Adrasteia - A Short Peptide Search Tool

Miles Hampson

This report is submitted as partial fulfilment
of the requirements for the Honours Programme of the
School of Computer Science and Software Engineering,
The University of Western Australia,
2006

Abstract

A new approach to short peptide searches, Adrasteia (Agrep for Doing Rapid Accurate Searching of Text from Externally Input Amino acids), uses the speedup provided by the bit parallel operations of the agrep utility to perform fast searching for proteins in biological databases. Given input in the form of short peptide sequences, it can score and evaluate likely protein matches, reducing the time taken for searches by using an indexing method that complements the matching work done by the agrep utility. It can search for more than one input peptide at a time, and allows the user to group together inputs that are expected to occur together, reducing the time taken to perform multiple searches compared with running them separately. The result is a fast short peptide search method that in many cases returns more accurate results than existing protein search programs.

Keywords: Algorithms, Approximate Matching, Biological Sequences, Probability Distributions, Alignment Scoring, Nondeterministic Finite Automata, Database Filtering, Indexing
CR Categories: J.3 [Computer Applications]: Life And Medical Sciences

Acknowledgements

My deepest thanks to my supervisor, Michael Wise, for the encouragement and insightful guidance throughout the course of the project. I would also like to thank Gordan Royle and Lyndon While for the constructive and helpful feedback on the contents and presentation of this report. Finally I am indebted, as always, to my friends and family for the constant support and optimism throughout the year.

Contents

List of Tables

List of Figures

CHAPTER 1

Introduction

1.1 Motivation

Events such as the completion of the draft human genome sequence on June 26, 2000, highlight the growing quantities of data in biological databases around the world. The practical use of this data is limited, however, if the functions and interactions of its expression within organisms are not understood. A first step in discovering the function of a newly discovered gene is often to look for homologies between the amino acids it codes for and sections of an amino acid database.

The Basic Local Alignment Search Tool (BLAST) [2] is the tool most frequently used for calculating this sequence similarity [11]. This project aims to examine the performance and accuracy of BLAST in two areas where it will potentially perform poorly. These areas are searching for short amino acid sequences, and searching for multiple short amino acid sequences simultaneously. An alternative method of searching for these sequences is then presented, and the performance and accuracy of an implementation of this method examined and compared to BLAST.

The aim of this research is to demonstrate differences in the performance and accuracy of the two search tools, and from this to draw conclusions as to the best method for searching for similarities in short and multiple amino acid sequences.

1.2 Terminology

This work describes a method of searching for peptide fragments in a protein database. A protein is an organic polymer, put together from a set of twenty amino acids. A protein can be modelled as a string composed of symbols from this alphabet of twenty, and a representation of a protein in this format can be stored as the body of a record in a database, along with header information

containing the name of the protein and what organism it came from. Peptides are short protein fragments, usually under 40 amino acids in length.

1.3 Development Methodology

The search method presented in this project is implemented in a program called Adrasteia, sections of which can be found in appendix B of this work. This program was developed in order to quantitatively test the search method against the method employed by BLAST, and as such it was necessary that the development and testing of this program follow well defined software engineering practices.

In order to choose a suitable development model for this project, it was necessary to consider the constraints under which work would need to proceed. Some of the more important considerations were:

- Limiting the large potential scope of the project.
- The need to develop a piece of software with a high confidence of correctness.
- The risk of the open nature of the project requiring redesigns at later stages.

To take account of these concerns, the development process was structured using an agile development framework [6]. The advantages of using an agile type method for developing this project were:

- Short development time frames reduce the risk of exploring new areas.
- Getting small pieces of software working allows for continual testing throughout development.
- Having working code with new functionality at regular intervals allows for better feedback from a project supervisor.
- The model encourages modular development of software, increasing flexibility if problems require a redesign of the program.

The use of these methods ensures that the deliverable product at the end of the project is well tested, able to provide a comparison with the BLAST program, and modular and easily upgradeable if future work requires it.

