6-23-1987

Matching Multiple Patterns From Right to Left

Samuel W. Bent
Dartmouth College

M A. Sridhar
University of South Carolina

MATCHING MULTIPLE PATTERNS FROM RIGHT TO LEFT

Samuel W. Bent

Technical Report PCS-TR90-143

Matching multiple patterns from right to left

Samuel W. Bent
Department of Mathematics and Computer Science
Dartmouth College
Hanover, NH 03755

M. A. Sridhar
Department of Computer Science
University of South Carolina
Columbia, SC 29208

June 23, 1987

Abstract

We address the problem of of matching multiple pattern strings against a text string. Just as the Aho-Corasick algorithm generalizes the Knuth-Morris-Pratt single-pattern algorithm to handle multiple patterns, we exhibit two generalizations of the Boyer-Moore algorithm to handle multiple patterns. In order to obtain worst-case time bounds better than quadratic, our algorithms remember some of the previous history of the matching. The first algorithm remembers at most $1 + \log_4 D$ previous matches, and runs in time $O(N \log D)$, where D is the length of the longest pattern. The second algorithm provides a time-space tradeoff: given an integer $k \geq 2$, it remembers at most $t/(k-1)$ previous nonperiodic matches (where t is the number of patterns), and runs in time $O(kN \log D)$.

Matching Multiple Patterns from Right to Left

Samuel W. Bent
Department of Mathematics and Computer Science
Dartmouth College
Hanover, NH 03755

M. A. Sridhar
Department of Computer Science
University of South Carolina
Columbia, SC 29208

1. Introduction.

The problem addressed in this paper belongs to the well-known family of *pattern-matching* problems. The general statement of such a problem is, "Given some finite set of *patterns* $p_1, \ldots, p_t$, and a *text* T, find occurrences of the patterns in the text." The text is usually a long string over some fixed alphabet Σ, and the patterns are often regular expressions over Σ. Special cases of the general problem of particular practical and theoretical interest include restricting the patterns to be fixed strings (regular expressions involving only concatenation), restricting to only one pattern ($t = 1$), asking for the first occurrence, or perhaps some combination of these restrictions. In this paper, we consider the problem of finding the first occurrence of fixed-string patterns, but we allow multiple patterns.

Of all the work on pattern-matching, we call attention to three algorithms in particular. For the single fixed-string case, Knuth, Morris, and Pratt published an algorithm that essentially simulates a finite-state machine accepting the single string p_1 [5]. The states of the machine correspond to the prefixes of p_1, with the start state corresponding to the empty prefix, and the accepting state corresponding to all of p_1. Each step of the algorithm reads a text character. If the character matches the next character in the pattern (extending the current prefix), the algorithm *succeeds*, advancing to the next state and next input character. Otherwise it *fails*, retreating to the most recent state whose corresponding prefix is a suffix of the current state and rereading the current text character. This process attempts to find an occurrence of the pattern that overlaps the already-matched prefix.

A single text character can cause many failures, but eventually progress is made since failures from the start state (empty prefix) always advance the text. The state transition diagram looks like a long line of states, augmented by failure transitions leading right to left, as in Figure 1.

Aho and Corasick devised a generalization of the Knuth-Morris-Pratt algorithm for the multiple fixed-string case [1]. Their algorithm again has states corresponding to prefixes of the patterns. Two (or more) patterns sharing a common prefix also share a common state, so that there is exactly one state for each distinct pattern prefix. A state may have

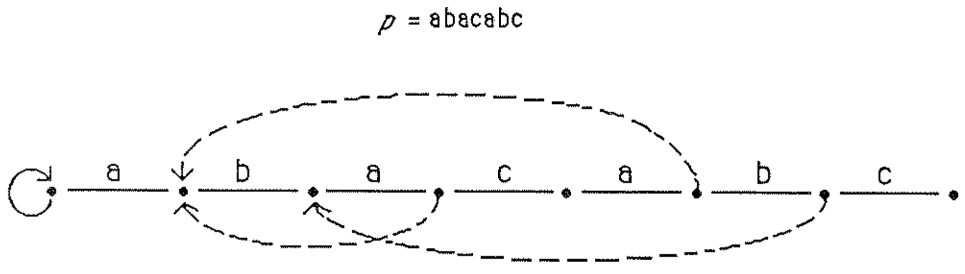

Figure 1.
Pattern matching machine.

