Chapter 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, 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 $O(n \times m)$ 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.

1.2 The preprocessing approach

Many string matching and analysis algorithms are able to efficiently skip comparisons by first spending "modest" time learning about the internal structure of either the pattern P or the text T. During that time, the other string may not even be known to the algorithm. This part of the overall algorithm is called the *preprocessing* stage. Preprocessing is followed by a *search* stage, where the information found during the preprocessing stage is used to reduce the work done while searching for occurrences of P in T. In the above example, the smarter method was assumed to know that character a did not occur again until position 5, and the even smarter method was assumed to know that the pattern abx was repeated again starting at position 5. This assumed knowledge is obtained in the preprocessing stage.

For the exact matching problem, all of the algorithms mentioned in the previous section preprocess pattern P. (The opposite approach of preprocessing text T is used in other algorithms, such as those based on suffix trees. Those methods will be explained later in the book.) These preprocessing methods, as originally developed, are "similar in spirit" but often quite different in detail and conceptual difficulty. In this book we take a different approach and do not initially explain the originally developed preprocessing methods. Rather, we highlight the similarity of the preprocessing *tasks* needed for several different matching algorithms, by first defining a *fundamental preprocessing* of P that is independent of any particular matching algorithm. Then we show how each specific matching algorithm uses the information computed by the fundamental preprocessing of P. The result is a simpler more uniform exposition of the preprocessing needed by several classical matching methods, and a simple linear time algorithm for exact matching based only on this preprocessing (discussed in Section 1.5). This approach to linear-time pattern matching was developed in [**?**].

1.3 Fundamental preprocessing of the pattern

Fundamental preprocessing will be described for a general string denoted S. In specific applications of fundamental preprocessing, S will often be the pattern P, but here we use S instead of P because fundamental preprocessing will also be applied to strings other than P.

The following definition gives the key values computed during the fundamental preprocessing of a string.

Definition Given a string S and a position $i > 1$, let $Z_i(S)$ be the *length* of the longest substring of S that *starts* at i and matches a prefix of S.

In other words, $Z_i(S)$ is the length of the longest *prefix* of $S[i..s]$ which

matches a prefix of S. For example, when $S = aabcaabxaaz$ then

$$Z_5(S) = 3 \quad (aabc...aabx...),$$
$$Z_6(S) = 1 \quad (aa...ab...),$$
$$Z_7(S) = Z_8(S) = 0,$$
$$Z_9(S) = 2 \quad (aab...aaz).$$

When S is clear by context, we will use Z_i in place of $Z_i(S)$.

To introduce the next concept, consider the boxes drawn in Figure 1.1. Each box starts at some position $j > 1$ such that Z_j is greater than zero. The length of the box starting at j is meant to represent Z_j. Therefore, each box in the figure represents a maximal-length substring of S that matches a prefix of S, and that doesn't start at position one. Each such box is called a *Z-box*. More formally,

Definition For any position $i > 1$ where Z_i is greater than zero, the *Z-box* at i is defined as the interval starting at i and ending at position $i + Z_i - 1$.

Definition For every $i > 1$, r_i is the rightmost endpoint of the Z-boxes that begin at or before position i. Another way to state this is: r_i is the largest value of $j + Z_j - 1$ over all $1 < j \leq i$ such that $Z_j > 0$. (See Figure 1.1.

We use the term l_i for the value of j specified in the above definition. That is, l_i is the position of *left end* of Z-box that ends at r_i. In case there is more than one Z-box ending at r_i, then l_i can be chosen to be the left end of any of those Z-boxes. As an example, suppose $S = aabaabcaxaabaabcy$. Then $Z_{10} = 7$, $r_{15} = 16$ and $l_{15} = 10$.

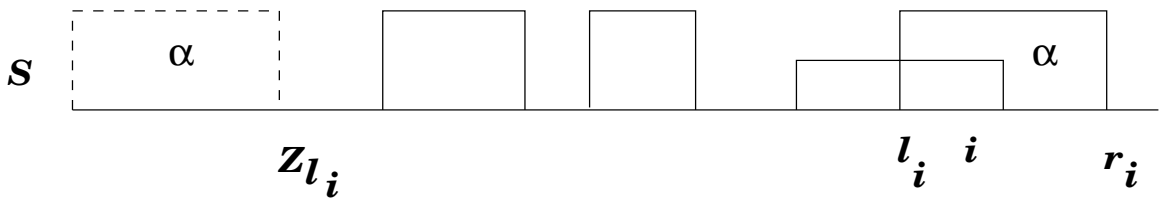

Figure 1.1: Each solid box represents a substring of S that matches a prefix of S and that starts between positions 2 and i. Each box is called a *Z-box*. We use r_i to denote the *rightmost* end of any Z-box that begins at or to the left of position i, and α to denote the substring in the Z-box ending at r_i. Then l_i denotes the left end of α. The copy of α that occurs as a prefix of S is also shown in the figure.

