WIKIPEDIA
The Free Encyclopedia

Rabin–Karp algorithm

In computer science, the **Rabin–Karp algorithm** or **Karp–Rabin algorithm** is a string-searching algorithm created by Richard M. Karp and Michael O. Rabin (1987) that uses hashing to find an exact match of a pattern string in a text. It uses a rolling hash to quickly filter out positions of the text that cannot match the pattern, and then checks for a match at the remaining positions. Generalizations of the same idea can be used to find more than one match of a single pattern, or to find matches for more than one pattern.

Rabin-Karp algorithm

Class	String searching
Worst-case performance	$O(mn)$ plus $O(m)$ preprocessing time
Average performance	$O(n)$
Worst-case space complexity	$O(1)$

To find a single match of a single pattern, the expected time of the algorithm is linear in the combined length of the pattern and text, although its worst-case time complexity is the product of the two lengths. To find multiple matches, the expected time is linear in the input lengths, plus the combined length of all the matches, which could be greater than linear. In contrast, the Aho–Corasick algorithm can find all matches of multiple patterns in worst-case time and space linear in the input length and the number of matches (instead of the total length of the matches).

A practical application of the algorithm is detecting plagiarism. Given source material, the algorithm 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.

Overview

A naive string matching algorithm compares the given pattern against all positions in the given text. Each comparison takes time proportional to the length of the pattern, and the number of positions is proportional to the length of the text. Therefore, the worst-case time for such a method is proportional to the product of the two lengths. In many practical cases, this time can be significantly reduced by cutting short the comparison at each position as soon as a mismatch is found, but this idea cannot guarantee any speedup.

Several string-matching algorithms, including the Knuth–Morris–Pratt algorithm and the Boyer–Moore string-search algorithm, reduce the worst-case time for string matching by extracting more information from each mismatch, allowing them to skip over positions of the text that are guaranteed not to match the pattern. The Rabin–Karp algorithm instead achieves its speedup by using a hash function to quickly perform an approximate check for each position, and then only performing an exact comparison at the positions that pass this approximate check.

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`. If two strings are equal, their hash values are also equal. For a well-designed hash function, the inverse is true, in an approximate sense: strings that are unequal are very unlikely to have equal hash values. The Rabin–Karp algorithm proceeds by computing, at each position of the text, the hash value of a string starting at that position with the same length as the pattern. If this hash value equals the hash value of the pattern, it performs a full comparison at that position.

In order for this to work well, the hash function should be selected randomly from a family of hash functions that are unlikely to produce many false positives, that is, positions of the text which have the same hash value as the pattern but do not actually match the pattern. These positions contribute to the running time of the algorithm unnecessarily, without producing a match. Additionally, the hash function used should be a rolling hash, a hash function whose value can be quickly updated from each position of the text to the next. Recomputing the hash function from scratch at each position would be too slow.

The algorithm

The algorithm is as shown:

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

Lines 2, 4, and 6 each require O(m) time. However, line 2 is only executed once, and line 6 is only executed if the hash values match, which is unlikely to happen more than a few times. Line 5 is executed O(n) times, but each comparison only requires constant time, so its impact is O(n). The issue is line 4.

Naively computing the hash value for the substring `s[i+1..i+m]` requires O(m) time because each character is examined. Since the hash computation is done on each loop, the algorithm with a naive hash computation requires O(mn) time, the same complexity as a straightforward string matching algorithm. For speed, the hash must be computed in constant time. The trick is the variable `hs` already contains the previous hash value of `s[i..i+m-1]`. If that value can be used to compute the next hash value in constant time, then computing successive hash values will be fast.

The trick can be exploited using a rolling hash. A rolling hash is a hash function specially designed to enable this operation. A trivial (but not very good) rolling hash function just adds the values of each character in the substring. This rolling hash formula can compute the next hash value from the previous 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.