more than one successor if its prefix is shared among patterns that disagree in the next character. Upon reading a text character, the algorithm succeeds if it matches one of the possible extensions of the current prefix, and fails otherwise. If it fails, it retreats to the state corresponding to the longest prefix (among all prefixes of any pattern) that is a proper suffix of the current state, again looking for a pattern occurrence that overlaps the matched text. The state transition diagram looks like a tree fanning out from left to right, augmented by failure transitions leading right to left (but not necessarily from descendant to ancestor), as in Figure 2.

Back in the single pattern world, Boyer and Moore invented an ingenious algorithm that matches the pattern from right to left (unlike the two algorithms above) [2]. As long as the current text character matches the pattern character, the fingers on each string move left. When a mismatch happens, the pattern is shifted to the right according to one of two heuristics. The *occurrence* heuristic suggests that the pattern be shifted so that the mismatching text character lines up under the rightmost occurrence of this character in the pattern, avoiding an obvious mismatch in that one position. However, this may not actually shift the pattern to the right if matching has already proceeded to the left of the rightmost occurrence. So the *match* heuristic suggests that the pattern be shifted so that a substring of the pattern matching the current successful suffix is lined up over the text that matched the suffix, again avoiding an obvious mismatch with respect to the successful suffix. The Boyer-Moore algorithm simply chooses the greater of the two shifts.

All three of these algorithms are remarkably efficient. Both Knuth-Morris-Pratt and Aho-Corasick make at most $2n$ queries on a text of length n. The proof is a one-liner:

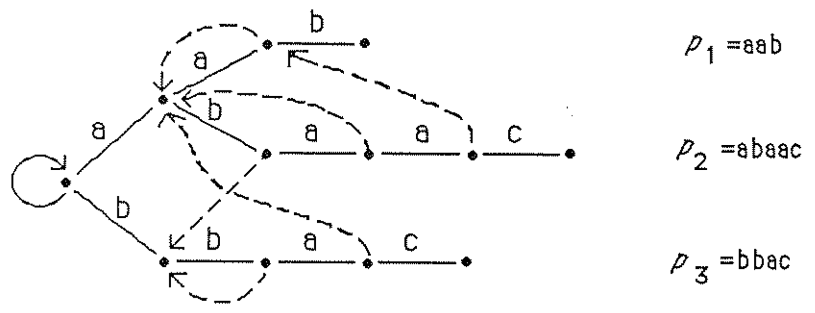

Figure 2.
Left-to-right pattern tree.

Charge a successful query to the queried text character, and charge a failure to the most recent character that succeded out of the failed-to state; each text character picks up a charge of at most 2.

The Boyer-Moore algorithm makes at most $6n$ queries, but the best known proof uses a much more sophisticated charging argument, and is very difficult [4]. Intuition suggests, however, that the Boyer-Moore algorithm is much more efficient on the average, since it is likely to make long shifts after reading only a few characters, either due to text characters that don't appear at all in the pattern or due to suffixes in the pattern that never recur. Experiments bear out this intuition, and the algorithm has been adopted in text editors and other real-life applications of string matching.

These admittedly incomplete descriptions of three well-known algorithms should serve to give the reader a picture of part of the pattern-matching landscape. Figure 3 shows the algorithms classified according to two attributes, namely whether they handle one or many patterns, and whether they scan the patterns left-to-right or right-to-left. The Boyer-Moore algorithm, by scanning right-to-left, improves over the Knuth-Morris-Pratt algorithm in practice without sacrificing much in the theoretical worst case. The research described herein explores the fourth cell of the figure, attempting to do for Aho-Corasick what Boyer and Moore did for Knuth-Morris-Pratt.

