



The University of Sydney

**Running Karp-Rabin Matching
and Greedy String Tiling**

Technical Report Number 463

March 1993

Michael J Wise

ISBN 0 86758 669 9

**Basser Department of Computer Science
University of Sydney NSW 2006**

Neweyes: A System for Comparing Biological Sequences Using the Running Karp-Rabin Greedy String-Tiling Algorithm¹

Michael J. Wise

Department of Computer Science,
University of Sydney, Australia
michaelw@cs.su.oz.au
FAX: +61 2 692 3838

Keywords: Efficient sequence search, structure matrices, transposed subsequences.

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. 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.

1. Submitted to Software - Practice and Experience

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

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. 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.

1. Introduction

What will be described in this paper is a system, called Neweyes², 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 as either a percentage match value, the length of the common portions or as a printout showing 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 [Wise93].)

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 due to Needleman and Wunsch [Need70] and uses dynamic programming. A similar approach evolved in the computer science literature, where the problem came to be known as the longest common subsequence³ (LCS) problem. The Needleman Wunsch system was in fact one of the earliest solutions to these 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 by Cormen, Leiserson, Rivest [Corm90] and Gonnet, Baeza-Yates [Gonn91]). 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 than 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 of the Needleman Wunsch algorithm that has become popular is due to Smith and Waterman [Smit81], and involves finding what have come to be known as local alignments. That is, by allowing gap penalties to increase as a function of gap or insertion size⁴, alignment scores may be reduced to zero. 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 $O(n^2)$

2. With the different perspective that the system offers 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 the Tom Lehrer song, *Lobachevsky*, "(Let) Noone Else's Work Evade Your EyeS" – reflecting the underlying algorithm's ancestry in the detection of suspected plagiarism [Wise93].

3. 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*.

4. The gap penalties are these days separated into a gap creation penalty and a gap extension penalty [Stat92].

[Krus83]. Furthermore, the choice 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 ([Lipm85], [Pear88], [Pear90]). 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 and Ninio [Duma82]. 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 $\alpha^k - \alpha$ 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 [Wilb83]. Local *regions* with a high density of k -tuples are then identified [Pear90]. In the second phase, the ten regions with the highest score are reevaluated using the PAM 250 matrix and for each of these regions, the subregion is found with the best score and reported as the *initl* 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 *initm* score. In the final phase, a Needleman Wunsch alignment is performed on those pairs of sequences that had the highest *initm* 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 *initm* scores. However, this strength is also a source of some weakness; As noted by several authors, e.g. States and Boguski [Stat92], choosing 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 [Alts90]. 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.

2. Running Karp-Rabin Greedy String Tiling – 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 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 /* Currently 20 */
stop := false
Repeat
  Lmax := scanpattern(s) /* Lmax is the size of the largest maximal-matches found in this iteration */
  if Lmax > 2 × s then s := Lmax /* Very long string; don't mark tiles but try again with larger s */
  else
    markstrings(s) /* Create tiles from matches takes from list of queues */
    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 [Karp87] 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 and Baeza-Yates [Gonn90].) Running Karp-Rabin matching extends Karp-Rabin matching in the following ways:

1. 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.

- 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  $\leq s$  then advance  $t$  to first unmarked token after next tile /* Just for efficiency */
  else create the KR hash-value for substring  $T_t \cdots T_{t+s-1}$  and add to hashtable
```

```
Starting at the first unmarked token of P, for each unmarked  $P_p$  do
  if distance to next tile  $\leq s$  then advance  $p$  to first unmarked token after next tile
  else
```

```
    create the KR hash-value for substring  $P_p \cdots P_{p+s-1}$ 
    check hashtable for hash of KR hash-value
    for each hash-table entry with equal hashed KR hash-value do
       $k := s$ 
      while  $P_{p+k} = T_{t+k}$  AND unmarked( $P_{p+k}$ ) AND unmarked( $T_{t+k}$ ) do
         $k := k + 1$  /* Extend match until non-match or element marked */
      if  $k > 2 \times s$  then return( $k$ ) /* abandon scan. will be restarted with  $s = k$  */
      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, while there is a non-empty queue do
  if the current queue is empty then drop to next queue /* corresponding to smaller maximal-matches */
  else
    remove match( $p, t, L$ ) from queue /* Assume the length of maximal-matches in the
                                          current queue is  $L$  */
    if match not occluded then
      if for all  $j: 0 \cdots s - 1, P_{p+j} = T_{t+j}$  then /* IE match is not hash artefact */
        for  $j := 0$  to  $L - 1$  do
          mark_token( $P_{p+j}$ )
          mark_token( $T_{t+j}$ )
          length_of_tokens_tiled := length_of_tokens_tiled +  $L$ 
      else if  $L - L_{occluded} \geq s$  then /* IE the unmarked part remaining of the maximal-match */
        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 [Wise93].)

