A Sublinear Algorithm for Approximate Keyword Searching[1]

Eugene W. Myers[2]

Abstract. Given a relatively short query string W of length P, a long subject string A of length N, and a threshold D, the *approximate keyword search problem* is to find all substrings of A that align with W with not more than D insertions, deletions, and mismatches. In typical applications, such as searching a DNA sequence database, the size of the "database" A is much larger than that of the query W, e.g., N is on the order of millions or billions and P is a hundred to a thousand. In this paper we present an algorithm that given a precomputed *index* of the database A, finds rare matches in time that is *sublinear* in N, i.e. N^c for some $c < 1$. The sequence A must be over a *finite* alphabet Σ. More precisely, our algorithm requires $O(DN^{pow(\varepsilon)} \log N)$ expected-time where $\varepsilon = D/P$ is the maximum number of differences as a percentage of query length and $pow(\varepsilon)$ is an increasing and concave function that is 0 when $\varepsilon = 0$. Thus the algorithm is superior to current $O(DN)$ algorithms when ε is small enough to guarantee that $pow(\varepsilon) < 1$. As seen in the paper, this is true for a wide range of ε, e.g., ε up to 33% for DNA sequences ($|\Sigma| = 4$) and 56% for proteins sequences ($|\Sigma| = 20$). In preliminary practical experiments, the approach gives a 50- to 500-fold improvement over previous algorithms for problems of interest in molecular biology.

Key Words. Approximate match, Dynamic programming, Index, Word neighborhood

0. Introduction

Given a relatively short query string W of length P, a long subject string A of length N, and a threshold D, the *approximate keyword search problem* is to find all substrings of A that align with W with not more than D insertions, deletions, and mismatches. More precisely, if $\delta(V, W)$ is the edit distance between V and W, and if $A[i..j]$ denotes the substring of A consisting of its i^{th} through j^{th} characters, then the problem is to find all pairs i, j such that $\delta(W, A[i..j]) \leq D$. For this problem, we say that the maximum *mismatch ratio* is $\varepsilon = D/P$ and that we are searching A for ε-*matches* to W.

This problem has been much studied. Sellers [Sel80] presented the obvious $O(PN)$ algorithm as a slight variation of the classic dynamic programming algorithm for the sequence vs. sequence comparison problem (here we are comparing a sequence vs. substrings of the other).

[1] This work was supported in part by the National Institutes of Health under Grant R01 LM04960-01 and the Aspen Center for Physics.

[2] Department of Computer Science, University of Arizona, Tucson, AZ 85721, U.S.A.

In roughly the same time frame, Ukkonen and Myers [Ukk85a,MyM86] both reported practical and simple $O(DN)$ expected time algorithms. Not long thereafter, Landau and Vishkin [LaV86] arrived at an $O(DN)$ worst case algorithm that required $O(N)$ space. At about the same time Myers [Mye86a] presented an algorithm with the same worst case time complexity but which required only $O(D^2)$ space. Recently, Galil and Park [GaP89] have reported another $O(DN)$ worst-case time algorithm that takes $O(P^2)$ space.

With the advent of applications such as those in molecular biology where the database will be massive, e.g. N in the billions, the need for algorithms that are less than linear in N is becoming of paramount importance. Recently, Chang and Lawler [ChL90] devised a method that takes $O(DN \log P/P)$ expected-time when the threshold D is less than $P/(\log P + O(1))$. It does so by quickly eliminating stretches of the "database" sequence A where a match cannot possibly occur. They term their algorithm sublinear in the sense of Boyer and Moore [BoM77], i.e., cN characters of A are examined where $c < 1$. Note however, that the algorithm still takes time linear in N and that as P gets larger the stringency of a match must be tightened as ε must be less than $1/(\log P + O(1))$.

In this paper we present an algorithm that given a precomputed index of the database A, finds rare matches in time that is truly sublinear in N, i.e. N^c where $c < 1$. More precisely, our algorithm requires $O(DN^{pow(\varepsilon)} \log N)$ expected-time where $pow(\varepsilon)$ is an increasing and concave function that is 0 when $\varepsilon = 0$. Thus the algorithm is superior to the $O(DN)$ algorithms when ε is small enough to guarantee that $pow(\varepsilon) < 1$. For example $pow(\varepsilon)$ is less than one when $\varepsilon < 33\%$ for $|\Sigma| = 4$ (DNA alphabet), and when $\varepsilon < 56\%$ for $|\Sigma| = 20$ (Protein alphabet). Figure 4 on page 9 precisely plots the curve for several choices of $|\Sigma|$. Apart from the fact that our algorithm is "truly" sublinear, it also has the advantage over the Chang and Lawler algorithm that the degree of sublinearity just depends on ε and not on P. On the other hand, we require a precomputed $O(N)$ space index structure, whereas their method is purely scanning based, requiring only $O(P)$ working storage. The bounding argument used in proving the expected complexity of our algorithm is rather crude. Consequently, performance in practice is much superior. In preliminary experiments, the approach appears to represent a 100- to 500-fold improvement over the $O(DN)$ search algorithms for problems of interest in molecular biology.

1. Overview

In this overview, we sketch the basic concepts and outline the algorithm embodying our result. The various sections of the paper then embellish upon the individual components.

The algorithm assumes that an index for the sequences in the database has already been constructed. An index for a large text is a data structure that allows one to rapidly find all occurrences of a given query string in the text. In our method, queries will all be of length $T = \log_{|\Sigma|} N$. Thus, each query can be uniquely encoded as an $O(N)$ integer, and we can store the results of all possible queries (which are lists of indices where the corresponding strings of size T appear in A) in an $O(N)$ table. This simple structure can be built in $O(N)$ time and $2N$ words of space as shown in Section 2.

Let $\delta(V, W)$ be the edit distance between V and W. Let the *D-neighborhood* of a string W be the set of all strings distance less than or equal to D from W, i.e., $N_D(W) = \{ V : \delta(V, W) \leq D \}$. Let the *condensed D-neighborhood* of W be the set of all strings in the D-neighborhood of W that do not have a prefix in the neighborhood, i.e., $\overline{N_D}(W) = \{ V : V$ in $N_D(W)$ and no prefix of V is in $N_D(W)$ $\}$. One way to find all approximate matches to W is to generate every string in the condensed D-neighborhood of W, then, for each such string, to find the locations at which the string occurs in the database using an index. Each such location is the leftmost position of an approximate match to W. The obvious problem is that as W or D become large the number of strings in $\overline{N_D}(W)$ quickly explodes, making the standard $O(DN)$ algorithms superior. However, for strings W whose length is $T = \log_{|\Sigma|} N$, the following are true:

Section 3: There exists an algorithm to generate the strings in $\overline{N_D}(W)$ in lexicographical order in $O(DN^{pow(\varepsilon)})$ worst-case time. The algorithm involves computing rows of a dynamic programming matrix in response to a backtracking search that essentially traces a trie of the words in $\overline{N_D}(W)$.

Section 4: $| \overline{N_D}(W) | \leq N^{pow(\varepsilon)}$ where $pow(\varepsilon) = \log_{|\Sigma|} (c+1)/(c-1) + \varepsilon \log_{|\Sigma|} c + \varepsilon$ and $c = \varepsilon^{-1} + \sqrt{1+\varepsilon^{-2}}$.

Section 5: Using the simple index described above, the algorithm above can look up the locations of these strings at no additional overhead, and under the assumption that A is the result of Bernouilli trials, finds $O(N^{pow(\varepsilon)})$ matches in expectation.

Therefore, for small strings the strategy of generating all words in the neighborhood of the query is effective for sufficiently small distances. To extend this strategy to larger queries requires the following observation detailed in Section 6. Consider dividing the query W in a binary fashion until all pieces are of size $\log_{|\Sigma|} N$. To model the various pieces, let W_α for $\alpha \in \{0, 1\}^*$ be recursively defined by the equations: $W_\varepsilon = W$, $W_{\alpha 0}$ = first_half_of(W_α), and $W_{\alpha 1}$ = second_half_of(W_α). The following lemma follows from a simple application of the Pigeon-Hole Principle.

Lemma: If W aligns to a string V with not more than D differences, then there exists a word α such that for every prefix β of α, W_β aligns to a substring V_β of V with not more than $\lfloor D/2^{|\beta|} \rfloor$ differences. Moreover, $V_{\beta a}$ is a prefix or a suffix of V_β according to whether a is 0 or 1.

This suggests that we can efficiently find approximate matches to W by first finding approximate matches to the words W_α of length T and then verifying that W_β approximately matches for progressively shorter prefixes β of α. At each stage, the string in question is twice as large and twice as much distance is allowed in the match, but the number of matches found drops hyperexponentially except where the search will reveal a distance D match to W.

Suppose for simplicity that the length of W, P equals $2^K T$ for some value of K. For $d = \lfloor D/2^K \rfloor$, we begin by generating the words in $\overline{N_d}(W_\alpha)$ for each of the 2^K words W_α of length T. From the above this takes $O(DN^{pow(\varepsilon)})$ total time and delivers this many d-matches. We then see if these d-matches can be extended to $\lfloor D/2^{K-1} \rfloor$-matches to the words W_β of length

$2T$. This step requires envisioning the result of the word generation lookups as delivering parallelogram shaped regions of a dynamic programming matrix or edit graph. All d-matches to a T-word of W are guaranteed to lie in one of these parallelograms. To find matches to $2T$-words it suffices to do a dynamic programming calculation over a $2T$-by-$2d$ parallelogram about each parallelogram of a T-word ‘‘hit’’. At the k^{th} stage of this process, the number of matches is reduced by a factor of $1/N^{2^k(1-pow(\varepsilon))}$. Thus while the time to extend matches grows by a factor of 4 at each stage, this is overwhelmed by the reduction in the number of surviving matches. In the end, the total time consumed in expectation is $O(DN^{pow(\varepsilon)} \log N + HDP)$ where H is the number of matches to W found.

Note that our algorithm’s expected time complexity is based on the assumption that the database is the result of random Bernouilli trials. In this case its expected complexity is $O(DN^{pow(\varepsilon)} \log N)$. However, if the database is not random but preconditioned to have H matches to W, then $O(HDP)$ time will be spent on each of these matches. Consequently, our algorithm does not improve upon the $O(DN)$ algorithm for the sequence-vs.-sequence problem. Nonetheless, for searching problems where the database is large and ‘‘sufficiently’’ random, this algorithm can find near matches with great efficiency and no loss in sensitivity.

2. A Simple Index Data Structure

An index for a large string $A = a_1 a_2 \cdots a_N$, is a data structure that allows one to efficiently locate all occurrences of a shorter query string within A. In this work, a very simple technique based on integer encodings can be used because all queries are of length $O(log\ N)$ and we assume that the underlying alphabet Σ is fixed and finite. From here on, let $T = \log_{|\Sigma|} N$ and assume all queries are of length between $T-D$ and $T+D$ where $D < T$.

Consider an arbitrary assignment (bijection) ϕ of the symbols in Σ to the integers 0 through $|\Sigma|-1$. ϕ is naturally extended to strings with the recursive definition $\phi(Wa) = |\Sigma|\,\phi(W) + \phi(a)$ where W is a string over Σ and a is a symbol in Σ. Essentially $\phi(W)$ is the integer obtained when the string is viewed as a radix-$|\Sigma|$ number. For $n \in [0, |\Sigma|^T - 1] \subseteq [0, N-1]$, let $Bucket(n) = \{\ i : \phi(a_i a_{i+1} \cdots a_{i+T-1}) = n\ \}$. That is, $Bucket(n)$ gives the indices of the leftmost character of each occurrence in A of the unique T-symbol string whose ϕ-code is n. This simple array of sets is our index structure.