It's not hard to imagine an algorithm of the sort just alluded to. Such an algorithm was reported by Commentz-Walter [3], and can be briefly described as follows. Imagine a finite-state machine with one state for each distinct suffix of one (or more) of the patterns. The transition diagram might look like a tree fanning out to the left, as in Figure 4.

	Left-to-right	Right-to-left
One pattern	Knuth-Morris-Pratt	Boyer-Moore
Multiple patterns	Aho-Corasick	?

Figure 3.
String matching algorithms.

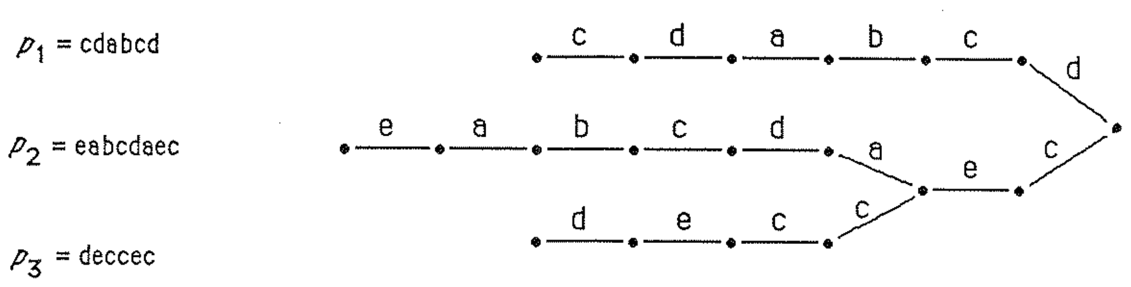

Figure 4.
Right-to-left pattern tree.

Initially, align the tree over the text so that the shallowest leaf lies just to the left of the first character of the text. (In general, always think of aligning the tree so that the states lie in between text characters). Now start at the right, matching from right-to-left in the obvious way. If there is a mismatch, shift the pattern tree right so that either:

a) the mismatched text character lies under at least one edge with matching label (occurrence heuristic), or
b) some deeper occurrence of the successfully matched pattern suffix lies over the successfully matched text (match heuristic), or
c) the shallowest leaf lies where the root used to lie.

The algorithm should choose the longer shift of (a) and (b), but in no event longer than (c). The last rule is necessary to avoid entirely skipping over an occurrence of the shortest pattern.

For many "typical" patterns and texts, this algorithm performs quite well, but unfortunately the worst case is horrible. If the two patterns are $p_1 = \mathtt{ba}^m$ and $p_2 = \mathtt{b}$, and the text is $\mathtt{a}^n$, the algorithm will look at $m+1$ characters before finding a mismatch, shift right by 1 according to rule (c), and repeat, leading to $O(mn)$ queries. The problem is that the work spent matching most of the long pattern is thrown away, because the shift was governed by the short pattern.

Our goal is to overcome this problem. The method we use is to furnish the algorithm with memory about previous matches. In the above example, for instance, if the algorithm remembers that it matched m a's and only shifted to look for p_2, it can abort the next match immediately upon reading an a. Neither p_1 nor p_2 can appear.

Remembering the entire history of previous matches is, of course, too unwieldy. We propose two strategies for remembering matches, both of which forget about a given match when the tree root is shifted far enough away from where the match occurred. The simple strategy remembers at most $1 + \log_4 D$ matches at any time, and makes $O(N \log D)$ queries of text characters. The more sophisticated strategy exibits a time-space tradeoff; it remembers at most $t/(k-1)$ nonperiodic matches while making $O(kN \log D)$ queries. Here $k \geq 2$ is an arbitrary parameter, t is the number of patterns, N is the length of the text, and D is the length of the longest pattern.

In short, both strategies make a linear number of queries for patterns of bounded length. The simple strategy needs memory related to the length of the patterns, while the second needs memory related to the number of patterns.

