Approximate String Matching: A Simpler Faster Algorithm*

Richard Cole[†] Ramesh Hariharan[‡]

Abstract

We give two algorithms for finding all approximate matches of a pattern in a text, where the edit distance between the pattern and the matching text substring is at most k. The first algorithm, which is quite simple, runs in time $O(\frac{nk^3}{m}+n+m)$ on all patterns except *mostly periodic* strings (defined later). The second algorithm runs in time $O(\frac{nk^4}{m}+n+m)$ on mostly periodic patterns. The two classes of patterns are easily distinguished in $O(m)$ time.

1 Introduction

The *approximate string matching* problem is to find all those positions in a given text which are the left endpoints of substrings whose *edit distance* to a given pattern is at most a given number k. Here, the edit distance between two strings is the minimum number of insertions, deletions and substitutions needed to convert one string to the other. It is convenient to say that such a substring *matches* the pattern.

This problem is of significant importance, especially in the context of identifying sequences similar to a query sequence in a protein or nucleic acid database. In this case, the insertions, deletions, and substitutions need to be appropriately weighted, however. This variant of the problem is touched on only briefly in this paper for there are other issues to resolve.

Let n be the length of the text and m the length of the pattern. Then an $O(nm)$ algorithm is easy to obtain. This algorithm is a dynamic programming algorithm that finds the edit distance between every prefix of the pattern and every prefix of the text, not counting any cost for characters in the text which are to the left of the pattern (we will refer to this as the *local edit distance*). The number of text-pattern prefix pairs is

*This work was supported by NSF grants CCR-9503309 and CCR-9800085. An abstract of this work appeared in the proceedings of the *ACM-SIAM Symposium on Discrete Algorithms*, 1998.

[†]Courant Institute, New York University, cole@cs.nyu.edu.

[‡]Indian Institute of Science, Bangalore, ramesh@csa.iisc.ernet.in. This work was done in part while visiting NYU.

$O(nm)$, and each pair can be processed in constant time, provided the pairs are processed in a certain natural order. The way to think about this order is to consider an array with columns associated with text prefixes of increasing length ordered towards the right and rows associated with pattern prefixes of increasing length ordered downwards. Each entry in this array represents the local edit distance of a text-pattern prefix pair. These entries are computed in an order such that all entries in rows $1 \ldots i$ are computed before row $i+1$ is computed, and the entries within a row are processed in order from left to right.

Landau and Vishkin[LV89] obtained an $O(nk)$ algorithm for this problem. This algorithm was based on the above dynamic programming paradigm as well. However, their order of computing the array entries was a clever one. They observed that in each 45-degree top-left to bottom-right diagonal, the entries are non-decreasing downwards and that one need compute only k entries in each diagonal, namely those entries whose value is different from the value of the preceding entry in the same diagonal. They show how these entries can be computed in constant time per entry, using a suffix tree of the pattern and the text.

Some other early work on this problem is described in [LV85, LV86, LV88, GG88, GP90].

The question that then arose was whether an $O(n+m)$ time algorithm was possible, at least for the case when k is small, e.g., $O(m^\epsilon)$, for some ϵ between 0 and 1. The intuition which suggests that this would be possible is that most of the pattern must match exactly when k is small.

An algorithm with an average case performance of $O(\frac{nk \log m}{m})$ time on random strings when $k < \frac{m}{\log m + O(1)}$ was given by Chang and Lawler[CL90]. While linear (sublinear, actually) on the average, the worst case performance of this algorithm was still $\Theta(nk)$. The assumption of the text being random is a strong one as random strings do not match with very high probability, but this algorithm may work well even on somewhat less random strings.

Baeza-Yates and Navarro [BN96] gave an algorithm with a running time of $O(n)$ for the case when $mk = O(\log n)$. In addition, they obtained another algorithm whose performance in the average case is $O(n)$ for medium k/m ratios. They also report finding this algorithm to be faster than previous algorithms experimentally, especially in the case when the pattern has moderate size, the error ratio k/m is not too high, and the alphabet size is not too small.

Recently, the above question was answered positively by Sahinalp and Vishkin[SV97], who obtained an algorithm with the following performance. Their algorithm takes $O(nk^{3+\frac{1}{\log 3}}(\frac{\log^* n}{m})^{\frac{1}{\log 3}} + n + m)$ time when there is *no periodicity* anywhere in the text or the pattern. Here, no periodicity means that even very local periodicity, e.g., two repeated characters, is not allowed. When there exists any periodicity in the text, the time taken by their algorithm is $O(nk^{8+\frac{1}{\log 3}}(\frac{\log^* n}{m})^{\frac{1}{\log 3}} + n + m)$. Their algorithm uses the technique of deterministic coin tossing in order to sparsify the set of diagonals which need to be processed in the above array, and then processes only these diagonals using the Landau-Vishkin algorithm. This technique and the associated proofs of complexity and correctness, especially when there is periodicity present, are fairly involved.

Our contribution in the paper is twofold.

1. We give a very simple way of sparsifying the set of diagonals which need to be processed in the above matrix. This method is completely different from the Sahinalp-Vishkin algorithm and does not use deterministic coin tossing. All it requires is finding all occurrences of a number of aperiodic pattern substrings[1] of suitable length in the text. This immediately gives us an $O(\frac{nk^3}{m} + n + m)$ time algorithm, except when the pattern and the text are *mostly periodic*. By mostly periodic, we mean that there are $O(k)$ substrings of size k^2 each such that the portions of the text and the pattern between these substrings (or *breaks*, as we call them) are all periodic. We believe that mostly periodic is a rather strict property and mostly periodic strings would be quite rare in practice.

2. We show how to process mostly periodic texts and patterns in $O(\frac{nk^4}{m} + n + m)$ time. While processing such strings in $O(\frac{nk^6}{m}+n+m)$ and even $O(\frac{nk^5}{m}+n+m)$ is quite easy, the $O(\frac{nk^4}{m}+n+m)$ time algorithm is non-trivial. The technical difficulties we face in obtaining this algorithm include the fact that the various periodic stretches between breaks need not have the same period and that periodic stretches in the pattern and the text need not align in a match of the pattern. Of course, there cannot be too many misalignments, since only k mismatches are allowed.

Thus, this paper gives an algorithm for approximate string matching which is not only faster and simpler than the Sahinalp-Vishkin algorithm, but also helps understand what kinds of text and patterns are hard to handle for this problem and why. We conjecture that the right bound is $O(\frac{nk^3}{m} + n + m)$ even for the mostly periodic case, but have been unable to obtain an algorithm with this performance. We also believe that obtaining an algorithm which takes $o(\frac{nk^3}{m} + n + m)$ time will be hard.

The rest of this paper is organized as follows. Section 2 gives some necessary definitions. Section 3 gives an overview of our algorithm and Section 4 describes our sparsification algorithm and how it gives an $O(\frac{nk^3}{m}+n+m)$ time algorithm for the case when either the text or the pattern is not mostly periodic. One of the tools used in this algorithm is a simple modification of the Landau-Vishkin algorithm [LV89] and is described in Section 5 (this modification is also used in [SV97]). Section 6 describes how to process the text when the pattern is mostly periodic but the text is not. Section 7 describes the first attempt at handling mostly periodic patterns and texts and obtains an $O(\frac{nk^6}{m}+n+m)$ time algorithm. Section 8 gives our more sophisticated scheme to handle such patterns and texts in $O(\frac{nk^4}{m} + n + m)$ time. Section 9 gives some intuition regarding the difficulties to

[1] Aperiodic, here and now on, refers to the usual notion of periodicity, i.e., the largest proper suffix of the substring which is also a prefix has length less than half that of the substring. This notion of aperiodicity is much weaker than that required by the Sahinalp-Vishkin algorithm.

be overcome in obtaining an $O(\frac{nk^3}{m} + n + m)$ time algorithm. Section 10 briefly discusses the weighted version of the problem.

2 Definitions and Preliminaries

We assume that suffix trees for the pattern and the text can be constructed in linear time [CR94, F98].

We will assume that m, the pattern length, is at least $5k^3$. The Landau-Vishkin $O(nk + m) = O(\frac{nk^4}{m} + n + m)$ time algorithm is used for shorter patterns.

We will also assume that the text has length $2m - 2k$ and the pattern has length m. If the text is longer then it is partitioned into pieces of length $2m - 2k$, with adjacent pieces overlapping in $m + k - 1$ characters. The reason this suffices is that any substring of the text which matches the pattern has length in the range $[m - k, m + k]$. Thus, all matches of the pattern are completely contained within some piece.