Use Case Name	**Index Database**
Actors	User
Preconditions	Database is specified
Postconditions	Index files exist for the database.
Process Description	**Trigger:** Program is started with the given database. 1. Database is parsed. 2. Database index files are created.
Use Case Name	**Indexed Match**
Actors	User
Preconditions	Index files exist for the database.
Postconditions	Results are output to the screen
Process Description	**Trigger:**User enters query details and selects indexed match. 1. Index files are searched for good match areas. 2. Match areas are checked using agrep. 3. Matches are ranked by likeliness. 4. Most likely matches are displayed to the user.
External Dependencies	Agrep
Use Case Name	**Direct Match**
Actors	User
Preconditions	Program is running
Postconditions	Results are output to the screen
Process Description	**Trigger:**User enters query details and selects direct match. 1. All possible match areas are checked using agrep. 2. Matches are ranked by likeliness. 3. Most likely matches are displayed to the user.
External Dependencies	Agrep

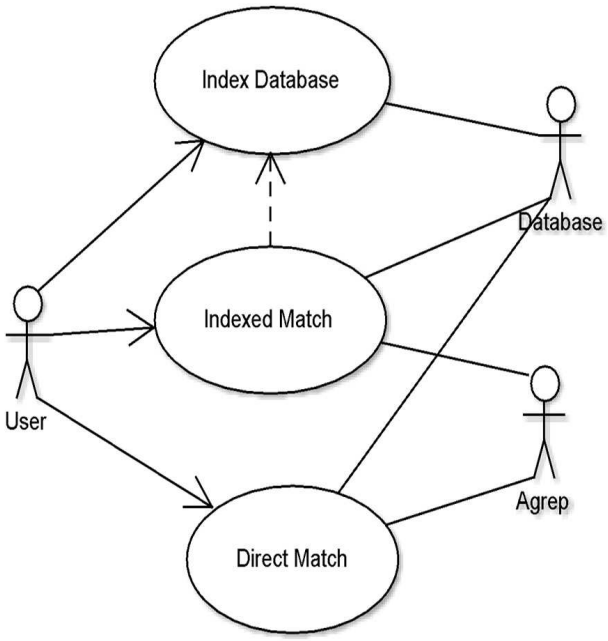

Figure 1.1: Functionality of the Adrasteia Program

1.4 Project Requirements

The initial analysis of the scope of the project is given in Appendix A. From this analysis the use cases in Figure 1.1 are apparent. At this high level, the functional requirements of the system are considered to be:

- Accept single and multiple amino acid queries from a user.
- Access and parse the contents of a protein database.
- Filter out sections of the database from the search.
- Compare database sections to queries in the externally implemented agrep tool.
- Rank the comparison results from highest to lowest closeness between the query and the database section in the match.
- Display the results that are above a specified cut-off to the user.

Some non-functional requirements can also be identified from this description of the project:

- The time performance for matching a given amino acid sequence in the range accepted by the program will be within an order of magnitude of BLAST.
- The number of returned results be comparable or better than those returned by BLAST for more than half of the tested short amino acid sequences.

These represent a basic, high level view of the capabilities of the system. In the agile methodology they are the high priority requirements for implementation, and the specification was left open for change during the course of the project.

CHAPTER 2

Background

2.1 Biological Sequence Comparison

The aim of this work is to discover a fast and accurate way for finding which record bodies in a protein database have sections similar to that of a given query peptide. In order to do this it is necessary to have a method of finding similarities between the query sequence and database sections. A large part of the problem of doing so is that a match between a query sequence and part of a protein database is likely to be only approximate, due to the fact that errors may have been introduced by biological processes. Figure 2.1 shows examples of the three different types of errors that need to be taken into consideration.

```
Substitution:          Insertion:          Deletion:
ATG -> AAG             ATG -> AITG         ATG -> AG
```