Under the assumption that A is the result of equi-probable Bernouilli trials, one can use *Bucket* to find all the, say H_V, occurrences of a query V within A in $O(T+DH_V)$ expected time as follows. If V is of length $U \le T$ then the leftmost indices of the occurrences of V are exactly the contents of $Bucket(k)$ for $k \in [\phi(V)|\Sigma|^{T-U}, (\phi(V)+1)|\Sigma|^{T-U} - 1]$. In this case, it takes time $O(U)$ to compute the integers defining the interval of codes and $O(H_V)$ time to list the H_V occurrences of V in A. Thus the total time is less $O(T+DH_V)$ as claimed. In the case where U is greater than T one knows that the occurrences of V must be a subset of those in $Bucket(\phi(V_T))$ where V_T denotes the string consisting of the first T symbols of V. It takes $O(T)$ time to compute the ϕ-code, and then it suffices to simply check whether the remaining $U-T$ symbols of V match at each of the locations in the given bucket. While potentially quite inefficient in the worst case this step works extremely well in the expected case where A is the result of equi-probable Bernouilli trials. Under this assumption the average number of coincidental matches to V_T is one and on average only $1/(|\Sigma|-1)$ additional symbols of such coincidental matches to V_T need be checked before they are discovered not to match V. Thus in expectation only a constant amount of time is spent investigating locations that do not match V. Moreover, $O(U-T) = O(D)$ time is spent checking additional characters at those locations that do match V. Thus only $O(T+DH_V)$ expected time is spent in the case where $U > T$.

Producing the index is also quite simple. First $\phi_i = \phi(a_i a_{i+1} \cdots a_{i+T-1})$ is computed for every index i. This is easily done in an $O(N)$ sweep of A using the observation that $\phi_i = a_i |\Sigma|^{T-1} + \lfloor \phi_{i+1} / |\Sigma| \rfloor$. Since the numbers ϕ_i are all in the range $[0, N-1]$, a simple $O(N)$ radix sort produces the list $Indices = < i_1, i_2, \cdots i_N >$ such that $\phi_{i_j} \le \phi_{i_{j+1}}$. Finally, the array $Header[n] = \min\{ j : \phi_{Indices[j]} = n \}$ is produced in an $O(N)$ sweep of *Indices*. The arrays, *Indices* and *Header*, together provide a realization of the *Bucket* sets. Namely, $Bucket(n) = \{ Indices[j] : j \in [\, Header[n], Header[n+1]-1 \,] \}$. Thus our index occupies $O(N)$ space and takes $O(N)$ time to construct.†

In the treatment immediately above two details were overlooked that require attention. First was the statement that T be set to $\log_{|\Sigma|} N$. T must be an integer, and so in fact one must round the logarithm either up or down. Rounding up implies that as much as $O(|\Sigma| N)$ time and *space* are required to construct the index. Rounding down implies that while only $O(N)$ space is used, long queries may take $O(T + |\Sigma| + DH_V)$ expected time because the average bucket size can approach $|\Sigma|$. If the cardinality of Σ is fairly large (e.g. 20 for protein sequences) then both of these alternatives are undesirable in practice. A simple repair is to round up, i.e., $T = \lceil log_{|\Sigma|} N \rceil$, but to use only the first $R = \lfloor \log_2 N - \lfloor \log_{|\Sigma|} N \rfloor \log_2 |\Sigma| \rfloor$ bits of the last symbol when computing a code. Specifically, if W is of length $T-1$ then the modified encoding ρ of Wa is given by $\rho(Wa) = \phi(W) 2^R + \phi(a) \; mod \; 2^R$. The treatment for constructing the index is as before except that ϕ_i is replaced with $\rho_i = \rho(a_i a_{i+1} \cdots a_{i+T-1})$. With this choice of encoding, there are no more than N codes and *there are no less than* $N/2$. Thus the structure takes $\Theta(N)$ time and space to construct independent of the size of Σ, and the average bucket size is between 1 and 2 so that searches using the ρ-code on the first T symbols of a query V will take no more than $O(DH_V)$ expected time. Note that the T^{th} symbol of V will have to be compared against its corresponding symbol for each bucket entry, but this does not add to the asymptotic complexity.

The second detail is that ϕ_i (ρ_i) is not properly defined for $i > N-T+1$ ($N-T$). This can rectified by simply adopting the convention that for $i > N$, a_i is any symbol not equal to a_N. This guarantees that the integer codes for indices $N-T+2$ to N are all distinct and thus contribute at most one extra element to any particular *Bucket* set. Thus the expected case analysis for searching still holds, although one must additionally check each index in a bucket and reject it if it is greater than $N-U$ where U is the length of the query.

†The structure requires exactly $2N + O(1)$ integers in the range 0 to N, and may be built with $\sigma(N)$ additional space with some very careful ‘‘in-place’’ manipulations. It is thus a very space efficient and practical index.

3. Generating Word Neighborhoods

The traditional sequence comparison of word $W = w_1 w_2 \cdots w_T$ with another word $V = v_1 v_2 \cdots v_U$ involves the computation of a dynamic programming matrix $L[0..U, 0..T]$ where $L[i, j] = \delta(V_i, W_j)$. The notation V_i denotes the prefix consisting of the first i symbols of V. Given a vector $R[0..T]$ and symbol a in alphabet Σ, let $row(R, a)$ be the vector $S[0..T]$ such that $S[0] = R[0]+1$ and for $j > 0$, $S[j] = \min\{ S[j-1] + 1, R[j] + 1, R[j-1] + (if\ a = w_j\ then\ 0\ else\ 1) \}$. It is well known that $L[0] = < 0, 1, 2, ... T >$ and for $i > 0$, $L[i] = row(L[i-1], b_i)$. Moreover, an induction reveals that entries in the matrix L increase by 0 or 1 along diagonals (e.g., $L[i, j] = L[i-1, j-1] + \{0, 1\}$), and by -1, 0, or 1 along rows (e.g., $L[i, j] = L[i, j-1] + \{-1, 0, 1\}$).

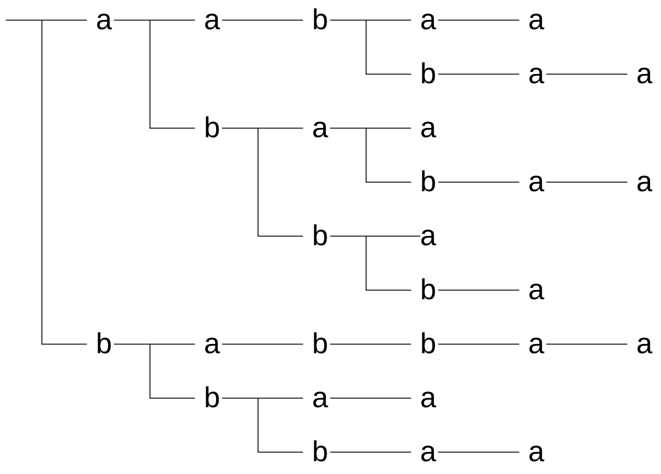


Figure 1: Trie for $\overline{N_1}(abbaa)$.

Consider the problem of generating the words in the condensed D-neighborhood of word W. Imagine a trie of all the words in this neighborhood and imagine traversing or delineating it with a backtracking search that explores the space of all words in lexicographical order. Figure 1 gives the trie for the neighborhood $\overline{N_1}(abbaa)$ over the alphabet $\Sigma = \{a, b\}$. Note that all vertices of the trie have outdegree equal to either $|\Sigma|$ or 1. The forthcoming algorithm of Figure 3 essentially provides a constructive proof of this fact. It also proves that every word in the condensed neighborhood is exactly distance D from W. More directly, this follows by observing that if a more closely matching word were in the neighborhood, then the prefix obtained by deleting its last symbol is also in the neighborhood. As the search generates words it computes the corresponding rows of the dynamic programming matrix of the current word versus W. It uses these rows to direct the search as follows. If a word is generated for which the last entry of the most current row is D, then a word in the condensed neighborhood has been reached. On the other hand, if all entries of a row are greater than D then the corresponding word and all extensions of it cannot be in the condensed neighborhood of W, and the search can backtrack. Otherwise there is some extension that is in the neighborhood and the search proceeds forward. The algorithm of Figure 2 details such a search.

```
array $L[0..T+D+1,\ 0..T]$
vector $V[1..T+D+1]$

1.  procedure GEN($i$)
2.    {  for $a \in \Sigma$ do
3.         {  $L[i] \leftarrow row(L[i-1], a)$
4.            $V[i] \leftarrow a$
5.            if $L[i, T] \le D$ then
6.                 $V[1..i]$ is in $N_D(W)$.
7.            else if $\min_j \{L[i, j]\} \le D$ then
8.                 GEN($i+1$)
9.         }
10.   }

11. $L[0] \leftarrow < 0, 1, 2, ...T >$
12. GEN(1)
```

Figure 2: Neighborhood Generator Algorithm

The correctness of this procedure requires several observations. First, the smallest entry of $row(R, a)$ is never smaller than the smallest entry of R and so it is correct to backtrack when a row is reached for which all entries are greater than D. Second, if a row contains an entry not greater than D then it contains an entry equal to D because successive entries in a row differ by -1, 0, or 1. Moreover, if the largest index of such an entry is j, then adding the suffix W^j = $w_{j+1} w_{j+1} \cdots w_T$ gives a word in the neighborhood, and this justifies the decision to search forward. Finally, the length of a longest word in the neighborhood is bounded by $T+D$ and so the sizes of L and V are adequate.

The algorithm of Figure 2 spends $O(|\Sigma| T)$ time per call to GEN and the number of calls is the number of characters in the trie of the neighborhood. The size of the trie is bounded by $O(TZ)$ where Z is the number of words in the neighborhood. Thus the algorithm has a worst case complexity of $O(|\Sigma| T^2 Z)$. One can do quite a bit better, namely $O(TZ)$ time, by better utilizing the information in the matrix rows.

Note that as the search progresses forward through the trie one must compute a row whose minimum entry is D before computing a row whose minimum entry is greater than D. Consider the case where such a row is reached and it is further true that the last entry is not D. In this case, the only extensions of the currently generated word that are in the condensed neighborhood are those that perfectly match the appropriate suffixes of W. That is, if the j^{th} entry is D, then adding W^j gives a word whose distance from W is D and this is the only way to get a word this close to W. The one difficulty is that while the word may be in the D-neighborhood, it may not be in the condensed neighborhood. For example, when *aba* has been generated in the example of Figure 1, the current vector is $< 3, 2, 1, 1, 1, 2 >$ and the possible extensions are the suffixes *baa*, *aa*, and *a* of W = *abbaa*. But *aba•aa* is not in the condensed neighborhood as *aba•a* is. In essence, of the available suffix extensions, one must choose only those that do not have another as a prefix.

This difficulty can be efficiently handled with the failure links of the Knuth-Morris-Pratt construction [KMP77] used for exact keyword search. For a word V, let $fail_V(0)$ = 0 and for

$j \in [1, |V|]$, let $fail_V(j) = \max\{ k : V_k \text{ is a suffix of } V_j \}$. An array recording the values of $fail_V$ can be computed in time linear in the length of V. For our problem, let $Jump[T-j] = T - fail_{W^R}(j)$ where W^R is the reverse of W. For index j, the indices $Jump[j]$, $Jump[Jump[j]]$, $Jump[Jump[Jump[j]]]$, $\cdots$ are exactly those whose suffix extensions are prefixes of j's suffix extension. Thus to check if j's extension gives a word in the condensed neighborhood simply requires checking the above sequence until an index whose entry is D is reached (in which case reject) or until index T is reached (in which case accept). These checks are realized in lines 8 to 15 of the algorithm of Figure 3. In order to only spend $O(T)$ time, the indices are checked in decreasing order and *Quick*, a ''short-circuited'' version of *Jump*, is built on the fly. After index j is processed in lines 10 to 12, $Quick[j]$ either contains the smallest index on the *Jump*-chain from j whose entry is D, or T if there is no such index. This permits index j to be checked for suffix extension in constant time in lines 13 and 14.

```
array $L[0..T+D, 0..T]$
vector $V[0..T+D]$, $Jump[0..T]$, $Quick[0..T]$

1.  procedure GEN($i$)
2.    {  for $a \in \Sigma$ do
3.         {  $L[i] \leftarrow row(L[i-1], a)$
4.            $V[i] \leftarrow a$
5.            if $L[i, T] = D$ then
6.                 $V[1..i]$ is in $\overline{N_D}(W)$.
7.            else if $\min_j \{L[i, j]\} = D$ then
8.                 $Quick[T] \leftarrow T$
9.                 for $j \leftarrow T-1, T-2, \cdots 0$ do
10.                  {  $Quick[j] \leftarrow Jump[j]$
11.                     if $L[i, Quick[j]] \neq D$ then
12.                          $Quick[j] \leftarrow Quick[Quick[j]]$
13.                     if $Quick[j] = T$ and $L[i, j] = D$ then
14.                          $V[1..i] \bullet W^j$ is in $N_D(W)$.
15.                  }
16.           else
17.                GEN($i+1$)
18.        }
19.   }

20. Compute $Jump[0..T]$.
21. $L[0] \leftarrow <0, 1, 2, ...T>$
22. if $D = 0$ then
23.      $W$ is the only member of $\overline{N_D}(W)$.
24. else
25.      GEN(1)
```

Figure 3: Refined Neighborhood Generator Algorithm

Figure 3 gives the improved variation of the algorithm of Figure 2. The search is more efficient because it stops as soon as a row whose minimum is D is reached. Each iteration of the loop of lines 2 to 18 takes $O(T)$ time provided the concatenation in line 14 is not actually performed. It is shown in Section 5 that it is indeed unnecessary to actually concatenate the two strings. An iteration is either charged to the one or more words in the neighborhood reported in lines 6 or 8 to 15, or if line 17 is executed than the iteration is charged to the vertex of the trie labeled with the current word. But there are only $O(Z)$ such vertices as each has outdegree $|\Sigma| > 1$. Thus the total time spent in the algorithm is $O(TZ)$.