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 $2 \times s$) 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 [Wise93] 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 were 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.

3. 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 Dayhoff et al. [Schw78]. 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 L thy et al. [Luet9], 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 PAM 250 matrix found in [Schw78] 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: $G \equiv T$ and $G \equiv P$. (The order of amino acids within groups and between the various groups is irrelevant.)

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 [Dayh78]). 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 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) do
  if LogOdds(AA1, AA2) > threshold then
    if AA1 <> AA2 then /* AA1 ≡ AA2 already part of group */
      pos_links := 0; /* Number of above threshold edges added to graph by addition of new edge */
      neg_links := 0; /* Number of edges at or below threshold */
      temp_excl := {}; /* Set of excluded edges due to 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 components */
        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 PAM 250 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 this time using a set of groups and excluded mutations resulting from the application of the `formgroups` algorithm to the *Beta* matrix from [Luet91]. The resulting groups are:

```

AGTSQKRNE D
WFYH
LIVM
C
P

```

from which the following mutations are explicitly disallowed:

$G \equiv N$, $G \equiv K$, $A \equiv D$, $A \equiv E$, $A \equiv R$, $A \equiv K$, $A \equiv Q$

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.

4. 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 then 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 covered by tiles divided by the length

- of the smaller biosequence (expressed as a percentage)
- Then total number of tokens from one biosequence marked (i.e. covered by tiles)
- The length of the longest tile.
- The length of the average tile (for a single biosequence).

Assuming the database file contains a number of different sequences (including unrelated ones), a mean and standard deviation are also returned for the selected statistic.

The second mode of operation is in a "micro" capacity, where a single pair of biosequences are being compared. In this case, more complex output formats are possible, including a dot-plot (for which tiling is switched off and just the maximal-matches are recorded). Alternatively, one can obtain a listing displaying the pattern string overlaid with the matching portions of the text string (or vice versa).

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

- Selection of exact matching or matching-by-group.
- Where matching-by-group is being employed, it is possible to specify an ascii file listing the groups (one per line) and a second file listing 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 string (so several text substrings may form tiles with the same pattern substring).
- When a new match is to be retrieved from the topmost queue in the list of queues, (in `markstrings`) the queue is first examined to see whether a match exists that is "close" to a tile that has already been laid, i.e. within search-length tokens of an existing tile on one string, and within 20 token on the second string. If no such maximal-match exists, the first on the current queue is chosen. This has the effect of extending ascending sequences of tiles.

5. Discussion

At first appearances, Running Karp-Rabin Greedy String Tiling (RKR-GST) algorithm has similarities with the algorithms underlying both FASTA and BLAST. In fact there are significant differences which probably 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 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 encode of tuples or a finite-state-machine, i.e. 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 required by RKR-GST grows in line with length of the input strings, and while the amount of dynamic storage is dependent on the number of maximal-matches, in practice it also grows in line with the lengths of the input strings. 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.

In terms of what is produced, when a tiling is done (as opposed to returning all possible matches) the output resembles multiple BLAST alignments. Two examples are presented.

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 group matching with the groups derived from L thy et al.'s beta matrix and a minimum-match-length of 3 the statics of the tiling are:

```
RASH_HUMAN (189) : RAS1_YEAST (309)
Number of marked tokens 165 / 189 (87.30%) in 16 tiles
Longest tile: 28
Average tile: 10.31
```

Appendix 1 is a dotplot showing the alignment, and Appendix 2 is a listing of the text-string, RAS1_YEAST, with RASH_HUMAN superimposed on it where there is a match. Appendix 3 is a dotplot of the same alignment, but this time using groups derived from L thy et al.'s alpha matrix. Notice that it tells essentially the same story, with the main difference being the emergence of two long tiles near the ends of the biosequences. The statistics for this match are:

RASH_HUMAN (189) : RAS1_YEAST (309)
Number of marked tokens 179 / 189 (94.71%) in 18 tiles
Longest tile: 29
Average tile: 9.9

As a second example, this time illustrating a weak match, Appendix 4 contains the alignment of CYC_KATPE (Cytochrome C, AC. P00025) against PAPA_CARPA (Papain Precursor, AC. P00784). First, the basic statistics of the matching-by-group tiling, with groups derived from L thy et al.'s alpha matrix and minimum-match-length set to 3, are:

CYC_KATPE (103) : PAPA_CARPA (345)
Number of marked tokens 96 / 103 (93.20%) in 14 tiles
Longest tile: 13
Average tile: 6.86

Appendix 5 contains a listing of the text-string (PAPA_CARPA) with the matching pieces of CYC_KATPE superimposed, Appendix 6 a dotplot of that tiling, while Appendix 7 is a dotplot of all the matches of length 5 or greater (for greater clarity. However, the groups used for matching are the same). Note that in both examples, smoothing as been use to improve the contiguity of the tilings. Note that the percentage match value is not a good indicator of a weak match; while a low value does indicate a weak match, a high value may be the result of a large number of very short matches. This is what has occurred in the comparison of CYC_KATPE and PAPA_CARPA as is evidenced by the far smaller average tile length of 6.86 versus 10.31 for RASH_HUMAN and RAS1_YEAST.