Periodicity. The *period length* of a string is defined to be the smallest i, such that two instances of the string, one shifted i to the right of the other, match wherever they overlap. A string is said to be *aperiodic* if its period length is more than half the string length, and *periodic* otherwise. A string is *cyclic* if it can be written as u^i, $i \geq 2$. A periodic string can be written as $u^i v$, where u is acyclic, $i \geq 2$, and v is a prefix of u. The following properties of periodic strings are well known (see [CR94]) and will be used implicitly throughout this paper.

1. The period length of a string can be determined in linear time, and so can its lexicographically least cyclic shift.

2. An acyclic string is not identical to any of its cyclic shifts. Therefore, a string s cannot be written as $u^i v$ and as $u'^i v'$, where $u \neq u'$, u, u' are acyclic, v is a prefix of u, v' is a prefix of u', and $|u| + |u'| \leq |s|$.

3. If u is acyclic, then every cyclic shift of u is acyclic as well.

4. If string s has period length u but string sa (a is a single character) does not have period length u, then any suffix of sa of length at least $2u$ is aperiodic.

Mostly Periodic Strings. The pattern is said to be *mostly periodic* if it contains at most $2k - 1$ disjoint aperiodic substrings of length k^2. The text is said to be *mostly periodic* if it contains at most $10k - 2$ disjoint aperiodic substrings of length k^2.

Lemma 2.1 *It can be determined in $O(m)$ time if the text and the pattern are mostly periodic. Further, if the pattern is not mostly periodic then $2k$ disjoint aperiodic substrings of the pattern of length k^2 each can be found in $O(m)$ time. Similarly, if the text is not mostly periodic then $10k - 1$ disjoint aperiodic substrings of the text of length k^2 each can be found in $O(m)$ time.*

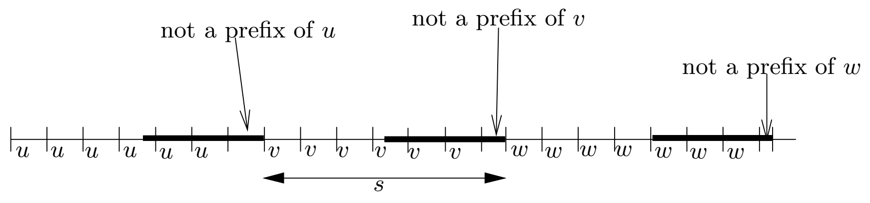

Figure 1: A mostly periodic string. Thick regions correspond to aperiodic substrings of length k^2. u, v, w are all at most $k^2/2$ in length.

Proof: We consider only the pattern here. The text is processed similarly.

We process the pattern from left to right, performing various rounds. In each round, a new aperiodic length k^2 substring disjoint from all previously found substrings is determined. The stretches between any two consecutive substrings determined above will have period length at most $k^2/2$. Finally, if the collection of aperiodic strings constructed above has size less than $2k$ then the pattern is mostly periodic. The time taken by all rounds together will be $O(m)$.

A round is performed as follows. The portion of the pattern to the right of the last aperiodic length k^2 substring determined earlier is considered in this round (if this is the first round, then the pattern is considered starting from its leftmost character). The shortest prefix s of this portion (see Fig.1) of the pattern with the following properties is determined: $|s| \geq k^2$ and the length k^2 suffix of s is aperiodic. This computation will take $O(|s|)$ time and is described in the next paragraph. The length k^2 suffix of s is added to the collection of disjoint aperiodic strings being constructed. The total time taken over all rounds is clearly $O(m)$.

It remains to show how s is determined in $O(|s|)$ time. Let s' denote the length k^2 prefix of the above portion of the pattern. First, the period length δ of s' is determined in $O(k^2)$ time (see property 1 above). If $\delta > k^2/2$, then $s = s'$. Otherwise, the leftmost pattern character $p[i]$ to the right of s' with the property that $p[i] \neq p[i-\delta]$ is determined; s is the extension of s' up to and including $p[i]$. The time taken above is clearly $O(|s|)$. By property 4, the length k^2 suffix of s is aperiodic, as required. $\Box$

Canonical Periods. Consider a periodic string u^iv, where v is a prefix of u, $i \geq 2$, and u is not cyclic. Note that all cyclic shifts of u are distinct since u is not cyclic. The *canonical period* of u^iv is the string y which is the lexicographically smallest circular shift of u. Note that u^iv can be written in a unique way as xy^jz, where x is a suffix of y and z is a prefix of y. Also note that, given u, y can be determined in $O(n)$ time (see property 1 above) and that y is acyclic as well (see property 3 above).

The following lemma about edit distance between periodic strings will be instrumental in the design of our algorithm.

Lemma 2.2 *Consider two strings u^i and xw^jy of the same length, where x is a suffix of w, y a prefix of w, and u, w are canonical periods of their respective strings. If $|u^i| = |xw^jy| \geq k^3 + k^2$ and $|u|, |w| \leq \frac{k^2}{2}$, then the edit distance between these two strings is at least $k + 1$, unless $u = w$.*

Proof: Suppose $u \neq w$. Note that $i \geq 2(k + 1)$. There are two cases.

First, suppose $|u| = |w|$, u and w cannot be cyclic shifts of each other as they are both canonical periods. It follows that each occurrence of u must incur at least one mismatch. The lemma follows in this case.

Second, suppose $|u| \neq |w|$. Partition u^i into disjoint substrings of length $|u| + |w|$. There must be at least $k + 1$ such substrings. In addition, there must be at least one insertion/deletion/substitution in each such substring by property 2 above. The lemma follows. $\square$

3 Overview

The algorithm first determines if the pattern is mostly periodic. More specifically, it determines whether there is a collection of $2k$ disjoint aperiodic length k^2 substrings in the pattern. Two cases are considered next, depending upon whether or not such a collection of substrings exists.

The Sparse Case. If such a collection exists then Section 4 describes a *sparsification* procedure that determines $O(\frac{m}{k^2})$ windows in the text, each of size k, which are the only locations where pattern matches can possibly begin. The matches starting in these windows are then found in $O(m)$ time by a simple modification of the Landau-Vishkin algorithm described in Section 5.

The Mostly Periodic Case. On the other hand, if no such collection exists, then the pattern is mostly periodic. In this case, in Section 6, we show that all matches of the pattern in the text must occur in a portion of the text which is mostly periodic. We also show how this portion can be found in $O(m)$ time. In Section 7, we show how to find all occurrences of mostly periodic patterns in mostly periodic texts in $O(k^6)$ time. This is improved to $O(k^4)$ time in Section 8. This leads to an overall complexity of $O(\frac{nk^4}{m}+n+m)$ for this case.

4 Sparsification

In this section, we assume that the pattern has $2k$ disjoint aperiodic length k^2 substrings and that these substrings have been found. We call these substrings *breaks*. We show how to determine $O(\frac{m}{k^2})$ text windows, each of size k, in which potential matches of the pattern can begin. This will take $O(m)$ time.

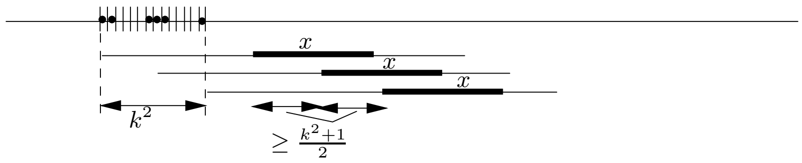

Figure 2: Windows which are marked for x. Filled circles indicate marks. Each instance of the pattern shown has x matching exactly.

First, we find all exact occurrences of each of these $2k$ breaks in the text. Note that these breaks have equal length. The time taken for this procedure is $O(m)$ using a standard multiple pattern matching algorithm [AC75].

Next, we partition the text into disjoint pieces of size k^2. Consider a particular piece $t[i \dots j]$. We partition it into disjoint windows of size k each. We will show how to determine at most 12 windows such that any pattern match beginning in this piece must begin in one of these windows.

Note that at least k of the breaks must match exactly in any match of the pattern. Consider one particular break x. As x is aperiodic, any two occurrences of x in the text are at least distance $\frac{k^2+1}{2}$ apart. It is not hard to show that any pattern match beginning in $t[i \dots j]$ with x matching exactly must begin in one of 6 size k windows in $t[i \dots j]$ (see Fig.2). We can represent this fact by putting a mark for x on each of these windows. For a match to begin in a particular window, it must receive a mark from each of at least k breaks. Since each break marks at most six windows, there are at most $12k$ marks in all and therefore, at most 12 windows in which matches can begin.

The verification of matches beginning in these windows is described next. It shows how all pattern matches beginning in a particular text window of length l can be found in time $O(k(k+l))$. Thus all matches beginning in a particular text window of length k can be found in $O(k^2)$ time. Then the time taken over all $O(\frac{m}{k^2})$ windows will be $O(m)$.

