Algorithms on Strings, Trees, and Sequences

COMPUTER SCIENCE AND COMPUTATIONAL BIOLOGY

Dan Gusfield

University of California, Davis

CAMBRIDGE
UNIVERSITY PRESS

1

Exact Matching: Fundamental Preprocessing and First Algorithms

1.1. The naive method

Almost all discussions of exact matching begin with the *naive method*, and we follow this tradition. The naive method aligns the left end of P with the left end of T and then compares the characters of P and T left to right until either two unequal characters are found or until P is exhausted, in which case an occurrence of P is reported. In either case, P is then shifted one place to the right, and the comparisons are restarted from the left end of P. This process repeats until the right end of P shifts past the right end of T.

Using n to denote the length of P and m to denote the length of T, the worst-case number of comparisons made by this method is $\Theta(nm)$. In particular, if both P and T consist of the same repeated character, then there is an occurrence of P at each of the first $m - n + 1$ positions of T and the method performs exactly $n(m - n + 1)$ comparisons. For example, if $P = aaa$ and $T = aaaaaaaaaa$ then $n = 3, m = 10$, and 24 comparisons are made.

The naive method is certainly simple to understand and program, but its worst-case running time of $\Theta(nm)$ may be unsatisfactory and can be improved. Even the practical running time of the naive method may be too slow for larger texts and patterns. Early on, there were several related ideas to improve the naive method, both in practice and in worst case. The result is that the $\Theta(n \times m)$ worst-case bound can be reduced to $O(n + m)$. Changing "$\times$" to "+" in the bound is extremely significant (try $n = 1000$ and $m = 10{,}000{,}000$, which are realistic numbers in some applications).

1.1.1. Early ideas for speeding up the naive method

The first ideas for speeding up the naive method all try to shift P by more than one character when a mismatch occurs, but never shift it so far as to miss an occurrence of P in T. Shifting by more than one position saves comparisons since it moves P through T more rapidly. In addition to shifting by larger amounts, some methods try to reduce comparisons by skipping over parts of the pattern after the shift. We will examine many of these ideas in detail.

Figure 1.1 gives a flavor of these ideas, using $P = abxyabxz$ and $T = xabxyabxyabxz$. Note that an occurrence of P begins at location 6 of T. The naive algorithm first aligns P at the left end of T, immediately finds a mismatch, and shifts P by one position. It then finds that the next seven comparisons are matches and that the succeeding comparison (the ninth overall) is a mismatch. It then shifts P by one place, finds a mismatch, and repeats this cycle two additional times, until the left end of P is aligned with character 6 of T. At that point it finds eight matches and concludes that P occurs in T starting at position 6. In this example, a total of twenty comparisons are made by the naive algorithm.

A smarter algorithm might realize, after the ninth comparison, that the next three

```
   0        1          0        1          0        1
   1234567890123       1234567890123       1234567890123
T: xabxyabxyabxz    T: xabxyabxyabxz    T: xabxyabxyabxz
P: abxyabxz         P: abxyabxz         P: abxyabxz
   *                   *                   *
    abxyabxz            abxyabxz            abxyabxz
    ^^^^^^^*            ^^^^^^^*            ^^^^^^^*
     abxyabxz               abxyabxz            abxyabxz
     *                      ^^^^^^^^               ^^^^^
      abxyabxz
      *
       abxyabxz
       *
        abxyabxz
        ^^^^^^^^
```

Figure 1.1: The first scenario illustrates pure naive matching, and the next two illustrate smarter shifts. A caret beneath a character indicates a match and a star indicates a mismatch made by the algorithm.

comparisons of the naive algorithm will be mismatches. This smarter algorithm skips over the next three shift/compares, immediately moving the left end of P to align with position 6 of T, thus saving three comparisons. How can a smarter algorithm do this? After the ninth comparison, the algorithm knows that the first seven characters of P match characters 2 through 8 of T. If it also knows that the first character of P (namely a) does not occur again in P until position 5 of P, it has enough information to conclude that character a does not occur again in T until position 6 of T. Hence it has enough information to conclude that there can be no matches between P and T until the left end of P is aligned with position 6 of T. Reasoning of this sort is the key to shifting by more than one character. In addition to shifting by larger amounts, we will see that certain aligned characters do not need to be compared.

An even smarter algorithm knows the next occurrence in P of the first three characters of P (namely abx) begin at position 5. Then since the first seven characters of P were found to match characters 2 through 8 of T, this smarter algorithm has enough information to conclude that when the left end of P is aligned with position 6 of T, the next three comparisons must be matches. This smarter algorithm avoids making those three comparisons. Instead, after the left end of P is moved to align with position 6 of T, the algorithm compares character 4 of P against character 9 of T. This smarter algorithm therefore saves a total of six comparisons over the naive algorithm.