Figure 2.1: Examples of possible errors in protein sequences

2.2 Bit-parallel search potential

An active area of research over the last fifteen years has been the improvement of algorithms by use of the bit-parallelism of nondeterministic finite automata (NFA) states. A computer word is a fixed sized group of bits, acting as the basic unit of manipulation in the processor and memory of a computer. An algorithm can achieve significant speedup by encoding several sets of data in each computer word so that while it is held in a processing unit a bitwise operation on the word is manipulating several sets of data in parallel. A NFA is a model of behaviour that moves from one state to a number of possible others based on input symbols, and can be used, among other things, to model an algorithm working on a string

of input symbols. An algorithm that can be modelled as a NFA has a great potential for optimization [12]. Different states of the algorithm can be mapped to bits of a word, and the transitions run in parallel using the simple bitwise operators of most computer languages.

One particular computer science problem that benefits from bit-parallelism techniques is approximate string matching. Approximate string matching looks for matching alignments between an input string (the query) and a subject string (the database), even if the strings have had some symbols inserted, deleted (indel) or substituted. A notable implementation of a bit-parallel technique for approximate string matching is the algorithm used in the UNIX agrep utility [17]. This algorithm adds together results obtained by parallel operations of shifting and ORing as it works towards obtaining the best alignment between two strings. A potential use of this algorithm is in performing searches for text sequences within databases. The speed and accuracy with which agrep can match sequences should allow searching for short sequences in small to medium sized databases. For very large databases the time performance of agrep may not be sufficient, in which case it might be possible to use some form of database filtering to recover performance.

This research aims to apply the approximate matching capabilities of the agrep utility to the biological sequence comparison problem of approximatly matching peptide fragments against a database of proteins. In order to do this the performance of the search method must be adequate for large databases, and consideration must be given to the underlying probability distribution of the data so that accurate search results can be returned.

2.3 Modelling the data and scoring matches

When searching a database for approximate matches to an input query, more than one match may be obtained. In fact, searching for approximate matches to a small query in a large database may return a large number of matching records. If an approximate matching method is to be relevant to the user it must provide a method of ranking the matches it finds, and returning only those that are of sufficiently high rank. It does this by calculating an alignment score between the best alignment found by the approximate matching tool of the query, and the database segment it is being compared to. In order to calculate this score the underlying probability distribution of the data needs to be taken into account.

The query and subject sequences in protein database searches could be considered to be collections of random variables. These random variables are often considered to be independent, meaning that a symbol being present in one po-

sition would not affect likelihood of occurrence for symbols at other positions. Another possible simplification is to define the random variables as being identical, meaning that the likelihood of a given symbol occurring at one position in the string is the same as the likelihood of it appearing at any other. If these assumptions are adopted the collection of random variables is known as independent identically distributed (i.i.d).

The method of scoring each alignment position can be visualised with a substitution matrix. Such a matrix has rows corresponding to each possible amino acid in the query and columns corresponding to each possible amino acid in the subject. The alphabet is constant and so the matrix is square and the amino acids in the columns the same as those in the rows. Each entry in the matrix is the score for moving from the amino acid at a position in the query, to the amino acid at the same position in the subject. The simplest possible scoring matrix is the unitary scoring matrix, which has ones down its leading diagonal and zeros at all other positions, meaning that it scores all matches the same (with a one) and all mismatches the same (with a zero). More advanced substitution matrices, based on evolutionary distances and statistical studies of the likelihood of substitutions, are generally used, and are usually more appropriate for scoring matches [1]. Substitution matrices can be used both in calculating whether a segment matches, and in ranking matches for display.

The likelihood of occurrence at each position can be modelled as a probability distribution, giving the probability of occurrence of each symbol. The combination of all these distributions over the length of a string gives a final distribution mapping any random query string to an alignment score. If each distribution is i.i.d this is simply the process of calculating the score at each position in an alignment, using the same scoring matrix for each position, and summing the values to obtain an alignment score. The process is complicated by the fact that indel events may result in gaps in the best alignment, possibly requiring that a penalty is imposed on a gap opening up and a further penalty on that gap widening.