The linear time computation of Z values from S is the *fundamental* preprocessing task that we will use in all the classical linear-time matching

algorithms that preprocess P. But before detailing those uses, we show how to do the fundamental preprocessing in linear time.

1.4 Fundamental preprocessing in linear time

The task of this section is to show how to compute all the Z_i values for S in linear time, i.e., in $O(|S|)$ time. A direct approach based on the definition would take $\Theta(|S|^2)$ time. The method we will present was developed in [**?**] for a different purpose.

The preprocessing algorithm computes Z_i, r_i and l_i for each successive position i, starting from $i = 2$. All the Z values computed will be kept by the algorithm, but in any iteration i, the algorithm only needs the r_j and l_j values for $j = i - 1$. No earlier r or l values are needed. Hence the algorithm only uses a single variable, r, to refer to the most recently computed r_j value; similarly it only uses a single variable l. Therefore, in each iteration i, if the algorithm discovers a new Z-box (starting at i), variable r will be incremented to end of that Z-box, which is the rightmost position of any Z-box discovered so far.

To begin, the algorithm finds Z_2 by explicitly comparing, left to right, the characters of $S[2..|S|]$ and $S[1..|S|]$ until a mismatch is found. Z_2 is the length of the matching string. If $Z_2 > 0$, then $r = r_2$ is set to $Z_2 + 1$ and $l = l_2$ is set to 2. Otherwise r and l are set to zero. Now assume inductively that the algorithm has correctly computed Z_i for i up to $k - 1 > 1$, and assume that the algorithm knows the current $r = r_{k-1}$ and $l = l_{k-1}$. The algorithm next computes Z_k, $r = r_k$, and $l = l_k$.

The main idea is to use the already computed Z values to accelerate the computation of Z_k. In fact, in some cases, Z_k can be deduced from the previous Z values without doing any additional *character* comparisons. As a concrete example, suppose $k = 121$, all the values Z_2 through Z_{120} have already been computed, and $r_{120} = 130$ and $l_{120} = 100$. That means that there is a substring of length 31 that starts at position 100 and that matches a prefix of S (of length 31). It follows that the substring of length 10 starting at position 121 must match the substring of length 10 starting at position 22 of S, and so Z_{22} may be very helpful in computing Z_{121}. As one case, if Z_{22} is three, say, then a little reasoning shows that Z_{121} *must* also be three. So in this illustration, Z_{121} can be deduced without any additional character comparisons. This case, along with the others, will be formalized and proven correct below.

The Z Algorithm

Given Z_i for all $1 < i \leq k-1$ and the current values of r and l, Z_k and the updated r and l are computed as follows:

Begin

1. If $k > r$, then find Z_k by explicitly comparing the characters starting at position k to the characters starting at position 1 of S, until a mismatch is found. The length of the match is Z_k. If $Z_k > 0$, then set r to $k + Z_k - 1$ and set l to k.

2. If $k \leq r$, then position k is contained in a Z-box, hence $S(k)$ is contained substring $S[l..r]$ (call it α) such that $l > 1$ and α matches a prefix of S. Therefore character $S(k)$ also appears in position $k' = k - l + 1$ of S. By the same reasoning, substring $S[k..r]$ (call it β) must match substring $S[k'..Z_l]$. It follows that the substring beginning at position k must match a prefix of S of length at least the *minimum* of $Z_{k'}$ and $|\beta|$ (which is $r-k+1$). See Figure 1.2.

We consider two subcases based on what that minimum is.

2a. If $Z_{k'} < |\beta|$ then $Z_k = Z_{k'}$ and r, l remain unchanged (see Figure 1.3).

2b. If $Z_{k'} \geq |\beta|$ then the entire substring $S[k..r]$ must be a prefix of S and $Z_k \geq |\beta| = r - k + 1$. However, Z_k might be strictly larger than $|\beta|$, so compare the characters starting at position $r+1$ of S to the characters starting a position $|\beta| + 1$ of S until a mismatch occurs. Say the mismatch occurs at character $q \geq r + 1$. Then Z_k is set to $q - k$, r is set to $q - 1$ and l is set to k (see Figure 1.4).

End

Figure 1.2: String $S[k..r]$ is labeled β and also occurs starting at position k' of S.