The above example illustrates the kinds of ideas that allow some comparisons to be skipped, although it should still be unclear how an algorithm can efficiently implement these ideas. Efficient implementations have been devised for a number of algorithms such as the Knuth-Morris-Pratt algorithm, a real-time extension of it, the Boyer–Moore algorithm, and the Apostolico–Giancarlo version of it. All of these algorithms have been implemented to run in linear time ($O(n + m)$ time). The details will be discussed in the next two chapters.

4.4. Karp–Rabin fingerprint methods for exact match

The *Shift-And* method assumes that we can efficiently shift a vector of bits, and the generalized *Shift-And* method assumes that we can efficiently increment an integer by one. If we treat a (row) bit vector as an integer number then a left shift by one bit results in the doubling of the number (assuming no bits fall off the left end). So it is not much of an extension to assume, in addition to being able to increment an integer, that we can also efficiently multiply an integer by two. With that added primitive operation we can turn the exact match problem (again without mismatches) into an arithmetic problem. The first result will be a simple linear-time method that has a very small probability of making an error. That method will then be transformed into one that never makes an error, but whose running time is only expected to be linear. We will explain these results using a binary string P and a binary text T. That is, the alphabet is first assumed to be just $\{0, 1\}$. The extension to larger alphabets is immediate and will be left to the reader.

4.4.1. Arithmetic replaces comparisons

Definition For a text string T, let T_r^n denote the n-length substring of T starting at character r. Usually, n is known by context, and T_r^n will be replaced by T_r.

Definition For the binary pattern P, let

$$H(P) = \sum_{i=1}^{i=n} 2^{n-i} P(i).$$

Similarly, let

$$H(T_r) = \sum_{i=1}^{i=n} 2^{n-i} T(r+i-1).$$

That is, consider P to be an n-bit binary number. Similarly, consider T_r^n to be an n-bit binary number. For example, if $P = 0101$ then $n = 4$ and $H(P) = 2^3 \times 0 + 2^2 \times 1 + 2^1 \times 0 + 2^0 \times 1 = 5$; if $T = 101101010$, $n = 4$, and $r = 2$, then $H(T_r) = 6$.

Clearly, if there is an occurrence of P starting at position r of T then $H(P) = H(T_r)$. However, the converse is also true, so

Theorem 4.4.1. *There is an occurrence of P starting at position r of T if and only if $H(P) = H(T_r)$.*

The proof, which we leave to the reader, is an immediate consequence of the fact that every integer can be written in a unique way as the sum of positive powers of two.

Theorem 4.4.1 converts the exact match problem into a numerical problem, comparing the two numbers $H(P)$ and $H(T_r)$ rather than directly comparing characters. But unless the pattern is fairly small, the computation of $H(P)$ and $H(T_r)$ will not be efficient.[2] The problem is that the required powers of two used in the definition of $H(P)$ and $H(T_r)$ grow large too rapidly. (From the standpoint of complexity theory, the use of such large numbers violates the unit-time *random access machine* (*RAM*) model. In that model, the largest allowed numbers must be represented in $O[\log(n+m)]$ bits, but the number 2^n requires n bits. Thus the required numbers are exponentially too large.) Even worse, when the alphabet is not binary but say has t characters, then numbers as large as t^n are needed.

In 1987 R. Karp and M. Rabin [266] published a method (devised almost ten years earlier), called the *randomized fingerprint* method, that preserves the spirit of the above numerical approach, but that is extremely efficient as well, using numbers that satisfy the *RAM* model. It is a *randomized* method where the *only if* part of Theorem 4.4.1 continues to hold, but the *if* part does not. Instead, the *if* part will hold *with high probability*. This is explained in detail in the next section.

4.4.2. Fingerprints of *P* and *T*

The general idea is that, instead of working with numbers as large as $H(P)$ and $H(T_r)$, we will work with those numbers *reduced modulo* a relatively small integer p. The arithmetic will then be done on numbers requiring only a small number of bits, and so will be efficient. But the really attractive feature of this method is a proof that the probability of error can be made small if p is chosen randomly in a certain range. The following definitions and lemmas make this precise.

Definition For a positive integer p, $H_p(P)$ is defined as $H(P)$ mod p. That is $H_p(P)$ is the remainder of $H(P)$ after division by p. Similarly, $H_p(T_r)$ is defined as $H(T_r)$ mod p. The numbers $H_p(P)$ and $H_p(T_r)$ are called *fingerprints* of P and T_r.