This score is usually compared to the expected value for aligning a random sequence of the same length to the same database section to see how likely it is to have occurred by chance. This expected value will come from the probability distribution chosen to model sequences. The comparison of the alignment score to the expected value is represented to the user as a p-value, which, very roughly, is a value that indicates the likelihood that the match was obtained simply by chance. It is then possible to rank matches by p-value, in order to examine only those that are below a certain value.

2.4 Alternative methods

Over the years there have been many notable algorithms developed for approximate string matching, many of which have had immediate applications to biology and protein searches. One of the first algorithms of use in this field was the Needleman-Wunsch algorithm [13], which could be used to find the best possible alignment of a protein sequence, possibly subjected to indel events, in a database. This was later built upon to develop another dynamic programming solution to the problem, the Smith-Waterman algorithm [15]. This could be used for the gapped local alignments of biological sequences subject to indel or mutation events.

The Smith-Waterman algorithm aligns substrings of the query string against substrings of the database and checks for paths, of any length and possibly containing insertions and deletions, that could lead to a match. Unlike the Needleman-Wunsch algorithm, Smith-Waterman ignores paths leading to negative scores and thus concentrates only on local alignment in regions that have generally been preserved from change, making it more useful in computational biology. When set up with a scoring system that contains the probability of each possible insertion, deletion and substitution event and given a negative expectation score for a random sequence, the Smith-Waterman algorithm will find the optimal local alignment for that scoring system. Unfortunately, direct implementations of this algorithm are impractical for use on anything but small databases due to the slowness of searching large numbers of substring and substitution permutations on possible regions. This is slowly changing due to the increasing computational power available to researchers, making exhaustive searches on larger databases possible.

The Basic Local Alignment Search Tool (BLAST) [2] uses heuristics to find good (but not necessarily the optimal) alignments based on an approximation of the Smith-Waterman algorithm, dramatically speeding up the search process. Mainly because of its speed BLAST is currently the method of choice for almost all sequence searching [11].

Similarly to Smith-Waterman, BLAST searches for regions of similar structure. This is done by first finding an exact match for some substring of the query, and then trying to extend this match to find good alignments. It is inexact in that not all possible substrings are searched; however in most cases it returns a useful set of results much faster than Smith-Waterman. However the accuracy of the results returned for short queries is often questionable, due to the heuristic method not having much room to extend matches, the extreme value probability distribution model used by BLAST not being a particularly good model for

short protein queries, and the fact that matching alignments produces too many gaps, requiring the user to constrain the allowed number of gaps much more. This method of scoring gives short sequences high expected values, meaning that valid results could be excluded from the result set. As a partial compensation, the online BLAST search [7] provides a search for short, nearly exact matches, which is a basic search modified to have no low complexity filtering and a higher expected value.

In addition, there may be problems using BLAST to search for more than one query at a time. BLAST treats multiple queries as a batch request, running each in turn, giving results and scores for each one independent of the others. However a researcher will often have a number of peptide fragments that are suspected of being related, in which case there is a potential to improve the quality of results by assuming the queries are related and so prioritizing spatially close matches, or by returning to the researcher how likely the queries are to be related based on their spatial closeness, or some combination of the two. Newer profile search systems such as ELM [4] and PRINTS [3] have multiple search features, however the proteins must be given in the order they are to be found in the database. Current work [9] suggests there may be ways in which bit-parallelism can be used to optimize searching for several patterns simultaneously.

CHAPTER 3

Search Method

3.1 Approximate Matching

The problem addressed in this section is that given a query string by a user, find a method of comparing that string to every section of every database record body, and so find which sections have differences less then some predefined number. In order that Adrasteia be competitive with BLAST it is necessary to use a method with excellent time performance for short sequences.