2. Preliminaries.

Given a set of patterns $p_1, \ldots, p_t$, form a tree P whose vertices are in 1-1 correspondence with the suffixes of the patterns. Denote by $word(v)$ the suffix corresponding to node v. If $word(v)$ is not empty, then there is a unique vertex w with $word(v) = \mathtt{a}\,word(w)$; in this case let w be the parent of v, and label the edge between v and w by a. Thus reading the labels on the edges along the path from v to the root spells $word(v)$.

Denote the depth of v by $d(v)$; this is the distance from v to the root, or equivalently, $|word(v)|$. If v is a descendant of w, let $string(v, w)$ denote the string of labels along the path from v to w. Two strings *agree* if one is a suffix of the other. We say that z *occurs at* w if z agrees with $string(u, w)$ for some leaf descendant u of w. In other words, z occurs at w whenever matching z right-to-left from w either succeeds in matching all of z or hits a leaf. We will allow z to contain "don't care" characters (#), that match any pattern character.

A *configuration* is a sequence of pairs

$$C = ((v_1, d_1), \ldots, (v_r, d_r))$$

where each v_i is a node in the tree, each $d_i \geq 0$, and $d_1 = \infty$. This configuration represents a piece of recent history, in which the algorithm matched text characters back to v_1, shifted the tree $d_2 + d(v_2)$ characters to the right (possibly with intervening matches), matched text characters back to v_2, shifted right $d_3 + d(v_3)$ positions, and so forth. The algorithm thus knows that the text matches the *characteristic string of* C,

$$char(C) = word(v_1) \cdot \#^{d_2} \cdot word(v_2) \cdots \#^{d_r} \cdot word(v_r)$$

ending at the text position where the last remembered match v_r began.

The general step of our algorithm is to align the pattern tree over the text, to match right-to-left governed by the current configuration until failure or abortion, to replace the configuration by a new one, and finally to shift the tree to the right. Shifting is the easiest to explain. If C is the new configuration, we could shift by

$$s_m(C) = \min\{d(v) \mid d(v) \geq 1 \text{ and } char(C) \text{ occurs at } v\},$$

looking for the next deeper occurrence of $char(C)$ (match heuristic). Or we could shift by

$$s_o(\mathtt{a}) = \min\{d(v) \mid d(v) \geq 1 \text{ and some edge out of } (v) \text{ is labelled } \mathtt{a}\},$$

where a is the mismatching text character, looking for an occurrence of a (occurrence heuristic). Our algorithm picks the larger of these two shifts, but in no event larger than $s_{\min}$, the length of the shortest pattern.

Having shifted according to configuration C, it behooves us not to waste time exploring paths in the tree that don't lead to occurrences of $char(C)$. Define

$$goal(C, j, D) = \{v \mid d(v) = D, \text{ and } char((v_1, d_1), \ldots, (v_j, d_j)) \text{ occurs at } v\}$$

to be the set of nodes at depth D at which a prefix of configuration C occurs. When the algorithm aligns the tree D positions to the right of the text known to match C, then as it matches right-to-left, if it reaches a node whose descendants don't include a node in $goal(C, r, D)$, it can abort the match — there's no way to agree with the remembered matches. On the other hand, if it reaches a node in $goal(C, r, D)$, it knows that the next portion of the text matches $word(v_r)$, so it can skip immediately to the descendant w of the current node v with $string(w, v)$ agreeing with $word(v_r)$. Such a descendant exists, else we would have already aborted. At this point, we say that the current match *closes* the previous match that ended at v_r. The algorithm can now resume matching at node w, using $goal(C, r-1, D + d_r + d(v_r))$ in the criterion for abortion.

