IMPERIAL COLLEGE LONDON, DEPARTMENT OF MATHEMATICS

Scientific Computing (M3SC)

Peter J. Schmid

January 22, 2017

1 PATTERN SEARCHING (RABIN-KARP ALGORITHM)

Pattern searching is an important discipline of computer science and applied mathematics. It is concerned with the extraction of a given pattern p of size m from a (commonly very large) text t of length n. The pattern and text can consist of numbers, letters or other distinct structures or tokens. Typical applications of pattern searching are encountered in plagiarism detection software or in the sequencing of genetic material.
Here we will consider sequential pattern searching, where a large string of text is searched for occurrences of shorter patterns. For example, in a long sequence of nucleotides within a DNA molecule we may be interested in searching for locations and frequency of a short pattern made up of the DNA alphabet $\{A, C, G, T\}$ which denotes the four building blocks, i.e., the nucleic acids **A**denine, **C**ytosine, **G**uanine and **T**hymine.

1.1 NAIVE SEARCH

A naive approach to pattern searching is the sequential comparison of the pattern to text snippets of equal length, then sliding along the entire text sequence. While this naive pattern searching algorithm often works better than expected, it exhibits its worst-case performance (and run times) when looking for rather long patterns in very long search strings. With a text of length n and a pattern of length m, we have to make a comparison of m entries in the pattern and substring for each of the positions in the text t. The run time of the algorithm thus scales as $\mathcal{O}(nm)$.
An implementation of a naive search algorithm is given below, written as a python function.

```
1 def naive_search (string,pattern):
```

```
    # naive string matching algorithm
    n  = len(string)
    m  = len(pattern)
    ic = 0
    # element-by-element matching
    for i in range(n-m+1):
        # call the (logical) match function
        if match(string,pattern,i):
            ic += 1
            print i
    print "number of matches: ", ic
```

This function uses another python function `match` that we will use later in the more advanced Rabin-Karp algorithm (see below). It is given by

```
def match(string,pattern,i):
    # element-by-element matching of a pattern
    # in a string; i: position in string
    for k in range(len(pattern)):
        if pattern[k] == string[i+k]:
            pass
        else:
            return False
    return True
```

1.2 Speeding up the naive-search algorithm by hashing

One possibility of speeding up the naive-search algorithm is by appling more sophisticated manners of traversing and skipping through the text, rather than the slow sliding-window approach. Another and alternative possibility is the speed-up of the matching procedure. This latter technique is what the Rabin-Karp algorithm proposes. The Rabin-Karp algorithm, named after Michael Rabin and Richard Karp, speeds up the matching section of the naive-search algorithm by introducing **hash functions**.
In general, a hash function is a mapping of a string or number sequence onto a numerical value, the **hash value**. Based on this hash function, the search pattern and a string segment match when their respective hash values match. This hash value is akin to the familiar checksum, when comparing two data files. The Rabin-Karp algorithm thus replaces a direct comparison of a given pattern and an extracted string segment by a comparison of their associated hash values. In other words, in the text sequence we search for substrings that have the same hash value as the pattern's hash value.
When introducing hash functions into the naive-search algorithm, two issues arise.
First, the mapping of a pattern onto a hash value may produce false positives, i.e., the hash values of the pattern and the text substring coincide, even though the actual substrings do not match. This is also referred to as **hash collision**. It is a consequence of the fact that

different text-substrings can map to the same hash value. While this non-uniqueness does not pose a problem, we simply need to compare the pattern and the real text segment after we receive a flag from the matching hash values. In other words, hash value matching does a fast preprocessing step, before direct comparison confirms or rejects the potential match. Since a direct comparison is relatively costly, we have to choose a proper hash function that does not produce too many false positivesm but rather keeps direct comparisons a rare event.
Second, the computing of the hash value has to be efficient, otherwise there will not be any savings over a direct comparison. Ideally, the comparison should not scale with the pattern size, but instead be constant in time. In this latter case, the run time $\mathcal{O}(nm)$ of the naive search would reduce to $\mathcal{O}(n)$, which is as good as one can expect from a sequential search algorithm.
With the hash function introduced, the Rabin-Karp string searching algorithm calculates a hash value for the pattern, and for each m-character subsequence of the text. If the hash values are unequal, the algorithm will calculate the hash value for the next m-character sequence. If the hash values are equal, the algorithm will compare the pattern and the m-character sequence directly. In this way, there is only one hash comparison per text subsequence, and character matching is only needed when hash values match.