Already, the utility of using fingerprints should be apparent. By reducing $H(P)$ and $H(T_r)$ modulo a number p, every fingerprint remains in the range 0 to $p-1$, so the size of a fingerprint does not violate the *RAM* model. But if $H(P)$ and $H(T_r)$ must be computed before they can be reduced modulo p, then we have the same problem of intermediate numbers that are too large. Fortunately, modular arithmetic allows one to reduce at any time (i.e., one can never reduce too much), so that the following generalization of Horner's rule holds:

Lemma 4.4.1. $H_p(P) = \{[\ldots(\{[P(1)\times 2 \bmod p + P(2)]\times 2 \bmod p + P(3)\}\times 2 \bmod p + P(4))\ldots] \bmod p + P(n)\} \bmod p$, *and no number ever exceeds* $2p$ *during the computation of* $H_p(P)$.

[2] One can more efficiently compute $H(T_{r+1})$ from $H(T_r)$ than by following the definition directly (and we will need that later on), but the time to do the updates is not the issue here.

For example, if $P = 101111$ and $p = 7$, then $H(P) = 47$ and $H_p(P) = 47 \bmod 7 = 5$. Moreover, this can be computed as follows:

$$\begin{aligned}
&1 \times 2 \bmod 7 + 0 = 2\\
&2 \times 2 \bmod 7 + 1 = 5\\
&5 \times 2 \bmod 7 + 1 = 4\\
&4 \times 2 \bmod 7 + 1 = 2\\
&2 \times 2 \bmod 7 + 1 = 5\\
&5 \bmod 7 = 5.
\end{aligned}$$

The point of Horner's rule is not only that the number of multiplications and additions required is linear, but that the intermediate numbers are always kept small.

Intermediate numbers are also kept small when computing $H_p(T_r)$ for any r, since that computation can be organized the way that $H_p(P)$ was. However, even greater efficiency is possible: For $r > 1$, $H_p(T_r)$ can be computed from $H_p(T_{r-1})$ with only a small *constant* number of operations. Since

$$H_p(T_r) = H(T_r) \bmod p$$

and

$$H(T_r) = 2 \times H(T_{r-1}) - 2^n T(r-1) + T(r+n-1),$$

it follows that

$$H_p(T_r) = [(2 \times H(T_{r-1}) \bmod p) - (2^n \bmod p) \times T(r-1) + T(r+n-1)] \bmod p.$$

Further,

$$2^n \bmod p = 2 \times (2^{n-1} \bmod p) \bmod p.$$

Therefore, each successive power of two taken mod p and each successive value $H_p(T_r)$ can be computed in constant time.

Prime moduli limit false matches

Clearly, if P occurs in T starting at position r then $H_p(P) = H_p(T_r)$, but now the converse does not hold for every p. That is, we cannot necessarily conclude that P occurs in T starting at r just because $H_p(P) = H_p(T_r)$.

Definition If $H_p(P) = H_p(T_r)$ but P does not occur in T starting at position r, then we say there is a *false match* between P and T at position r. If there is *some* position r such that there is a false match between P and T at r, then we say there is a false match between P and T.

The goal will be to choose a modulus p small enough that the arithmetic is kept efficient, yet large enough that the probability of a false match between P and T is kept small. The key comes from choosing p to be a *prime* number in the proper range and exploiting properties of prime numbers. We will state the needed properties of prime numbers without proof.

Definition For a positive integer u, $\pi(u)$ is the *number* of primes that are less than or equal to u.

The following theorem is a variant of the famous *prime number theorem*.

Theorem 4.4.2. $\frac{u}{\ln(u)} \le \pi(u) \le 1.26\frac{u}{\ln(u)}$, *where* $\ln(u)$ *is the base e logarithm of* u [383].

Lemma 4.4.2. *If $u \geq 29$, then the product of all the primes that are less than or equal to u is greater than 2^u* [383].

For example, for $u = 29$ the prime numbers less than or equal to 29 are 2, 5, 7, 11, 13, 17, 19, 23, and 29. Their product is 2,156,564,410 whereas 2^{29} is 536,870,912.

Corollary 4.4.1. If $u \geq 29$ and x is any number less than or equal to 2^u, then x has fewer than $\pi(u)$ (distinct) prime divisors.

PROOF Suppose x does have $k > \pi(u)$ distinct prime divisors $q_1, q_2, \ldots, q_k$. Then $2^u \geq x \geq q_1 q_2 \ldots q_k$ (the first inequality is from the statement of the corollary, and the second from the fact that some primes in the factorization of x may be repeated). But $q_1 q_2 \ldots q_k$ is at least as large as the product of the smallest k primes, which is greater than the product of the first $\pi(u)$ primes (by assumption that $k > \pi(u)$). However, the product of the primes less than or equal to u is greater than 2^u (by Lemma 4.4.2). So the assumption that $k > \pi(u)$ leads to the contradiction that $2^u > 2^u$, and the lemma is proved. □

The central theorem

Now we are ready for the central theorem of the Karp–Rabin approach.