The UNIX agrep utility [17] has this property. The algorithm bit-parallelizes a NFA, mapping the states of the automaton onto the bits of a computer word in order that the operations to update it can be run in parallel. In essence this is a circumvention of the problem of converting the automata to a (perhaps) larger deterministic model, by instead simulating some extra transitions that happen to be computable using the quick Shifting and ORing of many computer languages. In order to use these operations the automata state must be represented in a manner that allows these operations to perform the necessary transitions.

If the states of the automaton are pictured as lined up into m+1 columns and k+1 rows, where m is the length of the pattern and k is the number of errors, the aim is for the automaton to recognize a set of database input that matches m characters within k errors. Movement from one active state in the automata to another occurs every time a new character is input. A horizontal transition represents a direct match of the character to the character at the same position in the query, a vertical transition an insertion, and a diagonal a substitution or deletion, meaning that if any of the states on the right side of the automata become active then a match has been recognized [12]. An example of this process is given in Figure 3.1, which shows a NFA set up to recognize the pattern TL.

This is implemented in agrep by first pre-processing the query, building a bit mask of length m for each character in the query, with 1s set at every position in the query that has that character. Each row of the automation is fitted into a computer word, with 1s representing active states. If the query length is longer

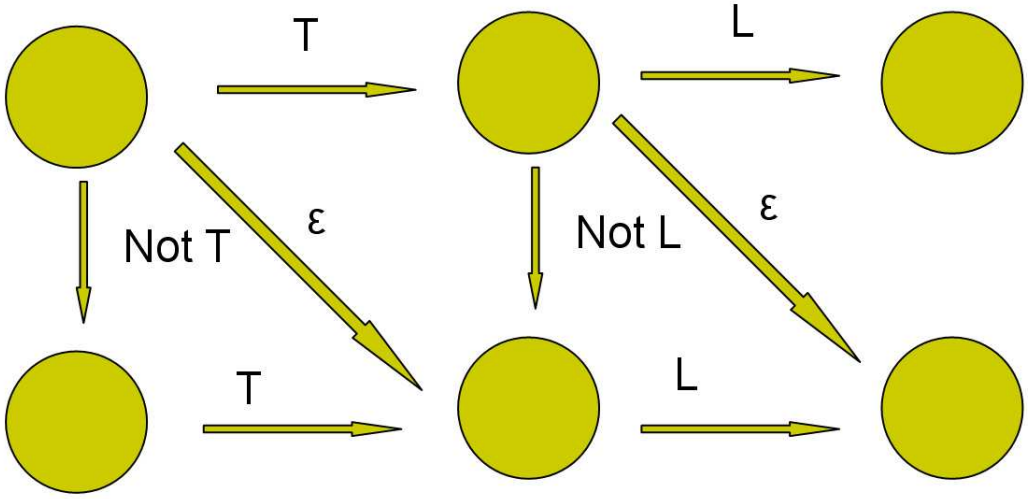

Figure 3.1: A machine for recognising the pattern TL with 1 error

than the word length on the machine more than one computer word can be used to hold the query. Each row is initialised to $R_i = 0^{m-i}1^i$ where i is the row number (starting from 0) and the right side of the equation is shorthand for a bit vector of m-i 0s followed by i 1s. As each new input character T is read from position j of the current text section, the state of the rows is found with the following update formula [12]:

$$R'_0 = ((R_0 << 1)|0^{m-1}1)\&B[T_j]$$

$$R'_{i+1} = ((R_{i+1} << 1)\&B[T_j])|R_i|(R_i << 1)|(R'_i << 1)$$

where B[] is the precomputed bit mask for T. The first row needs only a basic Shift-Or to match against the mask (the OR is to allow for the fact that the nondeterministic nature of the automata allows any match to start at the current position as it moves along the text). In the update of the other rows, the four sections between the ORs correspond respectively to the direct match, insertion, substitution and deletion transitions allowed by the automata.

