Neweyes: A System for Comparing Biological Sequences Using the Running Karp–Rabin Greedy String–Tiling Algorithm

Michael J. Wise

Department of Computer Science,
University of Sydney, Australia
michaelw@cs.su.oz.au

Abstract

A system for aligning nucleotide or amino acid biosequences is described. The system, called Neweyes, employs a novel string matching algorithm, Running Karp–Rabin Greedy String Tiling (RKR–GST), which involves tiling one string with matching substrings of a second string. In practice, RKR–GST has a computational complexity that appears close to linear. With RKR–GST, Neweyes is able to detect transposed substrings or substrings of one biosequence that appears rearranged in a second sequence. Repeated substrings can also be detected. Neweyes also supports a form of matching–by–group that gives the effect of different amino acid mutation matrices. Neweyes can be used in a macro mode (searching a database for a list of biosequences that are similar to a given biosequence) or in a micro mode, where two biosequences are compared and more detailed output formats are available.

Introduction

What will be described in this paper is a system, called Neweyes[1], for aligning nucleotide or amino acid biosequences. Stated in computer science terms and in its simplest form, the problem faced by any alignment program is, given two strings, to determine the degree of similarity between the strings. The result may then be expressed, for example, in terms of a percentage–match value or the length of the common portions. Neweyes employs a novel string matching algorithm, Running Karp–Rabin Greedy String Tiling (RKR–GST), which will be outlined below. (For a complete description, the reader should consult (Wise 1994).)

Before looking at the facilities provided by Neweyes and more specifically at the RKR–GST algorithm, it is worth canvassing the systems currently in use. The first system to address the biosequence alignment problem was reported in (Needleman & Wunsch 1970) and uses dynamic programming. A similar approached evolved in the computer science literature, where the problem came to be known as the longest common subsequence[2] (LCS) problem. The Needleman Wunsch system was in fact one of the earliest solutions to these and similar problems. Summarising the LCS problem, if S is a string, a subsequence of string S is formed by taking elements in order from S, where zero or more elements are ignored before another is taken. (By contrast, substrings allow no gaps.) The LCS of two strings S and T is the sequence of elements common to the two strings such that no longer sequence is available (though there may be multiple sequences with the same, maximal length). There are many discussions of LCS in the literature (e.g. recent ones (Cormen, Leiserson & Rivest 1990) and (Gonnet & Baeza–Yates 1991)). What is interesting about the Needleman Wunsch approach is they add the concept of gap penalties (which means, however, that the resulting sequence may have fewer gaps that the LCS, but may also cover fewer than the maximum number of elements. It is also worth noting that the Needleman Wunsch algorithm only had positive values (based on the number of base changes in each codon), so a *global alignment* is produced (i.e. involving the entirety of both input strings).

A form the the Needleman Wunsch algorithm that has become popular is due to (Smith & Waterman 1981), and involves finding what have come to be known as local alignments. That is, by making gap penalties a function of gap size alignment scores may be reduced to zero. (The gap penalties are these days separated into a gap creation penalty and a gap extension penalty, e.g. in (States & Boguski 1992).) This effect is strengthened by the authors' decision to also use negative values for non–matches. What is produced is a sequence that need not be bounded by the ends of the input strings. Once again, only a single (local) sequence is produced. The computational cost of algorithms based on dynamic programming can be shown to be $O(n^2)$ (Kruskal 1983). Furthermore, the

1. With the different perspective offered by the system the name is meant to suggest "looking at the world through new eyes". However, the name is in fact an acronym taken from a line of Tom Lehrer's song, *Lobachevsky*, "(Let) Noone Else's Work Evade Your EyeS" – reflecting the algorithm's ancestry in the detection of suspected plagiarism (Wise 1992).

2. This use of sequence should not be confused with the biologist's use of the term "sequence", which is a string or sequence of nucleotides or amino acids. I shall refer to the latter alternatives as *biosequences*.

choices of suitable gap creation and gap extension penalties appears problematic, but have substantial bearing on the results returned by the algorithms.

