EFFICIENT ALGORITHMS FOR MULTIPLE PATTERN MATCHING

by

M. A. Sridhar

Computer Sciences Technical Report #661

August 1986

EFFICIENT ALGORITHMS FOR MULTIPLE PATTERN MATCHING

M. A. Sridhar

Under the supervision of Assistant Professor Samuel W. Bent

We address the problem of finding an occurrence of one of several given patterns in a given text string.

The Aho-Corasick algorithm solves this problem by constructing a modified finite automaton and using this to scan the text string left to right. This yields an algorithm that runs in time linear in the text length. The Boyer-Moore algorithm scans the text right to left, looking for an occurrence of one pattern. It has a sublinear average running time, and can be modified to be linear-time in the worst case.

We extend the Boyer-Moore algorithm to handle multiple patterns. Two new algorithms are developed and analyzed. Both operate by remembering previous matches. Given a text string of length N and patterns with maximum length D, the first algorithm remembers no more than $1 + \log_4 D$ previous matches, and consults $O(N \log D)$ text characters. Algorithms for performing the necessary preprocessing are also developed.

The second algorithm is designed for a different time-space tradeoff. For any given k, it consults $O(kN \log D)$ text characters, and remembers no more than t/k nonperiodic matches at any time, where t is the number of patterns. The dominating feature of a sublinear average-case running time is retained by both algorithms.

Acknowledgements

First of all, I would like to thank my advisor Samuel Bent for the invaluable guidance that he provided me throughout the evolution of this work. I benefited enormously from Sam's remarkable ability to think on his feet. I learned a great deal from him, not only about theoretical computer science, but other things as well. I am deeply indebted to him for his invaluable criticisms and for his excellent proofreading.

I am very grateful to Rachel Manber for all encouragement she has provided me. I thank her and Debby Joseph for reading my thesis and suggesting the changes that brought it into the shape it now is in. I am also indebted to Eric Bach and Peter Spear for serving on my committee. I am very thankful to Peter Spear in particular, for providing me employment when I needed it.

My friends in Madison have provided me with the emotional support and assistance that has seen me through many difficult situations. Hari Madduri was always available to turn to for advice. Matthew was always willing to listen to me and help, and table-tennis with him helped ease the tensions I felt on many occasions. Mohan, Rajeev Arora, and Kishore contributed to an excellent environment. I would like to thank all of them.

Finally, I would like to thank my parents, aunts and uncles for the inspiration they have provided me over the years, without which I most certainly would not have achieved this. Last, but not by any means least, I thank my wife Suneetha for putting up with me and supporting me through some very difficult times.

Contents

1. Introduction

1.1. Problem description

The problem of pattern matching can be formulated in general terms as follows: given a text string, and a regular expression denoting a set of patterns, find an occurrence of one of the patterns in the text.

Instances of this problem, with varying degrees of restriction, arise quite frequently in practice. For example:

(i) Text editors invariably offer some form of searching and substitution commands, which involve matching a given set of patterns against lines of text.

(ii) Bibliographic search programs match keywords against titles of articles to retrieve relevant items from a database.

(iii) Utility programs such as *grep*, *awk* and *sed* of UNIX* provide pattern-matching facilities. Need for such functions also arises in command interpreters (such as the UNIX shell), e.g. for filename expansion.

(iv) Researchers in genetics use pattern-matching to detect occurrences of certain patterns in DNA chains in attempts to decode the "software" of the human body.

The problem of pattern matching has received considerable attention in the literature, most notably by Aho [1980] and by Aho, Hopcroft, and Ullman [1974]. The special case where the pattern is restricted to a single string (i.e., regular expression with concatenations only) was studied by Knuth, Morris and Pratt [1977] and by Boyer and Moore [1978].

In this thesis, we concern ourselves with the following restricted version of the pattern matching problem: given a text string, and a regular expression which is a union of concatenations (or equivalently, a set of pattern strings), determine whether any of the patterns occurs in the text. The problem as stated merely requires determination of the existence of an occurrence, but the algorithms obtained can be modified to determine all the positions of occurrence of all the patterns in the text.

* Trademark of AT&T Bell Laboratories

1.2. Review of related work

The obvious naive algorithm for detecting an occurrence of a pattern in a text is to start matching the pattern against the text from the left towards the right, and shifting the pattern right by one position when a mismatch is detected. This algorithm behaves badly in the worst case: it could take as much as $O(MN)$ time, where M is the length of the pattern and N that of the text. (For example, consider matching the pattern $a^{m-1}b$ against the text string a^n, with $n > m$.)

Numerous refinements of this approach have appeared in the literature, and we will discuss them briefly. We will always use M to denote the pattern length (or the sum of the pattern lengths in the case of multiple patterns), and N to denote the length of the text.

1.2.1. Thompson's algorithm

One of the earliest works on pattern matching was by Thompson [1968]. The essential idea of this paper was to construct a finite state machine corresponding to the given regular expression, and then run the machine on the text, declaring a match whenever the machine entered an accepting state. This approach could yield a machine with as many as 2^m states, where m is the length of the expression. This is true even when the regular expression denotes just a single pattern (see Reeker [1971]). Consequently, the storage requirement for this algorithm is significant.

1.2.2. Harrison's algorithm

Harrison [1971] suggested another approach to the problem when only one pattern is specified. In his approach, a hash function is computed on the pattern, and compared with the hash values obtained on segments of the text. Whenever the two values match, the text segment and the pattern are compared, character by character, to confirm the match. This technique is reasonable on the average, but performs badly in the worst case.

1.2.3. The Karp-Rabin algorithm

A class of randomized algorithms for the pattern-matching problem was invented by Karp and Rabin [1981]. Their algorithms represent long strings by much shorter ***fingerprints,*** and manipulate fingerprints instead of the much longer strings to achieve efficiency. The idea is to examine, at each step, a segment of text of length M, compute its fingerprint, and compare the result with the fingerprint of the pattern. If the two fingerprints match, the algorithm confirms the match by comparing the pattern character by character against the text. If the two fingerprints do not match, the pattern is "shifted right" one character on the text. The fingerprint functions are designed such that the updating required after a right shift can be done very efficiently.

1.2.3. Divide and conquer

Karp, Miller and Rosenberg [1972] address the problem of matching more general structures such as multi-dimensional arrays and trees. They formulate and solve the problem of finding repeated substructures using a technique that can be thought of as an application of the divide-and-conquer strategy. Their idea, as applied to a string, is to define an equivalence relation E_k on positions of the string, such that $i \ E_k \ j$ for positions i and j whenever the k-symbol substrings beginning at positions i and j are the same, and to observe that, for $i \leq j$, we will have $i \ E_{2k} \ j$ if and only if $i \ E_k \ j$ and $i + k \ E_k \ j + k$. This gives a way of computing E_{2k} from E_k .

This idea can be applied to our problem as follows. Let a single pattern x, of length M, and a text string y of length N over an alphabet Σ be given, and assume (to begin with) that M is a power of 2. We start by constructing the string $x\#y$, where $\#$ is a new symbol not in Σ. We then set up E_0 to be the relation containing pairs (i,i), for $1 \leq i \leq m+n$, and proceed to compute iteratively the relations E_{2^j} , for $1 \leq j \leq \log(M+N)$, using the idea described above. It is easy to see that x occurs in y at position i whenever $1 \ E_M \ i$. It is not difficult to extend this method to handle the case where M is not a power of 2.

1.2.4. The Knuth-Morris-Pratt algorithm

An algorithm for recognizing occurrences of single strings in text was constructed by Knuth, Morris and Pratt [1977]. Their algorithm exploits the occurrence of repeated segments in the pattern. The key idea it uses is to shift by as much as determined by the longest proper prefix of the pattern which is also a suffix of the current partial match, instead of a mere shift by 1 as done by the naive algorithm. The length of this shift is called a *period* of the partial match. (For example, in the first stage of the attempt to match the pattern *abcaba* against text beginning with *abcabc*, a mismatch is detected at the right end of the pattern. At this point, the naive algorithm would shift the pattern by one position, while the Knuth-Morris-Pratt algorithm shifts the pattern three places.)

These periods can be computed in a preprocessing phase in time $O(M)$, and the matching phase can then be executed in $O(N)$ time. An interesting property of this algorithm is that the text string is inspected sequentially from left to right with no backup.

The preprocessing phase constructs a table of shifts corresponding to positions in the pattern, and this obviously requires $O(M)$ space. By partial computation of this shift table, Galil and Seiferas[1980] have achieved significant space savings. While the Knuth-Morris-Pratt algorithm computes the period $f(q)$ for each position q of the pattern, Galil and Seiferas suggest computing and storing only the distinct values of the "truncated" function $f(q)/k$ for a fixed k. Whenever a mismatch occurs at a position for which the value of shift is unavailable, their algorithm shifts the pattern by a constant amount. They demonstrate that there are only $O(\log m/\log k)$ distinct values of this truncated function, thereby obtaining an exponential saving in space.

1.2.5. The Aho-Corasick algorithm

A generalized version of the Knuth-Morris-Pratt algorithm was developed independently by Aho and Corasick [1975]. This algorithm determines whether any of several patterns occurs in a given text. A preprocessing phase is used to construct, from the set of patterns, a finite state machine with only two transitions out of every state: a 'success' transition and a 'failure' transition. Then the text is used as input to this machine. When the input symbol corresponds to the success transition of the current state of the machine, the machine makes a success transition; otherwise, it makes a failure transition.

The number of states in the machine is $O(M)$, so the space requirement is $O(M)$. The algorithm also inspects the text sequentially from left to right without backing up, and the match phase takes $O(N)$ time.

1.2.6. Shyamasundar's algorithm

An interesting way to use boolean matrices for matching multiple patterns has been shown by Shyamasundar [1976]. Given a set of patterns $w_1, \ldots, w_t$, over an alphabet Σ, a boolean matrix M of size $|\Sigma| \cdot (|w_1| + \ldots + |w_t|)$ is constructed as follows. M is partitioned as $M = M_1 \ldots M_t$, and the bit $M_i(x, j)$ is set whenever the j-th symbol of w_i is x. Three other bit vectors U, V, and Q are used as follows: $U = U_1 \ldots U_t$, with $|U_i| = |w_i|$, with each U_i having a 1-bit in the first position and zero everywhere else. The vector V is similarly constructed, except that V_i has its last bit 1, and zeros everywhere else.

Q is initially zero. The iterative step is to inspect the i-th character x of the text, and set

$$Q \leftarrow (rightshift(Q) \wedge U) \vee M(x, *).$$

If $Q \wedge V$ is now nonzero, then the current position in the text marks the right end of the occurrence of one of the patterns.

The method inspects the text sequentially left to right. It is useful when bit vector operations can be directly performed, and can easily be extended to the case where some patterns are prefixes of others. It is obvious that the method is a simulation of the nondeterministic finite-state machine constructed from the patterns, using bit vectors, and therefore cannot be used with large patterns without excessive storage costs.

1.2.7. The Boyer-Moore algorithm

Boyer and Moore [1978] invented an algorithm which is significantly better than the Knuth-Morris-Pratt algorithm on the average. Their algorithm matches the pattern beginning at the *right* end of the pattern instead of the left end. This idea has the merit that, when the mismatching text character does not occur in the pattern, we may shift the pattern as much as to carry its left end past the mismatching text character. This makes the algorithm sublinear on the average, because most often, with a large alphabet size, the algorithm repeatedly discovers that the one text character it consulted does not occur in the pattern, and shifts right M characters; thus, approximately N/M characters are consulted on the average.

This algorithm makes use of a preprocessing phase to construct a table of next occurrences of partial matches in the pattern, so as to maximize the shift at a mismatch. This algorithm is very fast when all we need to know is whether the pattern occurs at all in the string; but it behaves as badly as the naive algorithm (i.e., needs $O(MN)$ time) when we need to find all occurrences of the pattern in the text, and there are a large number of overlapping occurrences to be found. This poor worst-case behaviour is because the algorithm needs to re-examine all the text characters matching the pattern every time it detects a match. A subsequent modification by Galil [1979] yields an algorithm which has a worst case time linear in the length of text.

The proof of linearity of the Boyer-Moore algorithm is nontrivial. The first such proof was given by Knuth (Knuth, Morris, and Pratt [1977]), but was quite terse and hard to understand. Knuth proved that the number of comparisons made by the Boyer-Moore algorithm in the worst case is at most $7N$. Later, Guibas and Odlyzko [1980] presented a lucid argument to show that the algorithm makes at most $4N$ comparisons in the worst case. By contrast, the proof of linearity of the Knuth-Morris-Pratt algorithm is much easier.

It is noteworthy that the Boyer-Moore algorithm, unlike the Knuth-Morris-Pratt algorithm, requires random access to the text array. This is because of the right-to-left matching strategy.

1.2.8. Commentz-Walter's algorithm

Of particular interest to us is the paper by Commentz-Walter [1979], which is an attempt to extend the Boyer-Moore algorithm to multiple patterns. This paper exhibits an algorithm which achieves this extension, but behaves badly in the worst case. Her idea is to construct a trie, similar to the one constructed by the Aho-Corasick algorithm, but based on *reversed* patterns, and then use it in a way similar to the finite state machine. The major shortfall here is that the algorithm could make a short shift followed by a long match along a path in the trie which was not intended by the shift, such as, for instance, when matching the patterns ac and ba^m against the text a^n. Thus the worst-case time complexity of this algorithm is $O(ND)$, where D is the length of the longest pattern.

1.2.9. Other related work

The Boyer-Moore algorithm is sublinear on the average, i.e., in the average case it examines fewer characters than the length of the text. But this is not true in the worst case. In fact, Rivest [1977] shows the strong result that for every pattern p and every algorithm A_p that correctly determines whether p occurs in an arbitrary string s, there exists a string s which causes A_p to examine at least $|s| - |p| + 1$ characters of s.