1.3 A BIT OF MATHEMATICS

Consider an m-character sequence as an m-digit number expressed in base b, where b is at least the number of letters in the alphabet used in our text t. The text subsequence $t[i,\ldots,i+m-1]$ can then be mapped to the number x_i according to

$$x_i = t[i]\cdot b^{m-1} + t[i+1]\cdot b^{m-2} + \cdots + t[i+m-1]. \tag{1.1}$$

This expression already acts as a hash function, assigning a number x_i to a string $t[i,\ldots,i+m-1]$. If the text to be processed consists of letters (such as in plagiarism detection, or in DNA pattern searching), an additional mapping from the letters to the $t[i]$'s has to be introduced (see the example below). The above way of expressing a text sequence as a number using a polynomial in b is known as the **Rabin fingerprint**. It has the considerable advantage that the subsequent hash value x_{i+1} of the right-shifted m-digit subsequence $t[i+1,\ldots,i+m]$ can be computed in constant time. This can be easily shown by

$$\begin{aligned}
x_{i+1} &= t[i+1]\cdot b^{m-1} + t[i+2]\cdot b^{m-2} + \ldots + t[i+m], && \text{(1.2a)}\\
&= x_i \cdot b \qquad \text{(shift left one digit)} && \text{(1.2b)}\\
&\quad - t[i]\cdot b^m \qquad \text{(subtract leftmost digit)} && \text{(1.2c)}\\
&\quad + t[i+m]. \qquad \text{(add new rightmost digit)} && \text{(1.2d)}
\end{aligned}$$

In this manner, we avoid the explicit computation of a new hash value; rather, we simply adjust the current hash value as we shift our substring in the text by one character to the right. Hash functions that allow this type of shortcut are referred to as **rolling hash** functions; there are many others besides the Rabin fingerprint.

In the above expression, there still remains one problem. If the size m of the pattern gets large and the alphabet of the text is rather rich (and thus requires a large base b), the value of b^m that appears in the expression above will become exceedingly high. To counteract this trend, we limit the size of the hash value by introducing the modulo function, i.e., we clip the hash values to a finite number range. More specifically, we compute our hash values by taking the modulo (with an appropriate number q) of each component in the above expression.
The modulo function $\mod$ is particularly convenient since it satisfies the mathematical expressions

$$[(x \mod q) + (y \mod q)] \mod q = (x+y) \mod q, \tag{1.3a}$$
$$(x \mod q) \mod q = x \mod q. \tag{1.3b}$$

Applied to our rolling hash expression, and introducing the notation $h_i = x_i \mod q$, we obtain

$$\begin{aligned} h_{i+1} &= \left(t[i+1]\cdot b^{m-1} \mod q + t[i+2]\cdot b^{m-2} \mod q + \ldots + t[i+m] \mod q\right) \mod q \\ &= (h_i \cdot b \mod q \quad \text{(shift left one digit)} && \text{(1.4a)} \\ &\quad - t[i]\cdot b^m \mod q \quad \text{(subtract leftmost digit)} && \text{(1.4b)} \\ &\quad + t[i+m] \mod q) \mod q \quad \text{(add new rightmost digit)} && \text{(1.4c)} \end{aligned}$$

This latter expression is at the heart of the Rabin-Karp algorithm.
It leaves the final questin of what value to choose for q. A small number will increase the frequency of hash collisions, the necessity to perform an excessive number of direct comparisons and, consequently, a serious degradation in performance. For example, when choosing $q = 3$, we only produce hash values of $h_i = 0, 1, 2$, which may be far too limited to properly represent all the different text substrings. Instead, a large prime number is commonly chosen for q.