Another system that has achieved prominence is FASTA/FASTP (Lipman & Pearson 1985; Pearson & Lipman 1988; Pearson 1990). The main reason for this prominence is that FASTA and its companion programs are computationally far cheaper than the earlier systems involving dynamic programming, but are able to retain reasonable accuracy. Where FASTA achieves much of this improvement is in its use of multiple phases in the construction of an alignment. Most of the improvement in speed is achieved in the first phase, where overlapping substrings from one of the biosequences are encoded in two one dimensional arrays using algorithms due to (Dumas & Ninio 1982). The substrings are of a fixed size specified by an input parameter, *k*, and are called k–tuples; the value of k is typically 1 or 2 for amino acids and 4 or 6 for nucleotides. The first array of size α^k – α either 4 for nucleotides or 20 for amino acids – marks the first instance of any particular k–tuple; the second array, with size equal to the input biosequence that is being encoded, links the successive occurrences of the various k–tuples (each cell points to the next occurrence of that k–tuple). By tracing through the second biosequence, all matching k–tuples can be found. The matching k–tuples are arranged in groups, called *diagonals*, where all the k–tuples on a given diagonal are the same distance from a notional perfect match (Wilbur & Lipman 1983). Local *regions* with a high density of k–tuples are then identified (Pearson 1990). In the second phase, the ten regions with the highest score are reevaluated using the PAM250 matrix and for each of these regions, the subregion is found with the best score and reported as the *init1* score. (The regions are still without gaps.) In the third phase, regions that are adjacent are joined depending on their respective scores and *joining penalty*. The new score is reported is the *initn* score. In the final phase, a Needleman Wunsch alignment is performed on those pairs of sequences that had the highest initn scores. In summary, FASTA achieves its greater speed due to its use of a fast but fairly insensitive first phase, which reduces the amount of work required in the more expensive third and fourth phases; joining of regions is expensive, but is limited to the best 10, and Needleman Wunsch alignments are expensive, but are now limited to those pairs of biosequences with the best initn scores. However, this strength is also a source of some weakness; As noted by several authors, e.g. (States & Boguski 1992), chosing a larger value for k, e.g. 2 in the case of amino acids, may mean that matches are missed due to the requirement for an exact matching over the entire k–tuple. That is, k–tuples of amino acids that mutationally equivalent will be by–passed in the first phase as they are not exactly equivalent, and therefore not be available for the second and subsequent phases.

The third system to have become very popular in recent times is BLAST (Altschul et al. 1990). While the Needleman Wunsch/Smith Waterman systems are able to deal with biosequences containing gaps, and FASTA introduces something similar to gaps in the process of joining adjacent regions, BLAST deals exclusively with ungapped biosequences. However, what is lost by disregarding gapped sequences is compensated for by the very fast execution times due to the construction of a finite state machine to recognise all substrings of some fixed size (called *w–mers*) of the query biosequence that score above a given threshold value. (Given the presence of mutations, the authors estimate that each residue in the query biosequence will typically contribute 50 *w*–mers to the finite state machine.) Hits generated by the scan are then extended until the score for the extended match falls below better scoring shorter matches.

Finally, a system of related interest is one described in (Karlin et al. 1988), which searches for sequence features such as direct and inverted repeats. Working within a window of fixed size, the system proceeds up a sequence searching for matching substrings of given size *x*. Matches of size *x* are linked. If a match exists between a substring and one further up the sequence, the match is extended as far as possible. Further, in the face of a non–match it is still possible to extend a match if the size of the error segment is no greater than a value that is also specified at the outset. Only repeats occurring within the window are considered, and each repeat is only paired with its closest occurrence further up the sequence. The algorithm has O(*n*) complexity.

Exact Matching

In this section the algorithm underlying Neweyes will be described. To keep matters reasonably simple, I shall just describe the algorithm for exact matching (e.g. as would be used when comparing DNA biosequences). The extension of the algorithm to proteins will be described in the next section. Unfortunately, in order to make more precise what I intend with this algorithm I must begin with some definitions. In this, I shall retain from the literature on strings the terms *pattern–string* (here labelled as P) and text–string (labelled as T), but in the present context the only difference is that pattern string is the shorter of the two.

Definition. A *maximal–match* is where a substring P_p of the pattern string starting at p, matches, element by element, a substring T_t of the text string starting at t. The match is assumed to be as long as possible, i.e until a non–match or end–of–string are encountered, or until one of the elements is found to be *marked*. (Marking will be discussed presently.) Maximal–matches are temporary and possibly not unique associations, i.e. a substring involved in one maximal–match may form part of several other maximal–matches.

Definition. A *tile* is a permanent and unique (one–to–one) association of a substring from P with a matching substring from T. In the process of forming a tile from a maximal–

match, tokens of the two substrings are *marked*, and thereby become unavailable for further matches.

With the definitions of tiles and maximal–matches in place it is worth noting that in many situations isolated, short maximal–matches can be ignored. For example, it is unlikely that a maximal–match of length 1 or 2 would be significant when comparing amino acid biosequences. For DNA biosequences, maximal–matches involving less than 6 bases can be ignored. The following definition is therefore made:

Definition. A *minimum–match–length* is defined such that maximal–matches (and hence tiles) below this length are ignored. The minimum–match–length can be 1, but in general will be an integer greater than 1.

Ideally what is being sought by the new algorithm is a maximal tiling of P and T, i.e. a coverage of non–overlapping substrings of T with non–overlapping substrings of P which, faced with a minimum–match–length greater than a single token, maximises the number of tokens (amino acids or bases) that have been covered by tiles. Unfortunately, an algorithm which produces a maximal tiling and computes in polynomial time is an open problem. Part of the difficulty lies in the possibility that several small tiles could collectively cover more tokens than a smaller number of larger tiles.