Another approach to the pattern-matching problem was taken by Bailey and Dromey [1980]. They suggest dividing up the pattern into b subkeys by selecting every b-th character of the pattern, starting with the first through the b-th characters. An Aho-Corasick machine is now constructed to simultaneously search for all the b subkeys by consulting every bth character of the text, starting at the b-th character. Whenever a subkey match is found, a match algorithm is used to match the entire pattern. They demonstrate that this algorithm behaves better on the average than the Boyer-Moore algorithm, although not in the worst case.

An interesting application of string pattern matching to the detection of patterns in multidimensional arrays is discussed by Baker [1978]. In the two-dimensional case, given the rectangular pattern and text arrays, the method first assigns labels to the rows of the pattern such that two rows have distinct labels if and only if they are different. It then constructs an Aho-Corasick pattern-matching machine, with the set of distinct rows of the pattern as the set of one-dimensional patterns for the machine. It also constructs a Knuth-Morris-Pratt machine from the string of labels assigned to the rows of the pattern, and associates an instance of this machine with each column of the text array.

The algorithm then proceeds to run the Aho-Corasick machine on the text matrix row by row, from the top left, producing an output matrix M (of the same dimensions as the text matrix) such that $M_{i,j}$ is set to l when the Aho-Corasick machine detects a match of the pattern row labeled l ending at position $[i,j]$ of the text matrix. When such a match is detected, the Knuth-Morris-Pratt machine for column j makes a transition on symbol l. A match of the whole pattern is declared whenever one of the Knuth-Morris-Pratt machines detects a match.

Another related work is by Hoffmann and O'Donnell [1982]. They address the problem of searching for a pattern in a text, with both pattern and text being labeled trees. They generalize some of the techniques of the Knuth-Morris-Pratt and the Boyer-Moore algorithms to obtain efficient algorithms for their problem.

The subject of parallel algorithms for the string matching problem has been studied by Galil [1984] and by Vishkin [1985]. They develop algorithms that utilize many processors with shared memory, and determine all occurrences of the pattern in the text. In particular, Vishkin's work shows how to use any number p of processors (up to $N/\log N$) so as to solve the problem in $O(N/p)$ time.

1.3. Overview of our work

In the subsequent chapters, we will describe two algorithms to determine whether one of several given patterns occurs in a given text string. Each of these algorithms involves a preprocessing phase, in which some of the structure of the patterns is examined, followed by a matching phase, in which the text string is scanned. Our algorithms are based on the Boyer-Moore algorithm, extended in a manner similar to that of Commentz-Walter. The first of our algorithms runs in time $O(N\log D)$, as compared with the $O(ND)$ time bound for Commentz-Walter's algorithm. Our first algorithm uses at most $1+\log_4 D$ additional memory cells.

Our second algorithm is an attempt to improve the space bound. It attempts to remember only those matches that could potentially cause nonlinear behaviour. Given a user-specified parameter k, this algorithm uses at most t/k memory cells to remember nonperiodic matches, and runs in time $O(kN \log D)$.

We will also show how to solve completely the problem of preprocessing for the first of the algorithms, and describe an approach to solving the preprocessing problem for the second.

Chapter 2 introduces some preliminary notions and describes the general setting for the two algorithms. Chapter 3 describes the simpler of the two algorithms, and proves time and space bounds for the algorithm. Chapter 4 describes the second algorithm, and chapter 5 describes the preprocessing algorithms.

2. Preliminaries

The algorithm described by Commentz-Walter [1979] generalizes the Boyer-Moore algorithm to handle multiple patterns. Her algorithm retains the sublinear average-case behaviour of the Boyer-Moore algorithm, but has a nonlinear worst-case bound. Our algorithms extend Commentz-Walter's ideas. In this chapter, we will discuss Commentz-Walter's algorithm and the motivation for our work. We will then introduce some preliminary notions needed, and describe the framework for our algorithms.

2.1. Commentz-Walter's algorithm

In this section, we will examine Commentz-Walter's algorithm from a viewpoint that is suitable for deriving the framework for our algorithms in section 2.2.

Given a set of patterns $P = \{p_1, \ldots, p_t\}$ over an alphabet Σ, Commentz-Walter's algorithm begins by constructing a trie T such that:

(a) the root r is labeled ϵ (the null string), and each node $v \in T$ is labeled with a symbol $\sigma \in \Sigma$;

(b) each node v of depth d represents a unique d-character suffix $word(v)$ of the set of patterns, and conversely, each d-character suffix has a unique node of depth d representing it;

(c) for each node $v \neq r$ with label σ, $word(v) = \sigma \cdot word(parent(v))$.

For instance, given the patterns *cdabcd*, *eabcdaec*, and *deccec*, the algorithm constructs the trie shown in Figure 2.1.

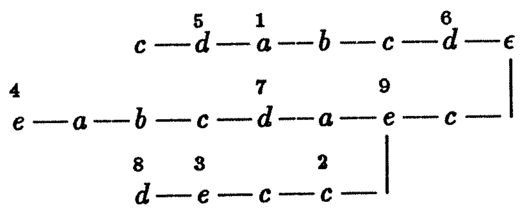

Figure 2.1

The idea is to extend the Boyer-Moore algorithm in a manner similar to the way in which the Aho-Corasick algorithm generalizes the Knuth-Morris-Pratt algorithm. So the trie is to be used as a state-transition graph while moving left along the text. In a manner analogous to the Aho-Corasick algorithm, a preprocessing phase is first executed, after which the trie is matched against the text. To motivate the preprocessing needed, we will describe the matching phase first.

The matching phase begins by setting the root r as the current node, and aligning the trie against the text such that r is aligned with the character $text[w_{min} + 1]$ (where w_{min} is the length of the shortest pattern), and the leaves are to the left of the root. At each step, the current node v of the trie is examined for a child labeled with the next text symbol σ. If v has such a child w, the algorithm proceeds by making the child w the current node, and moving left one character on the text; a successful match is declared when a node that represents a complete pattern is reached.

When the current node v does not have a child labeled with the text character σ, the trie is shifted right, the root becomes the current node, and the process is restarted. We will discuss the computation of the amount of this shift presently, in section 2.1.2. But first, we will introduce and describe the notion of a *match*.

2.1.1. The concept of a match

We will use the term *match* to denote the string of labels of trie nodes that are successfully matched by the algorithm during a right-to-left movement between shifts.

For instance, in Figure 2.2, the match named m_1 is the string *abcd*. Notice that, by definition, the match does not include its terminating character (the letter *e* in this example).

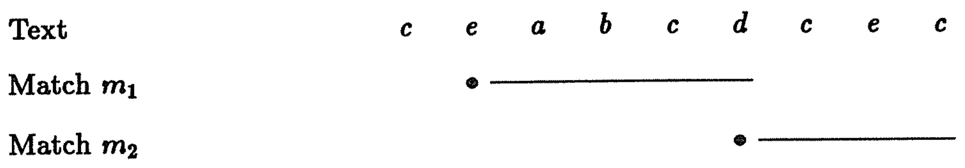

Figure 2.2

Some observations relating to matches can now be made.

(1) The total number of comparisons of text characters that the algorithm makes is comprised of two parts. One is the sum S of the lengths of the matches made. The second is the total number T of characters examined at the terminations of these matches; this quantity is not included in S, because the match does not include the terminating character. Now, on a text of length N, it must be that $T \leq N$. This is because every match begins (has right end) at a different text position, and we can charge each terminating character to the right end of the match; we will then have at most N such charged characters.

In analyzing our algorithms in chapters 3 and 4, we will derive bounds on the quantity S, the sum of the lengths of the matches. The following observation is now relevant.

(2) The length of a match m that ends at a certain trie node v is the same as the depth of v in the trie. We will denote the depth of a node v by $d(v)$. Also note that $d(v) = |word(v)|$.

As a notational convenience, we will often use the name of a string to also denote its length when no confusion arises.

2.1.2. Computing shifts

We are now ready to discuss the computation of the amount of shift. We will use the following notation in our discussion.

Notation. For nodes $v_1, v_2 \in T$ such that v_1 is a descendant of v_2, $string(v_1, v_2)$ denotes the string of labels along the path from v_1 to v_2, including both v_1 and v_2. Thus $word(v) = string(v, r)$ for all $v \in T$.

Notation. We say that a string $z \in \Sigma^+$ *occurs at* node w if either, for some descendant w' of w, $string(w', w) = z$, or, for some leaf descendant w'' of w, $string(w'', w)$ is a suffix of z. For instance, in Figure 2.1, the string *de* occurs at nodes 3 and 4, and the string *abcd* occurs at nodes 5, 6, 7, and 8.

The shift after a match is determined based on the following observations.

(1) A straightforward generalization of the Aho-Corasick idea indicates that the most shift that can be made from a current node v would be one less than the depth of a shallowest node w that is not a child of the root, such that $word(v)$ occurs at w. Let us denote this quantity by $s_1(v)$; i.e.,

$$s_1(v) = \begin{cases} \min\left(d(w) : word(v) \text{ occurs at } w \text{ and } d(w) > 1\right) - 1 \\ \infty, \quad \text{if there is no such } w. \end{cases}$$

This observation is the *match heuristic* in the terminology of Guibas and Odlyzko [1980].

(2) It is possible, however, to obtain a greater shift under some circumstances: if the text character σ that caused the mismatch occurs in the trie at the minimal depth d_σ, then we can shift the trie as far as $d_\sigma - d(v) - 1$, because any shorter shift would certainly fail to match σ. Formally,

$$d_\sigma = \begin{cases} \min\{d(w) : w \text{ is labeled } \sigma\} \\ \infty, \quad \text{if there is no such } w. \end{cases}$$

This observation is referred to as the *occurrence heuristic*.

(3) Under any circumstance, we cannot shift the trie right by more than w_{min}, the length of the shortest pattern, because any segment of text of length w_{min} could contain an occurrence of the shortest pattern.

Based on the above observations, we may define the amount of shift when a text character σ mismatches at the current node v:

$$s(v,\sigma) = \min\left(\max\left(d_\sigma, s_1(v)\right), w_{min}\right).$$

In the example of Figure 2.2, match m_1 fails at node 1, since node 1 does not have a child labeled e. The algorithm then shifts right three positions, since $s_1(1) = 3$. Match m_2 then fails at the child of node 2.

Thus the preprocessing phase for Commentz-Walter's algorithm computes the values of $s_1(v)$ for each node v of the trie, and the values of d_σ for each character $\sigma \in \Sigma$. This can be done in time $O(M + |\Sigma|)$, using the ideas of the preprocessing algorithm for the Aho-Corasick method.

For completeness, we will now describe the matching phase of Commentz-Walter's algorithm in formal terms.

Algorithm 2.1.

Inputs. The trie T with root r, the shift function s, and the text string *text* of length N.
Output. A decision whether *text* contains an occurrence of any of the patterns of the trie T.
Method.

```
begin
    $\tau \leftarrow w_{min} + 1$ { The position of the trie root }
    $v \leftarrow r$ { The current trie node }
    $i \leftarrow 1$
    repeat
        Let $\sigma = text[\tau - i]$
        if $v$ has a $\sigma$-child $w$
        then
            $v \leftarrow w$
            $i \leftarrow i - 1$
            if $v$ denotes a complete pattern
            then declare success and halt
        else
            $v \leftarrow r$
            $\tau \leftarrow \tau + s(v,\sigma)$
            $i \leftarrow 1$
    until $\tau > N$
end.
```

We can argue (informally) that in the average case, the size of the text alphabet is large compared with the number of distinct characters occurring in the trie. Therefore, most of the matches will consist of a single character that does not occur in the trie, so that the corresponding shifts will be by an amount equal to w_{min}. Thus, on the average, the algorithm consults N/w_{min} text characters.

The worst case, however, is not as good. Consider matching the trie constructed from the two patterns $P_1 = ba^m$ and $P_2 = ac$ against the text a^n. Each time the algorithm reaches the node labeled b, a mismatch occurs, followed by a shift of one character. The result is that in this case, the algorithm consults $O(mn)$ text characters.

2.2. The framework of our algorithms

The reason why Commentz-Walter's algorithm suffers from nonlinear worst-case bounds is that the algorithm "forgets" that certain text characters were already consulted, and reconsults those characters. Our goal is to incorporate a form of memory into the algorithm, such that, while retaining the sublinear average case behaviour, we will attempt to obtain a linear worst-case bound without requiring too much extra memory. When a mismatch occurs, we remember the node of the trie at which this occurs until the match is "paid for." This allows us to skip over regions of text that correspond to remembered matches. It also allows us to *abort* matches when, based on what the algorithm remembers, it decides that the path of the current match in the trie cannot possibly lead to a successful conclusion. These ideas are now illustrated.

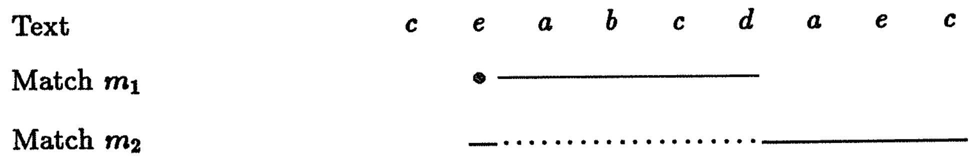

Figure 2.3

Consider matching the trie of Figure 2.1 against the text string *ceabcdaec*. Figure 2.3 shows the matches that occur in this case. During the time that the match m_2 is constructed, the node number 1, at which the match m_1 ended, is remembered; therefore, the algorithm "knows" (by preprocessing) where the string $word(1) = abcd$ occurs. This enables the algorithm to skip the over the text region shown by the dotted line. Under these circumstances, we treat m_2 as "subsuming" m_1, i.e., the text characters consulted by m_2 will be treated as all those between the left and right ends of m_2 (even though m_2 skipped over some of that region). Thus m_1 is "paid for" by m_2, and therefore need not be remembered after m_2.