When a match either fails, because no edge leaving the current node v matches the text, or aborts, as described above, it's time to update the configuration and to shift. In general, the algorithm may have skipped over $r - j$ of the remembered matches in C, so that v occurs between $word(v_j)$ and $word(v_{j+1})$ in $char(C)$. Initially, the algorithm appends new match v to the appropriate prefix of C, forming

$$C' = ((v_1, d_1), \ldots, (v_j, d_j), (v, d)),$$

where the last displacement d is the distance remaining between v and its descendants in $goal(C, j, \cdot)$. Next, it computes the distance to shift, as described above. Finally, it applies the *memory function*, deciding which of the matches in the configuration to retain for the next round of matching. The choice may depend both on the current configuration and on the shift distance. For example, if the algorithm shifts the tree far away from the text that matches $word(v_1)$, the memory function may decide to drop (v_1, d_1) from the configuration since no match can extend back far enough to need this bit of memory.

Figure 5 contains a more precise encoding of the ideas expressed above. It leaves the memory function unspecified. Our two algorithms differ only in the choice of memory function.

Variables:

r, the current text position of the root, initially $s_{\min}$
v, the current tree node, initially the root
$C = ((v_1, d_1), \ldots, (v_r, d_r))$, the current configuration, initally empty
j, the current goal prefix number, initially 0
D, the current goal depth, initially ∞
d, the distance to goal prefix, initially $s_{\min}$

Figure 5a.
Variables for the algorithm.

```
repeat
    let a = $text[r - d(v)]$ be the current text character
    if $v$ has a child $w$ with edge label a
    then { match can continue }
        if $w$ has a descendant in $goal(C, j, D)$
        then
            MoveLeft($w$)
        else { abort the match }
            Shift($w$)
    else { match failed }
        Shift($v$)
until no more text
```

Figure 5b.

The main loop.

```
procedure MoveLeft($w$)
    $d \leftarrow d - 1$
    if $d = 0$
    then { skip over remembered match }
        let $u$ be the descendant of $w$ such that $word(v_j)$ agrees with $string(u, w)$
        if some ancestor of $u$ terminates a pattern $p$
        then announce an occurrence of $p$ and halt
        $D \leftarrow D + d_j + d(v_j)$;    $d \leftarrow d_j$;    $j \leftarrow j - 1$
        $v \leftarrow u$
    else { match one character }
        $v \leftarrow w$
        if $w$ terminates a pattern $p$
        then announce an occurrence of $p$ and halt
end MoveLeft
```

Figure 5c.

Successful moves.

3. The Simple Algorithm.

A *match* consists of two parts, a tree node v and a character a. We write $m = (v, \mathtt{a})$ when the algorithm matches successfully from the root back to node v, but fails or aborts upon reading text character a. By abusing the notation, we will let m denote the match, or the successful portion of the match $word(v)$, or the length of the successful portion of the

```
procedure Shift($w$)
    append $(w, d)$ to the current configuration $C$
    compute $s = shift(C, \mathrm{a})$, as described above
    $C' \leftarrow \emptyset$;     $D \leftarrow s$
    for $i \leftarrow 1$ to $r + 1$ do
        if $memory(C, i, s)$ is true
        then { append $i$-th match to new configuration }
            append $(v_i, d_i)$ to $C'$
        else { forget $i$-th match and adjust displacement of $(i + 1)$-st match }
            $d_{i+1} \leftarrow d_{i+1} + d_i + d(v_i)$
            { by convention, let $d_{r+2}$ refer to $D$ }
    end for loop
$C \leftarrow C'$;     $\tau \leftarrow \tau + s$
end Shift
```

Figure 5d.
Unsuccessful moves.

match $d(v)$.

If match m began at text position r, define $R(m)$, the right neighborhood of match m, to be the range of text positions $r + 1$ to $r + m/4$. Our algorithms will never remember match m if the tree root has advanced outside of $R(m)$.

Define a node w to be *critical for match m* if w has an ancestor x such that

(i) $1 \leq d(x) \leq m/4$
(ii) $string(w, x)$ agrees with m
(iii) $d(w) \geq m/2$.

This says that some later match starting in $R(m)$ could overlap significantly with m. If this were to happen, we could waste time rereading text characters under m. This motivates the following memory function.

Memory Function. For a match m with right end at r, remember m if and only if

(a) the tree position τ lies in $R(m)$, and
(b) m has a critical node.