To motivate the transition to a computationally more reasonable measure of similarity it is worth observing that longer tiles are preferable to shorter ones because long tiles are more likely to reflect significant, similar regions in the source biosequences rather than chance similarities. Notice also that this model encompasses substrings, not subsequences, so debates arising from the choice of suitable gap penalties does not arise, but observe that the effect of gap penalties in the other algorithms, particularly when large values are chosen, is to coerce sequences into strings. With this in mind, the following greedy algorithm is proposed. The algorithm involves multiple passes, each pass having two phases. In the first phase, called *scanpattern*, all maximal–matches of a certain size or greater are collected. This size is called the *search length*. In the second phase, called *markstrings*, maximal–matches are taken, one at a time starting with the longest and tested to see whether they have been occluded by a sibling tile (i.e. part of the maximal–match is already marked). If not, a tile is created by marking the two substrings. When all the maximal–matches have been dealt with, a new, smaller search length is chosen. The top–level algorithm is:

```
search-length s := initial-search-length;
stop := false
Repeat
  /*$L_{max}$ is size of largest maximal–matches from this scan*/
  $L_{max}$ := scanpattern(s);
  /*If very long string no tiles marked. Iterate with larger s*/
  if $L_{max} > 2 \times s$ then s := $L_{max}$
  else
    /* Create tiles from matches taken from list of queues*/
    markstrings(s);
    if s > 2 × minimum_match_length then s := s div 2
    else if s > minimum_match_length then
      s := minimum_match_length
    else stop := true
until stop
```

So far, what has been described relates solely to Greedy String–Tiling. It is in the algorithm for `scanpattern` that Running Karp–Rabin matching is used to find all the maximal matches above a certain size. The basis of the well–known Karp–Rabin string matching algorithm (Karp & Rabin 1987) is that, if a hash–value exists for a string of length s starting at t, the hash–value for a string of length s starting at t+1 can be calculated using a simple recurrence relation. In particular, if the length of the pattern string is $|P|$, the hash–value of every substring of length $|P|$ from the text string is compared with the hash–value of the pattern string. If two hash–values are identical, the pattern and text substrings are compared element–by–element. (The version of the recurrence relation used in this work is due to (Gonnet & Baeza–Yates 1990).) Running Karp–Rabin matching extends Karp–Rabin matching in the following ways:

Firstly, instead of having a single hash–value for the entire pattern string, a hash–value is created for each (unmarked) substring of length s of the pattern string, i.e. for the substrings $P_p \cdots P_{p+s-1}$, p in the range $1 \cdots |P|-s$. Hash–values are similarly created for each (unmarked) substring of the length s of the text string. Secondly, each of the hash–values for the pattern string is then compared with the hash–values for text string and for those pairs of pattern and text hash–values found to be equal, there are possible matches between the corresponding pattern and text substrings. A hash–table of the text–string Karp–Rabin hash–values is used to reduce the otherwise $O(n^2)$ cost of this comparison. That is, rather than having to scan the entire text string for the matching hash–value corresponding to a particular pattern substring, the pattern Karp–Rabin hash–value is itself hashed and a hash–table search returns the starting positions of all text substrings (of length s) with the same Karp–Rabin hash–value. Note that after a successful match of both the Karp–Rabin hash–values and the actual substrings, the element–by–element matching continues until two elements fail to match or until a marked element or end–of–string are found. In this way, the matches are converted to maximal–matches. (In other words, length s is the minimum match–length being sought during one iteration.) The algorithm for `scanpattern(s)` – s the current search–length – is:

```
starting at the first unmarked token of T,
  for each unmarked $T_t$ do
    if distance to next tile ≤ s then
      move $t$ to first unmarked token after next tile
    else
      create KR hash-value for substring $T_t..T_{t+s-1}$
      add to hashtable

Starting at the first unmarked token of P,
  for each unmarked $P_p$ do
    if distance to next tile ≤ s then
      move $p$ to first unmarked token after next tile
    else
      create KR hash-value for substring $P_p..P_{p+s-1}$
      check hashtable for $P_p..P_{p+s-1}$ KR hash-value
      for each hit in hash-table do
        k: = s
        /*Extend match until non-match or element marked*/
        while $P_{p+k} = T_{t+k}$
              AND unmarked($P_{p+k}$)
              AND unmarked($T_{t+k}$) do
          k := k + 1
        if k > 2×s then
        /*abandon scan. Will restart with s = k*/
          return(k)
        else record new maximal-match
return(length of longest maximal-match)
```

The structure used to record the maximal matches is a doubly–linked–list of queues, where each queue records maximal–matches of the same length and the list of queues is ordered by decreasing length. A pointer is also kept to the queue onto which the most recent maximal–match was appended because there is a high probability that the next maximal–match will be similar in length to the last and therefore will be appended to the same queue or one that is close by. `markstrings` also has the parameter s, the search–length.

```
starting with the top-queue do
  if current queue is empty then drop to next queue
  /*corresponds to smaller maximal-matches */
  else with queue of maximal-matches length L do
    remove match(p, t, L) from queue
    if match not occluded then
      if for all $j$:0..$s-1$,$P_{p+j} = T_{t+j}$ then
      /* IE match is not hash artefact */
        for j:= 0 to L - 1 do
          mark_token($P_{p+j}$)
          mark_token($T_{t+j}$)
          count_tokens_tiled := count_tokens_tiled + L
    else if L - $L_{occluded}$ ≥ s then
      replace unmarked portion on list of queues
```

Note that the test of whether maximal–match is really a match has been deferred from `scanpattern` – where it normally would reside – to `markstrings` (remember that all putative matches found due to hashing must be tested element–by–element because equivalence of hash–values does not guarantee that the corresponding strings are equal). However, it has been observed that KR–hashing appears to fail so rarely that deferring the component–wise test to `markstrings` turns out to be far more efficient. (See discussion in (Wise 1994).)