As another example, if the algorithm is used to match the same trie against the text string *ceabcdcec* (the same text as in Figure 2.2), and the node number 1 is remembered when m_2 is constructed, the result is as shown in Figure 2.4. The difference between the cases illustrated in Figures 2.2 and 2.4 is that the match m_2 is aborted as soon as the node number 2 is reached. This

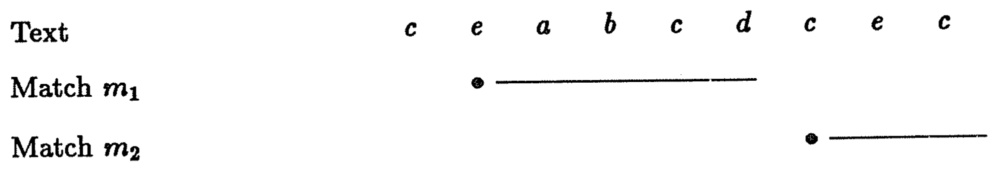

Figure 2.4

is possible because, by precomputation, the algorithm "knows" that there is no occurrence of the string $word(1) = abcd$ at the child of node 2.

In general, it is not sufficient to remember one previous match. Our algorithms will remember a sequence of nodes, representing the matches that ended at these nodes. The following observation is worth documenting.

Lemma 2.1. Any two matches remembered simultaneously must span disjoint text regions.

Proof. If two matches m_1 and m_2, occurring in that order, remembered by their terminating nodes v_1 and v_2, are such that m_2 overlaps m_1, then since m_1 was remembered when m_2 was constructed, it must be that m_2 skipped over the text region of m_1. But in that case, m_1 would not have been remembered after m_2.

$\square$

One consequence of this observation is that in order to make sense out of remembering the terminating nodes of several previous matches, we also need to remember the distances between them. This motivates the definition of a *configuration,* containing the matches remembered by the algorithm, as follows.

Definition. A *configuration* of the algorithm is a sequence of pairs,

$$C = ((v_1, d_1), \ldots, (v_{l-1}, d_{l-1}), (v_l, 0))$$

where each v_i is the terminating node of remembered match m_i, and d_i is the number of text characters between matches m_i and m_{i+1} that are not matched by either m_i or m_{i+1}.

For instance, in Figure 2.4, if the algorithm decided to remember both m_1 and m_2 after m_2 was constructed, the configuration would be $((1,1),(9,0))$. Here the value of d_1 is 1 because the terminating character e of the match m_2 is not included in the match m_2, by our convention. Note, in passing, that because of the existence of this terminating character, the value of d_i is always at least 1.

A second consequence of the idea of skipping over previous matches is that we have two kinds of matches:

Definition. A match m is said to be *open* if either

(a) no subsequent match overlaps m, or

(b) the first subsequent match that overlaps m does not skip over the text region of m.

A match that is not open is *closed.*

If match m_1 is skipped over by match m_2, then we will say that m_2 *closes* m_1. For example, in Figure 2.3, m_2 closes m_1.

An immediate consequence of this definition is the following observation.

Lemma 2.2. If an open match m has a subsequent overlapping match m', then m was forgotten before the construction of m' was begun.

Proof. If m had been remembered, then m' would either close m, or else have been aborted before reaching the text region of m.

$\square$

At this point, we have introduced the idea of remembering previous matches by the nodes at which they terminated. The questions that remain to be answered are:

(1) Which matches must be remembered, and for how long? and

(2) How much to shift when some matches are remembered?

We will discuss these questions in the opposite order in the following two subsections.

2.2.1. The shift function

The construction of the shift function is a straightforward extension of the observations we made in section 2.1.2 for obtaining the shift in Commentz-Walter's algorithm. Observations (2) and (3) made there are still valid. To obtain the proper generalization of the quantity we refered to as s_1 in observation (1), we will need to construct a string representing the sequence of remembered matches, along with don't-cares in intermediate positions. To this end, we will first introduce some notation.

Notation. The *characteristic string* of a configuration $C = ((v_1, d_1), \ldots, (v_l, 0))$, denoted $char(C)$, is the string

$$char(C) = word(v_1) \cdot \#^{d_1} \cdots \#^{d_{l-1}} \cdot word(v_l),$$

where the symbol # denotes a don't-care.

Thus, in Figure 2.1, for the configuration $C = ((2,3),(6,0))$, we have $char(C) = cec\#\#\#d$.

We can now extend the notion of occurrence to strings with don't-cares; for example, in Figure 2.1, the string *a#cd* occurs at nodes 5, 6, 7, and 8.

Next, we define the shift $s_1(C)$ from a configuration C, in a manner analogous to the definition in section 2.1.2:

Definition. The *goal* of a configuration C is the set of nodes at which $char(C)$ occurs:

$$goal(C) = \{w : char(C) \text{ occurs at } w \text{ and } d(w) > 1\}$$

and the *shift* $s_1(C)$ is defined to be

$$s_1(C) = \begin{cases} \min\{d(w) : char(C) \text{ occurs at } w, \text{ and } d(w) > 1\} \\ \infty, \text{ if there is no such } w. \end{cases}$$

Finally, accounting for the occurrence heuristic, we define the shift from a configuration C with a terminating text character σ to be

$$s(C, \sigma) = \min\left(\max(s_1(C), d_\sigma), w_{min}\right).$$

2.2.2. The memory function

Formally, given a configuration C and a particular match m_i in C, we define the *memory function* to be

$$\mu(C, i) = \begin{cases} 1 & \text{if } m_i \text{ is to be remembered in } C \\ 0 & \text{otherwise.} \end{cases}$$

It is clear that we can obtain a whole range of algorithms, depending upon our choice of μ. The approach we will take is to specify two different kinds of memory functions, one in each of chapters 3 and 4, and study the properties of the resulting algorithms.

2.2.3. Algorithm description

For completeness, we will now describe formally the match phase of our algorithms. We will say, in what follows, that two strings x and y *agree* if either x is a suffix of y or y is a suffix of x.

Algorithm 2.2.

Inputs.

Current configuration $CS = ((v_1, d_1), \ldots, (v_l, 0))$, initially empty.

Current trie node v, initally set to r, the root.

Current position τ of trie root, initially set to $w_{min} + 1$.

Current goal depth D, initially set to w_{min}. The quantity D counts the number of text characters between the current text position and the right end of the rightmost remembered match (not including either endpoint of that interval).

Function. Execute one move of the algorithm.

Method.

```
begin
    Let $\sigma = text[\tau - d(v) - 1]$, the current text character
    if $v$ does not have a $\sigma$-child
```

```
        then { match failed }
                Let $CS = ((v_1, d_1), \ldots, (v_l, 0))$
                Construct $CS' = ((v_1, d_1), \ldots, (v_l, D), (v, 0))$
                Shift $(CS')$;
                $CS \leftarrow CS'$;
        else
                Let $w$ be the $\sigma$-child of $v$
                if $w$ has descendant $w'$ such that
                        $w' \in goal(CS)$ and $d(w') - d(w) = D$
                or $w$ has a leaf descendant $w'$ with $d(w') - d(w) \leq D - 1$
                then
                        MoveLeft $(w, w')$
                else{ the match is to be aborted }
                        Construct $CS' = ((v_1, d_1), \ldots, (v_l, D), (v, 0))$
                        Shift $(CS')$;
                        $CS \leftarrow CS'$;
end.

procedure MoveLeft $(w, w')$
begin
        if $D > 1$
        then
                $D \leftarrow D - 1$;
                $v \leftarrow w$;
                if $w$ represents a complete pattern
                then halt ( match found at $w$ )
        else { need to skip over some earlier matches }
                Let $CS = ((v_1, d_1), \ldots, (v_{l-1}, d_{l-1}), (v_l, 0))$;
                Let $x$ be the deepest descendant of $w'$ such that
                $d(x) - d(w') < d(v_l)$ and $string(x, w')$ agrees with $word(v_l)$;
                if an ancestor of $x$ represents a complete pattern
                then halt (match found);
                $v \leftarrow x$;
                $D \leftarrow d_{l-1}$;
                $CS \leftarrow ((v_1, d_1), \ldots, (v_{l-1}, 0))$;
end {MoveLeft }

procedure Shift $(C)$
        { Shift based on current configuration $C$}
begin
```

```
        Let $C = ((v_1, d_1), \ldots, (v_l, 0))$;
        { Construct new configuration $C'$}
        $j \leftarrow 0$; { Current number of pairs in $C'$}
        for $k \leftarrow 1$ to $l$ do
        begin
                if $\mu(C, k) = 1$
                then
                        append $(v_k, d_k)$ to $C'$
                        $j \leftarrow j + 1$
                else
                        for the last match $(v', d')$ of $C'$,
                        $d' \leftarrow d' + d_k + d(v_k)$
        end { For loop }
        Let $C' = ((v'_1, d'_1), \ldots, (v'_j, d'_j))$;
        if $d'_j > 0$
        then
                $D \leftarrow d'_j + s(C, \sigma)$;
                $d'_j \leftarrow 0$;
        else
                $D \leftarrow s(C, \sigma)$;
        $\tau \leftarrow \tau + s(C, \sigma)$;{ Update trie root position }
        $C \leftarrow C'$; { Update configuration }
end Shift;
```

3. A simple algorithm

In chapter 2, we described the framework for our algorithms. There, we pointed out that different choices of the memory function lead to algorithms that behave differently. In this chapter we will provide one possible definition of the memory function. We will then analyze the algorithm resulting from this specification. On a text string of length N, this algorithm performs $O(N \log D)$ comparisons involving text characters, The algorithm remembers at most $1 + \log_4 D$ matches at any given time, where D is the length of the longest pattern, i.e., the height of the trie.

Section 3.1 describes the memory function, and the subsequent two sections analyze the time and space complexities of the resulting algorithm.

3.1. Memory function

The memory function that we will now describe is based on the following idea. If, for a match m, it is possible for a subsequent match m' to occur with right end not very far away from that of m, such that m' is comparable in length to m, then there could be a significant overlap between m and m', contributing to a nonlinearity. This condition is formalized in the following definition.

Definition. For a match m ending at a node v, a node w of the trie is *critical* if there is an ancestor x of w such that

(i) $2 \leq d(x) \leq \frac{1}{4}m - 1$

(ii) $string(w, x)$ agrees with $m = word(v)$

and (iii) $d(w) \geq \frac{1}{2}m$.

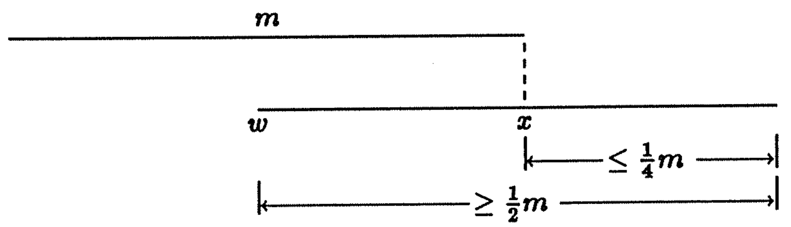

Figure 3.1

Figure 3.1 depicts the relationship between m and w. If m has a critical node w, then the path *word*(w) is the one that could be taken by a subsequent match m'. Condition (i) of the above definition stipulates, in essence, that m' should occur strictly further right of m, but not too far away from m. Condition (ii) says that m and m' agree on the region of text that they overlap on. Condition (iii) requires that the two matches be comparable in length.

If a match m ending at a node v does have a critical node w, then m is to be remembered until it is "paid for." We will think of the region of text to the right of m, of length $m/4$, as the region over which this happens. We now give this region a name.

Definition. For an open match m, with right end at text position r, we denote by $R(m)$ the range of text positions $r+1$ through $r+\frac{1}{4}m$, inclusive. We will informally refer to this text region as the *right neighbourhood* of m.

For a match m, with right end at r, the algorithm decides to remember m only if

(a) the match m has a critical node, and

(b) the current trie position τ satisfies $\tau - r \leq \frac{1}{4}m - 1$.

Otherwise, the algorithm forgets m.

In sections 3.2 and 3.3 we will establish time and space bounds for the algorithm using this memory function.

3.2. Time bound

The goal of this section is to show that the above algorithm consults $O(N \log D)$ text characters in the worst case. We will achieve this goal in steps. As pointed out in section 2.2, we need only account for open matches. First we will show, in lemma 3.1, that open matches of comparable lengths must occur at a certain minimum distance apart. We will then construct a charging argument to derive the time bound.

Lemma 3.1. For an open match m, every open match m' that begins in $R(m)$ is such that $m' < \frac{1}{2}m$.

Proof. Suppose, to the contrary, that for some open match m with right end r, there is an open match m' with right end r' such that $r' - r \leq \frac{1}{4}m$ and $m' \geq \frac{1}{2}m$, as shown in Figure 3.2.

It is easy to see that the node v' at which m' ends satisfies the conditions for a node that is critical with respect to m. However, the existence of such a node implies that m must have been remembered at the time m' was constructed, contradicting lemma 2.1.

□

We now need to establish a bound on the sum of the lengths of open matches that overlap with a given match m. To do this, we partition the matches that begin in $R(m)$ as follows.