5 The Landau-Vishkin Algorithm and Sparse Match Verification

First, we give an overview of the Landau-Vishkin algorithm. Subsequently, we show how to find all pattern matches beginning in a particular text window of length l in time $O(k(k+l))$. We assume that the suffix tree of the pattern and the text combined has been constructed and processed for Least Common Ancestor queries [SV88] so that the longest common prefix of any two suffixes in the text/pattern can be determined in $O(1)$ time.

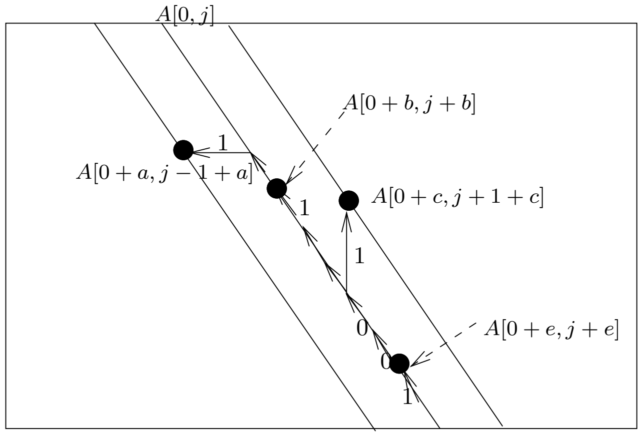

Figure 3: *Finding the bottommost point on* $A[0+*,j+*]$ *with shortest path value* $l+1$.

5.1 The Landau-Vishkin Algorithm

We review the Landau-Vishkin algorithm in this section. The classical approach to solving approximate string matching is to model it as a shortest paths problem on a graph defined on the entries of the following matrix.

Consider a matrix $A[0\dots m, 0\dots 2m-2k]$. $A[i,j]$ will be the value of the best match of $p[1\dots i]$ with any suffix of $t[1\dots j]$, for $1 \le i \le m, 1 \le j \le 2m-2k$. The 0th row and column are dummies put in for technical reasons which will become clear shortly.

The Dependency Graph. To determine the entries of A we define a *dependency graph* G with weighted edges as follows. For each $i \ge 1$ and $j \ge 1$, there is a directed edge from $A[i,j]$ to each of $A[i,j-1], A[i-1,j], A[i-1,j-1]$, with weights $1,1,y$ respectively, where y is 0 if $t[j]=p[i]$ and 1 otherwise. In addition, there is an edge from $A[i,0]$ to $A[i-1,0]$ with weight 1 and another from $A[0,j]$ to $A[0,j-1]$ with weight 0, for each $i \ge 1$ and each $j \ge 1$.

It is easy to see that the value of $A[i,j]$ is the weight of the shortest path from $A[i,j]$ to $A[0,0]$.

The Algorithm. This algorithm takes $O(k)$ time for each diagonal in A. Consider a diagonal $A[0+*,j+*]$ (here , * takes values from 1 to m). For each $l=1\dots k$, it computes the bottommost vertex on this diagonal whose shortest path has weight l. This is done for each l in sequence, each point taking constant time to compute.

Suppose the above has been done for a particular value of l for all diagonals. Consider $l+1$ now and the diagonal $A[0+*,j+*]$. The bottommost vertex with shortest path $l+1$ on this diagonal is computed in constant time as follows. Let $A[0+a,j-1+a], A[0+$

$b, j+b], A[0+c, j+1+c]$ be the bottommost vertices on their respective diagonals whose shortest paths have weight l (see Fig.3).

Consider the three points $A[0+a, j+a], A[0+b+1, j+b+1], A[0+c+1, j+c+1]$, and take the bottommost of these three points; call this bottommost point $A[0+d, j+d]$. Next, find the bottommost point $A[0+e, j+e]$ such that all the edges on the path from $A[0+e, j+e]$ to $A[0+d, j+d]$ have weight 0. $A[0+e, j+e]$ is the required point. The longest 0 weight path along a diagonal starting at any particular point on the diagonal can be found in constant time using a longest common prefix computation.

5.2 Sparse Match Verification

We show how to find all pattern matches beginning in a particular text window $t[i \dots i+l-1]$ of length l in time $O(k(k+l))$. In fact, our description will show how to find pattern matches ending in particular text window $t[i \dots i+l-1]$ of length l in time $O(k(k+l))$. Pattern matches beginning in the above window can be found using an analogous procedure (imagine reversing the pattern and the text).

Note that in order to determine pattern matches ending in the above text window, it suffices to determine the bottommost points whose shortest paths have weight k on each of the diagonals $A[0+*, i-m+*] \dots A[0+*, i-m+l-1+*]$. Let B denote the band formed by these diagonals. Let B' denote the band formed by the diagonals $A[1+*, i-m-k+*] \dots A[1+*, i-m+l-1+k+*]$. The algorithm is based on the following simple observation.

Observation: If the shortest path of a point in B has weight at most k, then this shortest path stays entirely within B'. This is true because horizontal and vertical edges have weight 1 in G. The shortest path of a point in $B' - B$ might leave B'. However, to compute shortest paths for points in B it is not necessary to compute shortest paths of points in $B' - B$ correctly; rather, it suffices to compute shortest paths using only edges in B'.

There are $O(k+l)$ diagonals in B'. Running the Landau-Vishkin procedure takes $O(k)$ time per diagonal, giving $O(k(k+l))$ time overall.

Remark. Suppose we are given two strings s_1, s_2, which are substrings of the text/pattern. Then note that the above procedure can easily be generalized to find the edit distances of s_2 with each of the $\Theta(k)$ longest suffixes of s_1, in time $O(k^2)$. Each such distance is determined correctly only if it is at most k. If it exceeds k, then the fact that it exceeds k is determined as well.

Further, note that the above procedure can also be generalized to find, for each of the $\Theta(k)$ longest suffixes of s_2, the edit distances with each of the $\Theta(k)$ longest suffixes of s_1, in time $O(k^2)$. As in the previous paragraph, each such distance is determined correctly if it is at most k; otherwise, the fact that it exceeds k is determined.

6 Text Processing for Mostly Periodic Patterns

We assume that the pattern has at most $2k-1$ breaks, i.e., disjoint substrings of length k^2, such that the stretches in between these breaks are periodic with period at most $k^2/2$. We show how to obtain a substring z of the text such that z is mostly periodic (i.e., has at most $10k-2$ breaks) and all potential matches of the pattern lie completely within z. This is done in $O(m)$ time.

Let x be the shortest text substring with its right end coinciding with the middle of the text and having $2(2k-1)+k+1=5k-1$ disjoint aperiodic substrings of length k^2. If no such x exists then x is just the first half of the text. Let y be the shortest substring beginning in the middle of the text and having $5k-1$ disjoint aperiodic substrings of length k^2. If no such y exists then y is just the second half of the text. We claim that all pattern matches must lie within $z=xy$.

Suppose a match of the pattern has its left end to the left of x. Recall that the text has length $2m-2k$. Then, this pattern occurrence must touch or overlap the boundary of x and y and therefore, it must overlap the whole of x (otherwise, more than k insertions/deletions would be required). But x has $5k-1$ disjoint aperiodic substrings of length k^2 and at most $2(2k-1)$ of them can overlap breaks in the pattern; the remaining $k+1$ (or more) aperiodic text substrings of length k^2 must incur at least one mismatch each (because an aperiodic substring of length k^2 when aligned with a periodic stretch with period length at most $k^2/2$ must incur at least one mismatch; also see Fig.1). Therefore, the pattern cannot match in the above configuration, a contradiction. Similarly, it can be shown that the pattern cannot match with its right end to the right of y.

Determining x, y: This is done in $O(m)$ time using an algorithm similar to the algorithm in Lemma 2.1.

7 Finding Matches of Mostly Periodic Patterns

In this section, we assume that both the text and the pattern are mostly periodic. Recall that there are at most $2k-1$ ($10k-2$, respectively) disjoint aperiodic length k^2 substrings in the pattern (text, respectively) such that the stretches between them are periodic with period length at most $\frac{k^2}{2}$. Recall that these substrings are called breaks.

7.1 The $O(k^6)$ Algorithm.

First, we classify all potential matches into two categories. The first category contains potential matches in which some break in the pattern or some endpoint in the pattern is within distance $2(k^3+k^2)+k^2$ from the beginning or end of some break or endpoint in the text. The remaining potential matches are in the second category.