The question arises as to what is an appropriate value for the parameter s passed to `scanpattern` and `markstrings`. More precisely, what is to be its initial value and how is that value to be decremented? While one might consider half the length of P as an appropriate starting value for s, it turns out in practice that a much smaller value will suffice. There are two reasons for this. Firstly, very long maximal–matches are rare, so in general a large initial value for s would generate a number empty sweeps by scanpattern until a match is finally found. Secondly, if a long maximal–match is found ("long maximal–match" defined to be where k, the maximal–match length is 2×s) the creation of a tile from this string will absorb a significant number of the pattern and text tokens. It is therefore worthwhile stopping the current scan and restarting with the larger initial value of s=k for this special case. This implies that the initial search–length can be set to a small constant value (it is currently 20), rather than being dependent on the string lengths.

Finally, in (Wise 1994) it is argued that although the worst–case complexity is $O(n^3)$ – n the sum of the lengths of the input strings – the circumstances where that arises are entirely pathological and using curve–fitting a complexity of $O(n^{1.12})$ is shown experimentally, i.e. close to linear. To see what the experimentally derived complexity would be in this domain, 37 pairs of proteins where compared, with combined string lengths ranging from 215 to 3533 amino acids. With the minimum–match–length set to 3, the experimentally derived complexity was $O(n^{0.90})$, i.e. still close to linear.

Matching Amino Acids by Group

The algorithm described in the previous section performs exact matching, in the first instance by comparing a hash–value of some number of tokens and only subsequently by comparing the actual tokens. The tokens in this instance are integer values representing, say, nucleotides. What has been observed many years ago in the case of amino acids is that over evolutionary time certain of them mutate into certain other amino acids. This propensity is usually displayed in the form of a matrix, the best known being the PAM250 of (Schwartz & Dayhoff 1978). It is this matrix that is used, for example, in the Smith Waterman algorithm. Looking beyond the matrix representation, however, what is being expressed is the notion that certain amino acids are substitutable, though perhaps in different contexts, e.g. as illustrated by the matrices due to (Lüthy, McLachlan & Eisenberg 1991), which reflect various features of secondary structure and hydrophilicity.

What is proposed is that, instead of using the matrices directly, groups of substitutable amino acids be formed, and it is the groups that will participate in the matching. In particular, a set of *imperfect equivalence classes* are formed (or alternatively, *equivalence classes with deleted edges*). That is, each member of an imperfect equivalence class, or simply group, is assumed to be substitutable for any other member of the group, except where that mutation has been explicitly disallowed. (Every amino acid is assumed to belong to one group, even if that group contains only itself, meaning that only mutations of that amino acid to itself are allowed.) For example, the groups drawn from the PAM250 matrix found in (Schwartz & Dayhoff 1978) are: {A G P S T}, {D E H N Q}, {I L M V}, {F Y}, {K R}, {C}, {W} where the mutations that are explicitly excluded are: G ≡ T and G ≡ P.

Assume that the information contained in the matrices is recast as a list of amino acid pairs followed by a number, say an integer, representing the log–odds of mutation occurring between the two amino acids (discussed in (Dayhoff, Schwartz & Orcutt 1978)). Assume also that the list is ordered by decreasing log–odds value, and that there is defined a function, `LogOdds`, that given a pair of amino acids, returns the log–odds mutation value. Finally, assume that a parameter passed to the algorithm, threshold, represents a value above which the corresponding mutation may occur. With these assumptions, the algorithm that is used to generate the the amino acid groups together with the list of excluded mutations is:

```
groups := {{A}, {C}, {D}, {E}, {F}, {G}, {H}, {I}, {K}, {L}, {M},
           {N}, {P}, {Q}, {R}, {S}, {T}, {V}, {Y}}
excluded_pairs := {}
for each (AA1, AA2), AA1 <>AA2 do
  /*Assume AA1 ≡ AA2 already part of group*/
  if LogOdds(AA1, AA2) > threshold then
    pos_links := 0;  /*Count above–threshold edges added
                        to graph by addition of new edge*/
    neg_links := 0;  /*Count edges at or below threshold*/
    temp_excl := { };  /*Set of edges excluded by addition
                          of new edge*/
    AA1_group := the group that contains AA1;
    AA2_group := the group that contains AA2;
    for each member i of AA1_group do
      for each member j of AA2_group do
        if LogOdds(i, j) > threshold then
          pos_links++;
        else
          neg_links++;
          temp_excl = temp_excl ∪ {(i, j)}
      if pos_links > neg_links then
      /*create new, joint group after removing parents*/
       groups := groups − AA1_group − AA2_group;
       groups := groups ∪ {AA1_group ∪ AA2_group}
       excluded_pairs := excluded_pairs ∪ temp_excl;
```

In the example of the groups formed from the PAM250 matrix, the threshold was set to 0. The way the groups are used is that, as each amino acid is being read in, rather than being converted to an integer value, it is given the index of the group to which it belongs. Then, in `markstrings`, when the component–wise tests is being undertaken a boolean valued matrix is used, so that if a mutation is allowed the cell is given a `TRUE` value; otherwise the cell is marked `FALSE`. (If exact matching is being done, only the diagonal elements will be labelled `TRUE`.)