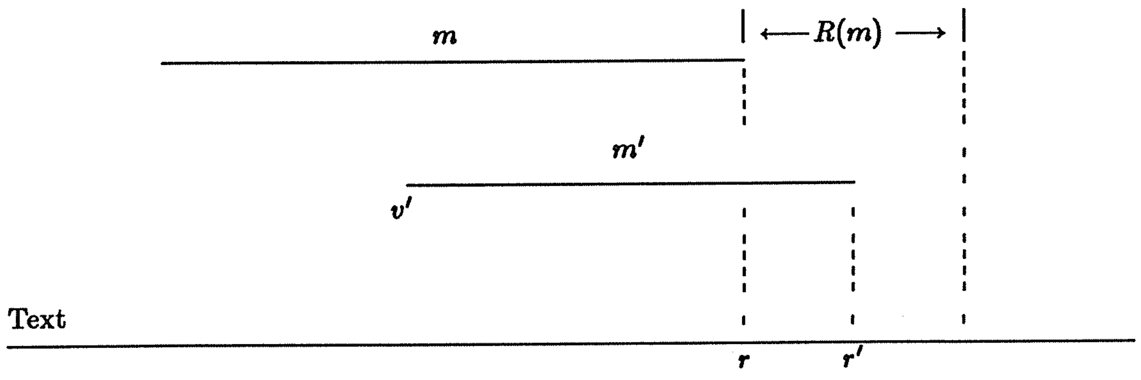

Figure 3.2

Definition. For a given open match m, denote

$$S_k(m) = \left\{m' : m' \text{ begins in } R(m), \text{ and } \frac{m}{2^{k+1}} < m' \leq \frac{m}{2^k}\right\}$$

for $k \geq 0$.

For any given open match m, the sets $S_k(m)$ partition the matches beginning in $R(m)$ according to length. By lemma 3.1, $S_0(m) = \emptyset$. Lemmas 3.2 and 3.3 bound the cardinalities of $S_k(m)$ for $k \geq 1$.

Lemma 3.2. For an open match m, and $k \geq 1 + \log D$, it must be that $|S_k(m)| = \emptyset$.

Proof. If $m' \in S_k(m)$, then $m' \leq m/2^k \leq D/2^k$ by definition of S_k, so that

$$k \leq \log \frac{D}{m'} \leq \log D$$

since $m' \geq 1$. Therefore, if $S_k(m)$ is nonempty, then $k \leq \log D$.

$\square$

Lemma 3.3. For an open match m, and $1 \leq k \leq \log D$,

$$|S_k(m)| \leq 2^{k+1}.$$

Proof. Suppose to the contrary that for some open match m, and some $k \geq 1$, the cardinality $|S_k(m)| \geq 2^{k+1} + 1$. Let matches $m_1, \ldots, m_{2^{k+1}+1} \in S_k(m)$, with right ends $r_1, \ldots, r_{2^{k+1}+1}$. Then, for each $i, 1 \leq i \leq 2^{k+1}$, it must be that $r_{i+1} - r_i > \frac{1}{4}m_i$; otherwise, m_{i+1} begins in $R(m_i)$ and $m_{i+1} \geq \frac{1}{2}m_i$ (by definition of $S_k(m)$), so that lemma 3.1 is contradicted with respect to m_i.

Since $m_i > m/2^{k+1}$, by definition of $S_k(m)$, we therefore have

$$r_{i+1} - r_i > \frac{1}{4}m_i > \frac{1}{4}\frac{m}{2^{k+1}}$$

so that

$$\begin{aligned} r_{2^{k+1}+1} - r_1 &= \sum_{1\le i\le 2^{k+1}} r_{i+1} - r_i \\ &> 2^{k+1}\cdot\frac{1}{4}\frac{m}{2^{k+1}} \\ &> \frac{1}{4}m \end{aligned}$$

which is impossible, because all the matches in $S_k(m)$ begin in $R(m)$.

□

Using these results, we can construct a charging argument to establish the time bound for the algorithm, as follows.

Theorem 3.4. The number $T(N)$ of text characters consulted by the algorithm in a text string of length N is bounded by

$$T(N) \le (4N + D)(2\log D + 1).$$

Proof. Charge the lengths of matches as follows. Set the current match to be the first match that occurred, and repeat the following steps:

(i) Charge to the current match m all the matches that begin in $R(m)$, and delete all the matches so charged;

(ii) Make current the first match remaining after m.

Now the total charge $c(m)$ accumulated by an open match m is

$$\begin{aligned} c(m) = \sum_{m' \text{ begins in } R(m)} m' &\le \sum_{k\ge 1}\ \sum_{m'\in S_k(m)} m' \\ &\le \sum_{1\le k\le \log D} 2^{k+1}\frac{m}{2^k} \quad \text{by definition of } S_k(m) \\ &\le 2m\log D. \end{aligned} \tag{1}$$

At the end of the deletions in this charging phase, call the remaining matches $m'_1, \ldots, m'_l$. For any two consecutive matches m'_i and m'_{i+1}, we must have

$$r'_{i+1} - r'_i > \frac{1}{4}m'_i$$

This follows because m'_{i+1} cannot begin in $R(m'_i)$, because of the charging phase. Therefore, adding these equations over all i,

$$N \geq r'_l - r'_1 > \frac{1}{4} \sum_{1\leq i\leq l-1} m'_i$$

$$\text{or} \sum_{1\leq i\leq l-1} m'_i \leq 4N$$

$$\text{i.e.,} \sum_{1\leq i\leq l} m'_i \leq 4N + D. \tag{2}$$

Thus

$$\begin{aligned} \sum_{m \text{ a match}} m &= \sum_{1\leq i\leq l} (m'_i + c(m'_i)) \\ &\leq \sum_{1\leq i\leq l} (m'_i + 2m'_i \log D) \qquad \text{by (1)} \\ &\leq (2\log D + 1) \sum_{1\leq i\leq l} m'_i \\ &\leq (2\log D + 1)(4N + D) \qquad \text{by (2).} \end{aligned}$$

□

3.3. Space bound

The algorithm remembers no more than $1 + \log_4 D$ matches at any time. This follows from the facts that any two matches remembered simultaneously must

(a) span disjoint text regions (by lemma 2.1), and
(b) differ by a factor of 4, by definition of the memory function.

Theorem 3.5. The algorithm remembers no more than $1 + \log_4 D$ matches at any given time.

Proof. Suppose, at some time, the algorithm remembers matches $m_1, \ldots, m_l$. Then it must be that the region of text currently being inspected lies within the range $R(m_i)$ for every i, $1 \leq i \leq l$, by definition of the memory function. Therefore, we must have $m_{i+1} \leq \frac{1}{4}m_i$, $1 \leq i \leq l-1$. Thus

$$m_l \leq \frac{m_1}{4^{l-1}}$$

or

$$l \leq 1 + \log_4 \frac{m_1}{m_l}$$

Since any match m must be in the range $1 \leq m \leq D$, we have

$$l \leq 1 + \log_4 D.$$

□

A question that arises naturally at this point is how to preprocess the trie so as to precompute the decisions of which matches to remember. In Chapter 5 we will discuss how to preprocess the trie efficiently.

4. A second time-space tradeoff

The algorithm of chapter 3 has time and space bounds dependent on the logarithm of the trie height. In general, this is undesirable, because we would like to have an algorithm that uses a bounded amount of resources independent of the trie size. In this chapter, we will describe an algorithm that makes some progress toward this goal. Specifically, the algorithm described here yields a time-space tradeoff as follows. Given any $k \geq 2$, the algorithm consults $O(kN \log D)$ text characters, and remembers at most $t/(k-1)$ nonperiodic matches at any time.

The restriction of the bound to nonperiodic matches turns out to be necessary because of the conjectured existence of classes of tries that can cause algorithms to require arbitrarily large amounts of time.

Definition. A string u is a *period* of string w if w is a prefix of u^k for some k. The shortest period of a string w is denoted $p(w)$.

Definition. A string w is said to be *periodic* if $p(w) \leq \frac{1}{4}w$, and *nonperiodic* otherwise. A match m ending at a node v is said to be *periodic* if $word(v)$ is periodic, and *nonperiodic* otherwise.

Conjecture 4.1. For each t and ϵ, there are tries with t leaves and text strings of length N that produce periodic matches such that any memory function (as defined in chapter 2) that remembers no more than $t-2$ matches would necessitate consulting $\Omega(N/\epsilon)$ text characters.

In the light of this conjecture, we will restrict ourselves to counting nonperiodic matches. Our algorithm will be required to remember periodic matches almost unconditionally until they are "paid for," but nonperiodic matches will be remembered only if the structure of the trie necessitates it. Evidence supporting this conjecture will be provided in chapter 6.

4.1. Algorithm description

The essential idea of this algorithm is the following. Given a sequence of remembered matches, and a newly constructed match, the algorithm examines the trie to determine whether it is worth remembering the new match. The criterion for remembering a match, however, is more complex than the one used in chapter 3. Given a match m and a configuration containing it, the algorithm looks for *critical paths* in the trie that have a certain structure. Intuitively, a path is critical with respect to a match m if

(1) it is possible for a previous match m' to have occurred along this path, and

(2) the distance between the right ends of m and m' is small, but the lengths of m and m' are comparable, so that there is significant overlap, contributing to a nonlinearity.

The algorithm decides to remember a match if there are approximately k such critical paths, where k is a user-specified parameter that serves as a measure of allowable nonlinearity.

We will now make precise these notions.

Definition. Strings u and v are said to ***agree*** if either u is a suffix of v or v is a suffix of u.

Notation. For nodes v and w of the trie, we write v ***pref*** w if $word(v)$ is a proper prefix of $word(w)$.

We are now ready to define the idea of a critical path.

Definition. A path $P = (v, w)$ from leaf v to node w in the trie is ***critical*** with respect to match m_i in configuration $C = ((v_1, d_1), \ldots, (v_l, 0))$ if all of the following conditions are satisfied (see Figure 4.1):

(C1) $word(w)$ is nonperiodic;

(C2) $d(w) \geq \frac{1}{2}m_i$;

(C3) For some node w' along the path of m_i such that $1 \leq d(w') - d(w) \leq \frac{1}{16}m_i$, we have w ***pref*** w';

and (C4) for each j, $1 \leq j \leq i-1$, if there is a node u_j at depth $d(w) + (m_i - d(w')) + d_{i-1} + \sum_{j+1 \leq l \leq i-1} (m_l + d_{l-1})$ along P (where w' is as defined in (C3)), then $string(v, u_j)$ agrees with m_j.

Given such a critical path (v, w), we will say that the quantity $d(w') - d(w)$ is the ***displacement*** of the critical path, where w' is as in the above definition. The fraction values 1/2 and 1/16 are chosen for reasons which will be clear subsequently.

Figure 4.1 shows an example of a path (v, w) that is critical with respect to match m_2 in a configuration containing the matches m_1 and m_2.

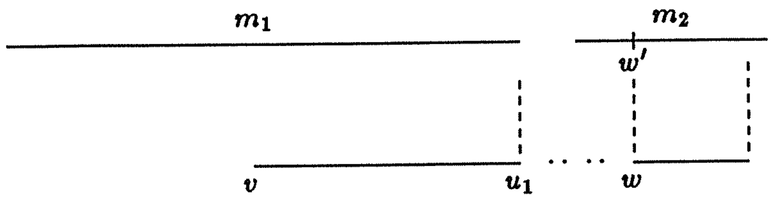

Figure 4.1

It is possible for a leaf v to have several ancestors $w_1, \ldots, w_p$, such that $(v, w_1), \ldots, (v, w_p)$ are all critical with respect to m_i in configuration C. However, we will show in lemma 4.7 that these paths cannot have distinct displacement values.

Definition. A *k-mesh* with respect to a match m_i in a configuration C is a set of $k-1$ critical paths with respect to m_i in C, such that all the critical paths have distinct displacement values.

The reason for insisting on distinct displacement values for the critical paths in a k-mesh is the following. If a match m has $k-1$ critical paths, each represents the path that might have been taken by an earlier match which m might overlap. Each of these earlier matches begins at a different text position, because the algorithm shifts by at least one position after every match. This observation is reflected in the requirement of distinct displacement values.

We will find it convenient to refer to regions of text surrounding a given match as the "neighbourhoods" of the match:

Definition. For match m with right end at text position r, the range of text positions $r+1$ through $r+\frac{1}{16}m$ is denoted $R(m)$ (the "right neighbourhood" of m) and the range $r+1-\frac{31}{16}m$ through $r-m$ is denoted $L(m)$ (the "left neighbourhood" of m).

The regions $R(m)$ and $L(m)$ are defined so that their lengths satisfy the relation $L(m)+m+R(m)=2m$.

The memory function is now specified as follows: for match m with right end at text position r, remember m in configuration C throughout $R(m)$ (i.e., whenever the current trie position satisfies $r+1-r \leq \frac{1}{16}m$) if

either (a) m is periodic,

or (b) m has a k-mesh in C.

These definitions imply that, once the algorithm decides to remember a match m, the only way that m is forgotten is when the trie position goes past the region $R(m)$. Moreover, if a match m_3 has a k-mesh in a configuration C containing matches m_1, m_2, and m_3 (occurring in that order), then m_3 still has a k-mesh in the configuration obtained by deleting m_2 from C.

4.2. Time bound

In this section, we will establish that the algorithm using the memory function defined above consults $O(N \log D)$ text characters. The main result that the proof hinges on is lemma 4.2, which shows that when several open matches of about the same length overlap, their right ends must be separated by a minimum amount. Once we have proved this result, we can construct a charging argument to establish the time bound.

In order to prove lemma 4.2, we will need two technical lemmas. The first is the following "gcd lemma," proved in Knuth, Morris and Pratt [1977]. Intuitively, the gcd lemma says that no string can have two short periods p and q without having an even shorter period that divides into both p and q.

Gcd lemma. If p and q are periods of a string x, and $p+q \leq x$, then $\gcd(p,q)$ is also a period of x.

$\square$

We will now apply this result to a very specific circumstance that arises in our proof. We will need, later, to use a fact that if we have two strings m_1 and m_2 aligned with right ends very close together, where m_1 has a very short period and m_2 has a long period, then it must be that many of the suffixes of m_2 have long periods. This fact is now formalized and proved.