A final improvement from $O(TZ)$ to $O(DZ + T)$ time is possible by observing that only a portion of the matrix L need be computed. As observed in several earlier papers [Ukk85b,Mye86b], only those entries $L[i, j]$ for which $|i - j| \leq D$ can have a value less than D. Thus all the row queries of the algorithms above can be answered by only computing these portions of each row. Since this portion consists of at most $2D + 1$ entries, the time for each execution of lines 3-18 can be reduced to $O(D)$ worst case time. This includes the extension step of lines 8 to 15 because it need only operate over the relevant indices. The $O(T)$ term remains for the computation of the *Jump*-vector.

4. Hit Probabilities and Neighborhood Sizes

In this subsection bounds are determined on the number of words in the condensed D-neighborhood of a word of length T, and on the probability of matching one of these words at a given position in a large and random database. Formally, let $Z(T, D) = \max\{ |\overline{N_D}(W)| : W$ is a word of length $T \}$. Further let $Pr(T, D)$ be the maximum of $\sum_{V \in \overline{N}_D(W)} |\Sigma|^{-|V|}$ over all words W of length T. If a database is the result of equi-probable Bernouilli trials over alphabet Σ, then $|\Sigma|^{-|V|}$ is the probability of matching word V at a given position in the database. Thus $Pr(T, D)$ is the maximum probability of matching a word in a condensed D-neighborhood of a word of length T at a given position in the database. Call Pr the *hit* probability and observe that if the database is of size N then the number of occurrences of words in a neighborhood, or equivalently, the number of hits is $N\,Pr(T, D)$. Expressions that bound both of these quantities from above are derived below.

Every word V in the condensed D-neighborhood of a word W is exactly edit distance D from W as noted near the start of Section 3. Thus a very crude bound on $Z(T, D)$ is to count the number of D-operation edit scripts on an arbitrary word of length T. This is an upper bound since some distinct scripts will produce exactly the same word, and others produce words not in the neighborhood. For example, if $W = abbaa$, then deleting the fourth and fifth symbol produce the same word (*abba*) and inserting a b after the third symbol produces a word (*abbbaa*) that is not in $\overline{N_1}(abbaa)$ because *abbba* is. However, in making such an estimate one can avoid counting obviously redundant scripts that, for example, delete a symbol and insert another, as opposed to simply substituting the inserted symbol. Specifically, it suffices to consider only *normalized* scripts that may (0) insert some number of symbols *before* the *first* symbol of W, and at each position/symbol of W may either (1) do nothing, (2) delete the symbol, (3) insert some number of symbols *after* the position, or (4) substitute a *different* symbol and insert zero or more symbols after the position.

Let $S(T, D)$ be the number of D-operation edit scripts that adhere to restrictions (1) to (4) above. These scripts do not allow one to insert symbols before the first character of a word (restriction 0 above). Lemma 1 below presents a recurrence for S and a bound on Z in terms of S.

Lemma 1.

$$S(T, D) = \begin{cases} 1 & \text{if } D = 0 \\ 2|\Sigma| & \text{if } D = 1 \text{ and } T = 1 \\ (2|\Sigma| - 1)|\Sigma|^{D-1} & \text{if } D > 1 \text{ and } T = 1 \\ S(T-1, D) + S(T-1, D-1) + (2|\Sigma| - 1)\sum_{j=1}^{D} |\Sigma|^{j-1} S(T-1, D-j) & \text{otherwise} \end{cases}$$

$$Z(T, D) \leq \sum_{j=0}^{D} |\Sigma|^{j} S(T, D-j)$$

Proof. $S(T, 0) = 1$ as there is only one empty edit script. In general, note that at a given position there is 1 script that deletes the symbol there, $|\Sigma|^{j}$ scripts that insert j symbols, and $(|\Sigma| - 1)|\Sigma|^{j-1}$ scripts that substitute a non-identical symbol and then insert $j-1$ new symbols. Thus $S(1, 1)$ is $2|\Sigma|$ because one may perform one delete (1 script), perform one substitute ($|\Sigma| - 1$ scripts), or perform one insert ($|\Sigma|$ scripts). For $S(1, D)$ for $D > 1$, there is only one position at which to perform D operations and deleting is not a possibility. Thus one may perform D inserts ($|\Sigma|^{D}$ scripts) or a substitute and $D-1$ inserts ($(|\Sigma| - 1)|\Sigma|^{D-1}$ scripts). Finally, in the general and recursive case one may either (1) do nothing at the first position and perform D edits at the remaining $T-1$ positions ($S(T-1, D)$ scripts), (2) delete the symbol at the first position and perform $D-1$ edits at the remaining $T-1$ positions ($S(T-1, D-1)$ scripts), (3) insert j symbols after the first position for $j \in [1, D]$ and perform $D-j$ edits at the remaining $T-1$ positions ($|\Sigma|^{j} S(T-1, D-j)$ scripts), or (4) substitute a non-identical symbol and insert $j-1$ symbols at the first position for $j \in [1, D]$ and perform $D-j$ edits at the remaining $T-1$ positions ($(|\Sigma| - 1)|\Sigma|^{j-1} S(T-1, D-j)$ scripts).

Certainly $Z(T, D)$ is bounded from above by the total number of normalized scripts. The only scripts not counted by $S(T, D)$ are those with inserts before the first symbol. The number of normalized scripts where $j \in [0, D]$ symbols are inserted before the first symbol is $|\Sigma|^{j} S(T, D-j)$. Thus $Z(T, D)$ is bounded by the summation given in the statement of the Lemma. ■

To bound the probability *Pr*, a recurrence analogous to that for *N* is developed. In Lemma 2 below, $Q(T, D)$ is the sum of the probabilities of matching each word generated by a normalized D-operation script that does not insert before the first character. Since different scripts generate the same word, its contribution may be summed several times, and so Q is not necessarily less than 1. It is a bound and not a probability.

Lemma 2.

$$Q(T, D) = \begin{cases} 1/|\Sigma|^T & \text{if } D = 0 \\ (3|\Sigma| - 1)/|\Sigma| & \text{if } D = 1 \text{ and } T = 1 \\ (2|\Sigma| - 1)/|\Sigma| & \text{if } D > 1 \text{ and } T = 1 \\ Q(T-1, D)/|\Sigma| + Q(T-1, D-1) + \sum_{j=1}^{D} Q(T-1, D-j) & \text{otherwise} \end{cases}$$

$$Pr(T, D) \le \sum_{j=0}^{D} Q(T, D-j)$$

Proof. The argument mimics exactly the proof of Lemma 1 except that now one multiplies by $1/|\Sigma|$ for each character that must be matched. For example, for the case $T = 1$ and $D > 1$, each of the $|\Sigma|^D$ insertion scripts produce a word that matches with probability $|\Sigma|^{-D}$, and each of the $(|\Sigma| - 1)|\Sigma|^{D-1}$ scripts that substitute and insert produce words that also match with this probability. Thus the sum of the probabilities is $|\Sigma|^D/|\Sigma|^D + (|\Sigma| - 1)|\Sigma|^{D-1}/|\Sigma|^D = (2|\Sigma| - 1)/|\Sigma|$. As the second and final example, consider the recursive formula for Q. If one does nothing at the first position then it will match with probability $1/|\Sigma|$ and the extensions, words obtained by performing D operations on the $T-1$ other positions, will match with probability less than $Q(T-1, D)$. This yields the $Q(T-1, D)/|\Sigma|$ term in the recurrence. If one deletes the first symbol then one must match a word obtained by performing $D-1$ operations on the other $T-1$ positions and this happens with probability less than $Q(T-1, D-1)$. Inserting j symbols gives $|\Sigma|^j$ new symbols which along with the first position match with probability $1/|\Sigma|^{j+1}$. The extensions match with probability less than $Q(T-1, D-j)$ for a total contribution of $Q(T-1, D-j)/|\Sigma|$. Finally, the substitute and $j-1$ insert case yields a contribution of $Q(T-1, D-j)(|\Sigma| - 1)/|\Sigma|$ to the bound. Summing the four cases and doing a bit of algebraic simplification gives the central recurrence of the Lemma. ■

With these recurrences in hand, the bounding expressions for Z and Pr given in Lemma 3 can easily be verified. Slightly tighter bounds for Pr are possible but not necessary since their use does not improve the complexity analysis which is dominated by the expression for Z.

Lemma 3. Let $Bnd(T, D, c) = (\frac{c+1}{c-1})^T c^D |\Sigma|^D$. For all $c > 1$,

$$N(T, D) \le Bnd(T, D, c)\; Z(T, D) \le \frac{c}{c-1} Bnd(T, D, c)$$

$$Q(T, D) \le Bnd(T, D, c)/|\Sigma|^T Pr(T, D) \le \frac{c}{c-1} Bnd(T, D, c)/|\Sigma|^T$$

Proof. A simple induction using Lemmas 1 and 2 suffices to verify the correctness of the bounds. ■

In the analyses of the algorithmic components that follow, T will be $\log_{|\Sigma|} N$ or a multiple thereof, where N is the size of the database being searched. Letting $\varepsilon = D/T$ be the permissible mismatch ratio, Lemma 4 below shows that both the Z and Pr quantities are proportional to a power of N that is a concave increasing function of ε.

Lemma 4. For $T = \log_{|\Sigma|} N$ and $D \le T$,

$$Z(T, D) < 2N^{pow(D/T)} \text{ and } Pr(T, D) < 2N^{pow(D/T)-1}$$

$$\text{where } pow(\varepsilon) = \log_{|\Sigma|} \frac{c+1}{c-1} + \varepsilon \log_{|\Sigma|} c + \varepsilon \text{ and } c = \varepsilon^{-1} + \sqrt{1+\varepsilon^{-2}}.$$

Proof. When $T = \log_{|\Sigma|} N$ and $\varepsilon = D/T$, some algebraic manipulation shows that $Bnd(T, D, c) = Bnd(log_{|\Sigma|} N, \varepsilon \log_{|\Sigma|} N, c) = N^{\alpha(\varepsilon, c)}$ where $\alpha(\varepsilon, c) = \log_{|\Sigma|} \frac{c+1}{c-1} + \varepsilon \log_{|\Sigma|} c + \varepsilon$. A straightforward application of calculus further shows that the value of $\alpha(\varepsilon, c)$ is minimized when $c = \varepsilon^{-1} + \sqrt{1+\varepsilon^{-2}}$. In the statement of the lemma we let $pow(\varepsilon) = \alpha(\varepsilon, c)$ for this choice of c. Since ε ranges from 0 to 1, it follows that c ranges from $1+\sqrt{2}$ and up, and thus $c/(c-1)$ is always less than 2. Thus it follows that $Z(T, D) < 2N^{pow(D/T)}$. The bound for Pr follows easily from the final observation that $|\Sigma|^{\log_{|\Sigma|} N} = N$. ■

Figure 4 shows a plot of $pow(\varepsilon)$ for Σ of sizes 4 and 20. Note that $pow(\varepsilon) \le 1$ for $\varepsilon \le .3303$ and $\varepsilon \le .5671$ for these two choices of $|\Sigma|$. Further note that the function is concave and for a fixed choice of c, $\alpha(\varepsilon, c)$ is an affine function of ε that bounds $pow(\varepsilon)$ and is a tangent of the curve. For example, when $|\Sigma| = 4$, $pow(\varepsilon) \le \alpha(\varepsilon, 6.520) = .2230 + 2.352\varepsilon$. When $|\Sigma| = 20$, $pow(\varepsilon) \le \alpha(\varepsilon, 3.971) = .1718 + 1.460\varepsilon$. The bounding lines in these two examples are plotted in Figure 4. The value of c chosen for each line was that for which $\alpha(\varepsilon, c) = 1$ exactly when $pow(\varepsilon) = 1$.

