Rabin–Karp algorithm

From Wikipedia, the free encyclopedia
(Redirected from Rabin-Karp string search algorithm)

In computer science, the **Rabin–Karp algorithm** is a string searching algorithm created by Michael O. Rabin and Richard M. Karp in 1987 that uses hashing to find any one of a set of pattern strings in a text. For text of length n and p patterns of combined length m, its average and best case running time is $O(n+m)$ in space $O(p)$, but its worst-case time is $O(nm)$. In contrast, the Aho–Corasick string matching algorithm has asymptotic worst-time complexity $O(n+m)$ in space $O(m)$.

A practical application of Rabin–Karp is detecting plagiarism. Given source material, Rabin–Karp can rapidly search through a paper for instances of sentences from the source material, ignoring details such as case and punctuation. Because of the abundance of the sought strings, single-string searching algorithms are impractical.

Contents

Shifting substrings search and competing algorithms

A brute-force substring search algorithm checks all possible positions:

```
1 function NaiveSearch(string s[1..n], string sub[1..m])
2    for i from 1 to n-m+1
3       for j from 1 to m
4          if s[i+j-1] ≠ sub[j]
5             jump to next iteration of outer loop
6       return i
7    return not found
```

This algorithm works well in many practical cases, but can exhibit relatively long running times on certain examples, such as searching for a string of 10,000 "a"s followed by a "b" in a string of 10 million "a"s, in which case it exhibits its worst-case $\Theta(mn)$ time.

The Knuth–Morris–Pratt algorithm reduces this to $\Theta(n)$ time using precomputation to examine each text character only once; the Boyer–Moore algorithm skips forward not by 1 character, but by as many as possible for the search to succeed, effectively decreasing the number of times we iterate through the outer loop, so that the number of characters examined can be as small as n/m in the best case. The Rabin–Karp algorithm focuses instead on speeding up lines 3-6.

Use of hashing for shifting substring search

Rather than pursuing more sophisticated skipping, the Rabin–Karp algorithm seeks to speed up the testing

of equality of the pattern to the substrings in the text by using a hash function. A hash function is a function which converts every string into a numeric value, called its *hash value*; for example, we might have hash("hello")=5. Rabin–Karp exploits the fact that if two strings are equal, their hash values are also equal. Thus, it would seem all we have to do is compute the hash value of the substring we're searching for, and then look for a substring with the same hash value.

However, there are two problems with this. First, because there are so many different strings, to keep the hash values small we have to assign some strings the same number. This means that if the hash values match, the strings might not match; we have to verify that they do, which can take a long time for long substrings. Luckily, a good hash function promises us that on most reasonable inputs, this won't happen too often, which keeps the average search time good.

The algorithm is as shown:

```
1 function RabinKarp(string s[1..n], string sub[1..m])
2     hsub := hash(sub[1..m]);  hs := hash(s[1..m])
3     for i from 1 to n-m+1
4         if hs = hsub
5             if s[i..i+m-1] = sub
6                 return i
7         hs := hash(s[i+1..i+m])
8     return not found
```

Lines 2, 5, and 7 each require Θ(m) time. However, line 2 is only executed once, and line 5 is only executed if the hash values match, which is unlikely to happen more than a few times. Line 4 is executed *n* times, but only requires constant time. So the only problem is line 7.

If we naively recompute the hash value for the substring `s[i+1..i+m]`, this would require Θ(*m*) time, and since this is done on each loop, the algorithm would require Ω(mn) time, the same as the most naive algorithms. The trick to solving this is to note that the variable `hs` already contains the hash value of `s[i..i+m-1]`. If we can use this to compute the next hash value in constant time, then our problem will be solved.

We do this using what is called a rolling hash. A rolling hash is a hash function specially designed to enable this operation. One simple example is adding up the values of each character in the substring. Then, we can use this formula to compute the next hash value in constant time:

```
s[i+1..i+m] = s[i..i+m-1] - s[i] + s[i+m]
```

This simple function works, but will result in statement 5 being executed more often than other more sophisticated rolling hash functions such as those discussed in the next section.