1.4 Algorithm

With this we can state the Rabin-Karp algorithm.
The Rabin-Karp pattern search algorithm computes a hash value for a search pattern p of length m and for each m-character substring of a text t. The two hash values are compared, rather than the actual strings. Only after the two values match does the algorithm compare the two patterns (search pattern and substring) to confirm or reject a match, thus resolving a hash collision. If no match is found, the algorithm shifts to the next substring of t. When a rolling hash is used, this latter shift can be computed very efficiently.
We have the following python functions, implementing the Rabin-Karp string search algorithm.

```
1 import sys
2 import time
```

```
3
4  def RabinKarp(str,pat,base=256,modu=16647133):
5      # Rabin-Karp string matching algorithm
6      #    str:  text to be searched
7      #    pat:  pattern to match
8      #    base: base of the Rabin-fingerprint
9      #    modu: modulus of the hash-function
10     ic = 0
11     n  = len(str)
12     m  = len(pat)
13     # compute the hash values for pattern
14     # and initial string (Horner's scheme)
15     hp = 0
16     for i in pat:
17         hp = (base*hp + ord(i))%modu
18     hs = 0
19     for i in str[:m]:
20         hs = (base*hs + ord(i))%modu
21     # check for initial match
22     if hs == hp and match(str[:m],pat):
23         print 0
24         ic = 1
25     # compute b^m
26     bm = 1
27     for i in range(m-1):
28         bm = (bm*base)%modu
29     # main loop
30     for i in range(1, n - m + 1):
31         hs = (base*(hs - bm*ord(str[i-1])) + \
32               ord(str[i+m-1]))%modu
33         if (hs == hp):
34             if match(str[i:i+m],pat):
35                 print i,
36                 ic += 1
37     print "\n number of matches found: ", ic
```

Finally, the main code contains material that applies the Rabin-Karp algorithm and the naive search algorithm to two applications: a data-base of genetic material from the DNA of the fruit fly (*Drosophila melanogaster*) which contains more than 23 million nucleotide, and the first 100000 digits of π.

```
if __name__ == '__main__':

    # read 'Genetic Material' file
    f     = open('GeneticMaterialLarge','r')
    strGM = f.read()
    f.close()
    strGM = strGM.replace("\n","")

    # patterns to search for
    p1  = 'CACAATATATGATCGC'
    p2  = 'GTGCCAACATATTGTGCTCTCTATATAATGACTGCCTCT'
    p3  = 'GCGAATATGGAAAAAAAGGCAGACACACACTTGTTGATTTCTTTATGCG'
    pa  = 'ATTTCTACTGTAAAAATCTTTCTTTGTCCAAAAAATCTTTGATAGTGTTT'
    pb  = 'CCCTATGTTTGTGTGCACGGACACGTGCTAAAGCCAAATATTTTGCTAGT'
    pc  = 'CAATATATAACAAAAAATATTTTTCGGGCTTGCGTACTGTAAAGATGAAT'
    p4  = pa + pb + pc
    patGM = p1

    # read 'Digits of Pi' file
    f     = open('DigitsOfPi','r')
    strPi = f.read()
    f.close()

    # pattern to search for
    p1 = '8888'
    p2 = '2479114016957900338356'
    patPi = p1

    # scan the Genetic Material file
    t0    = time.time()
    RabinKarp(strGM,patGM)
    t1    = time.time()
    total = t1-t0
    print 'Genetic Material (Rabin-Karp algorithm): ',total

    t0    = time.time()
    naiveSearch(strGM,patGM)
    t1    = time.time()
    total = t1-t0
    print 'Genetic Material (naive search): ',total

    # scan the Digits of Pi file
    t0    = time.time()
```

```
44      RabinKarp(strPi,patPi)
45      t1    = time.time()
46      total = t1-t0
47      print 'Digits of Pi (Rabin-Karp algorithm): ',total
48
49      t0    = time.time()
50      naiveSearch(strPi,patPi)
51      t1    = time.time()
52      total = t1-t0
53      print 'Digits of Pi (naive search): ',total
```