The time performance of this implementation is O(kn) if m is less than or equal to w [12], in other words when the length of the query is less than or equal to the word length. It is thus fastest for small queries that can be encoded onto a single computer word, and slower for longer queries. A disadvantage of agrep is that for queries longer than the word length the time performance is O(k[m/w]n), and so for very long queries other search methods are generally a better option.

Apart from the advantage of performance, there are a number of other reasons why agrep is especially suitable for use in this project:

- It is possible to modify the algorithm to use different scoring schemes for insertions, deletions and substitutions, by increasing or decreasing the number of row or column positions an automata transition jumps over.
- The bit parallelism in agrep is very similar to the method for filtering data that this program will employ, giving future opportunities for optimization by combining sections of the two.
- If this is done, agrep is one of the few basic pattern matching algorithms flexible enough to be able to incorporate the scoring matrices used in biological sequence comparisons without radically changing the algorithm. Whether this could be done without a significant performance penalty is an open question.
- A few short queries could be put together in a single word for searching.

The Adrasteia program uses agrepy [16], a python port of agrep, for performing approximate matching. In direct matching mode with a single query, it first takes the input query and a maximum error distance and uses them to compile a matcher for the pattern. Then, it sequentially reads in each record of the database, extracts and formats the relevant information from the record, and attempts to match this against the query within the predefined error distance. All successes for each record are noted, and when the database parsing is finished they are ranked and displayed to the user. Agrepy allows the specification of a maximum error distance between 1 and 8.

3.2 Generating Indices

Indexing is the process of pre-scanning a database, storing the important attributes of each record so that later queries to the database can be quickly matched against the attributes rather than the database itself. In an approximate matching context, indexing works by filtering out areas of database text that cannot possibly match a given query, leaving likely areas to be examined more closely by the approximate matching tool. Biological sequence databases are potentially hard to index well because of the difficulties of choosing a set of attributes to index that definitively link a query to all records in the database that could contain it.

It may be possible to get large performance improvements by using indexing schemes that return likely areas of the database without guaranteeing that all possible areas that could contain the query are searched. However, this approach

is not taken by the Adrasteia program, as trading off accuracy for speed does not make sense in a protein search program. In addition, if desired the program can run in direct match mode, which requires no indexing as it approximately matches the query against every record body in the database.

The indexing mode of Adrasteia is based on applying the method of superimposed code words (SCWs), described in [14], to a biological database. The basis of the method is to create a signature for each record by superimposing a set of vectors hashed from a set of record attributes. A set of attributes for each record is chosen (see next section), and each one is converted to a number unique to its value by way of a hashing function. Adrasteia uses a djb2 hashing algorithm, which appears to be fast and have a good distribution of results.

Once the hash has been computed it is used to seed a random number generator, which then generates a predefined and constant number of integers in sequence. The values of these integers are constrained to lie between 0 and a maximum bound, and correspond to a set of positions. If an integer is generated that maps to a position that is already taken it is regenerated until it does not. This process is completely deterministic, so that given the initial hash value, as long as the same random number generator is used, the sequence of integers can be determined exactly. A binary code word (an array of 1s and 0s), of length equal to the maximum bound, is then generated. The values at all positions are set to 0, except for the positions that match the randomly generated integers. This gives a binary code word for the record, and for a large enough maximum bound this word will be unique for every value of an attribute.

The binary code words of each attribute in a record are ORed together to form a SCW for the record. Then, when a query is given, each attribute in it is encoded in the same process used to create a binary code word for the record attributes. These binary code words for the query can then be combined in some manner and compared against the SCWs for each record in the database. This method has the important property that while it does not guarantee that ONLY matching records will be returned, it does guarantee that ALL matching records will be returned. The reason that it does not guarantee that only matching records will be returned is that there is the possibility that 'false drops' may occur, that is due to overlaps in the hashing function or different attributes being allocated the same integer values (especially if the maximum bound is too low) more than one possible attribute value may map to a code word.