Lemma 4.1. Let m_1 be a string with $p(m_1) \leq \frac{2}{15}m_1$, and let m_1 be a proper prefix of string m_2, where $m_2 \leq \frac{17}{16}m_1$ and $p(m_2) > \frac{1}{4}m_2$. Then if u is any suffix of m_2 such that $u \geq \frac{7}{16}m_2$, then $p(u) > \frac{1}{2}u$. In particular, $p(m_2) > \frac{1}{2}m_2$.

Proof. Let v be the prefix of u that is a suffix of m_1. Figure 4.2 depicts the relationship between these strings.

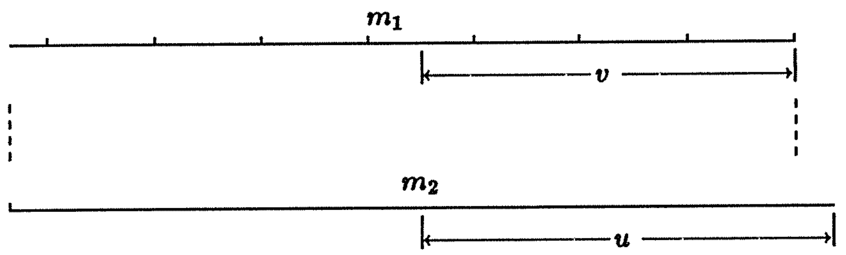

Figure 4.2

Then

$$\begin{aligned} v &\geq u - \frac{1}{16}m_1 \\ &\geq \frac{7}{16}m_2 - \frac{1}{16}m_1 \\ &\geq \frac{7}{16}m_1 - \frac{1}{16}m_1 \\ &\geq \frac{3}{8}m_1. \end{aligned}$$

So there are at least $\frac{3}{8} \cdot \frac{15}{2} = \frac{45}{16}$ occurrences of the basic period of m_1 in v, so $p(v) \leq \frac{16}{45}v$.

We also have the following bound for v in terms of u:

$$\begin{aligned} v &\geq u - \frac{1}{16}m_1 \\ &\geq u - \frac{1}{16}m_2 \\ &\geq u - \frac{1}{16} \cdot \frac{16}{7}u \\ &= \frac{6}{7}u. \end{aligned}$$

Now if $p(u) \geq v$, then $p(u) \geq \frac{6}{7}u$, and we are done. Otherwise (i.e., $p(u) < v$), there is some period of v of length $p(u)$. Now $p(v)$ does not divide $p(u)$, for otherwise m_2 would share the same period with m_1, making m_2 periodic. Therefore, when the gcd rule is applied to v, it must be that

$$p(u) + p(v) > v$$

for otherwise we would have a period of length $\gcd(p(u), p(v)) < p(v)$ for both u and v, making m_2 periodic. Thus,

$$\begin{aligned} p(u) &> v - p(v) \\ &\geq v - \frac{16}{45}v \\ &= \frac{29}{45}v \\ &\geq \frac{29}{45} \cdot \frac{6}{7}u \\ &> \frac{1}{2}u. \end{aligned}$$

□

We are now ready to prove our most important lemma, that we cannot have too many open matches that are about the same length too close together. The method of proof is to argue that if we did have many such matches close together, then at least one of them must have had a k-mesh in the configuration in which it was forgotten. However, in order to exhibit such a k-mesh, we will need to obtain nonperiodic strings in order to meet condition (C1). This is where the result of lemma 4.1 will be applied.

Lemma 4.2. Let $m_1, \ldots, m_{2k}$ be nonperiodic open matches with right ends $r_1, \ldots, r_{2k}$ in that order, such that any two matches differ in length by at most a factor of 2, i.e., for all i and j,

$$\frac{1}{2}m_i \leq m_j \leq 2m_i.$$

Then $r_{2k} - r_1 > \frac{1}{16}m_l$, where m_l is shortest among all the matches.

Proof. Suppose to the contrary that $r_{2k} - r_1 \leq \frac{1}{16}m_l$. Figure 4.3 depicts the relationship between the matches.

Since all the matches are open, $m_1, \ldots, m_{2k-1}$ must all have been forgotten, because they all have subsequent overlapping matches. Therefore, it suffices to show that there is some match among $m_1, \ldots, m_{2k-1}$ that has a k-mesh in the configuration in which it was forgotten.

Let $x_1, \ldots, x_{2k}$ be the nodes at the left ends of $m_1, \ldots, m_{2k}$. Consider some match m_j that was forgotten in configuration C_j. The matches $m'_1, \ldots, m'_p$ that occurred earlier than m_j and were remembered in C_j were certainly remembered at each of $m_1, \ldots, m_{j-1}$. So, by definition

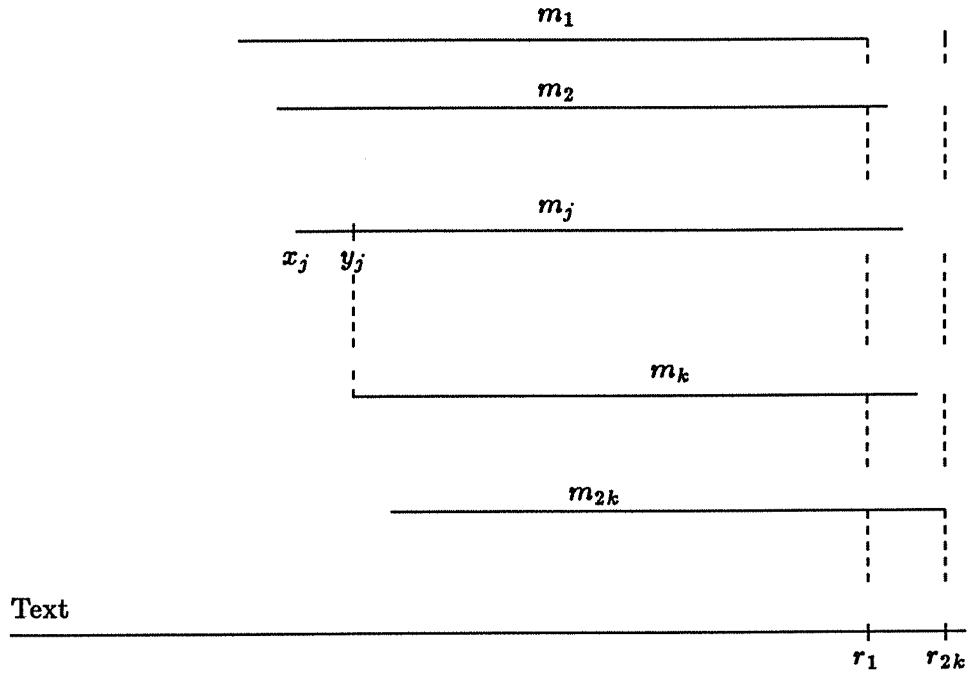

Figure 4.3

of the shift function, there are leaf descendants $v_1, \ldots, v_{j-1}$ of $x_1, \ldots, x_{j-1}$ such that the paths $(v_1, x_1), \ldots, (v_{j-1}, x_{j-1})$ all have "images" of $m'_1, \ldots, m'_p$. So each of these paths satisfy condition (C4) of the definition of critical path with respect to m_j in C_j.

The idea of our proof is to show that either m_k or m_{2k-1} must have had a k-mesh in the configuration in which it was forgotten. We will first attempt to construct a k-mesh for m_k. If we can choose nodes $w_1, \ldots, w_{k-1}$ along the paths of $m_1, \ldots, m_{k-1}$ respectively, such that the paths $(v_1, w_1), \ldots, (v_{k-1}, w_{k-1})$ satisfy the rest of the conditions, we would have a set of $k-1$ critical paths for m_k in C_k. Since the matches $m_1, \ldots, m_{k-1}$ have distinct right ends, the corresponding critical paths all have distinct displacement values, and thus we would have a k-mesh for m_k. Now we may choose, for each i, any ancestor w_i of x_i that satisfies (C3), and the paths (v_i, w_i) would satisfy (C4). It remains to choose the w's to satisfy (C1) and (C2). Therefore, suppose it is not possible to make such a choice. Then, in particular, there is some x_j such that none of its ancestors can be so chosen. We will now construct a k-mesh for m_{2k-1}.

Since m_j itself is nonperiodic and has the appropriate length (by hypothesis), it must be that m_k ends under m_j (otherwise we could choose $w_j = x_j$). Furthermore, the node y_j along m_j that aligns with x_k is such that $d(y_j) \geq \frac{16}{17} m_k$, since $r_k - r_j \leq \frac{1}{16} m_l$. So

$$p(word(y_j)) \leq \frac{1}{4} \cdot \frac{1}{2} m_k,$$

because otherwise we could choose for w_j the ancestor of y_j at depth $\frac{1}{2}m_k$. Thus

$$\begin{aligned} p(word(y_j)) &\le \frac{1}{8}\cdot\frac{17}{16}d(y_j) \\ &= \frac{17}{128}d(y_j) \\ &< \frac{2}{15}d(y_j). \end{aligned}$$

We now have $word(y_j)$ and $word(x_k)$ satisfying the conditions of lemma 4.1, so it follows that every suffix u of m_k with $u \ge \frac{7}{16}m_k$ must have $p(u) > \frac{1}{2}u$. Now each of the matches $m_k, \ldots, m_{2k-2}$ shares with m_k a suffix u of m_k of length at least $\frac{7}{16}m_k$, because $r_{2k} - r_1 \le \frac{1}{16}m_l$ and any two matches differ by at most a factor of 2.

We can now construct a k-mesh for m_{2k-1} by deriving, for each i in the range $k \le i \le 2k-2$, a critical path for m_{2k-1} from m_i. These critical paths will all have distinct displacement values, since the right ends of the corresponding matches are all distinct.

Case 1. m_{2k-1} goes at least as far left as m_i.

Choose $w_i = x_i$. This satisfies (C1), because m_i is nonperiodic, by hypothesis. It also satisfies (C2), because $m_i < m_{2k-1}$ (since m_{2k-1} goes at least as far left as m_i) and $m_i \ge \frac{1}{2}m_{2k-1}$, by hypothesis.

Case 2. m_i goes farther left than m_{2k-1}.

Choose $w_i = y_i$, where y_i is the ancestor of x_i that lines up with the left end x_{2k-1} of m_{2k-1}. This choice satisfies (C2), because $r_{2k-1} - r_i \le \frac{1}{16}m_i$ so that $d(y_i) \ge \frac{16}{17}m_{2k-1} > \frac{1}{2}m_{2k-1}$. This choice also satisfies (C1), because of the following: if m_k ends at or right of y_i, then $word(y_i)$ contains all of m_k as a substring, in which case

$$p(word(y_i)) \ge p(m_k) > \frac{1}{2}m_k \ge \frac{1}{4}d(y_i),$$

since m_k and m_i differ by at most a factor of 2. If m_k ends left of y_i, then the suffix u of m_k that is a prefix of $word(y_i)$ is also a prefix of m_{2k-1}, so that

$$\begin{aligned} u &\ge m_{2k-1} - \frac{1}{16}m_k \\ &\ge \frac{1}{2}m_k - \frac{1}{16}m_k \qquad \text{(by hypothesis)} \\ &\ge \frac{7}{16}m_k. \end{aligned}$$

Thus $p(u) > \frac{1}{2}u$. Now since u is a prefix of m_{2k-1} and a suffix of m_k, we must have

$$\begin{aligned} u &= m_{2k-1} - (r_{2k-1} - r_k) \\ &\ge m_{2k-1} - \frac{1}{16}m_{2k-1} \text{ by hypothesis} \\ &\ge \frac{15}{16}m_{2k-1}. \end{aligned}$$

Therefore,

$$d(y_i) \le m_{2k-1} \le \frac{16}{15}u$$

so that, since u is a substring of $word(y_i)$,

$$\begin{aligned} p(word(y_i)) &> \frac{1}{2}u \\ &\ge \frac{1}{2} \cdot \frac{15}{16} d(y_i) \\ &> \frac{1}{4} d(y_i). \end{aligned}$$

Consequently, we can always choose nodes $w_k, \ldots, w_{2k-2}$ along the paths of $m_k, \ldots, m_{2k-2}$ respectively, such that the paths $(v_k, w_k), \ldots, (v_{2k-2}, w_{2k-2})$ constitute a k-mesh for m_{2k-1} in the configuration in which it was forgotten, contradicting our assumption that m_{2k} begins in $R(m_{2k-1})$. (This contradiction is the only reason we needed m_{2k}.)

□

This result will now be applied to more specific situations. Our method of accounting for matches is the following. For a given open match m, consider the subsequent matches that have right ends close to that of m and overlap a significant amount with m. We can divide these matches into three classes:

(1) matches that go strictly further left than m, and have left ends far from that of m;

(2) matches that go strictly further left than m, but have left ends close to that of m; and

(3) matches that do not go further left than m, but overlap m significantly.

We will first show (in lemma 4.3) that for a given match m, there cannot be very many matches of the class 2. Next we will define precisely what we mean by a "significant overlap" of class 3, and show (in lemma 4.5) that the matches of class 3 must sum up to a constant times m. These two results will then allow us to argue, in the charging phase (theorem 4.6), that we can charge matches of the above two categories to m, and delete all such matches, thus simplifying our problem.

Lemma 4.3. For an open match m, at most $2k-1$ open matches begin in $R(m)$ and end in $L(m)$.

Proof. Only the last of the matches that begin in $R(m)$ and end in $L(m)$ can be periodic, because any earlier periodic match would be remembered through $R(m)$. Now if there are at least $2k$ matches beginning in $R(m)$ and ending in $L(m)$, then the match m, together with the first $2k-1$ matches that begin in $R(m)$ and end in $L(m)$, satisfy the conditions of lemma 4.2, and m is the shortest among these matches. This would contradict the conclusion of lemma 4.2.