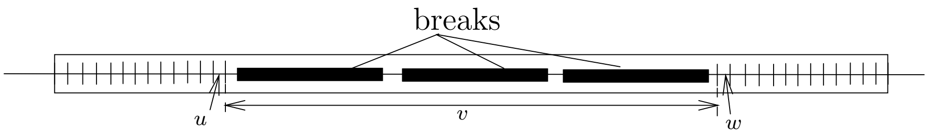

Figure 4: *An interval.*

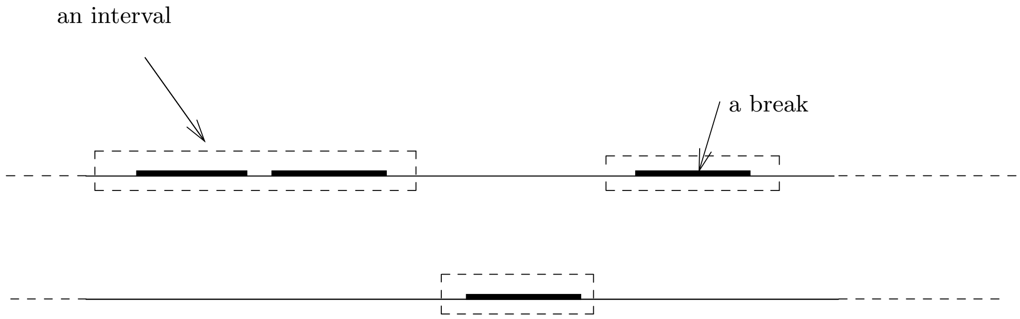

Figure 5: *Placement of a portion of the pattern in the second category.*

7.1.1 The First Category

Note that matches in the first category must begin in one of $O(k^2)$ windows, each of size $O(k^3)$. All matches in these windows can be found using the algorithm in Section 5 in $O(k^6)$ time. It remains to find matches in the second category.

7.1.2 The Second Category

Note that all potential matches in the second category also begin in one of $O(k^2)$ windows. Within each window, the order in which the various text and pattern intervals appear from left to right remains the same. The problem is that these windows could be long. Consider one such window.

Definitions. We need to form *intervals* in the text and the pattern before proceeding. First, we forms groups of breaks in the pattern and the text. A group is a maximal sequence of breaks such that the periodic stretch between neighboring breaks has length less than $2(k^3+k^2)+k^2$. An *interval* is a substring which includes all breaks in a group and extends on either side by a further amount described below. Let u denote the canonical period of the stretch to the left and let w denote the canonical period of the stretch to the right. On the left side, the interval extends to the least distance between k^3+k^2 and $k^3+k^2+\frac{k^2}{2}$ so as to have an integral number of occurrences of u (see Fig.4). On the

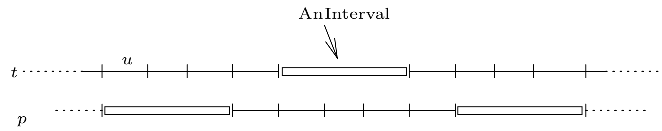

Figure 6: *Locked Intervals.*

right side, the interval extends to the least distance between k^3+k^2 and $k^3+k^2+\frac{k^2}{2}$ so as to have an integral number of occurrences of w. Note that there are at least $2(k+1)$ occurrences of the canonical period on either side. Thus, each interval can be written in canonical form as u^ivw^j, where $|u^i|, |w^j| \geq k^3+k^2$, $i, j \geq 2(k+1)$ and u, w are not cyclic. We call u the *lcp* (left canonical period) of this interval and w the *rcp* (right canonical period).

Note that there are potentially two exceptions to the rules above, namely the first and the last intervals in the pattern/text. The leftmost interval may be terminated by the left end of the pattern/text and therefore, may not satisfy $|u^i| \geq k^3+k^2$. Similarly, the rightmost interval may be terminated by the right end of the pattern/text and therefore, may not satisfy $|w^j| \geq k^3+k^2$. Prematurely terminated intervals are called *incomplete*; others are called *complete*.

The case when the pattern has only one interval, which is incomplete on both the left and the right needs to be treated as a special case. We will address this case later. Until then, assume that each interval in the pattern is complete either on the right or on the left.

Properties of Second Category Matches. Note that in all matches in this category, an interval or endpoint in the pattern (text, respectively) cannot overlap or touch an interval in the text (pattern, respectively). This is because the period length of any periodic stretch is at most $\frac{k^2}{2}$ and if such an overlap occurs then some break or endpoint in the text would be distance at most $2(k^3+k^2)+k^2$ from some break or endpoint in the pattern. Further, in all matches in this category, the endpoints of an interval will be *locked*, i.e., in alignment with the canonical periods in the overlapping periodic stretch (see Fig.6), as is proved in the following lemmas. This property enables us to process this category efficiently.

Lemma 7.1 *The pattern has a second category match only if the lcp's and rcp's of all pattern intervals (except possibly the lcp of the first interval, in case it is incomplete on the left, and the rcp of the last interval, in case it is incomplete on the right) and of all text intervals overlapping the pattern are identical (denoted by, say, u). In addition, in any second category match of the pattern, all periodic stretches in the text (pattern, respectively) overlapping intervals in the pattern (text, respectively) must have canonical period u.*

Proof: First, consider the first part of the lemma. Order the intervals involved in this match (i.e., all pattern intervals and all text intervals overlapping the pattern) from left to right in order of occurrence. We show that the rcp of one interval s_1 in this sequence is identical to the lcp of the next interval s_2 in this sequence. Further, if interval s in this sequence is either complete or in the text, we show that the lcp and rcp of s are identical. The first part of the lemma follows.

Let u denote the rcp of s_1 and w denote the lcp of s_2. Note that $|u|, |w| \leq \frac{k^2}{2}$. Further, the suffix of s_1 which is cyclic in u has length at least $k^3 + k^2$ and likewise for the prefix of s_2 which is cyclic in w.

There are two cases for showing that the rcp of s_1 is identical to the lcp of s_2. First, suppose both intervals are in the pattern. Then they are both overlapped by a periodic stretch in the text with canonical period, say, x. But by Lemma 2.2, $x = u$ and $x = w$. Therefore, $u = w$, as required. Second, suppose s_1 is in the text and s_2 is in the pattern (the case when s_1 is in the pattern and s_2 is in the text is symmetric). Then s_1 is overlapped in the pattern by a stretch with canonical period w. By Lemma 2.2, $u = w$, as required.

Next, suppose s is in the text or is complete. We show that its lcp and rcp are identical. First, note that s will be complete in either case (i.e., if the pattern overlaps the first/last text interval and this interval is incomplete, then this match is in the first category). Assume that s is a complete interval in the text (the case when s is a complete interval in the pattern is similar). Let u be the lcp of s and v the rcp. Since s is complete, the prefix of s which is cyclic in u has length at least $k^3 + k^2$ as does the suffix of s which is cyclic in v. Note that $|u|, |v| \leq \frac{k^2}{2}$. The portion of the pattern overlapping s is periodic with canonical period, say, x. Then, by Lemma 2.2, $x = v = u$, as required.

Now, consider the second part of the lemma. Consider a periodic stretch in the text having canonical period, say, w, which overlaps an interval s in the pattern. Assume that s is complete on the right (a similar proof holds for the case when s is complete on the left; any interval must be complete either on the right or on the left, by our assumption above). Note that $|u|, |w| \leq \frac{k^2}{2}$. Further, the suffix of s which is cyclic in u has length at least $k^3 + k^2$. It follows from Lemma 2.2 that $u = w$. A similar proof holds for the case of a periodic stretch with canonical period w in the pattern overlapping an interval s in the text (as stated earlier in this proof, s is necessarily complete in this case). $\Box$

Lemma 7.2 *Consider a match of the pattern in the second category. Consider any interval s involved in this match (i.e., all pattern intervals and all text intervals overlapping the pattern). If s is complete on the left and has lcp u then the portions of the text and pattern between s and the next interval (text or pattern; if there is no such interval, then consider the endpoint) to the left have suffix u and incur no cost for edits. If s is complete on the right and has rcp u then the portions of the text and pattern between s and the next interval (text or pattern; if there is no such interval, then consider the endpoint) to the right have prefix u and incur no cost for edits.*

Proof: We show that if s is complete on the left and has lcp u then the portions of the text and pattern between s and the next interval (text or pattern; if there is no such interval, then consider the endpoint) to the left have suffix u and incur no cost for edits. The other case is symmetric. We assume that s occurs in the pattern. The case when it occurs in the text is similar.

Recall from the definition of intervals that there are at least $2(k+1)$ occurrences of u at the beginning of s. Some instance of u in s amongst the rightmost $k+1$ such instances must match the text exactly. There are two cases now.

First, suppose there are no intervals to the left of s involved in the above second category match. Then the portion of the pattern to the left of (and including) the above exact match of u has the form vu^i, where v is a suffix of u. Therefore, the portion of the text which overlaps the above portion of the pattern is also periodic with canonical period u (for an exact match of u occurs within s and the portion of the text overlapping s is part of a periodic stretch with canonical period u, by Lemma 7.1). The lemma follows from the fact that u is a prefix of s.