Note that in all the above tilings, the `smooth`, option has been used.

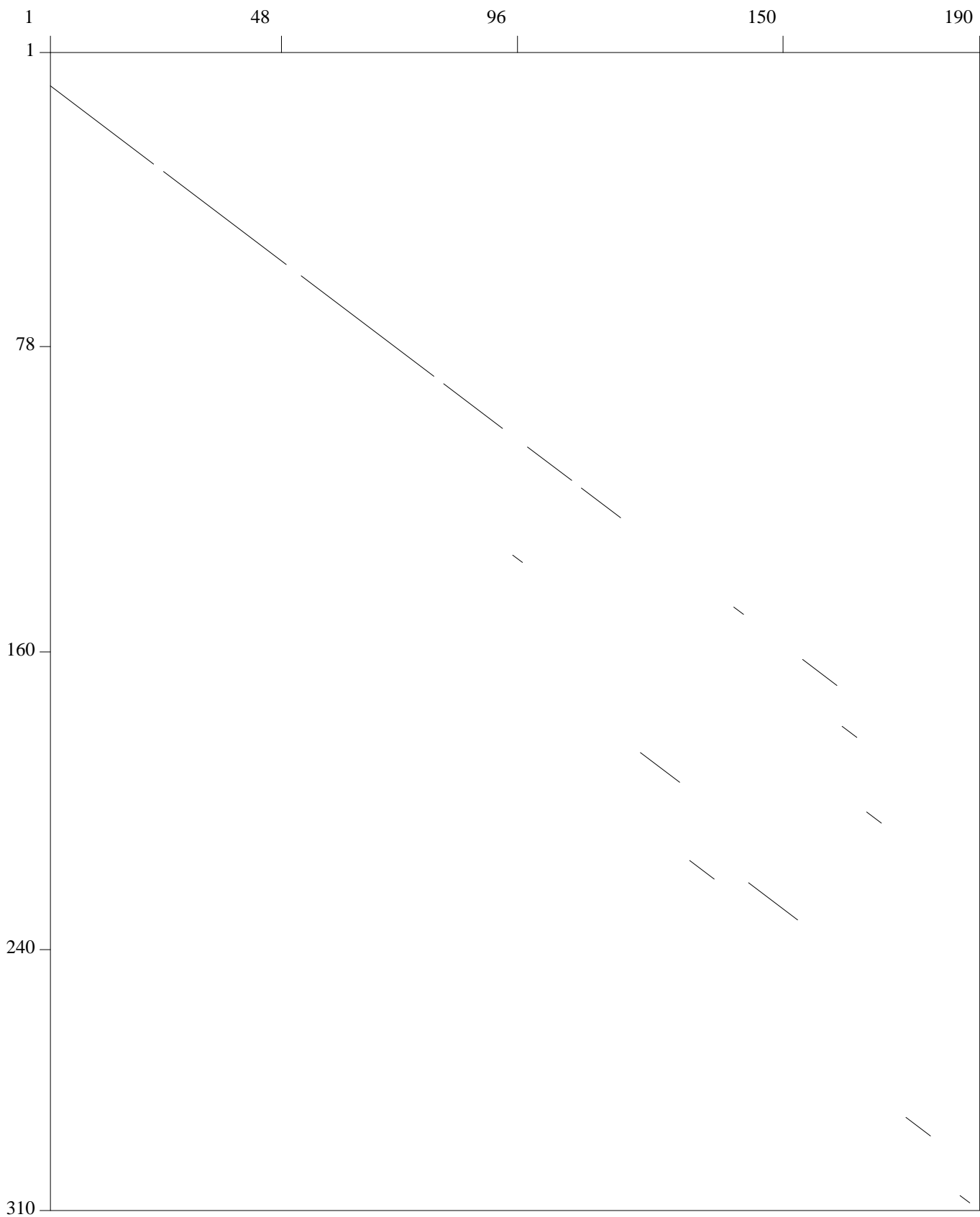
As the system currently stands, the pairwise comparison facilities described above have been implemented, and when run on a MIPS 3030 with a clock-frequency of 25MHz and 64 Mbytes memory, a rate of speed of around 15,000 residues/second is achieved. (On our DEC Alpha, the speed is around 24,000 residues/second.) However, comparisons between a number of biosequences is currently being handled by a shell-script, so the next step will be to bring the file comparisons within the main software system and to improve the overall speed of execution. Once this has been achieved (it should be completed by May), it will then be possible to compare the performance of the Neweyes with that of the other systems as a vehicle for database searching. The one facility that Neweyes has that the other systems lack is the ability to detect transposed substrings, so it will be interesting to investigate the prevalence of transposed substrings in the databases.

