4

Seminumerical String Matching

4.1. Arithmetic versus comparison-based methods

All of the exact matching methods in the first three chapters, as well as most of the methods that have yet to be discussed in this book, are examples of *comparison-based* methods. The main primitive operation in each of those methods is the comparison of two characters. There are, however, string matching methods based on *bit operations* or on *arithmetic*, rather than character comparisons. These methods therefore have a very different flavor than the comparison-based approaches, even though one can sometimes see character comparisons hidden at the inner level of these "seminumerical" methods. We will discuss three examples of this approach: the *Shift-And* method and its extension to a program called *agrep* to handle inexact matching; the use of the Fast Fourier Transform in string matching; and the random fingerprint method of Karp and Rabin.

4.2. The *Shift-And* method

R. Baeza-Yates and G. Gonnet [35] devised a simple, bit-oriented method that solves the exact matching problem very efficiently for relatively small patterns (the length of a typical English word for example). They call this method the *Shift-Or* method, but it seems more natural to call it *Shift-And*. Recall that pattern P is of size n and the text T is of size m.

Definition Let M be an n by $m+1$ *binary* valued array, with index i running from 1 to n and index j running from 1 to m. Entry $M(i, j)$ is 1 if and only if the first i characters of P exactly match the i characters of T ending at character j. Otherwise the entry is zero.

In other words, $M(i, j)$ is 1 if and only if $P[1..i]$ exactly matches $T[j-i+1..j]$. For example, if $T = californian$ and $P = for$, then $M(1, 5) = M(2, 6) = M(3, 7) = 1$, whereas $M(i, j) = 0$ for all other combinations of i, j. Essentially, the entries with value 1 in row i of M show all the places in T where a copy of $P[1..i]$ ends, and column j of M shows all the prefixes of P that end at position j of T.

Clearly, $M(n, j) = 1$ if and only if an occurrence of P ends at position j of T; hence computing the last row of M solves the exact matching problem. For the algorithm to compute M it first constructs an n-length binary vector $U(x)$ for each character x of the alphabet. $U(x)$ is set to 1 for the positions in P where character x appears. For example, if $P = abacdeab$ then $U(a) = 10100010$.

Definition Define *Bit-Shift*$(j-1)$ as the vector derived by shifting the vector for *column* $j-1$ down by one position and setting that first to 1. The previous bit in position n disappears. In other words, *Bit-Shift*$(j-1)$ consists of 1 followed by the first $n-1$ bits of column $j-1$.

For example, Figure 4.1 shows a column $j-1$ before and after the bit-shift.

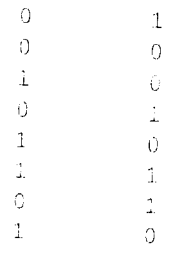

Figure 4.1: Column $j-1$ before and after operation *Bit-Shift*$(j-1)$.

4.2.1. How to construct array M

Array M is constructed column by column as follows: Column one of M is initialized to all zero entries if $T(1) \neq P(1)$. Otherwise, when $T(1) = P(1)$ its first entry is 1 and the remaining entries are 0. After that, the entries for column $j > 1$ are obtained from column $j-1$ and the U vector for character $T(j)$. In particular, the vector for column j is obtained by the bitwise **AND** of vector *Bit-Shift*$(j-1)$ with the U vector for character $T(j)$. More formally, if we let $M(j)$ denote the jth column of M, then $M(j) =$ *Bit-Shift*$(j-1)$ **AND** $U(T(j))$. For example, if $P = abaac$ and $T = xabxabaaxa$ then the eighth column of M is

$$\begin{matrix}1\\0\\1\\0\\0\end{matrix}$$

because prefixes of P of lengths one and three end at position seven of T. The eighth character of T is character a, which has a U vector of

$$\begin{matrix}1\\0\\1\\1\\0\end{matrix}$$

When the eighth column of M is shifted down and an **AND** is performed with $U(a)$, the result is

$$\begin{matrix}1\\0\\0\\1\\0\end{matrix}$$

which is the correct ninth column of M.

To see in general why the *Shift-And* method produces the correct array entries, observe that for any $i > 1$ the array entry for cell (i, j) should be 1 if and only if the first $i-1$ characters of P match the $i-1$ characters of T ending at character $j-1$ and character $P(i)$ matches character $T(j)$. The first condition is true when the array entry for cell $(i-1, j-1)$ is 1, and the second condition is true when the ith bit of the U vector for character $T(j)$ is 1. By first shifting column $j-1$, the algorithm **AND**s together entry $(i-1, j-1)$ of column $j-1$ with entry i of the vector $U(T(j))$. Hence the algorithm computes the correct entries for array M.

4.2.2. *Shift-And* is effective for small patterns

Although the *Shift-And* method is very simple, and in worst case the number of bit operations is clearly $\Theta(mn)$, the method is very efficient if n is less than the size of a single computer word. In that case, every column of M and every U vector can be encoded into a single computer word, and both the *Bit-Shift* and the **AND** operations can be done as single-word operations. These are very fast operations in most computers and can be specified in languages such as C. Even if n is several times the size of a single computer word, only a few word operations are needed. Furthermore, only two columns of M are needed at any given time. Column j only depends on column $j-1$, so all previous columns can be forgotten. Hence, for reasonable sized patterns, such as single English words, the *Shift-And* method is very efficient in both time and space regardless of the size of the text. From a purely theoretical standpoint it is not a linear time method, but it certainly is practical and would be the method of choice in many circumstances.