□

Definition. For match m with right end at r, we say that an open match m' that ends under m, with right end at r' in $R(m)$, is *m-heavy* if $r' - r \le \frac{1}{2}m'$. If r' is in $R(m)$, but $r' - r > \frac{1}{2}m'$, then we say that m' is *m-light*.

Informally, our intent is to classify the matches that have right ends close to m as *heavy* if they have a significant overlap with m, and *light* otherwise. Our immediate objective is to bound the sum of the lengths of the m-heavy matches, for every match m. To do this, we proceed as follows. First, for an open match m, we partition the m-heavy matches as follows, according to the length of the region of overlap with m.

Definition. For an open match m, and for each $i \ge 0$, denote

$$S_i(m) = \left\{ m' : m' \text{ is } m\text{-heavy and nonperiodic, and } \frac{m}{2^{i+1}} \le m' - (r' - r) < \frac{m}{2^i} \right\}.$$

Next we show (in lemma 4.4) that the number of elements in each class in this partition is bounded by a constant, and establish a bound on the maximum length and number of elements of such a class. Lemma 4.5 will then establish the bound on the sum of lengths.

Lemma 4.4. For an open match m, with right end r,

$$\max\{m' : m' \in S_i(m)\} \le \min\left\{ \left(\frac{1}{2^i} + \frac{1}{16}\right) m, \frac{m}{2^{i-1}} \right\}$$

and

$$|S_i(m)| \le \min\left\{2^{i+2}k - 1, 64k - 1\right\}.$$

Proof. We will first establish the bound on the lengths. If match m' with right end at r' is in $S_i(m)$, then

$$\begin{aligned} \frac{1}{16}m &\ge r' - r \quad \text{by definition of } m\text{-heaviness} \\ &> m' - \frac{m}{2^i} \quad \text{by definition of } S_i(m) \end{aligned}$$

so that

$$m' \le \left(\frac{1}{2^i} + \frac{1}{16}\right) m.$$

Also, observe that if $m' \in S_i(m)$ is a match that has right end r' as far right as possible, then $r' - r \le m/2^i$, for otherwise, since $r' - r \le \frac{1}{2}m'$ (by m-heaviness),

$$m' \ge 2(r' - r)$$

or

$$m' - (r' - r) \ge r' - r > \frac{m}{2^i}$$

which is impossible, since $m' \in S_i(m)$. Now, by definition of $S_i(m)$,

$$m' - (r' - r) < \frac{m}{2^i}$$
$$\text{or} \quad m' < \frac{m}{2^{i-1}}.$$

Next, we will show the bound on the cardinality of $S_i(m)$. Suppose there are lk matches $m_1, \ldots, m_{lk}$ in $S_i(m)$, with right ends at $r_1, \ldots, r_{lk}$ respectively. Then by lemma 4.2, we have for all j,

$$r_{2jk} - r_{(2j-2)k+1} > \frac{1}{16}\frac{m}{2^{i+1}}$$

so that

$$r_{lk} - r_1 > \frac{l}{2} \cdot \frac{1}{16}\frac{m}{2^{i+1}}.$$

Therefore, $l \geq 2^{i+2}$ implies that

$$r_{lk} - r_1 > \frac{m}{16}$$

which cannot be, since all the matches begin in $R(m)$. Also, $l \geq 64$ implies that

$$r_{lk} - r_1 > \frac{m}{2^i},$$

which we have already observed to be impossible.

□

We can now establish a bound on the sum of the lengths of the m-heavy matches, for any given match m: we need only separate the perioidic and the nonperiodic m-heavy matches, and apply lemma 4.4 to obtain a bound on the nonperiodic matches.

Lemma 4.5. For an open match m,

$$\sum_{m' \text{ is } m\text{-heavy}} m' \leq \left(\frac{143}{4}k - \frac{5}{16}\right) m.$$

Proof. We will account for the periodic and the nonperiodic m-heavy matches separately. Let $m_1, \ldots, m_p$ be the sequence of periodic m-heavy matches, with right ends at text positions $r_1, \ldots, r_p$ respectively. Then, for each i, m_i is remembered unconditionally through its right neighbourhood, by definition of the memory function. Also, any two of these matches must overlap at least at the rightmost character of m, since all these matches are m-heavy. Therefore,

$$r_{i+1} - r_i > \frac{1}{16}m_i \quad \text{for } 1 \leq i \leq p-1$$

and adding all these equations,

$$r_p - r_1 > \frac{1}{16} \sum_{1 \le i \le p-1} m_i.$$

Since $r_p - r_1 \le \frac{1}{16} m$ (by definition of m-heaviness),

$$\frac{1}{16} m > \frac{1}{16} \sum_{1 \le i \le p-1} m_i$$

i.e.,

$$m + m_p > \sum_{1 \le i \le p} m_i.$$

But $m_p \le \frac{17}{16} m$, since m_p ends under m, so that

$$m + \frac{17}{16} m > \sum_{1 \le i \le p} m_i$$

and thus

$$\sum_{\substack{m' \text{ is } m\text{-heavy} \\ \text{and periodic}}} m' \le \frac{33}{16} m.$$

Now using lemma 4.4 and the definition of $S_i(m)$,

$$\begin{aligned} \sum_{\substack{m' \text{ is } m\text{-heavy} \\ \text{and nonperiodic}}} m' &\le \sum_{i \ge 0} |S_i(m)| \cdot \max\{m'' : m'' \in S_i(m)\} \\ &\le \sum_{0 \le i \le 3} m \left(\frac{1}{2^i} + \frac{1}{16} \right) (2^{i+2} k - 1) + \sum_{i \ge 4} \frac{m}{2^{i-1}} (64k - 1) \\ &\le \left(\frac{143}{4} k - \frac{19}{8} \right) m. \end{aligned}$$

□

We are now ready to prove our time bound. The proof technique is a charging argument in four phases. For a given open match m, we divide the matches with right ends close to that of m into three classes, as described earlier, and account for each of these classes in the first three phases. The last phase sums up the resulting bounds.

Theorem 4.6. The algorithm consults $O(kN \log D)$ text characters.

Proof. As discussed in chapter 2, we need only account for open matches. The proof is a charging argument in phases.

Phase 1. Iterate the following step: Charge to the first remaining match m all the matches that begin in $R(m)$ and either end in $L(m)$ or are m-heavy; delete all the matches so charged.

By lemmas 4.3 and 4.5, each match m picks up a cost $c(m)$ of

$$\begin{aligned} &\le \left(\frac{143}{4}k - \frac{5}{16}\right) m + (2k-1)\,2m \\ &\le \left(\frac{159}{4}k - \frac{37}{16}\right) m. \end{aligned}$$

At the end of this phase, one of the following holds for any two matches m_1 and m_2 occurring in that order:

(1) m_2 does not start in $R(m_1)$;

(2) m_2 is m_1-light; or

(3) m_2 goes left of $L(m_1)$.

Each of the subsequent phases will now deal with, and get rid of, one of the above conditions.

Phase 2. Charge each match m, as well as $c(m)$, to the first subsequent match m' such that m' begins in $R(m)$ and goes strictly left of $L(m)$ (if there is such an m'), and delete m.

To account for the total cost picked up by a match, let $S(m)$ denote the set of matches charged *directly* to m by phase 2. Define, for match m,

$$height(m) = \begin{cases} 0 & \text{if } S(m) = \emptyset \\ 1 + \max\ \{height(m') : m' \in S(m)\} & \text{otherwise.} \end{cases}$$

Let $len_i(m)$ be the sum of the lengths of the matches charged to a match m of height i by phase 2. We will show by induction on height that

$$len_i(m) \le m \sum_{1 \le j \le i} \left(\frac{18}{31}\right)^j .$$

Suppose m has height 1, and let $m_1, \ldots, m_l$, occurring in that order, be the set of matches charged to m directly by phase 2. Then m starts in $R(m_i)$ for all i, and therefore m_j is m_i-light for $i < j$ (because of phase 1). Therefore, if $r_1, \ldots, r_l$ are the right ends of $m_1, \ldots, m_l$, respectively, then

$$r_{i+1} - r_i \ge \frac{1}{2} m_{i+1} \text{ for each } i,$$

and adding all these equations,

$$r_l - r_1 \ge \frac{1}{2} \sum_{2 \le j \le l} m_j$$

or

$$\sum_{1 \le j \le l} m_j \le m_1 + 2(r_l - r_1).$$

Now $r_l - r_1 \leq \frac{1}{16}m_1$, because all the m_i begin in $R(m_1)$, since m itself does. Therefore,

$$\begin{aligned}\sum_{1\leq j\leq l} m_j &\leq m_1 + 2\left(\frac{1}{16}m_1\right)\\ &= \frac{9}{8}m_1\\ &\leq \frac{18}{31}m,\end{aligned}$$

because $m_1 \leq \frac{16}{31}m$ since m goes strictly left of $L(m_1)$.

For the inductive step, if m has height i, then all the matches $m_1, \ldots, m_l$ have height at most $i-1$, by definition of height. Therefore the cost charged to m is

$$\begin{aligned}len_i(m) &= \sum_{1\leq j\leq l}(m_j + len_{i-1}(m_j))\\ &\leq \sum_{1\leq j\leq l}\left(m_j + m_j \sum_{1\leq p\leq i-1}\left(\frac{18}{31}\right)^p\right) \qquad \text{(by inductive hypothesis)}\\ &\leq \left(\sum_{1\leq j\leq l} m_j\right)\left(1+\sum_{1\leq p\leq i-1}\left(\frac{18}{31}\right)^p\right)\\ &\leq \frac{18}{31}m\left(1+\sum_{1\leq p\leq i-1}\left(\frac{18}{31}\right)^p\right)\\ &= m\sum_{1\leq p\leq i}\left(\frac{18}{31}\right)^p.\end{aligned}$$

Therefore, the greatest possible sum of lengths of matches that a match m can pick up directly through phase 2 is

$$len_i(m) \leq m \sum_{1\leq j\leq\infty}\left(\frac{18}{31}\right)^j \leq \frac{18}{13}m$$

and the total cost charged to m through phases 1 and 2 is

$$\begin{aligned}&\leq \left(1+\frac{18}{13}\right)\left(\frac{159}{4}k - \frac{37}{16}\right)m\\ &\leq \left(\frac{4929}{52}k - \frac{1147}{208}\right)m.\end{aligned}$$

At the end of phase 2, for any two matches m_1 and m_2 occurring in that order, at least one of the following is true:

(1) m_2 does not begin in $R(m_1)$, or

(2) m_2 is m_1-light.

Phase 3. Iterate the following step: Charge to the first remaining match m all the matches that are m-light; delete all the matches so charged.

We now derive a bound on the cost picked up by a given match through phase 3. As with the m-heavy matches, we will divide the m-light matches into classes based on their lengths. Define, for each i,

$$L_i(m) = \left\{ m' : m' \text{ is } m\text{-light and } \frac{m}{2^{i+1}} \le m' < \frac{m}{2^i} \right\}.$$

First observe that $L_0 = \emptyset$ (because no m-light match can have length $m/2$ or greater), and that $L_i = \emptyset$ for all $i \ge \log m - 1$ (by definition of L_i). Since the longest that any match m can be is D characters, it follows that $L_i(m) = \emptyset$ whenever $i \ge \log D - 1$.

Now, for some $1 \le i \le \log D$, let $L_i(m) = \{m_1, \ldots, m_p\}$, occurring in that order. Then, for any two matches m' and m'' in $L_i(m)$ occurring in that order, m'' does not begin in $R(m')$; otherwise, since m' and m'' differ by at most a factor of 2 (since they are both in $L_i(m)$), m'' would be m'-heavy and would have been taken care of by phase 2. Therefore, if $r_1, \ldots, r_p$ are the right ends of $m_1, \ldots, m_p$, we must have

$$r_{j+1} - r_j > \frac{1}{16} m_j, \quad \text{for } 1 \le j \le p-1$$

so that, summing over all j,

$$\frac{1}{16} m \ge r_p - r_1 > \frac{1}{16} \sum_{1 \le j \le p-1} m_j.$$

(The first inequality above holds because all the m_j begin in $R(m)$.) Now, noting that, since m_p is m-light, $m_p \le \frac{2}{16} m$, we have

$$\sum_{m' \in L_i(m)} m' < \frac{9}{8} m.$$

Since this bound holds for all the sets $L_i(m)$, we have

$$\sum_{m' \text{ is } m\text{-light}} m' \le \frac{9}{8} m \log D.$$

Thus the total cost charged to a match m through phases 1, 2 and 3 is

$$\le \left(\frac{4929}{52} k - \frac{1147}{208} \right) \left(1 + \frac{9}{8} \log D \right) m.$$

Phase 4. For the remaining matches $m_1, \ldots, m_l$ with right ends $r_1, \ldots, r_l$,

$$\begin{aligned} r_{j+1} - r_j &> \frac{1}{16} m_j \\ \text{so that} \quad r_l - r_1 &> \frac{1}{16} \sum_{1 \le j \le l-1} m_j \\ \text{or} \quad \sum_{1 \le j \le l} m_j &< m_l + 16\left(r_l - r_1\right) \\ &< 17N. \end{aligned}$$

Since each match now carries a charge,

$$\begin{aligned} \sum_{m \text{ a match}} m &< \sum_{1 \le j \le l} \left(m_j + c\left(m_j\right)\right) \\ &< \sum_{1 \le j \le l} \left(m_j + \left(1 + \frac{9}{8} \log D\right) \left(\frac{4929}{52} k - \frac{1147}{208}\right) m_j\right) \\ &= O(kN \log D). \end{aligned}$$

□

4.3. Space bound

In this section, we will establish a bound on the number of nonperiodic matches that the algorithm remembers. Notice that a bound very similar to that of theorem 3.5 holds for this algorithm as well: by an argument very similar to that of theorem 3.5, we can show that at most $1 + \log_{16} D$ matches (periodic or otherwise) are remembered at any time. To prove the bound on the nonperiodic matches remembered, we will need to examine the way in which the k-meshes of the remembered matches interact. This is done in lemmas 4.7 and 4.8. But first, we need some notation for the set of leaves of a k-mesh:

Definition. For a nonperiodic match m remembered in configuration C, denote

$$\lambda_C(m) = \{v : (v, w) \text{ is part of a } k\text{-mesh for } m \text{ in } C\}.$$

Now we can show that there are no shared leaves either within a k-mesh or between k-meshes.