6. Bibliography

- [Alts90] Altschul, Stephen F., Warren Gish, Webb Miller, Eugene W. Myers and David J. Lipman, "Basic Local Alignment Search Tool", *Journal of Molecular Biology* **215**(3), pp. 403–410 (1990).
- [Corm90] Cormen, Thomas H., Charles E. Leiserson and Ronald L. Rivest, *Introduction to Algorithms*, MIT Press (1990).
- [Dayh78] Dayhoff, M. O., R. M. Schwartz and B. C. Orcutt, "A Model of Evolutionary Change in Proteins", *Atlas of Protein Sequence and Structure*, ed. M. O. Dayhoff, pp. 345–352, National Biomedical Research Foundation, Washington (1978).
- [Duma82] Dumas, Jean-Pierre and Jacques Ninio, "Efficient Algorithms for Folding and Comparing Nucleic Acid Sequences", *Nucleic Acids Research* **10**(1), pp. 197–206 (1982).
- [Gonn90] Gonnet, G. H. and R. A. Baeza-Yates, "An Analysis of the Karp-Rabin String Matching Algorithm", *Information Processing Letters* **34**, pp. 271–274 (1990).

- [Gonn91] Gonnet, G. H. and R. Baeza-Yates, *Handbook of Algorithms and Data Structures (Second Edition)*, Addison-Wesley (1991).
- [Karp87] Karp, Richard M. and Michael O. Rabin, “Efficient Randomized Pattern-Matching Algorithms”, *IBM Journal of Research and Development* **31**(2), pp. 249–260 (March 1987).
- [Krus83] Kruskal, Joseph B., “An Overview of Sequence Comparison”, *Time Warps, String Edits and Macromolecules: The Theory and Practice of Sequence Comparison*, ed. David Sankoff and Joseph B. Kruskal, pp. 1–44, Addison Wesley (1983) (Chapter 1).
- [Lipm85] Lipman, David J. and William R. Pearson, “Rapid and Sensitive Protein Similarity Searches”, *Science* **227**, pp. 1435–1441 (March 1985).
- [Luet91] Luthy, Roland, Andrew D. McLachlan and David Eisenberg, “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 (1991).
- [Need70] Needleman, Saul B. and Christian D. Wunsch, “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 (1970).
- [Pear88] Pearson, William R. and David J. Lipman, “Improved Tools for Biological Sequence Comparison”, *Proceedings of the National Academy of Science U.S.A.* **85**, pp. 2444–2448 (1988).
- [Pear90] Pearson, William R., “Rapid and Sensitive Sequence Comparison with FASTP and FASTA”, *Molecular Evolution: Computer Analysis of Protein and Nucleic Acid Sequences*, ed. Russell .F. Doolittle, pp. 63–98, Academic Press (1990) (Methods in Enzymology, Vol. 183).
- [Schw78] Schwartz, R. M. and M. O. Dayhoff, “Matrices for Detecting Distant Relationships”, *Atlas of Protein Sequence and Structure*, ed. M. O. Dayhoff, pp. 353–358, National Biomedical Research Foundation, Washington (1978).
- [Smit81] Smith, T. F. and M. S. Waterman, “Identification of Common Molecular Subsequences”, *Journal of Molecular Biology* **147**, pp. 195–197 (1981).
- [Stat92] States, David J. and Mark S. Boguski, “Similarity and Homology”, *Sequence Analysis Primer*, ed. Michael Gribskov and John Devereux, pp. 89–157, W. H. Freeman (1992).
- [Wilb83] Wilbur, W. J. and David J. Lipman, “Rapid Similarity Searches of Nucleic Acid and Protein Data Banks”, *Proceedings of the National Academy of Science, U.S.A.* **80**, pp. 726–730 (February 1983).
- [Wise93] Wise, Michael J, “Running Karp–Rabin Matching and Greedy String Tiling”, Bassler Department of Computer Science Technical Report 463 (March 1993) (Submitted to Software – Practice and Experience).

Appendix 1: Plot of **ras1_yeast.y3** (Y-axis) against **rash.y3** (X-axis)
Matching by amino-acid group beta used



Appendix 2: Listing of Text string ras1_yeast.y3 (with rash.y3)

Matching By amino-acid group beta used

1: MQGNKST ^{MT} ^L ^A **IREYKIVVVG** ^{NH} **GGGVGKSALTIQ** F ^{GE} ^L **IQSYFVDEYDPTIEDSYRKQVVIDK** VS **IL**
 1 79 24 93 107 52

61: **DILDTAGQEEYSAMRE** ^D **QYMRTGEGFL** L ^{FAINNTK} ^{EDI} **VYSVTSRNSFDEL** LSYY ^E ^K **QIQRVKDS** Y ^V ^M ^L **IPVV**
 117 97 81 142 98 161 109