A match that is closed (skipped over by a later match while still being remembered) can be considered paid for by the closing match. The remaining matches, which we call *open*, can be accounted for by the following series of lemmas and definitions.

Lemma 3.1. *Suppose matches m_1 and m_2 are open. If match m_2 begins in $R(m_1)$, then $m_2 < m_1/2$.*

Definition. If m is open, partition the subsequent matches that begin in $R(m)$ by setting

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

Lemma 3.2. *If $1 \leq k \leq \log D$, then $|S_k(m)| \leq 2^{k+1}$. And if $k > \log D$, then $S_k(m) = \emptyset$.*

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 $k \leq \log D$.

Now suppose that $1 \leq k \leq \log D$, and that for some open match m, 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

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

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

We now account for the time spent matching open matches using the following charging argument. Charge to the first open match m all subsequent matches beginning in $R(m)$. Remove all the matches so dealt with, and repeat the process with the first remaining match. Continue until matches have been dealt with.

By the lemmas, an open match m accumulates at most $2m \log D$ units of charge, and any two matches that receive charge must start fairly far apart from each other. Also, if a sequence $m_1, \ldots, m_l$ of matches is remembered at any time, then their lengths must decrease by factors of 4. As a result, we obtain good time and space bounds.

Theorem 3.3. *The number $T(N)$ of text characters queried in a text string of length N is bounded by*

$$T(N) \leq (4N + D)(2 \log D + 1) = O(N \log D).$$

Furthermore, the algorithm remembers at most $1 + \log_4 D$ matches at any one time.

4. The Sophisticated Algorithm.

The simple algorithm requires memory related to the length of the longest pattern. Naturally, we'd like to do away with this dependence, although it seems inevitable that the memory will be related to the number of patterns. As usual in pattern-matching, most of the trouble comes from periodic strings, and we are unable to eliminate the problems of periodicity from our algorithm.

Definition. The (fundamental) period of string w, denoted $p(w)$, is the smallest integer p such that $w\#^p$ agrees with ww. We say w is *periodic* if $p(w) \leq w/4$. A match $m = (v, \mathrm{a})$ is *periodic* if $word(v)$ is periodic.

We conjecture that any memory function that doesn't remember $t - 1$ matches while searching for t patterns must query $\omega(N)$ text characters, for some collection of t patterns and some text. In other words, you must remember something about all but one pattern in order to avoid extravagant reëxamination of the text. All our examples involve periodic matches, so we can hope to use less memory for nonperiodic matches.

Analagous to our previous definition, define $R(m)$, the right neighborhood of match m at position r, to be the range of text positions $r + 1$ through $r + m/16$, and define $L(m)$, the left neighborhood, to be positions $r - 31m/16$ through $r - m$. Thus $L(m)$, m, and $R(m)$ together span $2m$ text positions.

We also define an analog to a critical node, to capture the notion of a potential reëxamination of text. This definition is much more complicated, and involves the current configuration to a much greater degree. Whereas in the simple algorithm a critical node for m corresponds to a potential future reëxamination of text, the definition of critical path below corresponds to a possible previous reëxamination.

Definition. A path P from leaf u to node w is *critical for match m_i in configuration* $C = ((v_1, d_1), \ldots, (v_r, d_r))$ if all the following conditions are satisfied:

(C1) $word(w)$ is not periodic

(C2) $d(w) \geq m/2$

(C3) $word(w)$ is a proper prefix of $word(w')$, for some node w' along the path of m_i with $1 \leq d(w') - d(w) \leq m_i/16$

(C4) for $1 \leq j < i$, if there is a node u_j on P at depth $d(w) + (m_i - d(w')) + d_i + \sum_{j+1 \leq l < i} (m_l + d_l)$, then $string(u, u_j)$ agrees with m_j.

Given such a critical path from u to w with node w' in m_i, call the quantity $d(w) - d(w')$ the *displacement* of the path. Figure 6 illustrates a path (u, w) that is critical for m_2 in a configuration containing matches m_1 and m_2.

