Rabin–Karp algorithm

From Wikipedia, the free encyclopedia

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 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 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.

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 pattern[1..m])
2    for i from 1 to n-m+1
3       for j from 1 to m
4          if s[i+j-1] ≠ pattern[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 pattern string of 10,000 "a"s followed by a single "b" in a search string of 10 million "a"s, in which case it exhibits its worst-case $O(mn)$ time.

The Knuth–Morris–Pratt algorithm reduces this to $O(n)$ 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 3-5.

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`. The algorithm exploits the fact that if two strings are equal, their hash values are also equal. Thus, string matching is reduced (almost) to computing the hash value of the search pattern and then looking for substrings of the input string with that hash value.

However, there are two problems with this approach. First, because there are so many different strings and so few hash values, some differing strings will have the same hash value. If the hash values match, the pattern and the substring may not match; consequently, the potential match of search pattern and the substring must be confirmed by comparing them; that comparison can take a long time for long substrings. Luckily, a good hash function on reasonable strings usually does not have many collisions, so the expected search time will be acceptable.

The algorithm is as shown:

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

Lines 2, 5, and 7 each require O(*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.

Simply recomputing 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 simple hash computation requires O(mn) time, the same complexity as a straightforward string matching algorithms. For speed, the hash must be computed in constant time. The trick is the variable `hs` already contains the hash value of `s[i..i+m-1]`. If that value can be used this 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 good hashing function for the encountered data. If the hashing is poor (such as producing the same hash value for every input), then line 5 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 Rabin fingerprint 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 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 101 as base is:

```
// ASCII a = 97, b = 98, r = 114.
hash("abr") = (97 × 101^2) + (98 × 101^1) + (114 × 101^0) = 999,509
```

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. 97×101^2, multiplying by the base and adding for the last a of "bra", i.e. 97×101^0. Like so:

```
//              base   old hash    old 'a'         new 'a'
hash("bra") = [101 × (999,509 - (97 × 101^2))] + (97 × 101^0) = 1,011,309
```

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, (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

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 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 the Rabin–Karp algorithm 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
    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, totalling in O(*(n+m)k*) time. In contrast, the variant algorithm above can find all *k* patterns in O(*n*+*km*) 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". *IBM Journal of Research and Development*. **31** (2): 249–260. CiteSeerX 10.1.1.86.9502. doi:10.1147/rd.312.0249.
- 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.
- Candan, K. Selçuk; Sapino, Maria Luisa (2010). *Data Management for Multimedia Retrieval*. Cambridge University Press. pp. 205–206. ISBN 978-0-521-88739-7. (for the Bloom filter extension)

External links

- "Rabin–Karp Algorithm/Rolling Hash" (PDF). *MIT 6.006: Introduction to Algorithms 2011- Lecture Notes*. MIT.

Categories: String matching algorithms | Hashing