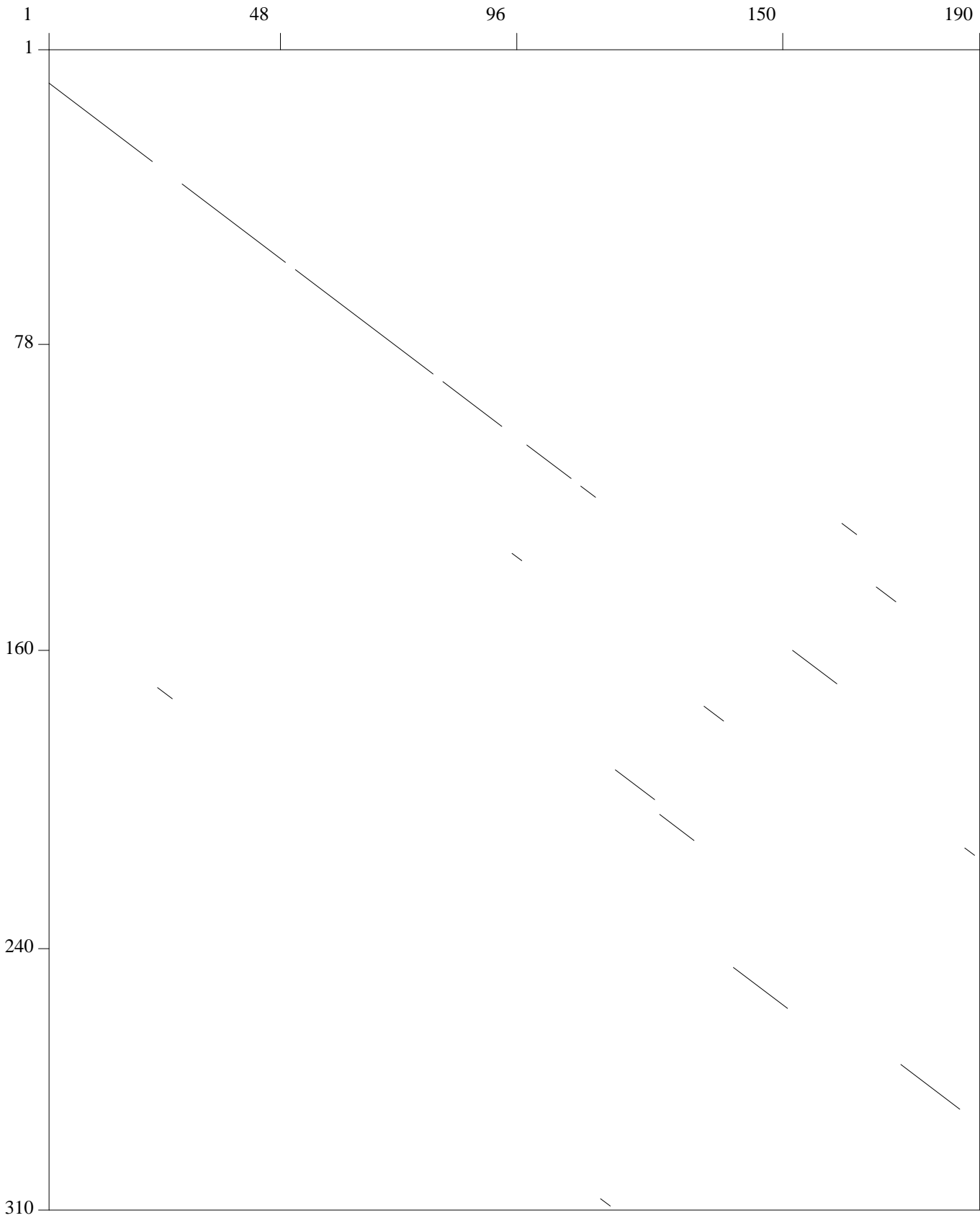
121: **VGNK** LDLENERQV ^Q ^R **SYE** DGLRLAKQLNA ^{YI} **PFL** ETSAKQAINVD ^D ^T ^V **EAFYSLIR** LVRDDGGKYN ^E **S**
 165 129 95 170 140 136 154 153 162

181: ^{IRQ} **MNR** QLD ^{AARTVESRQ} **NINEIRDSE** LTSSATA ^{KLR} **DIEK** KNNGSYVLD ^{QD} ^{ARS} ^E ^{SAKTRQG} ^E **NSLINA** **GTGSSSKSAVN** HNGETTKR
 121 167 131 143 180