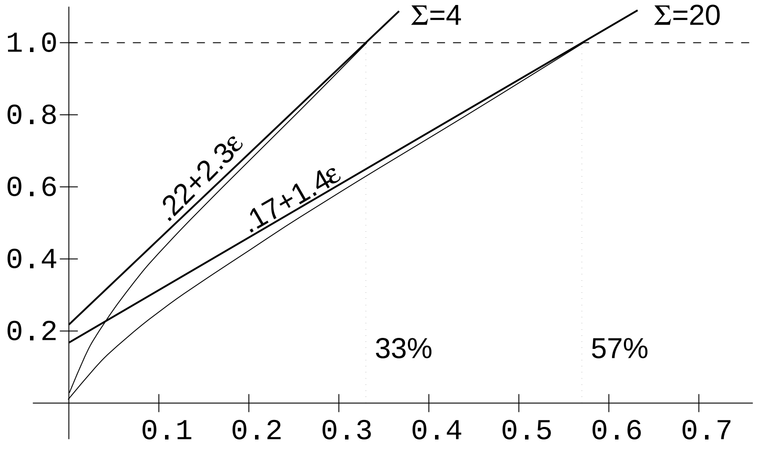


Figure 4: Plot of $pow(\varepsilon)$ and Sample Bounding Lines

As noted in Section 2, T must be chosen to be an integer and so, in general, one must round $\log_{|\Sigma|} N$ up or down. Considering $|\Sigma|$ as a factor in the complexity, rounding up can increase $Z(T, D)$ by a factor of $|\Sigma|^{pow(\varepsilon)}$ and decrease $Pr(T, D)$ by the same amount. Rounding down can decrease neighborhood size but increase match likelihood by the same factor. While in theory this is not important since Σ is assumed to be of a fixed and finite size, in practice we choose to round up for several reasons. As will be seen in the next section, this has the effect of increasing neighborhood generation *time* some (Z is larger) but decreases the *space* consumed by the record of positions matched in the database (*Pr* is smaller). So our first reason to round up, is that we prefer to trade time (which is unbounded) for space (which is bounded). Secondly, the next phase of the algorithm requires $O(D \log N)$ time per match versus the $O(D)$ time spent per neighborhood word. Thus reducing the number of matches is more desirable than reducing the number of words generated. Finally, we have shown in Section 2 that we can conveniently accommodate rounding up for the index structure without increasing the time or space complexity of this facet. With this said, we henceforth assume $|\Sigma|$ is a constant when expressing asymptotic complexity claims.

5. Finding Hits with the Index

This subsection deals with the details of combining word generation with index lookups and characterizes the complexity and results of this first phase of the total algorithm. Consider the following statement of this first phase problem. One is given a database $A = a_1 a_2 \cdots a_N$, a word W of length $T = \lceil \log_{|\Sigma|} N \rceil$, and a threshold D. Let $Score_W(i) = \min_{h \geq i} \{ \delta(a_i a_{i+1} \cdots a_h, W) \}$. That is, $Score_W(i)$ is the score of the closest word to W that begins at position i of the database. Also let $Hits_W(D) = \{ i : Score_W(i) \leq D \}$, i.e., the set of all positions in the database where a word in the D-neighborhood of W begins. The task is to compute $Hits_W(D)$.

The solution is to simply run the generator algorithm of Figure 3 and as each word is generated, to look up the indices of the left ends of all occurrences of this word in A with the index. This set of positions, $\{ i : \exists V \in \overline{N_D}(W), V = a_i a_{i+1} \cdots a_{i+|V|-1} \}$, is exactly the desired set $Hits_W(D)$. This follows because if $Score_W(i) \leq D$ then there is some word in the D-neighborhood whose leftmost character is at index i, and certainly some prefix of this word is in the condensed D-neighborhood.

Looking up a word V in the neighborhood takes $O(T + DH_V)$ expected time. Recall that the $O(T)$ term is for computing the ϕ (ρ) code, and that the $O(DH_V)$ term is for verifying the H_V instances found in the relevant index bucket. If realized exactly as described above the time to find all occurrences of all words in the neighborhood would thus be $O(TZ + DH)$ expected-time where Z is the size of the condensed neighborhood and $H = |Hits_W(D)|$ is the number of hits. However, the T-term is eliminated by noting that codes can be generated in parallel with the neighborhood words. That is, as each character is added to the string V in Figure 3, the ϕ-code is easily updated using its defining recurrence, $\phi(Va) = \phi(V)|\Sigma| + \phi(a)$. Moreover, for those words that consist of concatenating the current word with a suffix of W in line 14 of Figure 3, one does not need to explicitly perform the concatenation to do the lookup. If the current word, V, is of length less than T, than consulting a precomputed, *(T+1)*-element table of the codes of every prefix of W allows the needed code to be delivered in $O(1)$ time.† Whatever suffix of W remains is then used in checking for matches at each position in the appropriate bucket. Thus, as promised earlier, all words, as well as their codes are effectively computed in $O(DZ + T)$ worst-case time and the appropriate buckets of the index are checked for hits in $O(DH)$ expected time.

† If in line 14 the neighborhood word is $V[1..i] \bullet W^j$ and $i < T$ then we need the code for $V \bullet W[j+1..j+(T-i)]$ and the remainder, suffix $W^{j+(T-i)-1}$, must be checked against bucket positions (or suffix $W^{j+(T-i)}$ in the case that the parameter $R = 0$ for the index structure). But the needed code is simply $\phi(V)|\Sigma|^{T-i} + \phi(W[j+1..j+(T-i)])$ and with the table of codes of every prefix of W, the

Lemma 5. Given a precomputed index as described in Section 2, there exists an algorithm to compute $Hits_W(D)$ in $O(DN^{pow(D/T)} + T)$ expected-time and $H < 2N^{pow(D/T)}$.

Proof. From Lemma 4 it follows that for a database of size N, H is on average $Pr(T, D) \times N < 2N^{pow(D/T)} / \kappa^{1-pow(D/T)}$ where $\kappa = |\Sigma|^{T - \log_{|\Sigma|} N} \in [1, |\Sigma|]$. Thus the result on the size of H follows. Lemma 4 also asserts that Z is $O(N^{pow(D/T)})$. But then the result immediately follows as the procedure just described takes $O(D(Z+H) + T)$ expected-time. ■

As will be subsequently seen, we will also need to solve a "reverse" version of the first phase problem. Specifically, let $\overline{Score}_W(i) = \min_{h \le i} \{ \delta(a_h a_{h+1} \cdots a_i, W) \}$ and let $\overline{Hits}_W(D) = \{ i : \overline{Score}_W(i) \le D \}$. These quantities are analogous to their unbarred counterparts above except that they address where matches *end* as oppose to where they *begin*. Computing $\overline{Hits}_W(D)$ requires a simple modification of the word generator and index lookup. The key observation is that $\overline{Hits}_W(D) = \{ i : \exists V \in \overline{N}_D(W^R), V^R = a_{i-|V|+1} a_{i-|V|+2} \cdots a_i \}$ where W^R, the reverse of W, is $w_T w_{T-1} \cdots w_1$.‡ Thus it suffices to generate the condensed neighborhood of the reverse of W and then lookup the positions at which the reverse of the neighborhood words match A. The one subtlety is that we have an index for left-to-right matching and we cannot afford the time to reverse a neighborhood word. This is easily solved by computing the "forward" codes of the reverse words as they are generated by observing that $\phi((Va)^R) = \phi(aV) = \phi(a)|\Sigma|^{|V|} + \phi(V)$. In addition, the "concatenation" problem for line 14 of Figure 3 can be solved with the $O(T)$ table of prefix codes in a fashion similar to that proposed above. Thus we can find the left end of a match to the reverse of a neighborhood word at no additional overhead. For each such location i at which V^R matches, we need simply record in $\overline{Hits}_W(D)$ the right end of the match, $i + |V| - 1$.

6. Extending Hits

We now turn to the problem of handling a query W of length $P \gg T = \lceil \log_{|\Sigma|} N \rceil$. Throughout this section we will assume that P/T is a power of 2, i.e., $P = 2^K T$ for some K. The case where it is not will be treated at the conclusion of this section. To begin, we review traditional dynamic programming approaches and their graph-theoretic interpretation as finding shortest paths in an edit graph. With this machinery we prove the decomposition Lemma described in the overview. Finally, we show how to apply this lemma to finding all approximate matches to W in the database A.

As noted at the start of Section 3, the comparison of word W against another word A can be achieved with the computation of a dynamic programming matrix $L[0..N, 0..P]$ where $L[i, j] = \delta(A_i, W_j) = \min\{ L[i, j-1]+1, L[i-1, j]+1, L[i-1, j-1] + (if\ a_i = w_j\ then\ 0\ else\ 1) \}$ for $i, j > 0$. For the cases where i or j is zero, we have $L[i, 0] = L[0, i] = i$. From a graph-

code for $W[k..h]$ is simply $\phi(W_h) - \phi(W_{k-1})|\Sigma|^{h-k+1}$.

‡ Care must be taken here to realize that $\{ V^R : V \in \overline{N}_D(W^R) \}$ is not equal to $\overline{N}_D(W)$. On the other hand, equality does hold in the case of $N_D(W)$.

theoretic perspective, we can also view the problem as follows. Given A and W, construct a graph with vertices (i, j) for $i \in [0, N]$ and $j \in [0, P]$, arranged in a $N+1$ by $P+1$ grid or matrix as illustrated in Figure 5. For vertex $(i\ j)$ there are up to three edges directed out of it: (1) a *deletion* edge to $(i+1, j)$ (iff $i < N$), (2) an *insertion* edge to $(i, j+1)$ (iff $j < P$), and (3) an *alignment* edge to $(i+1, j+1)$ (iff $i < N$ and $j < P$). In the resulting *edit graph*, all paths from *source* vertex $(0, 0)$ to *sink* vertex (N, P) model the set of all possible alignments between A and W with the following simple interpretation: a deletion edge to (i, j) models leaving a_i unaligned, an insertion edge to (i, j) models leaving w_j unaligned, and an alignment edge to (i, j) models aligning a_i and w_j. If one weights deletion and insertion edges 1 and alignment edges 0 or 1 according to whether a_i equals w_j, then the problem of finding a minimal cost alignment between A and W is equivalent to finding a minimum cost source to sink path in the corresponding edit graph. The correlation to the matrix L is that $L[i, j]$ is the cost of the minimum path from the source to (i, j). Since the edit graph is acyclic, the shortest paths to each vertex may be computed in any topological order of the vertices using the recurrence defining L.