Next, suppose there is an interval to the left of s involved in the above second category match. Let s' be the rightmost such interval. By Lemma 7.1, the rcp of s' must also be u. As before, some instance of u in s amongst the rightmost $k+1$ instances must match the pattern exactly. Similarly, some instance of u in s' amongst the leftmost $k+1$ instances must match the pattern exactly. The portions of the pattern and the text between the above two exact matches of u are both cyclic in u. The number of edit operations in these portions is at least the difference in their lengths and is at most k. This number only reduces if all these edits are transferred so that they occur at the right end of these portions. Since there are at most k such edits, all of them will now appear either within s or the portion of the text overlapping s. It follows that the portion of the text starting from the above matching instance of u in s' and extending up to (but not including) the location aligned with the starting character of s matches the pattern exactly and is cyclic in u, as required. $\Box$

Definition. For each interval of length l, define its *locked edit distance* to be the minimum over all g, $h' \leq g \leq h$, of the edit distance between this interval and u^g. Here, h is the number such that $|u|(h-1) < l+k \leq |u|h$ and h' is the number such that $|u|(h'-1) < l-k \leq |u|h'$. We compute the locked edit distance for each interval in the text and the pattern. Here, note that this distance is needed only if it is at most k. So we will only compute this distance if it is at most k. This computation takes $O(k^3)$ time as there are $O(k)$ intervals, and for each interval, the algorithm given in Section 5 will take $O(k^2)$ time (essentially, the shortest paths from certain points lying on some of at most $2k+1$ diagonals need to be determined if these paths have cost at most k).

We need special handling for incomplete intervals. Note that incomplete intervals in the text do not play a role in second category matches (i.e., if the pattern overlaps the first/last text interval and this interval is incomplete, then this match is in the first category). For the up to 2 incomplete intervals in the pattern, we need to redefine *locked edit distance* as follows.

If the rightmost interval in the pattern is incomplete on the right, its locked edit distance is the the minimum edit distance between this interval and some prefix of u^h, where, h is the number such that $|u|(h-1) < l + k \leq |u|h$. If the leftmost interval in the pattern is incomplete on the left, its locked edit distance is the the minimum edit distance between this interval and some suffix of u^h, where h is the number such that $|u|(h-1) < l+k \leq |u|h$. As before, these locked edit distances need to be computed only if they do not exceed k. This can be done in $O(k^2)$ time using the algorithm in Section 5 (see the remarks at the end of that section).

Corollary 7.3 *The edit distance between the pattern and the text for a second category match is just the sum of the locked edit distances over all pattern intervals and all text intervals overlapped by the pattern.*

Proof: Consider any second category match. By Lemma 7.1, each interval in the pattern is aligned with a periodic stretch in the text which is cyclic in u. Similarly, each interval in the text overlapped by the pattern is aligned with a periodic stretch in the pattern which is cyclic in u. From Lemma 7.2, it follows that each complete interval in the pattern (text, respectively) is aligned with a string in the text (pattern, respectively) which is cyclic in u. Further, an incomplete interval in the pattern which is complete on the right (left, respectively) is aligned with a string in the text which is periodic with period u and has suffix (prefix, respectively) u (note that if the endpoint of the text were closer, preventing a suffix of u, this would be a first category match). The corollary follows from the definition of locked edit distance. $\square$

Remark. Our aim is to determine all locations in the text where matches of the pattern begin. Consider any second category match. Recall from the last paragraph of the proof of Lemma 7.2 that edits in this match can be transferred to within pattern and text intervals (or to within portions in the text and pattern, respectively, overlapping these intervals) without increasing the edit distance. This transfer will not change the starting point of the match in the text, except when the edits transferred are to the left of the leftmost interval (pattern or text, whichever is first). Therefore, edits to the left of the leftmost interval will have to be treated differently while determining the starting points of the various matches. Next, we show how these starting points can be determined in time $O(k^4)$ plus the number of matches.

Algorithm for Second Category Matches. All locked edit distances are computed in $O(k^3)$ time as above. Recall from the beginning of this section that all second category matches begin in one of $O(k^2)$ windows. Consider one such window. The pattern occurs in this window if and only if the sum of the locked edit distances of the relevant intervals is at most k. The precise necessary and sufficient conditions for the pattern to occur starting at a particular text location i in this window are described next.

Note that fixing the window under consideration also fixes the left to right sequence of text and pattern intervals involved in matches in this window. Consider the leftmost interval (pattern or text, whichever is first). There are two cases, depending upon whether this interval is in the pattern or in the text. We consider each case in turn. In each case, we show that the total time taken to output all matches in this window is $O(k^2)$ plus the number of matches.

Case 1. First, consider the case when this interval is in the pattern. Let x be the prefix of the pattern up to and including the right end of this interval. Recall from Lemma 7.2 that x must be aligned with a text substring with period u and having suffix u, in any match in this window. Let u' be a proper suffix of u such that the substring of the text beginning at location i has the form u' followed by several occurrences of u. Let $leftval$ be defined as the the minimum edit distance between x and some string y with the following property: y begins with u', ends with u, has canonical period u, and $|x|-k \leq |y| \leq |x|+k$. The pattern occurs starting at text location i if and only if $leftval$ plus the locked edit distance of all other pattern and text intervals involved in matches in this window sum to at most k.

It remains to describe how $leftval$ is determined in this case. We compute the edit distances of x and each of the $2k+1$ longest suffixes of the unique string having canonical period u, suffix u, and length $|x|+k$. This takes $O(k^2)$ time using the algorithm in Section 5 (see the remarks at the end of that section). For any relevant location i, $leftval$ is easily determined in constant time from this information.

Case 2. Second, consider the case when the leftmost interval is a text interval. Let x be the substring of the text starting from location i and extending up to and including the right end of this interval. Recall from Lemma 7.2 that x must be aligned with a pattern substring with period u and having suffix u, in any match in this window. Let u' be a proper suffix of u such that the pattern begins with u' followed by several occurrences of u. Let $leftval$ be defined as the the minimum edit distance between x and some string y with the following properties: y begins with u', ends with u, has canonical period u, and $|x|-k \leq |y| \leq |x|+k$. The pattern occurs starting at text location i if and only if $leftval$ plus the locked edit distance of all other pattern and text intervals involved in matches in this window sum to at most k.

$leftval$ is determined as follows for this case. Note that x depends on the value of i and that there are too many values of i which need to be considered. The key to fast computation of $leftval$ in this case is that the i's can be partitioned into $O(k)$ equivalence classes based on their offsets with respect to the canonical period u. Specifically, if the left end of the pattern is shifted by distance k, then the edit distance, if no more than k, is unchanged (so long as the left end of the pattern remains in the window in question). For the left end of I has at least $2(k+1)$ disjoint occurences of u; one of them is aligned with a copy of u in the pattern. A shift of the pattern left end by k units to the right can be thought of as removing this copy of u from the text, thereby leaving the edit distance unchanged.

We perform the following computation. Let z denote the suffix of u of length $|u'|+k$, if $|u| \geq |u'|+k$, and the string u, otherwise. Let x' be formed by concatenating z with the leftmost interval (which is a text interval in this case). Let y' denote the string with canonical period u, suffix u, and length $|x'|+k$. For each of the k longest suffixes of x', we find the edit distances with each of the $3k$ longest suffixes of y'. This takes $O(k^2)$ time using the algorithm in Section 5. (In fact, we are really only interested in suffixes of y' with prefix u'.) For any relevant text location i, $leftval$ is easily determined in constant time from this information.

Let u'' be that proper suffix of u such that the text substring starting at location i has prefix u'' followed by several occurrences of u. From Lemma 7.2, we note that if $|u| \geq |u'|+k$ and $leftval$ is at most k, then $|u''| \leq |u'|+k$. Consider that suffix x'' of x' which has the form u'' followed by the leftmost interval. We claim that if $leftval$ is at most k then it equals the minimum edit distance of x'' with a suffix y'' of y' whose size is within k of x'' and which begins with u'. To see this, the following three observations suffice. First, x can be obtained from x'' and y from y'' by inserting strings cyclic in u. Second, the leftmost interval is a suffix of both x, x'' and has at least $2(k+1)$ disjoint occurrences of u at its left end. Third, in any approximate match of x and y of value at most k, all but k of the various disjoint occurrences of u must match exactly; the same is true of any approximate match of x'', y'' of value at most k.

The Special Case. We consider the case when there is only one interval in the pattern and it is incomplete in both directions. Thus the whole pattern is a single interval. In a second category match, the entire pattern is overlapped by a single periodic stretch in the text with canonical period, say w, of length at most $\frac{k^2}{2}$. Clearly, in this situation, it suffices to find matches beginning in any window of length $|w|$; all second category matches in which the pattern is completely aligned with the periodic stretch with canonical period w can be interpreted from this information as in the previous paragraph. All matches beginning in a window of length $|w| \leq \frac{k^2}{2}$ can be found in $O(k^3)$ time using the algorithm in Section 5.