Good performance requires a good hashing function for the encountered data. If the hashing is poor (such as producing the same hash value for every input), then line 6 would be executed O(n) times (i.e. on every iteration of the loop). Because character-by-character comparison of strings with length m takes O(m) time, the whole algorithm then takes a worst-case O(mn) time.

Hash function used

The key to the Rabin–Karp algorithm's performance is the efficient computation of hash values of the successive substrings of the text. The Rabin fingerprint is a popular and effective rolling hash function. The hash function described here is not a Rabin fingerprint, but it works equally well. It treats every substring as a number in some base, the base being usually the size of the character set.

For example, if the substring is "hi", the base is 256, and prime modulus is 101, then the hash value would be

```
 [(104 × 256 ) % 101  + 105] % 101  =  65
 (ASCII of 'h' is 104 and of 'i' is 105)
```

'%' is 'mod' or modulo, or remainder after integer division, operator

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 using a rolling hash such as the Rabin fingerprint 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, the hash of the first substring, "abr", using 256 as the base, and 101 as the prime modulus is:

```
// ASCII a = 97, b = 98, r = 114.
hash("abr") =  [ ( [ ( [  (97 × 256) % 101 + 98 ] % 101 ) × 256 ] %  101 ) + 114 ]   % 101   =  4
```

We can then compute the hash of the next substring, "bra", from the hash of "abr" by subtracting the number added for the first 'a' of "abr", i.e. 97 × 256^2, multiplying by the base and adding for the last a of "bra", i.e. 97 × 256^0. Like so:

```
//           old hash   (-ve avoider)*   old 'a'   left base offset      base shift    new 'a'    prime
modulus
hash("bra") =     [ ( 4   + 101         -  97 * [(256%101)*256] % 101 ) * 256         +    97 ] % 101
=  30
```

* (-ve avoider) = "underflow avoider". Necessary if using unsigned integers for calculations. Because we know all hashes $h \leq p$ for prime modulus \$p\$, we can ensure no underflow by adding p to the old hash before subtracting the value corresponding to the old 'a' (mod p).

the last '* 256' is the shift of the subtracted hash to the left

although ((256%101)*256)%101 is the same as 256^2 % 101, to avoid overflowing integer maximums when the pattern string is longer (e.g. 'Rabin-Karp' is 10 characters, 256^9 is the offset without modulation), the pattern length base offset is pre-calculated in a loop, modulating the result each iteration

If we are matching the search string "bra", using similar calculation of hash("abr"),

```
hash'("bra") =  [ ( [ ( [ ( 98 × 256) %101  + 114] % 101 ) × 256 ] % 101) + 97 ] % 101 = 30
```

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 the previous hash by the first character value, then multiplying by the new last character's value. 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, (see hash function article). Meanwhile, naive hash functions do not produce large numbers quickly, but, just like 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 the Rabin–Karp algorithm.

Multiple pattern search

input will be given all at once ??

The Rabin–Karp algorithm 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, it is a useful algorithm for multiple pattern search.

To find any of a large number, say *k*, fixed length patterns in a text, a simple variant of the Rabin–Karp algorithm 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
    foreach 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 $O(n+m)$ time, totaling in $O((n+m)k)$ time. In contrast, the above algorithm can find all *k* patterns in $O(n+km)$ expected time, assuming that a hash table check works in $O(1)$ expected time.

References

Sources

- Candan, K. Selçuk; Sapino, Maria Luisa (2010). *Data Management for Multimedia Retrieval* (https://books.google.com/books?id=Uk9tyXgQME8C&pg=PA205). Cambridge University Press. pp. 205–206. ISBN 978-0-521-88739-7. (for the Bloom filter extension)
- Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2001-09-01) [1990]. "The Rabin–Karp algorithm". *Introduction to Algorithms* (2nd ed.). Cambridge, Massachusetts: MIT Press. pp. 911–916. ISBN 978-0-262-03293-3.
- Karp, Richard M.; Rabin, Michael O. (March 1987). "Efficient randomized pattern-matching algorithms". *IBM Journal of Research and Development*. **31** (2): 249–260.