4.2.3. *agrep*: The *Shift-And* method with errors

S. Wu and U. Manber [482] devised a method, packaged into a program called *agrep*, that amplifies the *Shift-And* method by finding *inexact* occurrences of a pattern in a text. By inexact we mean that the pattern either occurs exactly in the text or occurs with a "small" number of *mismatches* or *inserted* or *deleted* characters. For example, the pattern *atcgaa* occurs in the text *aatatccacaa* with two mismatches starting at position four; it also occurs with four mismatches starting at position two. In this section we will explain *agrep* and how it handles mismatches. The case of permitted insertions and deletions will be left as an exercise. For a small number of errors and for small patterns, *agrep* is very efficient and can be used in the core of more elaborate text searching methods. Inexact matching is the focus of Part III, but the ideas behind *agrep* are so closely related to the *Shift-And* method that it is appropriate to examine *agrep* at this point.

Definition For two strings P and T of lengths n and m, let M^k be a binary-valued array, where $M^k(i, j)$ is 1 if and only if at least $i-k$ of the first i characters of P match the i characters up through character j of T.

That is, $M^k(i, j)$ is the natural extension of the definition of $M(i, j)$ to allow up to k mismatches. Therefore, M^0 is the array M used in the *Shift-And* method. If $M^k(n, j) = 1$ then there is an occurrence of P in T ending at position j that contains at most k mismatches. We let $M^k(j)$ denote the jth column of M^k.

In *agrep*, the user chooses a value of k and then the arrays $M, M^1, M^2, \ldots, M^k$ are computed. The efficiency of the method depends on the size of k – the larger k is, the slower the method. For many applications, a value of k as small as 3 or 4 is sufficient, and the method is extremely fast.

4.2.4. How to compute M^k

Let k be the fixed maximum permitted number of mismatches specified by the user. The method will compute M^l for all values of l between 0 and k. There are several ways to organize the computation and its description, but for simplicity we will compute column j of each array M^l before any columns past j will be computed in any array. Further, for every j we will compute column j in arrays M^l in increasing order of l. In particular, the

zero column of each array is again initialized to all zeros. Then the jth column of M^l is computed by:

$$M^l(j) = M^{l-1}(j) \textbf{ OR } [\textit{Bit-Shift}(M^l(j-1)) \textbf{ AND } U(T(j))] \textbf{ OR } M^{l-1}(j-1).$$

Intuitively, this just says that the first i characters of P will match a substring of T ending at position j, with at most l mismatches, if and only if one of the following three conditions hold:

- The first i characters of P match a substring of T ending at j, with at most $l-1$ mismatches.
- The first $i-1$ characters of P match a substring of T ending at $j-1$, with at most l mismatches, and the next pair of characters in P and T are equal.
- The first $i-1$ characters of P match a substring of T ending at $j-1$, with at most $l-1$ mismatches.

It is simple to establish that these recurrences are correct, and over the entire algorithm the number of bit operations is $O(knm)$. As in the *Shift-And* method, the practical efficiency comes from the fact that the vectors are bit vectors (again of length n) and the operations are very simple – shifting by one position and **AND**ing bit vectors. Thus when the pattern is relatively small, so that a column of any M^l fits into a few words, and k is also small, *agrep* is extremely fast.

4.3. The match-count problem and Fast Fourier Transform

If we relax the requirement that only bit operations are permitted and allow each entry of array M to hold an integer between 0 and n, then we can easily adapt the *Shift-And* method to compute for each pair i, j the number of characters of $P[1..i]$ that match $T[j-i+1..j]$. This computation is again a form of inexact matching, which is the focus of Part III. However, as was true of *agrep*, the solution is so connected to the *Shift-And* method that we consider it here. In addition, it is a natural introduction to the next topic, match-counts. For clarity, let us define a new matrix MC.

Definition The matrix MC is an n by $m+1$ integer-valued matrix, where entry $MC(i, j)$ is the number of characters of $P[1..i]$ that match $T[j-i+1..j]$.

A simple algorithm to compute matrix MC generalizes the *Shift-And* method, replacing the **AND** operation with the *increment by one* operation. The zero column of MC starts with all zeros, but each $MC(i, j)$ entry now is set to $MC(i-1, j-1)$ if $P(i) \neq T(j)$, and otherwise it is set to $MC(i-1, j-1)+1$. Any entry with value n in the last row again indicates an occurrence of P in T, but values less than n count the *exact number* of characters that match for each of different alignments of P with T. This extension uses $\Theta(nm)$ additions and comparisons, although each addition operation is particularly simple, just incrementing by one.

If we want to compute the entire MC array then $\Theta(nm)$ time is necessary, but the most important information is contained in the last row of MC. For each position $j \geq n$ in T, the last row indicates the number of characters that match when the right end of P is aligned with character j of T. The problem of finding the last row of MC is called the *match-count* problem. Match-counts are useful in several problems to be discussed later.