Theorem 4.4.3. *Let P and T be any strings such that $nm \geq 29$, where n and m are the lengths of P and T, respectively. Let I be any positive integer. If p is a randomly chosen prime number less than or equal to I, then the probability of a false match between P and T is less than or equal to $\frac{\pi(nm)}{\pi(I)}$.*

PROOF Let R be the set of positions in T where P does *not* begin. That is, $s \in R$ if and only if P does not occur in T beginning at s. For each $s \in R$, $H(P) \neq H(T_s)$. Now consider the product $\Pi_{s \in R}(|H(P) - H(T_s)|)$. That product must be at most 2^{nm} since for any s, $H(P) - H(T_s) \leq 2^n$ (recall that we have assumed a binary alphabet). Applying Corollary 4.4.1, $\Pi_{s \in R}(|H(P) - H(T_s)|)$ has at most $\pi(nm)$ distinct prime divisors.

Now suppose a false match between P and T occurs at some position r of T. That means that $H(P) \bmod p = H(T_r) \bmod p$ and that p evenly divides $H(P) - H(T_r)$. Trivially then, p evenly divides $\Pi_{s \in R}(|H(P) - H(T_s)|)$, and so p is one of the prime divisors of that product. If p allows a false match to occur between P and T, then p must be one of a set of at most $\pi(nm)$ numbers. But p was chosen randomly from a set of $\pi(I)$ numbers, so the probability that p is a prime that allows a false match between P and T is at most $\frac{\pi(nm)}{\pi(I)}$. □

Notice that Theorem 4.4.3 holds for any choice of pattern P and text T such that $nm \geq 29$. The probability in the theorem is not taken over choices of P and T but rather over choices of prime p. Thus, this theorem does not make any (questionable) assumptions about P or T being random or generated by a Markov process, etc. It works for any P and T! Moreover, the theorem doesn't just bound the probability that a false match occurs at a fixed position r, it bounds the probability that there is even a single such position r in T. It is also notable that the analysis in the proof of the theorem feels "weak". That is, it only develops a very weak property of a prime p that allows a false match, namely being one of at most $\pi(nm)$ numbers that divide $\Pi_{s \in R}(|H(P) - H(T_s)|)$. This suggests that the true probability of a false match occurring between P and T is much less than the bound established in the theorem.

Theorem 4.4.3 leads to the following random fingerprint algorithm for finding all occurrences of P in T.

Random fingerprint algorithm

1. Choose a positive integer I (to be discussed in more detail below).
2. Randomly pick a prime number less than or equal to I, and compute $H_p(P)$. (Efficient randomized algorithms exist for finding random primes [331].)
3. For each position r in T, compute $H_p(T_r)$ and test to see if it equals $H_p(P)$. If the numbers are equal, then either declare a probable match or check explicitly that P occurs in T starting at that position r.

Given the fact that each $H_p(T_r)$ can be computed in constant time from $H_p(T_{r-1})$, the fingerprint algorithm runs in $O(m)$ time, excluding any time used to explicitly check a declared match. It may, however, be reasonable not to bother explicitly checking declared matches, depending on the probability of an error. We will return to the issue of checking later. For now, to fully analyze the probability of error, we have to answer the question of what I should be.

How to choose I

The utility of the fingerprint method depends on finding a good value for I. As I increases, the probability of a false match between P and T decreases, but the allowed size of p increases, increasing the effort needed to compute $H_p(P)$ and $H_p(T_r)$. Is there a good balance? There are several good ways to choose I depending on n and m. One choice is to take $I = nm^2$. With that choice the largest number used in the algorithm requires at most $4(\log n + \log m)$ bits, satisfying the *RAM* model requirement that the numbers be kept small as a function of the size of the input. But, what of the probability of a false match?

Corollary 4.4.2. When $I = nm^2$, the probability of a false match is at most $\frac{2.53}{m}$.

PROOF By Theorem 4.4.3 and the prime number theorem (Theorem 4.4.2), the probability of a false match is bounded by

$$\frac{\pi(nm)}{\pi(nm^2)} \leq 1.26 \frac{nm}{nm^2} \frac{\ln(nm^2)}{\ln(nm)} = 1.26 \frac{1}{m} \left[\frac{\ln(n) + 2\ln(m)}{\ln(n) + \ln(m)}\right] \leq \frac{2.53}{m}. \qquad \square$$

A small example from [266] illustrates this bound. Take $n = 250$, $m = 4000$, and hence $I = 4 \times 10^9 < 2^{32}$. Then the probability of a false match is at most $\frac{2.53}{4000} < 10^{-3}$. Thus, with just a 32-bit fingerprint, for any P and T the probability that even a single one of the algorithm's declarations is wrong is bounded by 0.001.

Alternately, if $I = n^2 m$ then the probability of a false match is $O(1/n)$, and since it takes $O(n)$ time to determine whether a match is false or real, the *expected* verification time would be constant. The result would be an $O(m)$ *expected* time method that never has a false match.