To get some indication of the impact that matching–by–groups has on system performance, the set of protein comparisons mentioned earlier were run again, but his time using a set of groups and excluded mutations resulting from the application of the `formgroups` algorithm to the *Beta* matrix from (Lüthy, McLachlan and Eisenberg 1991). The resulting groups are: {A G T S Q K R N E D}, {W F Y H}, {L I V M}, {C}, {P}, from which the following mutations are explicitly disallowed: G≡N, G≡K, A≡D, A≡E, A≡R, A≡K, A≡Q When curve–fitting is performed, the experimentally derived complexity rises to $O(n^{1.09})$, which is still close to linear, so the effect on performance does not appear to be excessive.

Facilities Provided by Neweyes

Neweyes has been designed to work in two different, but related, ways. In the first, a "macro" capacity, a file containing a single biosequence (in EMBL format) can be compared to a file containing multiple biosequences. The user selects which of a number of statistics will be used to sort the list possible matches that result from the comparison of the query sequence with the database. The statistics that are currently available are:

- A percentage match value, which is the number of bases or amino acids marked in one biosequence, i.e. covered by tiles, divided by the length of the smaller biosequence (expressed as a percentage).
- Then total number of marked tokens from one biosequence.
- The length of the longest tile.
- The length of the average tile.
- The length of the RMS (root–mean–square) tile
- A score, $\sum_{t=1}^{N} x \times (x - minimum\_match\_length)$

The second mode of operation offers a "micro" capacity, where a single pair of biosequences are being compared. In this case, more complex output formats are possible, including:

- Dotplot
- List of the tiles, showing the start and end points for each tile in the respective biosequences.
- Listing of either biosequence with matching portions from the other biosequence superimposed.

A number of facilities are available across both modes of operation. They include:

- Exact matching or matching–by–group.
- Where matching–by–group is being employed, it is possible to specify an ascii file which lists the groups (one per line) and a second file which lists the pairs of mutations that are to be explicitly disallowed.
- Where the user wishes to examine possible repeats of parts of the pattern biosequence occurring in the text

biosequence, it is possible to suppress marking of the pattern or text strings string (so several text substrings may form tiles with the same pattern or text substring).

- It is also possible to switch off tiling altogether, so that just the maximal-matches are recorded. They may then be displayed, for example, on a dotplot or list-of matches.
- It is, of course, possible to match a sequence against itself, but to do so also means suppressing matching along the main diagonal (which would otherwise be the default sequence returned). To simplify matters, and the facilitate traversal through a database, it is possible to specify that each sequence be compared against itself or against its complement.

Discussion

At first appearances, Running Karp-Rabin Greedy String Tiling (RKR-GST) algorithm has similarities with the algorithms underlying both FASTA and BLAST. There are, however, significant differences which outweigh the similarities. Like both FASTA and BLAST, the RKR-GST algorithm employs a search for substrings of some fixed length, and like BLAST, having found a match the match is then extended. However, in the case of RKR-GST the search-length (i.e. k-tuple or *w*-mer size) varies and can increase if it appears that very long strings exist. (This derives from the fact that the former systems use direct encoding of tuples or a finite-state-machine, i.e. they are essentially table driven and therefore expensive to recode for a different tuple size, while RKR-GST, being based on Karp-Rabin matching, is easily changed.) Furthermore, unlike both the earlier systems, there is no penalty for increasing the search-length (compared to the exponential increase in memory required to store the tables driving FASTA and BLAST). Relatedly, the static storage reqired by RKR-GST grows proportionally to the lengths of the input strings, and while the amount of dynamic storage is dependent on the number of maximal-matches, in practice the number is small (matches turn out to be relatively rare) and only weakly proportional to the lengths of the input strings. Finally, the use of multiple passes, each covering a range of maximal-match sizes also helps to reduce the number of maximal-matches that need to be stored.

Two examples are presented, the first illustrating Neweyes' micro-search features, the other illustrating macro-search. The first example illustrates an alignment between two sequences that are clearly related, RASH_HUMAN (Transforming Protein P21/H-RAS-1, AC. P01112) and RAS1_YEAST (Transforming Protein Homologue RAS1, AC. P01119). When these are aligned using a minimum-match-length of 3 the statics of the tiling are:

```
RASH_HUMAN (189) : RAS1_YEAST (309)
Number of marked tokens 97 / 189 (51.32%) in 15 tiles
Longest tile: 20
Average tile: 6.47
RMS tile: 8.30
```

As an alternative to the conventional dotplot (also provided by Neweyes) Appendix 1 is a listing of the text-string, RAS1_YEAST, with RASH_HUMAN superimposed on it where there is a match.