The preceding treatment was for the problem of comparing all of A against all of W. For approximate keyword searching, we seek substrings of A that align to W with less than D differences. In the respective edit graph, a d-path (i.e., a path of cost d) from vertex $(i, 0)$ to vertex (j, P), models an alignment between $A[i+1..j]$ and W with d differences. Thus, we are seeking paths from "row" 0 to "row" P whose cost is not greater than D. That is, in this version of the problem, any vertex with $j = 0$ can be a potential source vertex, and any vertex with $j = P$, a potential sink. We can accommodate this shift by simply changing the boundary for the recurrence for L to be $L[i, 0] = 0$, i.e., all values along row 0 are set to zero. With this modification it is easy to show that $L[i, j] = \min_h \{\ \delta(A[h..i], W_j)\ \}$, the shortest path to (i, j) from *some* vertex in row 0. Thus, a_i is the right end of an approximate match iff $L[i, P] \le D$. In the treatment that follows, we will be computing the matrix L for a number of different query words. Thus, we let F_W be the approximate match matrix for query W. We use F to denote "forward", for there will also be occasion to view this problem in its reverse sense. Namely, let $R_W[i, j] = \min_h \{\ \delta(A[i+1..h], W^j)\ \}$, the shortest path from (i, j) to some vertex in row P. In this case, the recurrence for R_W is by analogy seen to be $R_W[i, j] = \min\{\ R_W[i, j+1]+1,\ R_W[i+1, j]+1,\ R_W[i+1, j+1] + (\mathit{if}\ a_{i+1} = w_{j+1}\ \mathit{then}\ 0\ \mathit{else}\ 1)\ \}$ for $i < N$ and $j < P$. The boundaries are given by $R_W[N, j] = P - j$ and $R_W[i, P] = 0$. Note that in this case, a_{i+1} is the *left* end of an approximate match iff $R_W[i, 0] \le D$.

Let *diagonal* k of an edit graph be the set of vertices $\{\ (i, j) : i - j = k\ \}$. Note that a d-path that begins or ends in diagonal k must lie entirely between diagonals $k - d$ and $k + d$ as it requires a deletion or insertion to move from one diagonal to another. In the algorithm that follows there will be occasion to determine if there is a D-path from row 0 to row P lying between two diagonals $k < h$. Note that it suffices to apply either the forward or reverse recurrence over just the vertices lying in the parallelogram shaped region between the diagonals, i.e., if the value at a vertex between the diagonals depends on the value of a vertex outside the diagonals, simply ignore that vertex's contribution to the 3-way minimum of the recurrence. Let $F_W^{k,\,h}$ denote the values of the forward recurrence when evaluated over just this region of the edit graph. Naturally $F_W^{k,\,h}[i,\,j]$ is the value of a minimum cost path to $(i,\,j)$ over all paths lying between diagonals k and h. Thus there is a D-path from row 0 to row P between diagonals k and h iff $R_W^{k,\,h}[i,\,0] \le D$ for some $i \in [k,\,h]$, or iff $F_W^{k,\,h}[i,\,P] \le D$ for some $i \in [k+P,\,h+P]$.

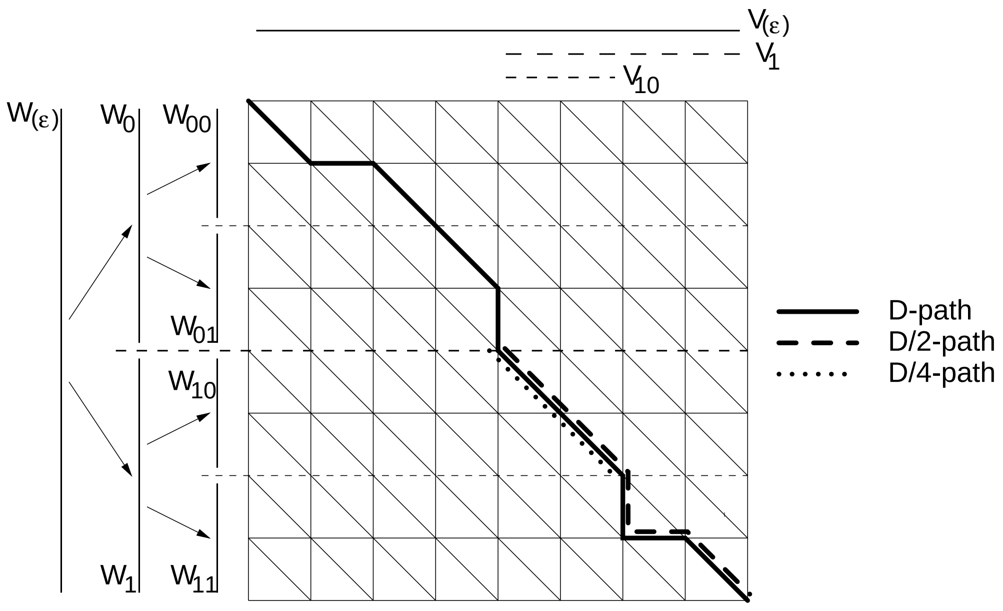


Figure 5: A Sample Edit Graph and Illustration for Lemma 6

With these preliminaries, we now proceed to the central lemma which we lever to efficiently extend matches to subwords of W of length T, to approximate matches to all of W. Consider dividing the query W in a binary fashion until all pieces are of size $T = \lceil \log_{|\Sigma|} N \rceil$ (recall we are assuming $P = 2^K T$). To model the various pieces, let W_α for $\alpha \in \{0,\,1\}^*$ be recursively defined by the equations: $W_\varepsilon = W$, $W_{\alpha 0} = W_\alpha[1..|W_\alpha|/2]$ (the first half of W_α), and $W_{\alpha 1} = W_\alpha[|W_\alpha|/2+1..|W_\alpha|]$ (the second half of W_α). Figure 5 illustrates the decomposition as well as the proof of Lemma 6 below. Note that for a given length $k \le K$, there are 2^k distinct labels α of that length and the strings W_α are all of length $2^{K-k}T = P/2^k$. Let $P_\alpha = P/2^{|\alpha|}$ denote the length of W_α and let $D_\alpha = \lfloor D/2^{|\alpha|} \rfloor$ be the match stringency to W_α required in Lemma 6 below.

Lemma 6. If W aligns to a string V with not more than D differences, then there exists a word α such that for every prefix β of α, W_β aligns to a substring V_β of V with not more than D_β differences. Moreover, $V_{\beta a}$ is a prefix or a suffix of V_β according to whether a is 0 or 1.

Proof. We show that under the hypothesis it follows that either W_0 or W_1 aligns to a prefix or suffix, respectively, of V with not more than $\lfloor D/2 \rfloor$ differences. Applying this observations inductively gives the result. If W aligns to V with not more than D differences, than as illustrated in Figure 5 there is a source to sink path of cost D-or-less in the corresponding edit graph. The path passes through one or more vertices on row $P/2$. Consider the two subpaths consisting of that part of the path from the source to its first vertex in row $P/2$ and the part from its last vertex in row $P/2$ to the sink. By the Pigeon Hole Principle, one of these two subpaths must have cost not greater than $\lfloor D/2 \rfloor$. If its true for the first part, then simply observe that the subpath aligns W_0 and a prefix of V. If its true for the later subpath, then there is an alignment of cost $\lfloor D/2 \rfloor$-or-less between W_1 and a suffix of V. ■

Note that the conclusion of Lemma 6 may be rephrased as: there exists a label α such that for every prefix β of α, W_β ε-matches a substring V_β of V. This follows simply because for each β the mismatch ratio of the match between W_β and V_β is $D_\beta / |W_\beta| = \lfloor D/2^{|\beta|} \rfloor / (P/2^{|\beta|}) \leq D/P = \varepsilon$. Moreover, while the induction of the proof yields the conclusion on progressively smaller subwords, the Lemma gives the strategy for extending approximate matches to progressively larger subwords of W. For example, if one finds an ε-match to W_{01011}, then one checks for an ε-match to W_{0101} and if successful then one checks for an ε-match to W_{010}, and so on, until either one fails to match at some level or succeeds in matching all of $W = W_\varepsilon$. If this extension strategy is applied to all ε-matches to all subwords W_α of length T, then one is guaranteed to detect all ε-matches to W by the Lemma.

The one difficulty in applying the extension strategy is that in a region where there is a particularly stringent match, one can spend an excessive amount of time if one proceeds one match at a time. For example, if $F_W[i, P] = 0$, then it is guaranteed that $F_W[i \pm d, P] \leq d$ because entries along a row or column of the dynamic programming matrix change by -1, 0, or 1. That is, if there is a 0-match at a particular position, then there are guaranteed to be $2D$ additional matches in the immediate neighborhood. Moreover, since W exactly matches this location, then W_α matches at corresponding locations for every α. Each of these exact subword matches implies D_α matches immediately about them all of which if extended individually would uncover the same matches at the level above. Extending each match of length T would result in the $O(D)$ matches to W being discovered $O(D^K)$ times. So clearly, one must accumulate the matches at each level before proceeding to the next. Moreover, one must pursue extensions of adjacent matches simultaneously as otherwise the $O(D)$ matches at the upper level will be uncovered $O(D^2)$ times by the level below.

Let $F_\alpha = \{ i : F_{W_\alpha}[i, P_\alpha] \leq D_\alpha \}$, the set of positions at which an ε-match to W_α ends. Similarly, let $R_\alpha = \{ i : R_{W_\alpha}[i, 0] \leq D_\alpha \}$, the set of positions at which an ε-match to W_α begins. Our goal is to compute a representation of either F_α or R_α for all α in decreasing order of label length. Certainly this suffices because F_ε is the set of right ends of ε-matches to W and R_ε gives the left ends. Term a list of ordered pairs $C = <(l_1, u_1), (l_2, u_2), \cdots (l_n, u_n)>$ where $l_k \leq u_k < l_{k+1}$, a *covering list* of set X if and only if $\bigcup_{k=1}^{n} [l_k, u_k] \supseteq X$. For each α of length less than K our algorithm computes a covering list for $F_{\alpha 0}$ and one for $R_{\alpha 1}$. Moreover, these coverings are parsimonious in that if C covers F_α (R_α) then $l_k, u_k \in F_\alpha$ (R_α) and $l_{k+1} - u_k > D_\alpha + 1$ for all k. Note that these additional conditions uniquely determine the covering list. Furthermore, because values change along a row by -1, 0, or 1, it follows that for every pair (l, u) either $F_{W_\alpha}[l, P_\alpha] = F_{W_\alpha}[u, P_\alpha] = D_\alpha$ or $R_{W_\alpha}[l, 0] =$ and $R_{W_\alpha}[u, 0] = D_\alpha$ depending on whether the list covers F_α or R_α.

```
1.   function FGEN($w, d$)
2.     { $S \leftarrow \overline{Hits}_w(d)$
3        Sort $S$
4.       $G \leftarrow \emptyset$
5.       $u \leftarrow -d-2$
6.       for $k \in S$ in increasing order do
7.         { if $k-d > u+1$ then
8.             { if $u \geq 0$ then $G \leftarrow G \bullet (l, u)$
9.                 $l \leftarrow k$
10.            }
11.            $u \leftarrow k$
12.        }
13.      if $u \geq 0$ then $G \leftarrow G \bullet (l, u)$
14.      return $G$
15.    }
```

Figure 6: Generating F_α's covering list when $|\alpha| = K$

Initially, the process is started by computing coverings for F_α (R_α) where $P_\alpha = T$ using the generator algorithm of Lemma 5 as a subroutine. Simply observe that $F_\alpha = \{ i : \min_{h \leq i}\{ \delta(a_h a_{h+1} \cdots a_i, W_\alpha) \} \leq D \} = \overline{Hits}_{W_\alpha}(D_\alpha)$. Thus producing a covering list for F_α consists of simply invoking the reverse generator to compute $\overline{Hits}_{W_\alpha}(D_\alpha)$, sorting the resulting set of indices with any $O(H\log H)$ sort, and then producing the desired covering list in a simple $O(H)$ scan of the sorted index list. This process is encapsulated in the procedure $FGEN(w, d)$ in Figure 6 above. Computing a covering for R_α follows analogously with the observation that it equals $Hits_{W_\alpha}(D_\alpha) - 1$ where the notation $X-1$ denotes $\{ i-1 : i \in X \}$. Assume that procedure $RGEN(w, d)$ computes coverings for R-sets for subwords of W of length T.