Theorem 1.4.1 *Using Algorithm Z, value Z_k is correctly computed and variables r and l are correctly updated.*

Proof In case 1, Z_k is set correctly since it is computed by explicit comparisons. Also (since $k > r$ in case 1), before Z_k is computed, no Z- box has been found that starts between positions 2 and $k-1$ and that ends at or after position k. Therefore when $Z_k > 0$ in case 1, the algorithm does find a

Figure 1.3: Case 2a. The longest string starting at k' that matches a prefix of S is shorter than $|\beta|$. In this case, $Z_k = Z_{k'}$.

Figure 1.4: Case 2b. The longest string starting at k' that matches a prefix of S is at least $|\beta|$.

new Z-box ending at or after k, and it is correct to change r to $k + Z_k - 1$. Hence the algorithm works correctly in case 1.

In case 2a, the substring beginning at position k can match a prefix of S only for length $Z_{k'} < |\beta|$. If not, then the next character to the right, character $k + Z_{k'}$, must match character $1 + Z_{k'}$. But character $k + Z_{k'}$ matches character $k' + Z_{k'}$ (since $Z_{k'} < |\beta|$) so character $k' + Z_{k'}$ must match character $1 + Z_{k'}$. But that would be a contradiction to the definition of $Z_{k'}$, for it would establish a substring longer than $Z_{k'}$ that starts at k' and matches a prefix of S. Hence $Z_k = Z_{k'}$ in this case. Further, $k + Z_k - 1 < r$, so r and l remain correctly unchanged.

In case 2b, β must be a prefix of S (as argued in the body of the algorithm) and since any extension of this match is explicitly verified by comparing characters beyond r to characters beyond the prefix β, the full extent of the match is correctly computed. Hence Z_k is correctly obtained in this case. Furthermore, since $k + Z_k - 1 \geq r$, the algorithm correctly changes r and l. □

Corollary 1.4.1 *Repeating algorithm Z for each position $i > 2$ correctly yields all the Z_i values.*

Theorem 1.4.2 *All the $Z_i(S)$ values are computed by the algorithm in $O(|S|)$ time.*

Proof The time is proportional to the number of iterations, $|S|$, plus the number of character comparisons. Each comparison results in either a match

or a mismatch, so we next bound the number of matches and mismatches that can occur.

Each iteration that does any character comparisons at all ends the first time it finds a mismatch, hence there are at most $|S|$ mismatches during the entire algorithm. To bound the number of matches, note first that $r_k \geq r_{k-1}$ for every iteration k. Now, let k be an iteration where $q > 0$ matches occur. Then r_k is set to $r_{k-1} + q$ at least. Finally, $r_k \leq |S|$, so the total number of matches that occur during any execution of the algorithm is at most $|S|$. □

1.5 The simplest linear-time exact matching algorithm

Before discussing the more complex (classical) exact matching methods, we show that fundamental preprocessing alone provides a simple linear time exact matching algorithm. This is the simplest linear-time matching algorithm we know of.

Let $S = P\$T$ be the string consisting of P followed by the symbol "\$" followed by T, where "\$" is a character appearing in neither P nor T. Recall that P has length n and T has length m, and $n \leq m$. So, $S = P\$T$ has length $n + m + 1 = O(m)$. Compute $Z_i(S)$ for i from 1 to $n + m + 1$. Since "\$" does not appear in P or T, $Z_i \leq n$ for every i. Any value of $i > n + 1$ such that $Z_i(S) = n$ identifies an occurrence of P in T starting at position $i-(n+1)$ of T. Conversely, if P occurs in T starting at position j of T, then $Z_{(n+1)+j}$ must be equal to n. Since all the $Z_i(S)$ values can be computed in $O(n + m) = O(m)$ time, this approach identifies all the occurrences of P in T in $O(m)$ time.

The method can be implemented to use only $O(n)$ space (in addition to the space needed for pattern and text) independent of the size of the alphabet. Since $Z_i \leq n$ for all i, position k' (determined in step 2) will always fall inside P. Therefore there is no need to record the Z values for characters in T. Instead, we only need to record the Z values for the n characters in P, and also maintain the current l and r. Those values are sufficient to compute (but not store) the Z value of each character in T and hence to identify and output any position i where $Z_i = n$.

There is another characteristic of this method that is worth introducing here. The method is considered an *alphabet independent* linear time method. That is, we never had to assume that the alphabet size was finite, or that we knew the alphabet ahead of time – a character comparison only determines whether the two characters match or mismatch, it needs no further information about the alphabet. We will see that this characteristic is also true of the Knuth-Morris-Pratt and Boyer-Moore algorithms, but not of the Aho-Corasick algorithm or methods based on suffix trees.