Notice that if we're very unlucky, or have a very bad hash function such as a constant function, line 5 might very well be executed *n* times, on every iteration of the loop. Because it requires Θ(m) time, the whole algorithm then takes a worst-case Θ(mn) time.

Hash function used

The key to Rabin–Karp performance is the efficient computation of hash values of the successive substrings of the text. One popular and effective rolling hash function treats every substring as a number in some base, the base being usually a large prime. For example, if the substring is "hi" and the base is 101, the hash value would be $104 \times 101^1 + 105 \times 101^0 = 10609$ (ASCII of 'h' is 104 and of 'i' is 105).

Technically, this algorithm is only similar to the true number in a non-decimal system representation, since for example we could have the "base" less than one of the "digits". See hash function for a much more detailed discussion. The essential benefit achieved by such representation is that it is possible to compute the hash value of the next substring from the previous one by doing only a constant number of operations, independent of the substrings' lengths.

For example, if we have text "abracadabra" and we are searching for a pattern of length 3, we can compute the hash of "bra" from the hash for "abr" (the previous substring) by subtracting the number added for the first 'a' of "abr", i.e. 97×101^2 (97 is ASCII for 'a' and 101 is the base we are using), multiplying by the base and adding for the last a of "bra", i.e. $97 \times 101^0 = 97$. If the substrings in question are long, this algorithm achieves great savings compared with many other hashing schemes.

Theoretically, there exist other algorithms that could provide convenient recomputation, e.g. multiplying together ASCII values of all characters so that shifting substring would only entail dividing by the first character and multiplying by the last. The limitation, however, is the limited size of the integer data type and the necessity of using modular arithmetic to scale down the hash results, for which see hash function article; meanwhile, those naive hash functions that would not produce large numbers quickly, like just adding ASCII values, are likely to cause many hash collisions and hence slow down the algorithm. Hence the described hash function is typically the preferred one in Rabin–Karp.

Rabin–Karp and multiple pattern search

Rabin–Karp is inferior for single pattern searching to Knuth–Morris–Pratt algorithm, Boyer–Moore string search algorithm and other faster single pattern string searching algorithms because of its slow worst case behavior. However, Rabin–Karp is an algorithm of choice for multiple pattern search.

That is, if we want to find any of a large number, say *k*, fixed length patterns in a text, we can create a simple variant of Rabin–Karp that uses a Bloom filter or a set data structure to check whether the hash of a given string belongs to a set of hash values of patterns we are looking for:

```
function RabinKarpSet(string s[1..n], set of string subs, m):
    set hsubs := emptySet
    for each sub in subs
        insert hash(sub[1..m]) into hsubs
    hs := hash(s[1..m])
    for i from 1 to n-m+1
        if hs ∈ hsubs and s[i..i+m-1] ∈ subs
            return i
        hs := hash(s[i+1..i+m])
    return not found
```

We assume all the substrings have a fixed length *m*.

A naïve way to search for *k* patterns is to repeat a single-pattern search taking O(*n*) time, totalling in O(*n k*) time. In contrast, the variant Rabin–Karp above can find all *k* patterns in O(*n*+*k*) time in expectation, because a hash table checks whether a substring hash equals any of the pattern hashes in O(1) time.

References

- Karp, Richard M.; Rabin, Michael O. (March 1987). *Efficient randomized pattern-matching algorithms* (http://www.research.ibm.com/journal/rd/312/ibmrd3102P.pdf) . **31**. http://www.research.ibm.com/journal/rd/312/ibmrd3102P.pdf. Retrieved 2008-10-14.
- Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2001-09-01). "The

Rabin–Karp algorithm". *Introduction to Algorithms* (2nd ed.). Cambridge, Massachusetts: MIT Press. pp. 911–916. ISBN 978-0-262-03293-3.
- K. Selçuk Candan; Maria Luisa Sapino (2010). *Data Management for Multimedia Retrieval* (http://books.google.com/books?id=Uk9tyXgQME8C&pg=PA205) . Cambridge University Press. pp. 205–206. ISBN 978-0-521-88739-7. http://books.google.com/books?id=Uk9tyXgQME8C&pg=PA205. (for the Bloom filter extension)