With basis of the induction handled by the generator algorithms, we now turn to the induction: given covering lists for $F_{\alpha 0}$ and $R_{\alpha 1}$, how do we compute a covering list for F_α (R_α)? Consider the edit graph for W_α versus A and a path between rows 0 and P_α of cost no greater than D_α. Suppose this path passes through row $P_\alpha / 2 = P_{\alpha 0} = P_{\alpha 1}$ at vertex $(i, P_{\alpha 0})$. Then by Lemma 6, i must be a member of either $F_{\alpha 0}$ or $R_{\alpha 1}$ and hence covered by a pair (l, u) of the appropriate covering list. We show that the entire path must lie between diagonals $l - P_{\alpha 0} - \Delta_\alpha$ and $u - P_{\alpha 0} + \Delta_\alpha$ where $\Delta_\alpha = D_\alpha - D_{\alpha 0}$. Suppose that $i \in F_{\alpha 0}$; the case where $i \in R_{\alpha 1}$ is entirely symmetric. This implies that the first part of the path from row 0 to vertex $(i, P_{\alpha 0})$ costs $D_{\alpha 0}$ or less, say its d. As observed earlier, since this d-subpath ends in diagonal $i - P_{\alpha 0}$, the entire subpath must lie between diagonals $i - P_{\alpha 0} - d$ and $i - P_{\alpha 0} + d$. But since $i \in [l, u]$ and $D_{\alpha 0} \leq \Delta_\alpha$ it follows that the first part of the path lies between the desired diagonals. Now, the remainder of the path from $(i, P_{\alpha 0})$ to row P_α has cost no greater than $D_\alpha - d$. As noted when covering lists were introduced, $F_{W_\alpha}[x, P_{\alpha 0}] = F_{W_{\alpha 0}}[x, P_{\alpha 0}] = D_{\alpha 0}$ for $x = l$ and $x = u$. Moreover, since values change by -1, 0, or 1 along a given row it follows that $d = F_{W_\alpha}[i, P_{\alpha 0}]$ cannot be less than $F_{W_\alpha}[l, P_{\alpha 0}] - (i-l)$ nor less than $F_{W_\alpha}[u, P_{\alpha 0}] - (u-i)$. Thus $d \geq D_{\alpha 0} - \min\{i-l, u-i\}$. But then it follows that the second part of the path must lie between diagonals $i - P_{\alpha 0} \pm (D_\alpha - d) \subseteq i - P_{\alpha 0} \pm (D_\alpha - (D_{\alpha 0} - \min\{i-l, u-i\})) = i - P_{\alpha 0} \pm (\Delta_\alpha + \min\{i-l, u-i\}) \subseteq [l - P_{\alpha 0} - \Delta_\alpha , u - P_{\alpha 0} + \Delta_\alpha]$.

```
1.  function UNION($L_1, L_2, d, p$)
2.    { $G \leftarrow \emptyset$
3.      $u \leftarrow -d-2$
4.      while $L_1 \neq \emptyset$ or $L_2 \neq \emptyset$ do
5.        { if $L_2 = \emptyset$ or $head(L_1).l < head(L_2).l$ then
6.              $(i, j) \leftarrow pop(L_1)$
7.          else
8.              $(i, j) \leftarrow pop(L_2)$
9.          if $i-d > u+1$ then
10.           { if $u \geq p$ and $l \leq N-p$ then $G \leftarrow G \bullet (l-p, u-p)$
11.             $l \leftarrow i-d$
12.           }
13.         $u \leftarrow j+d$
14.       }
15.     if $u \geq p$ and $l \leq N-p$ then $G \leftarrow G \bullet (l-p, u-p)$
16.     return $G$
17.   }
```

Figure 7: Merging Covering Lists

In the last paragraph we showed that if there is a path between rows 0 and P_α of cost D_α or less in the edit graph of W_α versus A, then it must lie entirely between diagonals $l - P_{\alpha 0} - \Delta_\alpha$ and $u - P_{\alpha 0} + \Delta_\alpha$ for some pair (l, u) in the covering list of $F_{\alpha 0}$ or $R_{\alpha 1}$. Let L_α be the covering list of $\cup \{ [l - P_{\alpha 0} - \Delta_\alpha, u - P_{\alpha 0} + \Delta_\alpha] : (l, u) \in F_{\alpha 0} \cup R_{\alpha 1} \}$ that is as parsimonious is possible, i.e., $span(L_\alpha) = \sum_{k=1}^{n} | u_k - l_k + 1|$ is minimal and among those lists whose span is minimal, L_α's cardinality, n, is smallest. This list is computable in time linear in the size of the lists $F_{\alpha 0}$ and $R_{\alpha 1}$ with the call $UNION(F_{\alpha 0}, R_{\alpha 1}, \Delta_\alpha, P_{\alpha 0})$ to the subroutine *UNION* shown in Figure 7. It consists of simply merging the two ordered lists while "expanding" each pair by Δ_α and "translating" each by $-P_{\alpha 0}$. Transformed pairs whose intervals overlap are fused into a single pair representing the combined interval.

From the two preceding paragraphs it follows that to compute F_α it suffices to compute F_{W_α} only between those diagonals given by pairs of the list L_α. Formally, $F_\alpha = \bigcup_{(l, u) \in L_\alpha} \{ i : F^{l,u}_{W_\alpha}[i, P_\alpha] \leq D_\alpha \}$. Replacing F with R gives the analogous result for R_α. Let $FSCAN(L, w, d)$ be a procedure that computes $F^{l,u}_{w}$ for each pair (l, u) on the list L and examines the last row of this computation to build a covering list of the entries that are less than d. The algorithm is sketched in Figure 8 primarily to confirm the details of the covering list construction. The time required by this procedure is $O(|w| \times span(L))$ and the space required is $O(max\{ l-u+1 : (l, u) \in L \})$. The space requirement could be $O(N)$ in the worst-case but in expectation it is $O(D)$. By virtue of the preceding remarks it follows that the list returned by the call $FSCAN(L_\alpha, W_\alpha, D_\alpha)$ is a covering list for F_α. We assume an analogous procedure $RSCAN(L, w, d)$ that computes a covering list for R-sets.

```
1.  function FSCAN($L, w, d$)
2.    { $G \leftarrow \emptyset$
3.      $u \leftarrow -d-2$
4.      while $L \neq \emptyset$ do
5.        { $(i, j) \leftarrow pop(L)$
6.          Compute vector $F_w^{i,j}[?, p = |w|]$
7.          for $k \leftarrow i+p$ to $\min\{j+p, N\}$ do
8.              if $F_w^{i,j}[k, p] \leq d$ then
9.                { if $k-d > u+1$ then
10.                   { if $u \geq 0$ then $G \leftarrow G \bullet (l, u)$
11.                     $l \leftarrow k$
12.                   }
13.                 $u \leftarrow k$
14.               }
15.       }
16.     if $u \geq 0$ then $G \leftarrow G \bullet (l, u)$
17.     return $G$
18.   }
```

Figure 8: Generating F_α's covering list from L_α when $|\alpha| < K$

The overall algorithm in terms of the subprocedures — $F(R)\,GEN$, $F(R)\,SCAN$, and *UNION* — is given in Figure 9. The recursive procedure $LIST(\alpha)$ returns a pointer to the covering list L_α. When $|\alpha| = K-1$, it does so by generating covering lists for $F_{\alpha 0}$ and $R_{\alpha 1}$ with the appropriate calls to the neighborhood-based algorithms *FGEN* and *RGEN*. It then combines these to form L_α with a call to *UNION*. When $|\alpha| < K-1$, the difference is that recursive calls to *LIST* produce $L_{\alpha 0}$ and $L_{\alpha 1}$ which are then used by *FSCAN* and *RSCAN* to produce the covering lists for $F_{\alpha 0}$ and $R_{\alpha 1}$. At the top level, when L_ε is returned it suffices to call *RSCAN* to

```
list of pairs $G$

1.  function LIST($\alpha$)
2.    { list of pairs $H$
3.      if $|\alpha| = K-1$ then
4.        { $H \leftarrow$ FGEN($W_{\alpha 0}, D_{\alpha 0}$)
5.          $G \leftarrow$ RGEN($W_{\alpha 1}, D_{\alpha 1}$)
6.        }
7.      else
8.        { $H \leftarrow$ FSCAN(LIST($\alpha 0$), $W_{\alpha 0}, D_{\alpha 0}$)
9.          $G \leftarrow$ RSCAN(LIST($\alpha 1$), $W_{\alpha 1}, D_{\alpha 1}$)
10.       }
11.     return UNION($G, H, \Delta_\alpha, P_{\alpha 0}$)
12.   }

13. $G \leftarrow$ LIST($\varepsilon$)
14. Report intervals in RSCAN($G, W, D$)
```

Figure 9: The Sublinear Algorithm

obtain a covering list of the positions at which approximate matches to W begin. A call to *FSCAN* would produce a covering list of the right ends. For these top level calls to the *SCAN* routines, one should remove the term $-d$ in line 9 of Figure 8 in order to produce covering lists whose covered positions are exactly the indices at which approximate matches begin or end.

The various covering lists are assumed to be implemented as simple linked lists of integer pairs. Note that each of the subroutines presented in Figures 6, 7, and 8 are careful to consume (via *pop*s) their input lists as they produce their resultant lists. Thus the algorithm of Figure 9 is carefully structured so that at a given instance there is never more than the list G of the current recurrence level, and one list H pending a *UNION* at each level of the recurrence. This feature is very important to the space requirement of the algorithm proven in Lemma 7.

Lemma 7. Given that A is the result of equi-probable Bernouilli trials and that $pow(\varepsilon) \leq 1$, the algorithm of Figure 9 in expectation takes $O(DN^{pow(\varepsilon)} \log N + P)$ time and $O(N^{pow(\varepsilon)} + P)$ working space (excludes the index). If A does have matches to W, then in the worst case $O(DP)$ time is spent on each of these occurrences.

Proof. First, consider the time spent in calls to *FGEN* and *RGEN*. The number of calls to each is $P/2T$. Observe that when $|\alpha| = K$, D_α equals $\sigma = \lfloor \varepsilon T \rfloor$. By Lemma 5, it takes $O((\sigma+1)N^{pow(\sigma/T)}+T)$ expected time to produce S in line 2 and it is of size $N^{pow(\sigma/T)}$. Further sorting S and producing the covering list takes $O(N^{pow(\sigma/T)} \log N^{pow(\sigma/T)})$ additional time. There are two cases to consider. First, if $\varepsilon < 1/T$ then $\sigma = 0$ and $N^{pow(\sigma/T)} = N^{pow(0)} = N^0 = 1$. Thus in this case, each call to *FGEN* or *RGEN* takes $O(T)$ time om expectation, for a total over all P/T calls of $O(P)$ time. For the other case where $\varepsilon \geq 1/T$, note that $P/T \leq \varepsilon P = D$. Thus the time taken in line 2 over all calls is $O(P/T\,(\sigma N^{pow(\sigma/T)}+T)) = O(P/T\,(\varepsilon TN^{pow(\varepsilon)}+T)) = O(\varepsilon PN^{pow(\varepsilon)}+P) = O(DN^{pow(\varepsilon)}+P)$. The time taken for the sort and covering list construction is $O(P/T\,(N^{pow(\sigma/T)} \log N^{pow(\sigma/T)}))$ $= O(DN^{pow(\varepsilon)} \log N)$. Thus, the total expected time spent in the *GEN* subroutines is within the bound of the Lemma.

