CS 1501 Essay 3

11/1/2017

Rabin-Karp Algorithm

Reach higher peak performance with our new invention!

Remember the old times of searching through your thirty pages plus documents in pursuit of a simple citation, or just simply recalling what you wrote on page ten, paragraph five, in the fourth sentence? Probably, you’re still doing that! If you change to our new string pattern matching algorithm, you will receive faster access to the information that you want! In fact, the larger, more detailed the document you create, the more it will outlast and outperform the slow, inefficient, atrocious pattern matching algorithm you are using right now! I and my colleague were able to reach speeds twice faster than other competitors through some of our detailed and well-planned test. Now who would not want our new invention? People who are sitting at work, home, or school and need to complete documents for their boss, papers for class, or making simply diary and journal would definitely want our new product rather than keep using slower software with outdated algorithms. We, inventors, know well that you must search the document for words or phrases at some point and you do not want to waste so much time just because current algorithm you are using is too slow. You have nothing to lose and everything to gain, so go ahead, please use our cutting-edge patter matching algorithm in full potential. Now let us explain about new innovative string patter matching algorithm, Rabin-Karp Algorithm.

**Description**

The method developed by us is very different approach to substring search that is based on hashing. We compute given hash function for the pattern and find the match by using same hash function for each possible M character substring (M: length of pattern string) of the some text. If text substring with same hash value as the pattern is found, we can check for match. The process is same as storing the pattern in hash table and then performing search for each substring of the text, however, we do not have to keep extra memory space for hash table because it will have only one entry. By just looking at above description, it seems that it would be slower than brute-force search because computing a hash function for every character is likely to be more expensive performance-wise than just comparing characters. But we actually find that it is easy to perform hash functions for M character substrings in O(1) time and it leads to linear time for search in practical situations.

Let’s assume string of length M corresponds to M-digit base-R number. In order to use a hash table (size Q) for keys of this type, it needs hash function to convert M-digit base-R number to int value between 0 and Q-1. Modular hashing (taking remainder when dividing number by Q) can be useful in this situation. For practical situation, we should use random prime Q to avoid overflow. To explain the method very simply, an example with small number Q and R being equal to 10 should be given. For example, to search the pattern 26535 in the text 3141592653589793, we choose hash table size Q as 997 (prime number) and compute hash value (modular hashing) by dividing 26535 by 997 and take remainder which is 613. Then, look for match by computing hash values for each five digit substring in the text. After some work being done, we get hash values 508, 201, 715, 971, 442, 929 before finding the match 613.

It is not so difficult when M is only five length (Five digit values). What about M being 100 or 1000. We may use simple application of Horner’s method (computes hash function for M-digit base-R number represented as char array in time proportional to M). But this will cost a lot for substring search in the text because of too much multiplication, addition, remainder calculation for each text character for total NM operations in worst case. Is there a way to improve this? Answer is yes.

Our Rabin-Karp method is based on efficiently computing hash function for position i +1 (given position i) in the text. Using notation ti for text.charAt(i), the number corresponding to M character substring of text starting at position i is: xi = tiRM-1 + ti+1RM-2 + . . . + ti+M-1R0 and assume that we know value of h(xi) = xi mod Q. So moving one position right in the text corresponds to replacing xi by number: xi+1 = (xi - tiRM-1) R + ti+M. We subtract leading digit, multiply by R and add trailing digit. Now important part is that we do not need to maintain values of numbers but values of remainders when divided by Q. Modular operation has a property that if we take the remainder when divided by Q after each arithmetic operation, then get the same answer as if perform all of the arithmetic operations, then take remainder when divided by Q. We take advantage of this property. It leads to effectively moving right one position in the text in constant time even though M is 100 or 100. How can we implement this algorithm? We can build the constructor that computes hash value patternHash for the pattern and the value of RM-1 mod Q in the variable RM. Then, we can make search() method that computes hash function for first M characters of the text and compares that value with the hash value for the pattern. If not matched, it goes through the text string by using technique described in above passage to sustain hash function for M characters at position i and comparing new hash value to patternHash.

There are other string pattern matching algorithms like brute-force search, KMP, and Boyer-Moore string pattern matching. The KMP matching algorithm uses one property: pattern has same sub patterns emerging more than once in the pattern. It builds DFA from the pattern. Key idea behind KMP is that whenever detecting a mismatch, we already know some of characters in the text of next window. We can take advantage of this information to avoid comparing the characters that we know will match no matter what. Its time complexity is O(n) in worst case. Another algorithm, Boyer-Moore also preprocesses the pattern. It is combination of two approaches: bad character heuristic (mismatch) and good suffix heuristic. Unlike KMP algorithm, Boyer-Moore starts matching from the last character of pattern. The idea of mismatched character heuristic is simple; upon mismatch, we shift the pattern until the mismatch becomes a match or pattern moves past the mismatched character. Its running time takes O(n/m) in best case. It seems that Boyer-Moore is very competitive algorithm against our Rabin-Karp since its performance is sublinear. However, our Rabin-Karp is simple and easy to implement. Also, both KMP and Boyer-Moore need to have extra space of R (256 ASCII characters) while Rabin-Karp literally does not take any extra space. Our algorithm is linear in most time. Also, many string pattern matching algorithms require backup in the text string while Rabin-Karp does not require backup when we need to make sure of true match, not hash collision. Instead, we can make hash table size as large as we want because we are not building hash table for testing collision with one key or pattern. Using very large number like greater than 1030 will make probability that a random key hashes to the same value as the pattern less than 10-30, a very small value. This can guarantee completion time to be more linear.

Lastly, our new invention can be really helpful to your document searching in any situation like school, home, or workplace. We acknowledge some string pattern matching algorithms such as Boyer-Moore are currently being used and Boyer-Moore can out-perform our Rabin-Karp in some situations. I think that is only shortcoming of our algorithm and we need to work on that part to improve our algorithm. Since Rabin-Karp has its own advantages over other algorithms, our future enhanced Rabin Karp with efficient time performance equal to or greater than Boyer-Moore will be best string pattern matching algorithm out there.