To compare Neweyes with Blast and Fasta, the RASH_HUMAN sequence was tested against the SwissProt database (as at October, 1994). In each case, the top 200 matches were recorded. Neweyes was run with a minimum-match-length of 5; the *score* metric was used to rank the matches. The number of matches that Blast and Fasta, Blast and Neweyes and Fasta and Neweyes have in common are (respectively): 183, 157 and 154.

As a second, somewhat harder experiment, the sequence EGF_HUMAN (Epidermal Growth Factor Precursor, kidney, AC. P01133) was used as the query sequence against the SwissProt database. Once again, a minimum-match-length of 5 and the *score* metric were used. This time, the numbers of matches in common, Blast vs Fasta, Blast vs Neweyes and Fasta vs Neweyes are 81, 42 and 49. Looking more closely, 36 matches are common to all three systems; 13 are only to be found in both Neweyes and Fasta while 6 are only to be found in both Neweyes and Blast. In other words, Neweyes has 55 matches in common with either Blast or Fasta. What both of these examples indicate is that Neweyes will generally find the same matches as either Blast or Fasta, but as the strength of the matches decreases the number of matches shared by all the systems will decrease. However, some differences between Neweyes and Blast or Fasta are inevitable because of Neweyes' different point of view. This will be particularly evident if subsequences have been shuffled in some way between a query sequence and those in the database; Neweyes will still see a stronger match than either Blast and Fasta (which may fail to find any significant match at all).

Finally, it should be noted that Neweyes, coming from a rather different point of view, provides an expansion of facilities offerred by the system described in (Karlin et al. 1988). That is, while both systems can find repeated substrings, in the system described by Karlin et al. each repeat is associated with its next occurrence, so to find all repeats generated by a particular substring the closure must be taken of the set of adjacent repeats. (The cost of doing this is limited, however, by the size of the window. On the other hand, the use of a window also implies that distant repeats may not be found.)

As the system currently stands, implementation of the facilities described above has recently been completed. When the EGF_HUMAN query sequence was run several times on a single processor of SUN SparcServer 1000 (50MHz) the query took an average of 90.93 to scan the SwissProt database (14,147,368 amino acids in 40,292 protein), or approximately 155,600 comparisons per second. (One might expect a somewhat better rate with DNA sequences because the sequences tend to be longer – amortizing the cost of reading in the sequences – and because direct matching is used rather than matching by groups.

Much work still needs to be done to improve the overall speed of execution. One possibity is to rework the system to perform the database searches in parallel, perhaps using the process–farm model. Work is also under way to find measures that more accurately characterize the similarity between sequences. For example, all of the current metrics are based on the length of the tiles that have been created. This tacitly assumes that matches are of equal likelihood, but but this is not the case; for single amino acids Leu is roughly 8 times more likely than Trp, and by the time one looks at tripeptides, the most common tripeptide (AAA) is 400 times more likely than than the least likely, MWW. Work is therefore currently being done to improve the scoring metric by considering metrics based on the sums of the log–odds values for particular polypeptide subsequences. Preliminary results are encouraging; using a metric similar to the original *score* metric, but based on summing dipeptide log–likelihoods (viewed as a Markov process), and a minimum–match–length of 4 the numbers of common matches were Blast vs Fasta 81, Blast vs Neweyes 60 and Fasta vs Neweyes 62. Looking more closely, 45 matches returned by Neweyes are also returned by Blast and Fasta, 15 are only shared with Blast and 17 only shared by Fasta. Appendix 3 contains a list of the matchess. The letters B and/or F before an entry indicates a common entry with either Blast or Fasta, respectively. In addition a number of matches are annotated with a ** or a ??, indicating probable and possible matches not seen by either Blast or Fasta (based on similarity of their descriptions with matches returned by Blast or Fasta).

In tandem with this work, experiments are currently underway looking at the application of Neweyes to different types of searches. In particular, because Neweyes has the ability to detect rearranged and repeated substrings, it will be interesting to investigate the prevalence of these sorts of phenomena in databases such as EMBL. In the final analysis, with its different viewpoint and its integrated range of facilities, Neweyes is not intended as a replacement for either Blast or Fasta, but as an additional service, in some sense orthogonal to the latter.

Appendix 1:Listing of Longer string RAS1_YEAST with RASH_HUMAN superimposed

```
                5     10             22                         47
   1:MQGNKSTIR EYK I VVVG G GGVGKSALTIQ FIQSY FVDEYDPTIEDSYRKQVVID DKVSI L
               3     7      12                26                         53
                  68           79                            107
  61:DILLDTAGQEEYSAMR E QYMRTGEGFL LVYSVTSRNSFDELLSYYQQIQ RVKDSD YIPVVV
                        70                                102
       117                              147   85      157      161
 121:VGNK LDLENERQVSYEDGLRLAKQLNAPFL ETSAK Q AIN VDE AFY SLIR LVR DDGGKYNS
     114                             143     83      155      159
                164                                            50
 181:MNRQLDNTN EIR DSELTSSATADIEKKNNGSYVLDNSLTNAGTGSSSKSAVNHN GET TKR
               162                                            48
                      88
 241:TDEKNYVNQNNNNEG NTK YSSNGNGNRSDISRGNQNNALNSRSKQSAEPQKNSSANARKE
                     86

 301:SSGGCCIIC
```