The time spent in a call to *UNION* is linear in the sizes of the lists produced by the corresponding calls to *FSCAN* and *RSCAN*. Thus the total time spent in calls to the *SCAN* routines dominates the time spent in *UNION*. Extending the proof of Lemma 4, observe that $Bnd(\pi \log_{|\Sigma|}N,\, \pi\, \varepsilon \log_{|\Sigma|}N,\, c) = N^{\pi\alpha(\varepsilon,\, c)}$ and thus $Pr(\pi T, \pi D) < 2N^{\pi(pos(D/T)-1)}$. Thus it follows that the expected number of approximate matches to a word of length P_α with no more than D_α differences is less than $N \times Pr(2^a T,\, 2^a \varepsilon T) \leq 2N/N^{2^a(1-pow(\varepsilon))}$ where a denotes $K-|\alpha|$. Now a covering list for F_α or R_α has this many elements in expectation, each pair giving a $O(D_\alpha)$ width interval about an approximate match. Because L_α is the union of $F_{\alpha 0}$ and $R_{\alpha 1}$ it follows that it has less than $4N/N^{2^{a-1}(1-pow(\varepsilon))}$ intervals of expected width $O(D_\alpha)$. Thus, the expected time spent in a call to $FSCAN(L_\alpha, W_\alpha, D_\alpha)$ is $O(|L_\alpha|\, D_\alpha\, P_\alpha) =$ $O(\varepsilon NT^2 4^a/N^{2^{a-1}(1-pow(\varepsilon))})$. The same amount of time is spent in a call to *RSCAN* and there are a total of $P/(2^a T)$ such calls made on words of length P_α. Thus the total time spent on words of length P_α is $O(DNT(2^a/N^{2^{a-1}(1-pow(\varepsilon))})$. Over the entire course of the algorithm, a runs from 1 to K, so the total time spent in calls to *FSCAN* and *RSCAN* is

$O(DNT \sum_{c=0}^{K-1} 2^c / N^{2^c(1-pow(\varepsilon))})$. For ε in the range of interest we can assume $N^{1-pow(\varepsilon)} > 2$ and so the progression of terms in the summation above approach zero hyperexponentially. Thus the sum is asymptotically dominated by the first term and we can conclude that in expectation $O(DNT/N^{1-pow(\varepsilon)}) = O(DN^{pow(\varepsilon)} \log N)$ time is spent extending hits.

As stated in the first paragraph of the proof, the size of the ''Hit list'' S used in RGEN and FGEN is $N^{pow(\sigma/T)} = O(N^{pow(\varepsilon)})$. Thus the bound on space is observed at the lowest level of the recursion, since only one S is in existence at any given time. At an arbitrary point in the computation there are some number of H lists at a distinct levels of the recursion, awaiting the production of the G list to which they will be merged. Each H is a covering list of F_α for some α and so as argued above is of expected size $2N/N^{2^a(1-pow(\varepsilon))}$. Thus the total space occupied by the at most K H lists at distinct levels is $O(N \sum_{c=0}^{K} N/N^{2^c(1-pow(\varepsilon))})$. As noted previously, this sum is dominated by the first term which is $O(N^{pow(\varepsilon)})$. At any moment there is at most one G list in existence, so certainly the space claim is not exceeded by the covering lists created and destroyed during the course of the algorithm. ■

We conclude this section with a discussion of how to treat the case where P/T is not a power of 2. To make a beginning, consider the case were P is a multiple of T. The difficulty here is that progressively halving W does not lead to pieces of size T. The key to handling this is to observe that one could equally well have divided W into thirds, then split the thirds in third, and so on without changing the principle aspects of Lemma 6, the algorithm, and its complexity. Specifically, for a word W such that its length $P = 3^K T$ for some K, we could have let W_α for $\alpha \in \{0, 1, 2\}^*$ be recursively defined by the equations: $W_\varepsilon = W$, $W_{\alpha 0} = W_\alpha[1..|W_\alpha|/3]$ (the first third of W_α), $W_{\alpha 1} = W_\alpha[|W_\alpha|/3+1..|W_\alpha|2/3]$ (the second third of W_α), and $W_{\alpha 2} = W_\alpha[|W_\alpha|2/3+1..|W_\alpha|]$ (the last third of W_α). If we had then let $D_\alpha = \lfloor D/3^{|\alpha|} \rfloor$ be the match stringency to W_α then Lemma 6 as stated would remain true. Moreover, in analogy with the argument given for producing L_α in Figure 7, one can show that all paths of cost D_α or less in the edit graph of W_α versus A, must lie on the set of diagonals: $\cup \{ [l - P_{\alpha 0} - D_\alpha, u - P_{\alpha 0} + D_\alpha] : (l, u) \in F_{\alpha 0} \} \cup \{ [l - P_{\alpha 0} - P_{\alpha 1} - D_\alpha, u - P_{\alpha 0} - P_{\alpha 1} + D_\alpha] : (l, u) \in F_{\alpha 1} \cup R_{\alpha 2} \}$. Certainly, a covering list L_α of this set can be built with a three way merge of covering lists for $F_{\alpha 0}$, $F_{\alpha 1}$, and $R_{\alpha 2}$. Note that in this case we must expand each pair (l, u) by D_α as opposed to Δ_α and thus the $F(R)SCAN$ procedure over the covering list L_α may take twice as long as before, but this inefficiency does not affect the asymptotics of the complexity argument. Thus, in almost exact analogy with the development of the algorithm of Figure 8, we could have proceeded to build an algorithm based on three way merges. Moreover, the complexity would remain unchanged since the critical sum, $\sum_{c=0}^{K} 2^c/N^{2^c(1-pow(\varepsilon))}$, in the analysis becomes, $\sum_{c=0}^{K} 3^c/N^{3^c(1-pow(\varepsilon))}$, which still converges hyperexponentially.

Taking this idea a little further, observe that as we partition W into pieces we may split these pieces into halves or thirds on an individual basis. The only difficulty is how to distribute the errors in the case where an even split is not possible, e.g. if $P = 11T$ then ‘‘halving’’ it gives pieces of sizes $5T$ and $6T$. An easy extension of Lemma 6, shows that if W aligns to V with not more than D differences and $W = W_0 W_1$, then either W_0 aligns to a prefix of V with not more than $\lfloor D / |W_0| \rfloor$ errors, or W_1 aligns to a suffix of V with not more than $\lfloor D / |W_1| \rfloor$ errors. Thus an uneven split does not create a problem, we still seek ε-matches to the subparts. So our solution involves repeatedly halving W into pieces whose length is divisible by T until pieces of size T or $3T$ result. Those pieces of length $3T$ are split into thirds and then processed as a three way merge as discussed above. For example, if $P = 7T$, then $P_0 = 3T$, $P_1 = 4T$, $P_{00} = P_{01} = P_{02} = T$, $P_{10} = P_{01} = 2T$, and $P_{100} = P_{101} = P_{110} = P_{111} = T$. Such a subdivision method always applies when T divides P, and requires finding ε-matches at each level. Note that three-way merges are always confined to the deepest level and that the expected time still decreases hyperexponentially as one moves up the decomposition hierarchy. Thus this approach continues to guarantee $O(DN^{pow(\varepsilon)} \log N)$ expected-time under the more general condition where T divides P.

Finally, consider the case where P is arbitrary. Our technique for this case requires that P must not be less than T^2 or $\Omega(\log^2 N)$ in order to maintain the asymptotic complexity claim. In principle this is permissible since we need prove the result only for N and P sufficiently large. So suppose $P \geq T^2$ and let $a = \lfloor P/T \rfloor$ and $b = P\ (mod\ T)$, i.e. $P = aT + b$ where $b < T$. Now it is possible subdivide W into a pieces using the 2- and 3-splitting method, where b of the pieces are of length $T+1$ and the rest are of length T (this requires that $P \geq T^2$). For the pieces of length $T+1$, finding ε-matches to them requires the generation of $O(|\Sigma|^{pow(\varepsilon)} N^{pow(\varepsilon)})$ words for a $|\Sigma|$-factor increase in time for this phase (recall the discussion at the bottom of page 9). Since $|\Sigma|$ is assumed to be a constant from an asymptotic point of view, we are done. In practice this works very well since the per-word cost of generation is much less that than the per-hit cost of extension. For queries that are very short, we divide W into pieces of length $T \pm c$ for small c, in a fashion that gives the best performance possible.

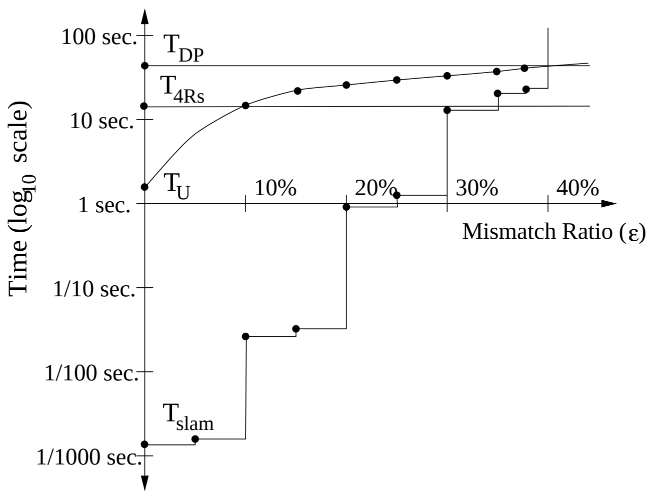
7. Practical Experience

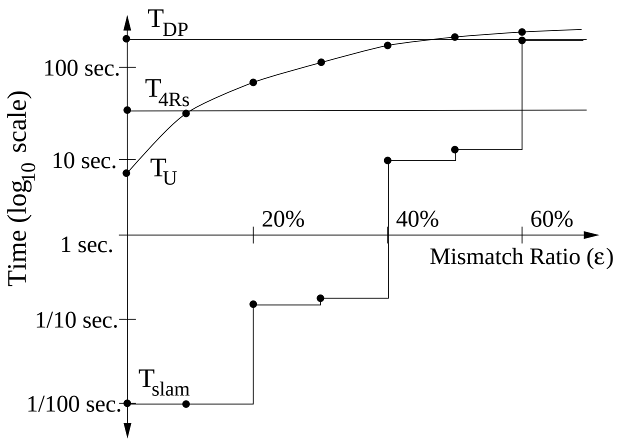
In order to determine the practical efficiency of our approach to the approximate keyword searching problem, the theoretical algorithm described in the preceding sections was implemented in the C programming language. The implementation effort amounted to about 1500 lines of software. The index data structure was implemented exactly as described in Section 2. For the algorithm proper, however, a number of practical considerations require slight variations on the theoretical design and these are described in the next two paragraphs.

Three small observations improved the practical performance of the word generation and lookup phase of the algorithm. First, in practice there is no advantage in going from the $O(TZ)$ version to the $O(DZ+T)$ version that computed only the relevant $2D+1$ entries of each row. For the small values of T and D actually involved (e.g., $T \in [5,10]$ and $D \in [0,4]$), the overhead of checking which entries to compute outweighs the straightforward calculation of the entire row. Secondly, using the KMP construction to avoid generating words not in the condensed neighborhood was similarly found to be ineffective in practice because it eliminates very few words in expectation. The third variation involves the interaction of the generation of words with their lookup in the index. Specifically, in the case where a word of length T has been generated that is still a proper prefix of a condensed neighborhood word (i.e., there is an entry less than D in the current row), then this word is looked up in the index immediately, and the extensions of the individual matches to this word are checked for membership in the condensed neighborhood by continuing the computation of the dynamic programming matrix on the extension. This is more efficient in practice because there are always at least $|\Sigma|$ extensions of the word in the condensed neighborhood, but on average only one occurrence of their T-symbol prefix in the database.

The last and most significant deviation from the theoretical algorithm described above, is in the way $\overline{Hits}_w(d)$ ($Hits_w(d)$) is recorded and sorted in lines 2 and 3 of *FGEN* (*RGEN*). An array S of bits is used to record the position of the hits. Initially all bits are set to zero and whenever a word is generated that matches at position i, then the i^{th} bit of S is set. Thus at the conclusion of word generation $S[i]$ is set iff $i \in \overline{Hits}_w(d)$. Simply reading off the set bits left to right gives the hit list in sorted order. We take this another step farther, by recording both the forward generation on $W_{\alpha 0}$ and the reverse generation on $W_{\alpha 1}$ for what would normally be the calls to *FGEN* and *RGEN* at the bottom level of the recursion of the function *LIST* in lines 4 and 5. That is, instead of calling these two routines, we establish the bit array S so that $S[i] = 1$ iff $i \in \overline{Hits}_{W_{\alpha 0}}(D_{\alpha 0}) \cup Hits_{W_{\alpha 1}}(D_{\alpha 1})$. Then we call a special version of *UNION* which produces the covering list for L_α in a single left-to-right pass over the array S. As regards space, this is not too great a cost, since S requires $N/8$ bytes versus the $6N$ bytes required by the index itself. While avoiding a couple of covering list constructions, the potential pitfall is that the scan of S takes $O(N)$ time as opposed to the $O(N^{pow(\varepsilon)} \log N)$ time taken by a sort over a listing of the hit set. This inefficiency for sparse problems (i.e., small ε) is rectified as follows. On the computers in our laboratory an integer occupies 32-bits and S is realized as an $N/32$ element array of integers. The left-to-right scan only needs to examine the bits of an integer if it is nonzero, i.e., one of its 32 bits is set. Thus the time for the scan is improved by a factor of 32 for sparse problems, but this may still be too inefficient. So a second array T of $N/32^2 = N/1024$ integers is maintained such that the j^{th} bit of T is set if and only if the j^{th} word of S is nonzero. This requires twice as much time when a bit, say i, of S must be set, because the $i/32^{th}$ bit of T must also be set. But for sparse problems, only $N/1024$ integers need to be checked during the scan, and only those 1024 position stretches containing hits are examined further. We could extend this idea recursively, essentially arriving at a logarithmic scheme, but we found that a two tiered approach was quite sufficient for problems where N is in the million to 10 million range.