8 The $O(k^4)$ Algorithm

Recall that the pattern is mostly periodic. However, the periods of the periodic stretches between various pairs of consecutive breaks could be different. Suppose the pattern has at most $2k-1$ and the text has at most $10k-2$ *bad segments* of length at most $4|u|$ each, such that the stretches between two adjacent bad segments are cyclic with canonical period u, for some string u. Such texts and patterns are called *even more periodic*. First, we will show how to handle patterns and texts which are not even more periodic in $O(k^4)$ time.

The even more periodic case is the hardest and is handled in Section 8.1.

The following steps are performed to determine whether or not the pattern is even more periodic, and to process it in case it is not.

Step 1. Recall that the periodic stretches in the pattern could have distinct periods. We choose a multiset U of disjoint substrings $u_1^2, \ldots, u_{2k+1}^2$ of the pattern as follows. The periodic stretches in the pattern are considered in non-increasing order of period length. For a particular stretch with canonical period, say, v, all (or as many as necessary to achieve the desired cardinality of $2k+1$) disjoint occurrences of v^2 in it are added to U. This procedure continues until U has exactly $2k+1$ substrings in it. Since the pattern is assumed to have length at least $5k^3$ (see beginning of Section 2), $2k+1$ such substrings always exist (recall all periods have length at most $k^2/2$, and the total length of breaks is at most $2k^3$). Let w denote u_{2k+1}.

There are two cases now. If $|w| \leq k$ then nothing is done in this step. Suppose $|w| > k$. Then we show how to obtain $O(\frac{m}{|w|})$ windows, each of size k, in which pattern matches can begin. In fact, we show something stronger, namely, in any window of size $|w|$ in the text, there are only a constant number of the above windows of size k in which pattern matches can begin.

All occurrences of the u_i^2's in the text are found in linear time using a standard multiple pattern matching algorithm [AC75]. Next, the text is partitioned into disjoint windows of size k each. Note that two occurrences of u_i^2 in the text occur at least distance $|u_i|$ apart (since u_i is a canonical period and therefore not cyclic; see property 2 in Section 2 as well). Pattern matches in which u_i^2 matches exactly must therefore begin in $O(\frac{m}{|u_i|}) = O(\frac{m}{|w|})$ windows; this is represented by putting a mark for u_i^2 on each such window. Over all strings in U, the total number of marks put is at most $(2k+1) * \frac{2m-2k}{|w|}$ (the size of the text is $2m-2k$) and only windows with at least $k+1$ marks can have pattern matches beginning in them. It follows that pattern matches can begin in only $O(\frac{m}{|w|})$ windows of size k each. In addition, any two windows which are more than distance k apart and which receive $k+1$ marks each must both receive a mark for some u_i^2. Since any two occurrences of u_i^2 occur at least distance $|u_i| \geq |w|$ apart, any two windows which are more than distance k apart and which receive $k+1$ marks each must actually be distance at least $w - 2k$ apart. It follows that in any window of size $|w|$ in the text, there are only a constant number of the above windows of size k in which pattern matches can begin. The total time taken is $O(m)$.

We now include all occurrences of all u_i's which are not identical to w as breaks in the pattern. The number of breaks in the pattern is still $O(k)$, each break being $O(k^2)$ in length. In addition, all periodic stretches have period lengths at most $|w|$. In the text, all periodic stretches with period lengths more than $|w|$ are partitioned into disjoint substrings of length k^2; these substrings are also included as breaks. Note that one substring in each stretch could have length less than k^2; this substring is just merged with the next break to the right. So the text has several breaks now, each of length between k^2 and $2k^2$. Now, as in Section 6, the text is trimmed so that it has only $O(k)$ breaks. The key property used in this trimming is that any break when aligned with a periodic stretch in the pattern (which now has period length at most $|w|$) must incur at least one mismatch. Thus, both the pattern and the text now have $O(k)$ breaks of length $O(k^2)$ each, with all intervening periodic stretches having period length at most $|w|$.

Step 2. We partition p into disjoint pieces of length $2|w|$. A piece-substring is a substring beginning and ending at piece boundaries. A piece-substring is *homogeneous* if at least three-fourths of the pieces in it have the same canonical period; it is *heterogeneous* otherwise.

Step 2: Case 1. If there exists a heterogeneous piece-substring of length $2|w| * (4k+1)$ in the pattern, then this piece-substring must overlap a break in the text in any match of the pattern. This is because any alignment of this piece-substring with a periodic stretch (which now has period length at most $|w|$) is guaranteed to give at least $k+1$ mismatches (at least $k+1$ pieces will have a canonical period different from the canonical period of the periodic stretch).

A heterogeneous piece-substring, if one exists, can be found in $O(m)$ time. In addition, if such a piece-substring exists then all matches of the pattern must begin in $O(k)$ windows, each of size $O(k^2 + k|w|)$. If $|w| \leq k$ then the total size of these windows is $O(k^3)$ and all matches beginning in these windows can be found in $O(k^4)$ time using the algorithm in Section 5. If $|w| > k$, then these windows can be further refined by taking intersections with the windows obtained in Step 1 (recall, pattern matches begin in only $O(1)$ length k windows in any length $|w|$ window) to give $O(k^2)$ windows each of size $O(k)$. Thus the total time taken to find all matches using the algorithm in Section 5 is $O(k*k^2*k) = O(k^4)$ in this case as well.

Step 2: Case 2. Suppose there is no heterogeneous piece-substring of length $2|w|*(4k+1)$ in the pattern. Then 3/4ths of the pieces in every piece-substring of length $2|w| * (4k+1)$ have the same canonical period, u say, $|u| \leq |w|$. Any periodic stretch which has canonical period different from u has length less than $2|w|(k+1)+4|w| = 2|w|(k+3)$, otherwise, there would be least $k+1$ complete pieces occurring contiguously within this periodic stretch, each having canonical period different from u; any piece-substring of length $2|w| * (4k+1)$ containing these pieces would then be heterogeneous.

We now make each periodic stretch in the pattern which has a canonical period different from u and length at least $2|w|$ a break (this is in addition to the existing breaks). Periodic stretches in the pattern with canonical periods different from u and length less than $2|w|$ are appended to the next breaks to the right. Thus, the pattern now has $O(k)$ breaks, each of length $O(|w|k + k^2)$ and all intervening periodic stretches have period u.

In the text, we redefine the breaks as follows. All existing breaks continue to be breaks. Recall that each of these has length between k^2 and $2k^2$. Call these breaks class 1 breaks. All periodic stretches with canonical period different from u also become breaks now; call these breaks class 2 breaks. Next, both classes of breaks are together reorganized into a new set of breaks so that each resulting break has length at least $4|w|k + 2k^2$ and at most $2(4|w|k+2k^2)$; this reorganization involves clubbing together existing breaks to form new breaks by including intervening strings and extending at the ends, or alternatively, partitioning a break into smaller breaks, if necessary. The length restrictions on the resulting breaks imply that the above reorganization allows for each class 1 break to be contained completely inside some resulting break; further, if a class 2 break is broken down and distributed over several resulting breaks, then each substring into which it is broken down has length at least $2|w|$. Then, each resulting break contains one of the following (below, the first two cases relate to those resulting breaks which include a class 1 break and the third relates to those resulting breaks which are derived from class 2 breaks):

1. A length k^2 aperiodic substring (these were the original breaks).
2. A substring with period length more than $|w|$ and having at least two consecutive occurrences of the canonical period (see the new breaks defined just before Step 2; also recall that $|w| \leq \frac{k^2}{2}$). Clearly, this canonical period will be different from u.
3. A substring of length $2|w|$ with period length at most $|w|$ and canonical period different from u, $|u| \leq |w|$.

Now, as in Section 6, the text is trimmed so that the total number of breaks in each half of the text is $O(k)$. The key property used in this trimming is that any text break when aligned with a periodic stretch with canonical period u in the pattern must incur at least one mismatch. This holds because of the properties listed above. Thus, both the pattern and the text now have $O(k)$ breaks of length $O(k^2 + |w|k)$ each, with all intervening periodic stretches having canonical period u.

Now consider those substrings of the pattern of length $2|u|$ which do not have canonical period u. There are two subcases now.

First, suppose there are at least $2k$ such disjoint substrings. Then at least k of these substrings must match exactly in any match of the pattern. For such a substring to match exactly, it must be aligned with a text substring which is not a periodic stretch of u's. Recall that the text has $O(k)$ breaks and that all intervening periodic stretches have canonical period u. It follows that there are $O(k)$ windows in which possible matches of the pattern can begin, each window having length $O(|w|k + k^2)$. If $|w| \leq k$ then all these matches can be found in $O(k * k^2 * k) = O(k^4)$ time using the algorithm is Section 5. And if $|w| > k$ then each of the above windows of size $O(|w|k + k^2)$ can be further refined by taking intersections with the windows obtained in Step 1 to get $O(k^2)$ windows of size $O(k)$ each; the $O(k^4)$ time bound follows in this case as well.