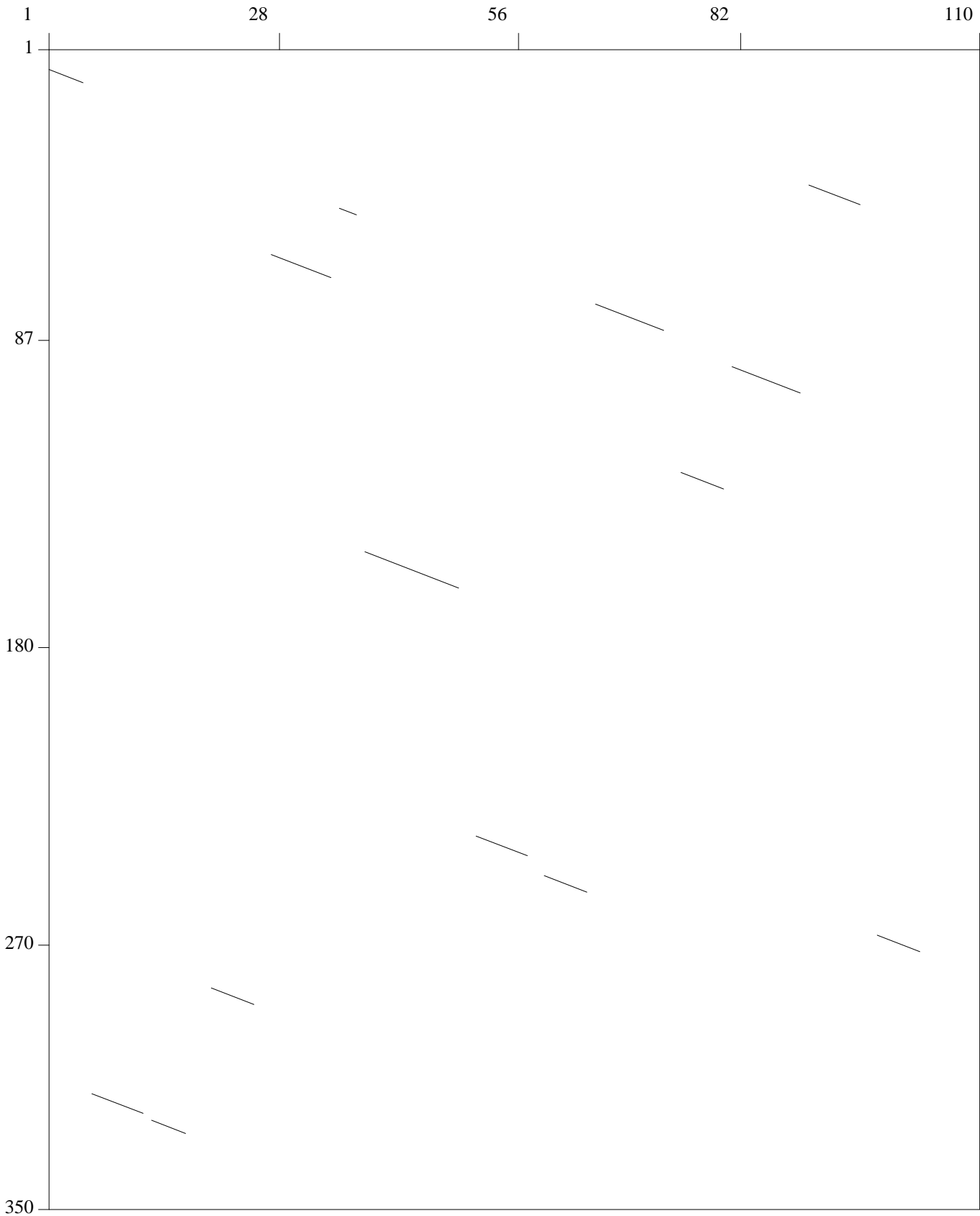
241: TDEKNYVNQNNNNEGNTKYSSNGNGNRSDISRGNQNNALNSRSK ^{DESG} ^G **QSAEPQ** KNSSANARKE
 175

301: SSGGC ^{VL} **CII** C
 186

Appendix 3: Plot of **ras1_yeast.y3** (Y-axis) against **rash.y3** (X-axis)
Matching by amino-acid group alpha used



Appendix 4: Plot of PAPA_CARPA.y3 (Y-axis) against CYC_KATPE.y3 (X-axis)
Matching by amino-acid group alpha used



Appendix 5: Listing of Text string PAPA_CARPA.y3 (with CYC_KATPE.y3)

Matching by amino-acid group alpha used

1: MAMI ^{GDVA}**PSISK** LLFVAICLFVYMGLSFGDFSIVGYSQNDLT ^{ERQD VA G}**STERLIQLFE** SWMLKHNKIYK

61: ^{KVGPNLWG}**NIDEKIYR** FEIFKDN ^{ME LENPK}**LKYIDEINK** KNNSYWLGLN ^{I GIKKKG}**VFADMSNDE** FKEKYTGSIAAGNYTTTE

121: LSYEEV ^{IPGTKM}**INDGDV** NIPEYVDWRQKGAVTPVK ^{RKTGQAEGYSYT}**NQSGCSCWAFS** AVVTIEGIIKIRTGNLNE

181: YSEQELLDCCRYSYGCNGGYPWSALQLVAQYGIHYRNTYPYEGVQRYCRSREKGPY ^{N S}**AAKT**

241: ^{K I}**DGV** RQVQP ^{W NT}**YNEGAL** LYSIANQPVSVV ^{KS TS}**LEAAGK** DFQLYRGGIF ^{ENG K}**VGPCGN** KVDHAVAAVGYG

301: PNYILIKNSWGTGW ^{KKTFVQ}**GENGYIR** I ^{CAQC}**KRGIG** NSYGVCGLYTSSSFYPVKN

Appendix 6: Plot of **PAPA_CARPA.y3** (Y-axis) against **CYC_KATPE.y3** (X-axis)
Matching by amino-acid group alpha used