In Adrasteia the code words for the database and the query are created, stored and compared using the int data type in the C programming language. This is done by breaking each code word in sections corresponding to the word length of the machine, and encoding that section as a single int. All operations on these

code words are then done by the extremely fast shifting, ANDing and ORing operations, allowing fast comparison of query code words to record code words. As suggested by Roberts, the record code words are held in fast main memory using a bit slice organisation; the first position of each SCW being stored as the first word, the second position as the second word, and so on. This makes matching queries much faster, as the position of each 1 in the query code word corresponds to the bit slice word that has a 1 in every position containing a record matching the query.

3.3 Record Filtering

Due to the use of fast bit level operations and usually being able to hold the code word index in main memory, with a complete and concise set of attributes the method of SCWs has been shown to provide order of magnitude speedups for database searching [14]. The problem now is to determine a possible set of attributes for a biological database. This could possibly be done by breaking each record into fixed length segments, encoding each segment, and then breaking the query into similar sized segments and only looking at records that contain an encoding that matches the encoding of some of the query pieces. The selection of a good set of attributes for the database is in this case is greatly complicated by the fact that there may be errors in the query, thus some of the query piece encodings may be invalid.

A reasonable and simple solution might be to look at all sequences containing a segment matching ANY of the query segments. In this solution the segment needs to be small, as there must be a region equal to the segment size that is preserved without errors between the query and the record in order to flag that section of the database for matching. However, the smaller the segment is made, the less text is filtered out, making the matching process slower. In order to guarantee that all possible matches are returned, the maximum number of errors cannot exceed the length of the query divided by the segment size. In the context of short peptide searches, a small segment length such as 3 letters is the best solution. If there are 20 possible amino acids, this gives 8000 possibilities for each segment. If the average record length is 480 amino acids, and they are broken into non-overlapping segments (in which case the query segments would need to overlap to catch all errors), there are 160 segments per record. If all segments are assumed to be equally likely, then a false drop would be expected only once in 50 records for each segment of the query, so for example a query of length 12 would incur false drops in an average of 1 in 4 records, which is still reasonable.

A more elegant solution for this program came from an idea described by the authors of agrep [10] (but not part of the standard matching in agrep, and not present in agrepy). They describe a partitioning approach to segment a query. If the query is compared to the same sized section of database, with an error rate k, if the query is partitioned into (k+1) blocks then the distance between the query and the database section is greater than k if none of the blocks of the query occur in the database section. To match the query with no more than k errors, a database section has to contain a substring that matches exactly one segment of the query.

Adrasteia's filtering scheme is based on an adaptation of this principle, in which several pre-processing runs are done on the database, each one encoding a different segment size. A query is then divided by the (user specified) maximum number of errors, plus one, to give the minimum number of segments it should be broken into. A static lookup table is used to find the best fitting database segments for each of the query segments (in order to avoid an expensive tree traversal), and the query segment further broken into these lengths, which are encoded and ANDed together for that query segment. Any records that match an OR of all query segments meet the filtering requirements.

The segment sizes 2,3,4,5 and 6 are a reasonable choice here as they make up most small query subdivision lengths in few pieces (ideally each piece would be matched by only one as this would produce the fewest false drops and be quickest to compute, however each extra segment size requires an extra pre-processing run and requires another complete database index that has to be held in memory). This filtering method imposes the restriction that a query's length divided by (the number of errors plus 1) is 2 or greater.

3.4 Multiple Queries

Adrasteia allows the input of multiple query peptides in both direct match and indexing modes. In direct match mode each record is checked against all of the queries in sequence, and any matches recorded. The advantages of multiple query matching over running each query by itself are in slightly faster running times. This is because the database is only iterated through once, so the overhead of fetching and formatting each record is reduced. In addition the scoring scheme can prioritise records which contain more than one query (see next section).