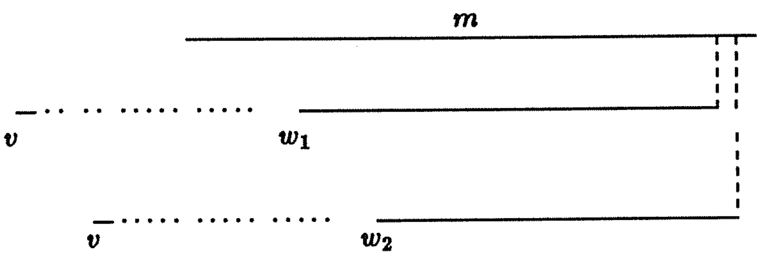

Figure 4.4

Lemma 4.7. For a nonperiodic match m remembered in configuration C,

$$|\lambda_C(m)| \geq k - 1.$$

Proof. If not, then there are critical paths (v, w_1) and (v, w_2) in the k-mesh that share the same leaf v, and with distinct displacement values (see Figure 4.4).

Assume without loss of generality that $d(w_2) > d(w_1)$. Then $word(w_1)$ and $word(w_2)$ agree at two distinct positions that are no more than $\frac{1}{16}m \leq \frac{1}{8}d(w_1)$, by conditions (C3) and (C2) of the definition of critical path. So the shorter of $word(w_1)$ and $word(w_2)$ is periodic, contradicting condition (C1).

□

Lemma 4.8. If m and m' are both nonperiodic matches remembered in configuration C, then

$$\lambda_C(m) \cap \lambda_C(m') = \emptyset.$$

Proof. Suppose to the contrary that $v \in \lambda_C(m) \cap \lambda_C(m')$. Assume wlog that m and m' occur in that order. So (v, w) is a critical path for m, and (v, w') is a critical path for m'.

Now by condition (C4) of the definition of critical path of m', m occurs along this path at depth at most $\frac{1}{16}m$, so that, in particular, $word(w)$ occurs along this path at depth at most

$$\frac{1}{16}m + \left(\frac{9}{16}m - \frac{1}{2}m\right) = \frac{1}{8}m \leq \frac{1}{4}word(w).$$

So $word(w)$ is periodic, contradicting condition (C1) for m.

□

The bound on the number of nonperiodic matches is now immediate:

Theorem 4.9. The algorithm remembers at most $t/(k-1)$ nonperiodic matches.

Proof. If nonperiodic matches $m_1, \ldots, m_l$ are remembered in configuration C, then by lemmas 4.5 and 4.6,

$$t \geq \sum_{1 \leq i \leq l} |\lambda_C(m_i)| \geq l(k-1)$$

or

$$l \leq \frac{t}{k-1}.$$

$\square$

5. Preprocessing algorithms

We now have two algorithms for matching a pattern trie against a text string, assuming that the required preprocessing can be done in a reasonable amount of time. In this chapter, we will show how to perform preprocessing for the simple algorithm (of chapter 3) efficiently. The problem of preprocessing for the algorithm of chapter 4 is much more complex, and we will describe some ideas that might be used in obtaining a good solution.

Throughout this chapter, we will use M to denote the sum of the lengths of the patterns, and D to denote the length of the longest pattern.

5.1. Preprocessing for the simple algorithm

Preprocessing for the simple algorithm involves precomputing the memory function, the shift function, and the decisions of when to abort matches. We will examine each of these problems in turn in the following subsections.

5.1.1. Computing the memory function

Since the memory function is defined for individual nodes without regard to the configuration in which the node (or match) is remembered, we can precompute the memory function on a node-by-node basis. In other words, we could perform a depth-first traversal of the trie, keeping track of the required information as we go, and label each node according to whether or not it has a critical node.

We will need to use the following notion of *cover*.

Definition. We will say that a node w ***covers*** node v if $\mathit{word}(v)$ is a proper prefix of $\mathit{word}(w)$ and $d(w) \leq 2d(v)$.

The intent of this definition is that if a node v has a critical node w, then v has an ancestor v' such that w covers v'. Thus, in order to compute the memory function, we will need, for each trie node v, a list of all nodes that cover v. This computation can be performed by a straightforward extension of the algorithm due to Aho and Corasick [1975].

In their work, Aho and Corasick show an algorithm to compute a ***failure function,*** defined as follows. For a node v, the failure function $f(v)$ is the deepest node w such that $\mathit{word}(w)$ is a proper prefix of word (v). Once the function f is constructed, the lists of covers that we need can be computed as follows.

Algorithm 5.1.

```
*Input.* The trie of patterns, and the failure function $f$.
*Output.* For each node $v$, a list $Cover(v)$ of nodes that cover $v$.
*Method.*
Begin
    for each node $w$ do $Cover(w) \leftarrow \emptyset$;
    for each node $w$ in the trie do
    begin
        $x \leftarrow f(w)$;
        while $d(x) \geq \frac{1}{2}d(w)$ do
        begin
            $Cover(x) \leftarrow Cover(x) \cup \{w\}$;
            $x \leftarrow f(x)$;
        end
    end
end.
```

Since the number of entries in the *Cover* lists could be as many as about M^2 in the worst case, this algorithm runs in $O(M^2)$ time.

We are now ready to discuss the computation of the memory function μ. Our algorithm for computing μ performs a depth-first search of the trie of patterns. It maintains a list of "depth intervals," each of which is a pair (d_l, d_h) such that each descendant w of the current node v (in the depth-first search) in the range of depths d_l through d_h (inclusive) has a critical node. This list is maintained sorted in lexicographic order. When a node is visited in the process of the search, its memory function is assigned. The depth-interval list is then augmented as follows. When a node w covers the current node v, any descendant v' of v that satisfies the depth bounds $d(v') \geq 4(d(w) - d(v))$ and $d(w) \geq \frac{1}{2}d(v')$ would have w as a critical node, by definition. Therefore, the pair $(4(d(w) - d(v)), 2d(w))$ is inserted into the depth interval list for each such node w. The children of the current node are then visited recursively.

Algorithm 5.2.

```
*Input.* The trie of patterns, with root $r$.
*Output.* The memory function $\mu$.
*Data structures used.* The list *Dlist* of depth-intervals.
*Method.*
Begin
    *Dlist* $\leftarrow \emptyset$;
    for each node $v$ do $\mu(v) \leftarrow 0$;
    *Search* ($r$, *Dlist*);
end.
```

```
procedure Search (v, Dlist)
begin
        Let $(d_l, d_h)$ be the first interval on Dlist;
        if $d_l \leq d(v) \leq d_h$
        then $\mu(v) \leftarrow 1$;
        if $d(v) = d_h$
        then delete $(d_l, d_h)$ from Dlist;
        for each $w$ that covers $v$ do
                Add to Dlist the pair $(4(d(w) - d(v)), 2d(w))$;
        for each child $x$ of $v$ do
                Search (w, Dlist);
        Undo the changes to Dlist;
end;
```

The first of the two loops takes $O(M^2 \log D)$ time, since each insertion into a sorted list takes $O(\log D)$ time. Therefore, the above algorithm runs in $O(M^2 \log D)$ time.

5.1.2. Computing the shift function

The proof of time bound of the simple algorithm, as described in chapter 3, does not rely very heavily on the amount of shift. In fact, it suffices to ensure that we do not consult the text in a match that is currently remembered. This means, in particular, that we can get by with shifting by an amount equal to the least depth at which $word(v_l)$ occurs, where v_l is the rightmost remembered match (or by w_{min}, the length of the shortest pattern).

To compute this shift, we will first need some information on the structure of the trie.

Definition. The relation *pref* on trie nodes is defined as follows. For nodes x and y of the trie, we will write x *pref* y if $word(x)$ is a prefix of $word(y)$.

Definition. For nodes x and y of the trie, we write $x \preceq y$ if

(i) x *pref* y, and

(ii) for all nodes z, if z *pref* y, then z *pref* x.

The following observations shed some light on the nature of the $\preceq$ relation.

Definition. The *failure function* of a node v (as defined by Aho and Corasick [1975]), denoted $f(v)$, is the deepest node w such that $word(w)$ is a prefix of $word(v)$.

It is evident from the definitions that for nodes x and y in the trie, $x \preceq y$ if and only if $f(y) = x$. Moreover, the relation $\preceq$ is a tree, because

(1) it is acyclic;

(2) if $x \preceq z$ and $y \preceq z$, then $f(z) = x$ and $f(z) = y$, implying that $x = y$ because f is well-defined; and

(3) the root r of the trie has label ϵ, and so (trivially) r *pref* x for every node x. Thus r qualifies as the unique root of the $\preceq$ relation.

The $\preceq$ tree can be obtained directly from the Aho-Corasick failure function f, since $x \preceq y$ if and only if $f(y) = x$.

Now we will define, for each node v, a quantity we denote $lpref(v)$, from which it will be easy to compute the required shift.

Definition. For node v, the function $lpref(v)$ is the depth of a shallowest leaf w such that v *pref* w, and $lpref(v) = \infty$ if there is no such leaf w.

This function can be computed by a straightforward depth-first search of the $\preceq$ tree, as follows.

Algorithm 5.3.

Input. The $\preceq$ tree, and the trie with root r.

Output. The function *lpref.*

Method.

```
    begin
        for  each node $v$ such that $r \preceq v$ do
            $lpref(v) \leftarrow search(v)$
    end.

    function  $search$ ($v$)
        { Returns the depth of the shallowest leaf $w$ such that $v$ $pref$ $w$}
    begin
        $p \leftarrow \infty$
        for  each node $w$ such that $v \preceq w$ do
        begin
            if $w$ is a leaf in the trie
            then
                $p \leftarrow \min(p, d(w))$;
                $s \leftarrow search(w)$; { Ignore returned value }
            else
                $p \leftarrow \min(p, search(w))$
        end
        $lpref(v) \leftarrow p$;
        return  ($p$)
    end
```

This depth-first search takes $O(M)$ time. Having constructed the function *lpref,* it is easy to compute the shift from a node v, as follows.

Algorithm 5.4.

Input. The trie with root r, the relation $\preceq$, and the function *lpref.*
Output. The shift function $s(v)$, defined to be the least depth at which $word(v)$ occurs, or w_{min}, whichever is smaller.
Method.

```
begin
    for each child $v$ of $r$ do
        $search(v, \infty)$
end.

procedure  $search(v, n)$
    $\{n = \min(lpref(x) - d(x) : x \text{ is an ancestor of } v)\}$
begin
    $s(v) \leftarrow \min(w_{min}, \{d(w) - d(v) : v \preceq w\}, n)$
    for  each child $w$ of $v$ do
        $search(w, \min(n, lpref(v) - d(v)))$
end
```

This algorithm is a depth-first traversal of the trie, in which every node is consulted once on entry to procedure *search* and at most once in the minimum calculation of the first line of procedure *search*. Therefore, the algorithm takes $O(M)$ time.

5.1.3. Computing abort decisions

When a set of nodes is currently remembered by the algorithm, and a node is reached from which there is no possibility of continuing to a successful match, the match must be aborted. These decisions of when to abort a match can either be precomputed, or else computed as and when they are needed. One possibility would be to preprocess the trie to compute a function *action* such that, given the current node w and the rightmost remembered node v of the trie, $action(v, w)$ specifies some minimal-depth descendant of w at which $word(v)$ occurs. We can compute *action* in a manner similar to computing the shift function. We can then make use of *action* when making abort decisions.

This raises the question of how much storage is needed for the function *action*. An obvious bound is $M^2/2$, assuming that $action(v, w)$ is defined for all possible nodes v and w. The observation that $action(v, w)$ is meaningful only when $d(w) \leq \frac{1}{4}d(v)$ yields a better bound of $M^2/8$ cells for the storage of *action*. It seems that even better bounds can be derived, because it is unlikely that *action* is defined for every node pair v, w that satisfy $d(w) \leq \frac{1}{4}v$.

5.2. Preprocessing for the second algorithm

Many approaches are possible for preprocessing for the second algorithm. At one extreme, given the pattern trie, we could precompute all the possible configurations that can arise from the trie, as well as which matches to remember in each configuration. This approach seems to require an inordinate amount of space. If at most l matches need to be remembered, then a simple estimate of the number of configurations that could arise is $O(N^l)$, where N is the number of trie nodes. It is certainly possible that the actual number of configurations is much less. Tighter bounds on this quantity need to be established.

At the other extreme, we could keep the preprocessing to a minimum, and the decisions of which matches to remember and how much to shift can be made during the match phase instead of during preprocessing. This approach would require excessive computation in situations where there are many short matches followed by short shifts. Of course, a whole range of other solutions between these two extremes are possible.

We will assume a hybrid approach, one in which some of the computation is done as part of the preprocessing and some during the match phase. In this section, we will address the problem of determining, given a configuration $C = ((v_1, d_1), \ldots, (v_l, 0))$ and a match $m_i \in C$, whether m_i has a critical path. An answer to this question provides part of the solution to the problem of preprocessing. Other questions remain to be settled, including computing the shift from a configuration, and precomputing abort decisions.