The second subcase arises when there are fewer than $2k$ disjoint substrings of length $2|u|$ with canonical period different from u in the pattern. As in Section 6, the text can now be trimmed so that it has at most $10k - 2$ of these. Clearly, all stretches in the text and in the pattern between the above substrings are periodic with canonical period u (this follows from the fact that if two substrings, both having length $2|u|$ and canonical period u, overlap in $|u|$ locations, then their union also has canonical period u, by definition). Next, by extending each substring of length $2|u|$ with canonical period different from u on either side, the intervening stretches can be made cyclic in u (earlier they were just periodic with canonical period u, but not necessarily cyclic); the length of each such substring can go up to $4|u|$ in the process. Our text and pattern are now both even more periodic (defined at the beginning of this section).

8.1 The $O(k^4)$ Algorithm for the Even More Periodic Case

To get a faster algorithm, we have to define intervals which have stronger properties than those defined in Section 7. We define an *interval* in the pattern (text, respectively) to be a set of disjoint substrings of the pattern (text, respectively). Roughly speaking, intervals are formed by extending *bad segments* (substrings of length between $2|u|$ and $4|u|$ which do not have canonical period u) at either end while skipping over other intervals. Intervals will always have the property that they end in at least one, possibly more, occurrences of the period u at each end. The *span* of an interval is the substring between and including the leftmost and the rightmost characters in the interval. In contrast to the intervals defined in Section 7, spans of intervals defined here could be nested one inside the other.

Recall the definition of locking from Fig.6. We say that an interval in the pattern (text, respectively) *locks* in a particular alignment if the portion of the text (pattern, respectively), if any, with which this interval is aligned is a cyclic repetition of u.

Our strategy will be to identify intervals in the pattern and the text with total length $O(k|u|)$, with each interval having length at least $2|u|$. These intervals will have the following property: in any match of the pattern, either some pattern interval overlaps some text interval, or all the pattern and text intervals are locked.

All matches in the first category clearly occur in at most $O(k^2)$ windows, each of length $\max\{k, |u|\}$. If $|u| \leq k$ then the total length of all these windows is $O(k^3)$ and all matches in these windows can be found in $O(k^4)$ time using the algorithm in Section 5. If $|u| > k$, then recall that $|u| \leq |w|$, that potential matches of the pattern have been determined in Step 1, and that there are only a constant number of windows of length k within any length $|w|$ window in which these matches can begin. It follows that all matches must again begin in $O(k^2)$ windows each of length $O(k)$; these matches can again be found in $O(k^4)$ time.

Matches in the second category will also occur in $O(k^2)$ windows, but of larger size. Whether or not the pattern matches in one such window will depend upon the *locked edit distance* of some of the intervals defined. These matches will be easy to find. In particular, if the pattern matches at a particular position in this window then it will match at all positions which are shifts of multiples of $|u|$ from this position in this window.

8.2 Defining Intervals

We show how the pattern is processed. The text is processed similarly.

We define intervals to contain all sufficiently small strings that are not repetitions of string u. More specifically, an interval I will be a string with a 2^i-fold repetition of string u at either end, for a suitable i. The best match of I with a string u^k, $(u^k| \geq |I|)$, for suitable k will be in locked alignment. Intervals are chosen to minimize i in a sense made precise below. Further, the intervals are chosen so that in any match in which the intervals in the text and pattern do not overlap, the intervals are all in locked alignment.

We define intervals as follows in $O(\log k)$ rounds. In each round, a set of partially formed intervals inherited from the previous round is processed. These intervals will be disjoint from each other. Some of the intervals being processed in the current round will be fully formed at the end of this round; these will not be processed in subsequent rounds. The remaining intervals will be processed further in the subsequent rounds.

The first round begins with a minimal collection of disjoint intervals, called *initial* intervals, where each initial interval is just a bad segment (defined at the beginning of Section 8.1). Recall that the portions of the string between the initial intervals are cyclic in u. The following procedure is performed in each round i, $i \geq 1$.

2^i-Extending Interval I. For each partially formed interval I being processed in the current round i, a 2^i-*extension* is determined as below. Starting from the left end of I, walk to the left skipping over any substrings in fully formed intervals until either another partially formed interval is reached or 2^i instances of u have been encountered. The same procedure is repeated at the right end. The substrings walked over in this process (ignore the substrings skipped over) along with the substrings in I together constitute the 2^i-extension of I.

An interval I processed in round i is said to be *successful* in this round if, after extension, it does not overlap or touch another extended interval on both the left and on the right.

Finally, we form new intervals by taking a *union* of the various extended intervals. Each new interval comprises maximal collections of extended intervals above such that consecutive extended intervals in each collection overlap or touch each other. Thus, if two extended intervals overlap or touch then they become part of the same interval now. Each new interval comprises exactly those pattern positions which belong to one of the extended intervals in the corresponding maximal collection of extended intervals. Of these new intervals, some will be fully formed, as described in the next paragraph. Those which are not fully formed will be carried over to the next round.

Condition for Full-Formedness. Each interval will have an *i-nested cost* to be defined below. Those intervals I whose span has locked edit distance (with respect to u) at most 2^i plus the i-nested cost of I will be fully formed at the end of this round; the remaining intervals will be processed again in the next round.

Definitions. The *i-current cost* of an interval I which is processed in round i is the locked edit distance of the span of I with respect to u, if it is fully formed by the end of round i, and 2^i plus its i-nested cost, if it is not yet fully formed at the end of round i. The *final cost* of an interval is its current cost at the end of the last round or its locked edit distance (with respect to u) if it is fully formed. The i-nested cost of I is the sum of the final costs of the fully formed intervals which were skipped over while forming I and the $(i-1)$-current costs of those partially formed intervals which are nested within I and were unsuccessful in round i. As the base case, we define the 0-current cost of an initial interval to be 1. Lemma 8.1 describes the motivation for the above definitions.

Lemma 8.1 *For all $i \geq 0$, the i-current cost of an interval I processed in round i is a lower bound on the cost of aligning the span s of I with a periodic stretch of u's.*

Proof: Consider a least cost match of s in a periodic stretch of u's. If $i = 0$, then the lemma follows from the fact that initial intervals have canonical periods different from u and therefore incur at least one mismatch. So assume that $i > 0$.

Note that s has 2^i occurrences of u at either end, possibly interspersed with intervals fully formed before round i. Some or all of these occurrences of u in s could be out of alignment with u's in text. If all these occurrences of u at the left end or at the right end are out of alignment then the cost of aligning s is at least 2^i plus, inductively, the i-nested cost. On the other hand, if at least one occurrence of u on either side aligns, then we claim that all occurrences of u further to the extremes of s from these two occurrences align as well. This is because the portions of s outside these two occurrences of u consist only of u's and other fully formed intervals, and fully formed intervals, by induction, cost at least (and, of course, at most) their locked edit distance. Therefore, the cost of the best match of s is the same as its locked edit distance with respect to u. The claim now follows from the fact that the i-current cost of I is the smaller of this distance and $2^i + i$-nested cost.
□

Termination Conditions for the Rounds. The ith round is the last round if the sum of the i-current costs of those intervals which are obtained in round i and are not nested inside other intervals and the sum of the final costs of those intervals that are fully formed earlier and not nested inside other intervals (we call both these kinds of intervals together *final* intervals) exceeds k, or if all intervals are fully formed. When the sum of the above costs is more than k, all matches of the pattern must have some interval in the text overlapping or touching some interval in the pattern. Clearly, the number of rounds is $O(\log k)$. The cost of processing a round, i.e., extending and computing the costs, is $O(k^3)$ (each of up to $O(k)$ intervals requires a locked edit distance calculation and each calculation is performed in $O(k^2)$ time using the algorithm described in Section 5). This can be reduced to $O(k^2)$ time, by performing the edit distance calculations more carefully, keeping in mind that the collective error that can be tolerated over all edit distance calculations is k. However, the bound of $O(k^3)$ per round suffices to achieve our final bound of $O(k^4)$.

Remark on the Text. A similar formation of intervals is done in the text, except that interval formation continues until each interval is either fully formed or $\log k + 1$ rounds are done, whichever is sooner.