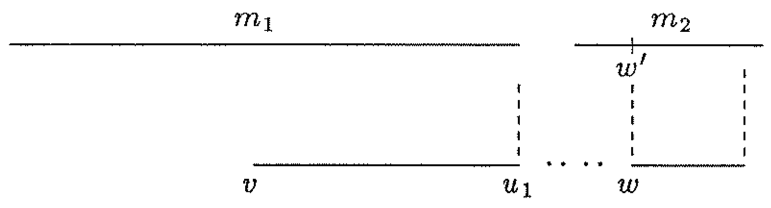

Figure 6.
(u, w) is critical for m_2.

Our second algorithm makes a time-space tradeoff, based on the parameter $k \geq 2$ that appears in the next definition.

Definition. A *k-mesh for match m_i in configuration C* is a set of $k - 1$ critical paths for m_i with distinct displacements.

Roughly speaking, a match m has a k-mesh if k previous matches could have read the same text as m. Such a match is a candidate for remembering; this leads to our second memory function.

Memory Function. Remember m in C if and only if the tree position τ lies in $R(m)$, and either

a) m is periodic, or
b) m has a k-mesh in C.

We now proceed to establish a time bound for this algorithm. 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 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 8 depicts the relationship between the matches.

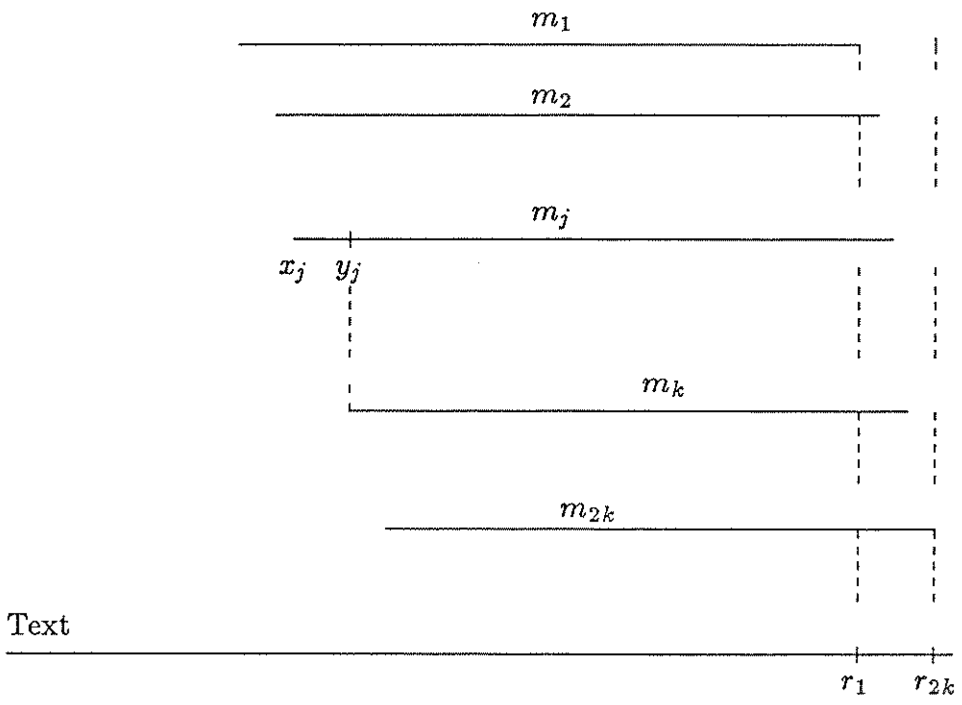

Figure 8.
A collection of overlapping matches.

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 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

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

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 $u \geq \frac{7}{16} m_k$ 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 $r_{2k} - r_1 \leq \frac{1}{16} m_l$ 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 $k \leq i \leq 2k-2$, 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 $m_i \geq \frac{1}{2} m_{2k-1}$, 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 $r_{2k-1} - r_i \leq \frac{1}{16}m_i$ so that $d(y_i) \geq \frac{16}{17}m_{2k-1} > \frac{1}{2}m_{2k-1}$. 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

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

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

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

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

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

Therefore,

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

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

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

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}.)