The main code includes a time comparison between the naive search and the Rabin-Karp algorithm. Even though the difference may not appear this drastic, one has to keep in mind that the algorithm is commonly applied to far larger data-bases and to larger patterns. In this case, the difference in run-time between naive searching and Rabin-Karp will be more pronounced.

1.5 Improvements and applications

The Rabin-Karp algorithm has been presented in its most basic form. However, even in this rudimentary form it is commonly applied in bioinformatics (when searching for patterns in nucleotides), text processing (such as plagiarism detection), music prototyping (for search engines) or data compression.
Nonetheless, there are improvements to the algorithms that increase its efficiency and applicability to a wide range of data. One of the most common is the use of prefix and suffix trees. Prefix and suffix strings are smaller substrings attached in front or at the end of a string that allow the early dismissal of mismatching strings and the skipping ahead by more than one record. Taking advantage of prefix and suffix strings can produce substantial speedup in the overall search for patterns.
The Rabin-Karp algorithm can quite easily be extended to multiple-pattern searches. In this case, we are looking for occurrences in the form of a vector-valued shift (s_1, s_2) (in the case of two patterns). Facial recognition and medical imaging are areas where two-dimensional pattern searches are commonly encountered.
Another extension of the Rabin-Karp algorithm involves the occurrence of wild cards, which are taken as matching *any* pattern. This situation is often encountered when dealing with DNA transcription factors. These factors need to be ignored (labelled as wild cards) when searching for valid patterns.

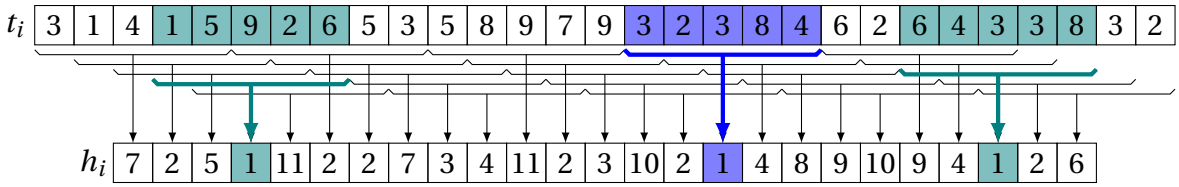

Figure 1.1: Demonstration of the Rabin-Karp algorithm: search for the string '32384' in the digits of π. We employ the algorithm with radix 10 (i.e., a representation of the digits in base 10) and a modulo 13 function. The hash value of the pattern is 1. We recognize the string in position 16 in the text (highlighted in blue), but we also encounter two hash collisions (in green) in positions 4 and 23, where the hash value matches, but not the actual substring.

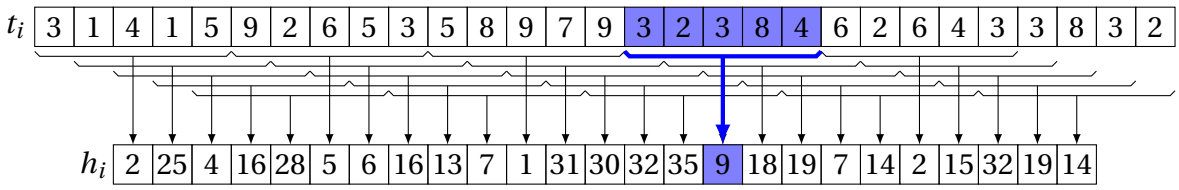

Figure 1.2: Demonstration of the Rabin-Karp algorithm: search for the same string '32384' in the digits of π. This time, we choose a modulo 37 function. The hash value of the pattern is 9, and we do not encounter hash collisions.