Appendix 2: List of Top 200 Matches Returned from Query EGF_HUMAN

BF EGF_HUMAN	BF EGF_RAT	BF PRTS_BOVIN	YBA4_YEAST	BF 7LES_DROME
BF EGF_MOUSE	BF LDLR_RABIT	POLG_PVYHU	** ITB1_XENLA	POLG_HCVB
BF FBN1_HUMAN	BF TGFB_HUMAN	BF UN52_CAEEL	F ITB0_XENLA	F RRPB_CVMJH
BF FBN2_HUMAN	BF LDVR_RABIT	B TSP3_MOUSE	BF EGFH_STRPU	F ITB2_HUMAN
BF NOTC_DROME	B VWF_HUMAN	B ITB1_MOUSE	RYNC_RABIT	F POLG_COXB5
BF TGFB_RAT	POLG_PVYN	B LMG1_MOUSE	GCN2_YEAST	?? ITA4_MOUSE
BF FBLD_MOUSE	BF NIDO_HUMAN	B YN81_CAEEL	NOS2_RAT	RRPB_CVMA5
BF FBL2_MOUSE	BF NIDO_MOUSE	BF TRBM_HUMAN	B ITB1_CHICK	HMD1_YEAST
BF YNX3_CAEEL	?? POLN_EEVVT	?? DYHC_DICDI	G156_PARPR	?? POLN_SINDO
BF FBLC_MOUSE	B POLN_EEVV3	F RRPA_CVH22	PCX_DROME	SPCN_CHICK
BF LDLR_HUMAN	BF CRB_DROME	BF LDL2_XENLA	TOXA_PSEAE	?? POLN_SINDV
BF NOTC_XENLA	?? POLN_EEVVP	F BAR3_CHITE	POLG_TBEVS	GLTB_ECOLI
BF LDLR_MOUSE	BF LDL1_XENLA	BF ITB2_BOVIN	F MCAS_MYCBO	E75A_DROME
BF LDLR_RAT	B LMA1_HUMAN	BF AGRI_RAT	BF ITB2_MOUSE	DPP4_RAT
F MUC2_HUMAN	BF PRTS_HUMAN	** FINC_RAT	RRPA_CVMJH	BF PERT_HUMAN
BF FBLB_HUMAN	NF1_HUMAN	F HTS1_COCCA	POLG_COXA9	BF PERU_HUMAN
BF FBLA_HUMAN	NF1_MOUSE	RPOA_LELV	SWH1_YEAST	FOR4_MOUSE
BF FBLD_HUMAN	F VGF1_IBVB	KPCL_HUMAN	POLG_DEN2J	C1TC_HUMAN
BF FBLC_HUMAN	B LMG1_HUMAN	F FAS_CHICK	BF LMA1_MOUSE	B AGRI_CHICK
BF LDLR_CRIGR	SPCB_DROME	BF SERR_DROME	G168_PARPR	F POLG_TBEVW
BF EGF_RAT	BF PRTS_BOVIN	YBA4_YEAST	BF 7LES_DROME	POLG_COXB3
BF LDLR_RABIT	POLG_PVYHU	** ITB1_XENLA	POLG_HCVB	BGAL_MOUSE
BF TGFB_HUMAN	BF UN52_CAEEL	F ITB0_XENLA	F RRPB_CVMJH	BF PERT_RAT
BF LDVR_RABIT	B TSP3_MOUSE	BF EGFH_STRPU	F ITB2_HUMAN	ANPA_RAT
B VWF_HUMAN	B ITB1_MOUSE	RYNC_RABIT	F POLG_COXB5	B GRN_CAVPO
POLG_PVYN	B LMG1_MOUSE	GCN2_YEAST	?? ITA4_MOUSE	BRC1_HUMAN
BF NIDO_HUMAN	B YN81_CAEEL	NOS2_RAT	RRPB_CVMA5	NEST_RAT
BF NIDO_MOUSE	BF TRBM_HUMAN	B ITB1_CHICK	HMD1_YEAST	IRR_CAVPO
?? POLN_EEVVT	?? DYHC_DICDI	G156_PARPR	?? POLN_SINDO	F LMB1_DROME
B POLN_EEVV3	F RRPA_CVH22	PCX_DROME	SPCN_CHICK	BF LI12_CAEEL
BF CRB_DROME	BF LDL2_XENLA	TOXA_PSEAE	?? POLN_SINDV	POLG_TMEVD
?? POLN_EEVVP	F BAR3_CHITE	POLG_TBEVS	GLTB_ECOLI	RRPL_UUK
BF LDL1_XENLA	BF ITB2_BOVIN	F MCAS_MYCBO	E75A_DROME	B COMP_RAT
B LMA1_HUMAN	BF AGRI_RAT	BF ITB2_MOUSE	DPP4_RAT	POLG_COXA4
BF PRTS_HUMAN	** FINC_RAT	RRPA_CVMJH	BF PERT_HUMAN	YLS4_CAEEL
NF1_HUMAN	F HTS1_COCCA	POLG_COXA9	BF PERU_HUMAN	POLG_COXB4
NF1_MOUSE	RPOA_LELV	SWH1_YEAST	FOR4_MOUSE	RRPO_ACLSV
F VGF1_IBVB	KPCL_HUMAN	POLG_DEN2J	C1TC_HUMAN	POLG_DEN27
B LMG1_HUMAN	F FAS_CHICK	BF LMA1_MOUSE	B AGRI_CHICK	SRN1_YEAST
SPCB_DROME	BF SERR_DROME	G168_PARPR	F POLG_TBEVW	POLG_DEN26

