|  |  |  |
| --- | --- | --- |
| Algorithm 1 | Algorithm 2 | Algorithm 3 |
| * Online * Uses k Randomly generated primes at a time | * Not Online * Not Requires one to one pattern checking on HIT * Considers only 1 prime | * Not Online * Not Requires one to one pattern checking on HIT * Considers a new prime every time we get a bad seed |

**Mathematical Analysis**

Expected Number of Trials to get the prime is ln M. The time to perform each primality test is O ((log M) 2) if we use Solovay and Strassen.

**Lemma 1:**

Proof: - Rosser and Schoenfeld in their paper stated that the

**Corollary 1:**

In short, we are saying here that if we can upper bound a power of 2 say u, then the number can have at most u - 1 prime divisors. We will use this later to proofs by counting the number of possible prime divisors

**Central Theorem:**

For a **fixed** a single false can occur if –

This also means that,

**p |**

Literally translated to “p completely divides the quantity ”

So, for 1 run of algorithm, for a single false match,

For each r, assuming that we are considering only binary alphabets. Max difference between the

Cosnsidering t shifts and we have maximum difference possible on each iteration,

By corollary 1, this product has < and it is for these primes will the false hit is expected. Total prime to choose from is at most. Therefore, the probability that false match will be –

This proves for how many primes out of M, a false match could occur. It does NOT give any idea about the entire run of the algorithm

QED

**Extension of Central Theorem**

If Algorithm 1 is executed, the

Why? We are picking k random independent prime, so probability of false match is equal to

… k times

YET! This wouldn’t provide the tight bound. Why? Because here we are considering that all of the primes resulted in false match. This may not happen – only 1 position out of t choices may have a false match. So probability that one of the hash is a false hit is

,

**So, # of choice for prime dividing this value at every position of r is**

Therefore, the probability that one of the primes result in false match at every for one run of the algorithm is,

…t times

**What is a good value of M?**

1. Large enough
2. Must support register operation
3. Good balance
4. Must support the Ram Model

How about if we take

With choice of largest number, we can have overflow operation at which requires bits. This is in order of number of bits require to

Applying Lemma 2,

This is good because it gives us a good probability to work with, depending only upon the available number of shifts.

Example: - Consider n = 250, t = 4000, M = nt2 = 4x109< 232. Then, for any p <= M, the bits required to store p is less than 32. So it can be stored in a register. Applying above formula, the probability of false match < 1000-1. For algorithm 1, if k = 4, Pr ≤ 2 x 10-22.

**Storage Requirement**

p is drawn from M ≤ nt2. So will require . Following are the operations needed for RK.

* 4 Fetch
* 3 Add
* 3 Compare
* 2 Subtraction
* 1 store

Hash computation is in constant time from Hr-1 to Hr.

Algorithm 1 requires (4k + 2) Registers

Algorithm 2, 3 requires 6 Registers

**Expected Time Spend in detecting the False Match:**

Algorithm 1 is real-time. So the complexity is O(m). Algorithm 2 & 3 requires confirmation and thus, the complexity is O(m+n).

So the expected time spend on detecting a false match is

Thus the complexity of Rabi Karp Algorithm is *O(m+n)*

**Novelty**

We implemented RK in three algorithms of Rabin Karp algorithm: -

1. A simple algorithm without caring about the overflow of fingerprint.
2. With mod prime restriction to fingerprint.
3. A parallel implementation of RK that essentially uses overlapping chunks of data writeen in CUDA. This algorithm is shelved currently as I am attempting to implement online version of the parallel algorithm.

*Pattern size n*

**Fig: -** Overlapping chunks From CUDA implementation

**References:**

1. Karp, Richard M., and Michael O. Rabin. "Efficient randomized pattern-matching algorithms." *IBM Journal of Research and Development* 31.2 (1987): 249-260.
2. Gusfield, Dan. *Algorithms on strings, trees and sequences: computer science and computational biology*. Cambridge university press, 1997.
3. Thomas H. Cormen et al. *Introduction to Algorithms*. 3nd edition. McGraw-Hill Higher Education, 2001. ISBN: 0070131511
4. Gonnet, Gaston H., and Ricardo A. Baeza-Yates. "An analysis of the Karp-Rabin string matching algorithm." *Information Processing Letters* 34.5 (1990): 271-274.
5. Barkley Rosser, J., and Lowell Schoenfeld. "Approximate formulas for some functions of prime numbers." *Illinois Journal of Mathematics* 6 (1962): 6z-9z.
6. Góngora-Blandón, Mario, and Miguel Vargas-Lombardo. "State of the art for string analysis and pattern search using cpu and gpu based programming." *Journal of Information Security* 3.04 (2012): 314.
7. Aronovich, Lior, et al. "The design of a similarity based deduplication system." *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*. ACM, 2009.
8. Some discreteyoutube videos and articles.