Special Cases. The above interval formation algorithm needs to be suitably modified to account for the endpoints of the text and the pattern. We will very briefly sketch the special handling of intervals which encounter premature termination at either the left or the right end. Consider an interval which is prematurely terminated on the left. Intervals prematurely terminated on the right are handled similarly. In future rounds, this interval will be extended only to the right, until it is fully formed. Recall that full-formedness is related to the locked edit distance of span of the interval (with respect to u). The locked edit distance for such intervals is defined as in Section 7 (i.e., the span of this interval need not be aligned with a cyclic string of u's but with a string whose canonical period is u and which has suffix u).

Interval Lengths. We need the following lemma before describing the remainder of the algorithm.

Lemma 8.2 *The length of the span s of an interval I obtained in round i is at most $8|u|*i$-current cost of I.*

Proof: Consider the various initial intervals J in s. For each such initial interval J, consider the interval $int_j(J)$ which is the unique interval processed in round j whose span contains J. There may not be such an interval, of course. J is said to be *alive* in round j if it is the leftmost (rightmost, respectively) initial interval in $int_j(J)$ at the beginning of round j and $int_j(J)$ hasn't yet reached the left endpoint (right endpoint, respectively). Let the last round in which J is alive be denoted by $last(J)$. The *contribution* of J to s is defined to be the sum of the lengths of all the strings involved in extending the intervals $int_1(J), \ldots, int_{last(J)}(J)$, plus the length of J itself. Clearly, the length of s is at most the sum of the lengths of the contributions of the various initial intervals in s.

The contribution of J is at most $2*(\sum_{l=1}^{last(J)} 2^l)*|u|+|J| \leq 2(2^{last(J)+1}-2)|u|+4|u| \leq 2*2^{last(J)+1}*|u|$, if $last(J) \neq 0$, and $|J| \leq 4|u|$, otherwise. The $(last(J)-1)$-current cost of $int_{last(J)-1}(J)$ is at least $2^{last(J)-1}$ plus its $(last(J)-1)$-nested cost, if $last(J) \geq 1$. We call the quantity $2^{last(J)-1}$ the *capacity* of J (unless $last(J) = 0$, in which case, the capacity is defined to be 1). It is easy to see that the capacities of the various initial intervals in s sum to at most the i-current cost of I. The lemma follows. □

The Algorithm. First, intervals are formed as above. Next, two minimal sets of final text intervals, one on either side of the middle of the text, each with total final cost exceeding k are chosen (if either of these two sets does not have final cost exceeding k, all the text intervals in the corresponding half are taken). By Lemma 8.2, the total lengths of the spans of these final text intervals and the final pattern intervals will be $O(k|u|)$. Ignore the remaining text intervals for the moment. Each match in the span of one of these text intervals that overlaps or touches the span of one of the final pattern intervals is found. In addition, matches in which one of the endpoints of the pattern is aligned with one of these text intervals in found. These matches occur in $O(k^2)$ windows each of size $O(\max\{k, |u|\})$. If $|u| > k$ then each of the above windows of can be further refined by taking intersections with the windows obtained in Step 1 to get $O(k^2)$ windows of size $O(k)$ each. All such matches can then be found in $O(k^4)$ time using the algorithm in Section 5.

Next, we consider the remaining matches of the pattern. Note that the spans of the the final intervals in the pattern cannot overlap with the spans of the above final intervals chosen in the text. In addition, the text can be trimmed so as to contain only the above final intervals, by an argument similar to the one used in Section 6. The reason for this is that, by Lemma 8.1, the above final intervals in the left half of the text will incur more than k mismatches if all of them are overlapped by the pattern (note that they must all be aligned with periodic stretches having canonical period u in the matches being considered), and likewise for the above final intervals in the right half. It follows that the spans of the final intervals in the pattern cannot overlap with the spans of any of the final intervals in the text, in any of the remaining matches.

The remaining matches occur in $O(k^2)$ windows as well. Consider one such window. Consider the final costs of the final intervals in the pattern and the final costs of those final intervals in the text which overlap the pattern. If any one of these text intervals is partially formed then the pattern cannot match the text, because the final cost of this text interval is more than k and Lemma 8.1 implies that the span of this interval must incur more than k edit operations when aligned with a periodic stretch of u's. Similarly, if any of pattern intervals is partially formed then again the sum of the final costs of these pattern intervals exceeds k and the pattern cannot match. So suppose that all these text and pattern intervals are fully formed. Then the final cost of each such interval is its locked edit distance. If the sum of these final costs is at most k then the pattern matches at intervals of $|u|$ in this window with all these final intervals locked, and otherwise it does not match anywhere in this window. The precise locations where matches occur can be determined as in the algorithm for second category matches described towards the end of Section 7.1.2.

9 Is $O(\frac{nk^3}{m} + n + m)$ Running Time Possible?

The following text and pattern appear to form a hard case for our problem. They are defined in terms of an acyclic string u. Apart from $\Theta(k)$ substrings, each of length equal to $|u| = \Theta(k)$, the text and the pattern are periodic with period u. Suppose these *bad* substrings appear at intervals of $\Theta(k^2)$ in the text and at intervals of $\Theta(k)$ in a length $\Theta(k^2)$ prefix of the pattern. There are $\Theta(k^2)$ windows, each of size k, in which one of these pattern substrings overlaps some text substring. Exactly one bad pattern substring overlaps a bad text substring in any of the pattern placements in these windows. Our current $O(k^4)$ algorithm will take $\Theta(k^2)$ time to determine matches, if any, in each window, giving an $\Theta(k^4)$ time algorithm for this case. However, it is conceivable that an algorithm which takes $O(k)$ amortized time per window can be obtained by observing that the average edit distance between pairs of text-pattern substrings must be $O(1)$, otherwise there can be few matches. The difficulty we face is that the occurrences of u among the bad substrings of the pattern need not align with occurrences of u in the text.

10 The Weighted Case

In the weighted case deletions of different characters and the various substitutions may have differing costs, but by way of normalization, all will be required to have cost at least 1.

The approximate matches with cost $\leq k$ can be found using essentially the same algorithm; the only change needed is to the Landau-Vishkin algorithm, to take account of the differing costs. The details are left to the reader.

A important application in the weighted case is to match a pattern against a database of strings. We would like to apply the above algorithm. For efficiency, one approach would be to have a precomputed suffix tree for the database of strings. This suffix tree would then need to be incremented to incorporate the pattern string so as to enable the above algorithm to be used. Following the match calculation, the modification to the suffix tree would need to be undone. It would also be useful to support both insertions and deletions to the database. We leave this topic as an open problem.

References

[AC75] A. V. Aho, M. Corasick, *Efficient String Matching: An aid to bibliographic search*, Communications of the ACM, 18, 1975, pp. 333-340.

[BN96] R. Baeza-Yates, G. Navarro, *Faster Approximate String Matching*, Proceedings of CPM, Springer-Verlag Lecture Notes No. 1075, 1996, pp. 1-23.

[CL90] W. I. Chang, E. Lawler, *Approximate String Matching in Sublinear Expected Time*, Proceedings of 31st Annual IEEE Symposium on Foundations of Computer Science, 1990, pp. 116-125.

[CR94] M. Crochemore, W. Rytter, *Text Algorithms*, Oxford University Press, New York, 1994, pp. 27-31.

[F98] M. Farach, *Optimal Suffix Tree Construction with Large Alphabets*, Proceedings of 38th Annual IEEE Symposium on Foundations of Computer Science, 1997, pp. 137-143.

[GG88] Z. Galil, R. Giancarlo, *Data Structures and Algorithm for Approximate String Matching*, Journal of Complexity, 4, 1988, pp. 33-72.

[GP90] Z. Galil, K. Park, *An Improved Algorithm for Approximate String Matching*, SIAM Journal on Computing, 19(1990), pp. 989-999.

[LV85] G. Landau, U. Vishkin, *Efficient String Matching in the Presence of Errors*, Proceedings of 26th Annual IEEE Symposium on Foundations of Computer Science, 1985, pp. 126-136.

[LV86] G. Landau, U. Vishkin, *Introducing Efficient Parallelism into Approximate String Matching and a New Serial Algorithm*, Proceedings of 18th Annual ACM Symposium on Theory of Computing, 1986, pp. 220-230.

[LV88] G. Landau, U. Vishkin, *Fast String Matching with k Differences*, Journal of Computer and System Sciences, 37, 1988, pp. 63-78.

[LV89] G. Landau, U. Vishkin, *Fast Parallel and Serial Approximate String Matching*, Journal of Algorithms, 10, 1989, pp. 158-169.

[SV97] S. Sahinalp, U. Vishkin, *Approximate Pattern Matching using Locally Consistent Parsing*, Manuscript. Abstract appeared in Proceedings of 37th IEEE Symposium on Foundations of Computer Science, Burlington, 1996, pp. 320-328.

[SV88] B. Schieber, U. Vishkin. *On Finding Lowest Common Ancestors: Simplification and Parallelization*. SIAM Journal on Computing, 17, 1988, pp. 1253-1262.