$\square$

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 $r' - r \leq \frac{1}{2}m'$. 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 $i \geq 0$, denote

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

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 $S_i(m)$,

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

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.

$\square$

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,

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

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

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

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 $S_i(m)$,

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

□

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 pointed out earlier, 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

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

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

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

and adding all these equations,

$$r_l - r_1 \geq \frac{1}{2} \sum_{2 \leq j \leq l} m_j$$

or

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

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

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

because $m_1 \le \frac{16}{31}m$ 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

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

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

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

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

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

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,

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

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 $i \geq \log m - 1$ (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 $i \geq \log D - 1$.

Now, for some $1 \leq i \leq \log D$, 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

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

so that, summing over all j,

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

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

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

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

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

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

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

Since each match now carries a charge,

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

□

We will now establish a bound on the number of nonperiodic matches that the algorithm remembers. To this end, 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.

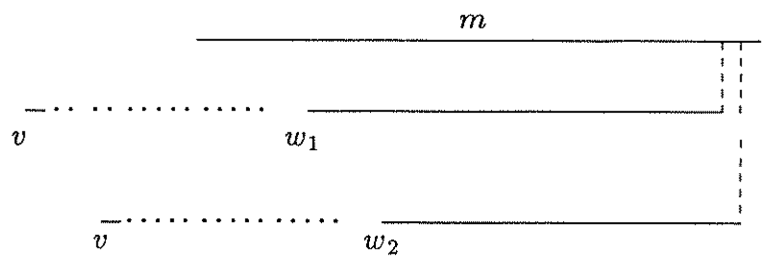

Figure 9.
Paths in a k-mesh.

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 9).

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. Discussion.

The folklore of pattern matching says that most of the theoretical difficulties are caused by patterns that rarely occur in practice, periodic patterns for example. Despite the rather formidible form of our second algorithm, and the extravagantly large constants that come out of its proof, the algorithm will never need to use its memory, nor will it come close to the theoretical bounds, unless the patterns are

(a) periodic,
(b) overlap each other to a high degree, or
(c) lengthening geometrically.

Examples that cause the algorithm to remember even two matches are difficult to construct.

We have omitted any discussion of the preprocessing needed to set up the memory and shift functions. Suffice it to say that a suitable adaptation of the pattern-matching algorithm itself can easily discover the required information. See [6] for details. In the worst case, this information could require a huge amount of space; but again, only if the patterns are pathologically bad as described above.

In summary, allowing multiple patterns introduces substantial difficulties over the Boyer-Moore algorithm, but they can mostly be dealt with satisfactorily (if not as spectacularly as Boyer-Moore). The algorithm behaves well in practice, as long as one avoids pathological patterns.

6. References.

1. Alfred V. Aho and Margaret J. Corasick, "Efficient string matching: an aid to bibliographic search". *CACM* **18** (1975), 333–340.
2. Robert S. Boyer and J. Strother Moore, "A fast string searching algorithm". *CACM* **20** (1977), 762–772.
3. Beate Commentz-Walter, "A string-matching algorithm fast on the average". *Proceedings of the Sixth Colloquium on Automata, Languages, and Programming* (Springer-Verlag, 1979), 118–132.
4. Leo J. Guibas and Andrew M. Odlyzko, "A new proof of the linearity of the Boyer-Moore string searching algorithm". *SIAM J. Computing* **9** (1980), 417–438.
5. Donald E. Knuth, James H. Morris, Jr., and Vaughan R. Pratt, "Fast pattern matching in strings". *SIAM J. Computing* **6** (1977), 323–350.
6. M. A. Sridhar, *Matching Multiple Patterns from Right to Left*. Ph.D. dissertation, University of Wisconsin–Madison, August 1986.