We begin by defining a ***potential critical path***:

Definition. For a leaf v and a node x, we will say that a path (v, w) from v to an ancestor w of v is ***potentially critical*** with respect to node x if w is a minimal-depth node such that

(1) $word(w)$ is nonperiodic;

(2) $d(w) \geq \frac{1}{2}d(x)$;

and (3) for some ancestor w' of x such that $1 \leq d(w') - d(w) \leq \frac{1}{16}d(x)$, we have $w \; pref \; w'$.

It is clear from this definition and the definition of critical path (chapter 4) that if (v, w) is critical with respect to some match m that ends at node x in configuration C, then (v, w) is potentially critical with respect to x, because the conditions above are almost identical to the conditions C1 through C3 of the definition of critical path. Let $pc(v, x)$ denote the minimal-depth node w such that (v, w) is potentially critical with respect to x. The function pc is well-defined, as can be shown in a manner similar to the proof of lemma 4.7.

We will first exhibit an algorithm to compute the function pc.

Algorithm 5.5.

Input. The trie with root r, and the $\preceq$ relation.

Output. The function pc defined above.

Method.

```
    begin
    for each leaf $v$ do
    begin
        Let $w_1, \ldots, w_p$ be the ancestors of $v$, in order of
            increasing depth, such that $d(w_1) \geq 8$ and $word(w_i)$
            is nonperiodic for all $i$;
        $low \leftarrow 8$
        for $i \leftarrow 1$ to $p$ do
        begin
            $high \leftarrow 2d(w_i)$
            for  each node $w'$ such that $w_i$ pref $w'$ and
                                $1 \leq d(w') - d(w_i) \leq \frac{1}{8}d(w_i)$ do
            begin
                for each descendant $w''$ of $w'$ with $low \leq d(w'') \leq high$ do
                    $pc(v, w'') \leftarrow w$
            end
            $low \leftarrow high + 1$
        end
    end
end.
```

Suppose m, ending at node x, has a potential critical path (v, w) such that w has the least depth possible. Then $word(w)$ is nonperiodic, so let $w = w_i$ in the ordering of the above algorithm description. By the minimality of depth of w, $d(w_{i-1}) < \frac{1}{2}d(x)$. Therefore, on the i-th iteration through the *for* loop, the algorithm will have set $low = 2d(w_{i-1}) + 1 \leq d(x)$ and $high = 2d(w_i) \geq d(x)$, so that the algorithm sets $pc(v, x)$ to w. Moreover, for any given v and x, the value of $pc(v, x)$ is set at most once, because the values taken by low and $high$ partition the trie into disjoint depth ranges. Consequently, the algorithm finds all potentially critical paths for nodes in the trie.

Now for a leaf v, the statement $pc(v, w'') \leftarrow w$ is executed at most M times, because the values taken by low and $high$ partition the trie into disjoint ranges of depths, and one such partition is explored in each iteration of the second *for* loop. Thus the algorithm takes time $O(Mt)$.

To explore the question of how "good" this algorithm is, consider the set of patterns specified by $p_i = a^{2n-i}b^i$, $1 \leq i \leq t$, for sufficiently large n. All suffixes of p_i of length at least $i + 1$ are nonperiodic. In the trie constructed from these patterns, for each leaf v_j, $1 \leq j \leq p - 1$, and each ancestor w_i of v_j, with $\max(i + 1, 16) \leq d(w_i) \leq n$, there are nodes $x_{j+1}, \ldots, x_t$ along the paths of $p_{j+1}, \ldots, p_t$ such that $pc(v_j, x_l)$ must be set to w_i, for $j + 1 \leq l \leq t$. Thus the number of pairs (v, w) for which the function pc is defined is $\Omega(Mt)$. Thus the above algorithm is almost optimal.

Now, given a configuration $C = ((m_1, d_1), \ldots, (m_l, 0))$ and a newly constructed match m_{l+1} that ends at node x, we may obtain critical paths for m_{l+1} by examining, for each leaf v, whether

$(v, pc(v,x))$ satisfies condition C4 of the definition of critical path. This can be done by checking, for $1 \leq i \leq l$, whether the match m_i occurs at the appropriate depth along this path, using the $pref$ relation. This check takes at most l queries of the $pref$ relation per leaf, so that the total time taken is $O(tl)$.

6. Summary and open questions

We have shown how to extend the Boyer-Moore pattern-matching algorithm to handle multiple patterns without incurring excessive worst-case time and space costs. We have also derived some results concerning preprocessing algorithms. It is worth pointing out that, as in the case of the single-pattern version of the Boyer-Moore algorithm, our algorithms remain sublinear in the average case when handling a large alphabet size. This is because, when the alphabet size is large, the occurrence heuristic is used much more often than the match heuristic when shifting the trie, so that not every text character is examined by the algorithm.

Our investigations raise several questions worth pursuing, and we will now discuss some of them briefly.

6.1. Pathological examples

Conjecture 4.1 claims that, when we are allowed to construct patterns that yield periodic matches, it is possible to construct sets of patterns that would cause any memory function that remembers fewer than $t-1$ matches to behave arbitrarily badly in terms of text characters consulted. We will now provide evidence to substantiate this conjecture. Given a number $t \geq 2$ and $\epsilon \geq 0$, we can construct a set of t patterns $p_1, \ldots, p_t$ over the alphabet $\Sigma = \{a_1, \ldots, a_t, z\}$ as follows. Let

$$p_1 = z(a_1 \ldots a_t)^n$$

and for $2 \leq i \leq t$, let

$$p_i = (a_1 \ldots a_t)^{2n} z(a_2 \ldots a_1)^{2\epsilon n} z \ldots z(a_{j+1} \ldots a_j)^{2\epsilon^j n} \ldots z^{t\epsilon^{i-1} n}(a_i \ldots a_{i-1})^{\epsilon^{i-1} n}.$$

Every pattern in this set has a different rightmost character, so that the root of the trie has t children and every other internal node has one child. The idea is to match this trie against the pattern $a_t(a_1 \ldots a_t)^m$.

For instance, suppose $t = 4$. The patterns in this case would be

$$\begin{aligned}
p_1 &= z(a_1a_2a_3a_4)^n \\
p_2 &= (a_1a_2a_3a_4)^{2n} z(zzzz)^{\epsilon n}(a_2a_3a_4a_1)^{\epsilon n} \\
p3 &= (a_1a_2a_3a_4)^{2n} z(a_2a_3a_4a_1)^{2\epsilon n} z(zzzz)^{\epsilon^2 n}(a_3a_4a_1a_2)^{\epsilon^2 n} \\
\text{and } p4 &= (a_1a_2a_3a_4)^{2n} z(a_2a_3a_4a_1)^{2\epsilon n} z(a_3a_4a_1a_2)^{2\epsilon^2 n} z(zzzz)^{\epsilon^3 n}(a_4a_1a_2a_3)^{\epsilon^3 n}.
\end{aligned}$$

Consider the computation when this trie is matched against the periodic text $a_4(a_1a_2a_3a_4)^n$ by an algorithm that remembers no more than $t - 2 = 2$ matches at any time. We can make two observations:

(1) Every match is a substring of the text, and is therefore of the form

$$a_j \dots a_4(a_1a_2a_3a_4)^k a_1 \dots a_l;$$

(2) There are no aborted matches, because any aborted match must end at a node with at least two children, and in the above trie the root is the only node meeting this condition.

When a match m is forgotten, the path of m in the trie will be reused before the algorithm makes a shift of more than ϵm. This suggests that the constant of linearity is about $1/\epsilon$.

It seems reasonable to conjecture that any such set of patterns would have a "logarithmic" structure, i.e., the matches produced would have to diminish in length by a multiplicative factor so as to facilitate "reuse" of a forgotten match before making significant progress along the text.

This suggests two possible conclusions. One is that the algorithm of chapter 3 is fairly close to optimal, since it does not remember any more than $O(\log D)$ matches, and since the longest that a match can be is the height of the trie. The second conclusion is that a closer study of such pathological examples might yield better criteria than those developed in chapter 4 for remembering matches, so that, for example, we might be able to obtain a better time-space tradeoff without having to remember periodic matches unconditionally through their right neighbourhoods.

6.2. Tries with bounded disparity

Throughout our analyses, our emphasis has been on attempting to prevent too much overlap between matches. We assumed that any two trie nodes v and w, such that some prefix of $word(v)$ is a suffix of $word(w)$, could contribute overlapping matches. This is probably too strong an assumption. It is usually not very hard to construct examples with almost arbitrary overlaps when the disparity in depths between the shallowest and deepest leaves is very large. However, the question of what happens when this disparity is bounded (e.g., tries in which the shallowest leaf is at least half as deep as the deepest) remains to be investigated. It seems likely that more efficient algorithms can be constructed under the latter constraint.

6.3. A lower bound on the time-space product

Another question worth examining is how close our time-space tradeoff comes to the optimal. To explore this issue, we would first need to define precisely the notion of a "right-to-left" style algorithm as against the left-to-right Aho-Corasick algorithm, because the space needed by the latter is merely that for a single node of the trie. A lower bound on the number of matches remembered by an algorithm in this class, that runs in linear time, would then have to be proved. An important aspect of such a definition would be to ensure that the Aho-Corasick machine could not be somehow encoded into the construction.

A related question is whether we can construct an algorithm that, given a pattern trie as input, yields the fewest number of matches that need to be remembered by a "right-to-left" technique without sacrificing the linear time bound.

6.4. Preprocessing issues

The simple algorithm of chapter 3 makes localized decisions of whether to remember a match without regard to what other matches are currently remembered. This makes it easy to preprocess the trie to precompute these decisions. The resulting memory function needs no more than $O(M)$ storage.

The algorithm for the general case, however, makes its decisions based on the current match as well as the currently remembered matches. In this situation, we must first decide whether to precompute all the decisions the algorithm has to make during the matching phase. At issue here is whether there exist alternative strategies that might compute some of the decisions during the match phase and still yield good results. Assuming that the memory function is entirely precomputed, a simple estimate of storage needed for the memory function is $O(M^l)$, when remembering at most l matches. But this bound is obtained assuming that all nodes can occur at all positions in a configuration, which is certainly not the case. It would be interesting to derive a better bound than this, by utilizing more of the structure of the trie.

6.5. Other extensions

One of reasons for remembering many nonperiodic matches is to circumvent difficulties due to matches along long paths followed by short shifts due to the existence of a short pattern. It would be interesting to make a fairly realistic assumption that the patterns do not differ greatly in length, and derive better space bounds.

Another nice extension would be along the lines of Baker [1978] to obtain algorithms for matching multidimensional patterns.

7. References

[1] Alfred V. Aho, "Pattern matching in strings," in *Formal language theory: perspectives and open problems*, Ronald V. Book (ed.), Academic Press, 1980, 325–347.

[2] Alfred V. Aho and Margaret J. Corasick, "Efficient string matching: an aid to bibliographic search," *Communications of the ACM* **18**, June 1975, 333–340.

[3] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman, *The design and analysis of computer algorithms*, Prentice-Hall, 1974, Chapter 9.

[4] Thomas A. Bailey and R. Geoff Dromey, "Fast string searching by finding subkeys in subtext," *Information Processing Letters* **11**, November 1980, 130–133.

[5] Theodore P. Baker, "A technique for extending rapid exact-match string matching to arrays of more than one dimension," *SIAM Journal on Computing* **7**, November 1978, 533–541.

[6] Robert S. Boyer and J. Strother Moore, "A fast string searching algorithm," *Communications of the ACM* **20**, October 1977, 762–772.

[7] Beate Commentz-Walter, "A string-matching algorithm fast on the average," *Automata, Languages and Programming,* Sixth Colloquium, Springer-Verlag, July 1979, 118–132.

[8] Zvi Galil, "On improving the worst-case running time of the Boyer-Moore string-searching algorithm," *Communications of the ACM* **22**, September 1979, 505–508.

[9] Zvi Galil, "Optimal parallel algorithms for string matching," *ACM Symposium on Theory of Computing,* 1984.

[10] Zvi Galil and Joel Seiferas, "Saving space in fast string matching," *SIAM Journal on Computing* **9**, May 1980, 417–438.

[11] Leo J. Guibas and Andrew M. Odlyzko, "A new proof of the linearity of the Boyer-Moore string searching algorithm," *SIAM Journal on Computing* **9**, November 1980, 672–682.

[12] Malcolm C. Harrison, "Implementing the substring test by hashing," *Communications of the ACM* **14**, December 1971, 777–779.

[13] Richard M. Karp, Raymond E. Miller, and Arnold L. Rosenberg, "Rapid identification of repeated patterns in strings, trees and arrays," *Eighth Annual ACM Symposium on Theory of Computing*, 1972, 125–136.

[14] Richard M. Karp and Michael O. Rabin, "Efficient randomized pattern-matching algorithms," Technical report 31-81, Aiken Computation Laboratory, Harvard University, Cambridge, Mass. 02138, 1981.

[15] Donald E. Knuth, James H. Morris, Jr., and Vaughan R. Pratt, "Fast pattern matching in strings," *SIAM Journal on Computing* **6** (1977), 323–350.

[16] L. Reeker, "A note on a badly nondeterministic automaton," *SIGACT News*, No. 13, December 1971, 22–24.

[17] Ronald L. Rivest, "On the worst-case behaviour of string-searching algorithms," *SIAM Journal on Computing* **6**, December 1977, 669–674.

[18] R.K.Shyamasundar, "A simple string-matching algorithm," Technical report, National Centre for Software Development and Computing Techniques, Tata Institute of Fundamental Research, Bombay, India, 1976.

[19] Ken Thompson, "Regular expression search algorithm," *Communications of the ACM* **11**, June 1968, 419–422.

[20] Uzi Vishkin, "Optimal parallel pattern-matching algorithms," *Automata, Languages and Programming,* Twelfth Colloquium, Springer-Verlag, 1985.