References

Altschul, Stephen F., Gish, Warren, Miller, Webb, Myers, Eugene W. and Lipman, David J. (1990), Basic Local Alignment Search Tool. *Journal of Molecular Biology* **215**(3), pp. 403–410.

Cormen, Thomas H., Leiserson, Charles E. and Rivest, Ronald L. (1990), *Introduction to Algorithms.* MIT Press.

Dayhoff, M. O., Schwartz, R. M. and Orcutt, B. C. (1978), A Model of Evolutionary Change in Proteins. In M. O. Dayhoff (eds),*Atlas of Protein Sequence and Structure.*, pp. 345–352, National Biomedical Research Foundation, Washington.

Dumas, Jean–Pierre and Ninio, Jacques (1982), Efficient Algorithms for Folding and Comparing Nucleic Acid Sequences. *Nucleic Acids Research* **10**(1), pp. 197–206.

Gonnet, G. H. and Baeza–Yates, R. (1991), *Handbook of Algorithms and Data Structures (Second Edition).* Addison–Wesley.

Gonnet, G. H. and Baeza–Yates, R. A. (1990), An Analysis of the Karp–Rabin String Matching Algorithm. *Information Processing Letters* **34**, pp. 271–274.

Karlin, Samuel, Morris, Macdonald, Ghandour, Ghassan and Leung, Ming–Ying (1988), Algorithms for Indentifying Local Molecular Sequence Features. *CABIOS* 4(1), pp. 41–51.

Karp, Richard M. and Rabin, Michael O. (1987), Efficient Randomized Pattern–Matching Algorithms. *IBM Journal of Research and Development* **31**(2), pp. 249–260.

Kruskal, Joseph B. (1983), An Overview of Sequence Comparison. In David Sankoff and Joseph B. Kruskal (eds),*Time Warps, String Edits and Macromolecules: The Theory and Practice of Sequence Comparison.*, pp. 1–44, Addison Wesley (Chapter 1).

Lipman, David J. and Pearson, William R. (1985), Rapid and Sensitive Protein Similarity Searches. *Science* **227**, pp. 1435–1441.

Luthy, Roland, McLachlan, Andrew D. and Eisenberg, David (1991), Secondary Structure–Based Profiles: Use of Structure Conserving Scoring Tables in Searching Protein Sequence Databases for Structural Similarities. *Proteins: Structure, Function and Genetics* **10**, pp. 229–239.

Needleman, Saul B. and Wunsch, Christian D. (1970), A General Method Applicable to the Search for Similarities in the Amino Acid Sequence of Two Proteins. *Journal of Molecular Biology* **48**, pp. 443–453.

Pearson, William R. (1990), Rapid and Sensitive Sequence Comparison with FASTP and FASTA. In Russell .F. Doolittle (eds),*Molecular Evolution: Computer Analysis of Protein and Nucleic Acid Sequences.*, pp. 63–98, Academic Press (Methods in Enzymology, Vol. 183).

Pearson, William R. and Lipman, David J. (1988), Improved Tools for Biological Sequence Comparison. *Proceedings of the National Academy of Science U.S.A.* **85**, pp. 2444–2448.

Schwartz, R. M. and Dayhoff, M. O. (1978), Matrices for Detecting Distant Relationships. In M. O. Dayhoff (eds),*Atlas of Protein Sequence and Structure.*, pp. 353–358, National Biomedical Research Foundation, Washington.

Smith, T. F. and Waterman, M. S. (1981), Identification of Common Molecular Subsequences. *Journal of Molecular Biology* **147**, pp. 195–197.

States, David J. and Boguski, Mark S. (1992), Similarity and Homology. In Michael Gribskov and John Devereux (eds),*Sequence Analysis Primer.*, pp. 89–157, W. H. Freeman.

Wilbur, W. J. and Lipman, David J. (1983), Rapid Similarity Searches of Nucleic Acid and Protein Data Banks. *Proceedings of the National Academy of Science, U.S.A.* **80**, pp. 726–730.

Wise, Michael J (1992), Detection of Similarities in Student Programs: YAP'ing may be Preferable to Plague'ing. *Twenty–Third SIGCSE Technical Symposium*, Kansas City, USA, pp. 268–271.

Wise, Michael J (1994), Running Karp–Rabin Matching and Greedy String Tiling. *Software – Practice and Experience* (Submitted to journal (revision of Basser Department of Computer Science Technical Report 463)).