We compared our implementation against an implementation of the standard $O(NP)$ dynamic programming algorithm [Sel80], the $O(DN)$ expected-time algorithm of Ukkonen [Ukk85a,MyM86], and a novel use of the 4-Russians paradigm that permits the dynamic programming matrix to be computed 5 entries at a step [WMM91]. In all cases the software had been written at an earlier time by this author and represent his best efforts at efficient code. All experiments were performed on a SparcStation 2 with 64 megabytes of memory and all code was compiled under the standard SunOS C-compiler with the optimization option on. For each timing result reported, we ran the given algorithm enough times so that the total elapsed time was at least 100 seconds and then averaged. Given that the system clock is accurate to about 0.1 to 0.2 seconds, timing results are figured to be accurate to the third digit. A random query of length 80 was searched against a random (every symbol equally likely) database of a million symbols for a four letter alphabet, and four million symbols for a twenty letter alphabet. A plot of the results is shown in Figure 10. The curves for the standard dynamic programming algorithm are labeled T_{DP}, those for the Ukkonen algorithm are labeled T_U, those for the 4-Russians algorithm are labeled T_{4R}, and those for our sublinear algorithm T_{slam}. A logarithmic time scale is used because the sublinear algorithm's time performance increases exponentially in *D*. Thus the curve for T_U is shaped like a log curve because it is actually a straight line on a normal scale. T_{DP} and T_{4R} are straight lines because the complexity of their underlying algorithms depend only on *P* and *N*.





$N = 1{,}000{,}000$ and $|\Sigma| = 4$ $N = 4{,}000{,}000$ and $|\Sigma| = 20$

Figure 10: Timing Plots for Queries of Length $P = 80$

Observe from the figures that for the case where $|\Sigma| = 4$, our algorithm is three orders of magnitude faster than any of the others when $\varepsilon < 10\%$. It is two orders of magnitude faster when $\varepsilon < 20\%$, and a single factor of 10 faster when $\varepsilon < 30\%$. Moreover, it crosses over with the best algorithms in the 30-40% range of ε exactly as suggested by the curve for $pow(\varepsilon)$ given in Figure 4. In the case where $|\Sigma| = 20$, our algorithm achieves slightly more modest factors of improvement for the intervals 0-20% (3 orders), 20-40% (2-orders), and 40-60% (factor of 4). When ε is above 60% it performs considerably word than the 4-Russians algorithm.

Tables 1 and 2 below, shows some of the exact numbers used to produce the plots of Figure 10 and also displays some statistics on the the number of hits and covering list spans for the sublinear algorithm. In studying these statistics, which readily explain the time performance, it is important to note that $T = 10$ for the experiments in Table 1, and $T = 5$ for the experiments in Table 1. Thus in the first case, the length of the query $P = 2^3 T$, and in the latter, $P = 2^4 T$. The column, *Hits*, gives the average number of matches to words in the neighborhood about each T subpiece of the query string. The columns labeled, $Span_j$, for some j, give the percentage of the database spanned by the average covering list L_α where $|\alpha| = j$. For example, when $\varepsilon = 20/80 = 25\%$, $Span_2 = 1.54$ in Table 1, indicating that on average $span(L_\alpha) = 1.54N/100 = 15,400$ when $|\alpha| = 2$. The interesting observation about these columns is that they reveal that as D is increased for the query, D_α increases at each level and the corresponding statistics increase exponentially, but more slowly at the higher levels. Some readers may wonder what happens when P becomes larger than 80. If P were doubled (without changing N) then each of the first three columns concerning time would double. But the numbers in the remaining columns would be exactly the same. However, the headers $Span_j$ would become $Span_{j+1}$. Thus for fixed ε and N, time varies proportionally with P while coverage statistics remain constant.

D	T_{slam} $(sec.)$	T_U $(sec.)$	T_{4R} $(sec.)$	*Hits*	$Span_2$ (%)	$Span_1$ (%)	$Span_0$ (%)	*Matches*
0	.0015	1.8	12.8	1	.0	.0	.0	0
4	.0017	7.6	12.8	1	.0	.0	.0	0
8	.037	13.0	12.8	54	.03	.0	.0	0
12	.045	18.7	12.8	54	.05	.0	.0	0
16	.97	24.3	12.8	1400	1.12	.0	.0	0
20	1.17	30.5	12.8	1400	1.54	.12	.0	0
24	10.8	36.6	12.8	17000	14.	1.2	.0	0
28	16.0	42.6	12.8	17000	18.	9.4	.4	0
30	17.3	45.5	12.8	17000	18.	10.5	2.2	1

Table 1: Times and Hit frequencies when P=80, $|\Sigma|$=4, and N=1,000,000

D	T_{slam} $(sec.)$	T_U $(sec.)$	T_{4R} $(sec.)$	*Hits*	$Span_3$ (%)	$Span_2$ (%)	$Span_1$ (%)
0	.0097	6.1	40.1	1	.0	.0	.0
8	.0097	38.5	40.1	1	.0	.0	.0
16	.184	71.0	40.1	220	.03	.0	.0
24	.223	104.	40.1	220	.05	.0	.0
32	9.8	140.	40.1	13000	2.5	0.3	.0
40	13.4	173.	40.1	13000	3.4	1.0	.0
44	13.8	190.	40.1	13000	3.4	1.1	.07
48	179.	204.	40.1	284000	46.	15.	1.1

Table 2: Times and Hit frequencies when P=80, $|\Sigma|$=20, and N=4,000,000

In a final experiment, we ran our algorithm over an older version of the PIR database containing 3,000,538 symbols. The query was the 104 symbol sequence for human Cytochrome C. This test was run to see how critical the uniformity assumption for the database was. The underlying alphabet had 23 characters, containing two codes that denoted one of two residues, and a wild card code, 'X', denoting any residue. These symbols appeared much less frequently than the others, and, in general the frequency of occurence of each letter was not uniform. Indeed, about 40% of all buckets in the index were empty, and there was one that had 553 positions in it. Nonetheless, note that the performance figures in Table 3 are very comparable to those in Table 2. Times are roughly about three times slower. As D increases the factor becomes less. The key thing to note is that there are many cytochrome C entries for other organisms in the database and consequently, this search is preconditioned to contain quite a few matches to the query. As noted in the overview, this effectively means that complete dynamic programming computations are run for each match. It is this time that is primarily responsible for the differential over simulated data and not the skew in character distribution.

D	T_{slam} $(sec.)$	*Hits*	$Span_3$ (%)	$Span_2$ (%)	$Span_1$ (%)	$Span_0$ (%)	*Matches*
0	.027	23	>.0	>.0	>.0	>.0	2
8	.067	23	.005	.005	.01	.01	18
16	.40	298	.06	.01	.02	.03	32
24	.58	298	.09	.02	.04	.07	44
32	12.4	19900	3.2	.1	.1	.1	74
40	17.2	19900	4.3	1.6	.1	.2	79

Table 3: Times and Hit frequencies for Searching a Protein Database

8. Conclusion

A sublinear algorithm for approximate keyword searching has been presented that not only represents an asymptotic improvement for the problem, but also provides order-of-magnitude speedups in practice. We close with several observations and conjectures. First observe that with $\varepsilon = 37.5\%$, a match was found just by chance as shown in Table 1 above. We conjecture, that the point at which D becomes large enough so that $Z(T, D) = N$ is strongly correlated to the point at which a (D/T)-match to a word W would be found purely by coincidence in a random database. Second, we observe that there is nothing in the algorithm itself that precludes using more general measures of similarity such as real-valued arbitrary scores for indels and substitutions. The only aspect of our treatment that was specifically tied to the simple unit measure was in the mathematics for bounding the sizes of neighborhoods. An open development is to demonstrate that the approach works well in practice for scoring schemes where neighborhoods aren't "too" large. Alternatively one needs to develop a formal treatment of how neighborhood size is a function of scoring scheme. Third, is there anyway to improve upon the $O(N^{pow(\varepsilon)})$

working storage required by the algorithm? Finally, we note that the essential idea of this paper can be summed up as: ‘‘find approximate matches to subparts using exact matches to neighborhoods as a filter to those locations where an extension strategy can be profitably employed.’’ There are potentially many other ways to instantiate this idea and perhaps there are better ones than that realized here. For example, this approach was the essentially idea behind a heuristic sequence comparison tool, BLASTA, now in popular use for protein database searches [AGM90].

Acknowledgement

The author wishes to thank his colleague and mentor Andrzej Ehrenfeucht for his advice, ideas, and encouragement early in the development of this work. Also thanks to George Corugedo for the courage to finish a long overdue paper.

References.

[AGM90] Altschul, S., W. Gish, W. Miller, E. Myers, and D. Lipman, ‘‘A Basic Local Alignment Search Tool,’’ *J. of Molecular Biology* **215** (1990), 403-410.

[BoM77] Boyer, R. and J. Moore, ‘‘A fast string searching algorithm’’, *Comm. ACM* **20**(10) (1977), 262-272.

[ChL90] Chang, W.I. and E.L. Lawler, ‘‘Approximate matching in sublinear expected time’’, *Proc. 31st IEEE Symp. on Foundation of Computer Science* (1990), 116-124.

[GaP90] Galil, Z. and K. Park, ‘‘An Improved Algorithm for Approximate String Matching’’, *SIAM J. on Computing* **19**(6) (1990), 989-999.

[KMP77] Knuth, D.E., J.H. Morris, and V.R. Pratt, ‘‘Fast pattern matching in strings’’, *SIAM J. on Computing* **6**(2) (1977), 323-350.

[LaV86] Landau, G.M. and U. Vishkin, ‘‘Introducing efficient parallelism into approximate string matching and a new serial algorithm’’, *Symp. on Theory of Computing* (1986), 220-230.

[Mye86a] Myers, E.W., ‘‘Incremental alignment algorithms and their applications’’, Tech. Rep. 86-22, Dept of Computer Science, U. of Arizona, Tucson, AZ 85721.

[Mye86b] Myers, E.W., ‘‘An O(ND) difference algorithm and its variants’’, *Algorithmica* **1** (1986), 251-266.

[MyM86] Myers, E.W. and D. Mount, ‘‘Computer program for the IBM personal computer that searches for approximate matches to short oligonucleotide sequences in long target DNA sequences’’, *Nucleic Acids Research* **14**(1) (1986), 501-508.

[Sel80] Sellers, P.H., ‘‘The theory and computation of evolutionary distances: pattern recognition’’, *J. Algorithms* **1** (1980), 359-373.

[Ukk85a] Ukkonen, E., ‘‘Finding approximate patterns in strings’’, *J. of Algorithms* **6** (1985), 132-137.

[Ukk85b] Ukkonen, E., ‘‘Algorithms for approximate string matching’’, *Information and Control* **64** (1985), 100-118.

[WMM91] Wu, S., U. Manber, and E.W. Myers, ‘‘Improving the running times for some string matching problems’’, Technical Report TR91-20, Dept. of Computer Science, U. of Arizona, Tucson, AZ 85721 (submitted to *Algorithmica*).