In indexing mode, the queries can be input as a list, and records that contain any of the queries will be selected by the filtering algorithm as in the single query case. The final set of matching records for each single query is ORed together,

and any matches in any record will be returned to the user. In order to speed up the process, Adrasteia allows elements in a list to be grouped together. If a group contains only a single element it will not have any effect, but specifying more than one element in a group has the effect that only records that contain ALL elements in a group will be searched for (elements within a group are ANDed together, and the groups are ORed). Unfortunately, false drops may cause a few extra records, matching only the OR rather than the AND condition, to be returned. There does not seem to be an obvious solution to this.

The main advantage of grouping elements together in this manner is it allows dramatically faster searches. This is the result of specifying much more specific constraints on which records are desired, and so being able to filter out much more of the database. Another advantage is that if the user is only interested in the queries that are in the same record, as is often the case with multiple query searches, this helps to clean up the result set.

3.5 Scoring Results

Adrasteia ranks results based on a statistical model of how likely the result is to be a genuine, rather than an accidental, match. A key point of this model is the use of a simple identity matrix for the scoring scheme (all matching elements are given a score of 1, non-matching elements a score of 0), rather than the use of a more sophisticated biological scoring scheme.

Firstly, the agrep algorithm is unaware of the biological context when determining matches, scoring transitions along the rows of the automata as a zero, and penalising those along the columns as a one. It is possible to alter the algorithm to penalise the first insertion or deletion much higher than subsequent indel events, and change the costs of insertions, deletions and mutations relative to each other. It may even be possible (most likely with a performance penalty) to incorporate scoring matrices by scoring transitions according to the amino acid they start from and the amino acid they end on. These are left as future work, in the current version results are matched without regard the likelihood of the event.

Once the results are matched, a similar scheme is used for assigning each one a score. The best alignment within the edit distance of the query to the match is found, and then each aligned pair is scored with a one, and each mismatched pair or gap with a zero. The use of this scoring scheme may appear more justified when one looks at how BLAST scores short queries. A short query search with the online BLAST tool recommends the use of a PAM30 matrix [7].

The PAM (Point Accepted Mutation) matrix, described in [8], contains the probabilities in log-odds scoring of substitutions between different amino acids, obtained from empirical observation. There are different versions of this matrix for comparisons where different amounts of substitution are expected, for example the PAM1 matrix contains the probabilities of amino acid substitutions given that 1% of the amino acids in the protein had changed. Higher value PAM matrices represent greater expected change between the sequences, and will score short sequences poorly due to the small amount of information present in even the best alignments. For smaller PAM matrices, the difference between the scores awarded for matches and mismatches increases, approaching a log-odds version of the unit matrix at PAM0. Our scoring system is an approximation to these smaller PAM matrices, and the accuracy appears to be reasonable. It is noted for future work that there would be value in building in and testing PAM and other substitution matrices in this program.

Once a score has been obtained, a p-value needs to be calculated. This is done based on a normal probability distribution model for the distribution of random scores. The score values are assumed to be modelled by random variable X, having a normal distribution with mean M and standard deviation S. The values for M and S are obtained from precompiled statistical data obtained from the TREMBL [5] amino acid database. A function for a user to create their own tables of statistical data from a given amino acid database is included in the program. Otherwise it automatically uses the precompiled table of means shipped with program, and a standard deviation of 1. This background data is then used to normalize X to the standard normal distribution, by using the conversion:

$$Z = X - M/S$$

where Z is the random variable modelling the standard normal distribution. Thus, any result score value in X can be converted to an equivalent value in Z. This normalised value can be fed into the Gaussian function, which uses the erfc function included in the standard C libraries to find the probability that X is greater than the score and X is less than the negative of the score. This is the p-value for that score.

The results are output to the user from lowest to highest p-value, that is the most likely results are shown first. Adrasteia has a global constant that specifies the maximum p value to report to the user, allowing the user to clean up their result set by stopping scores with values above this from being reported.

For multiple query proteins, the p-values for the score of each query are multiplied together if they are in the same record, which accurately reflects the decreased probability of X being greater than the score and X being less than

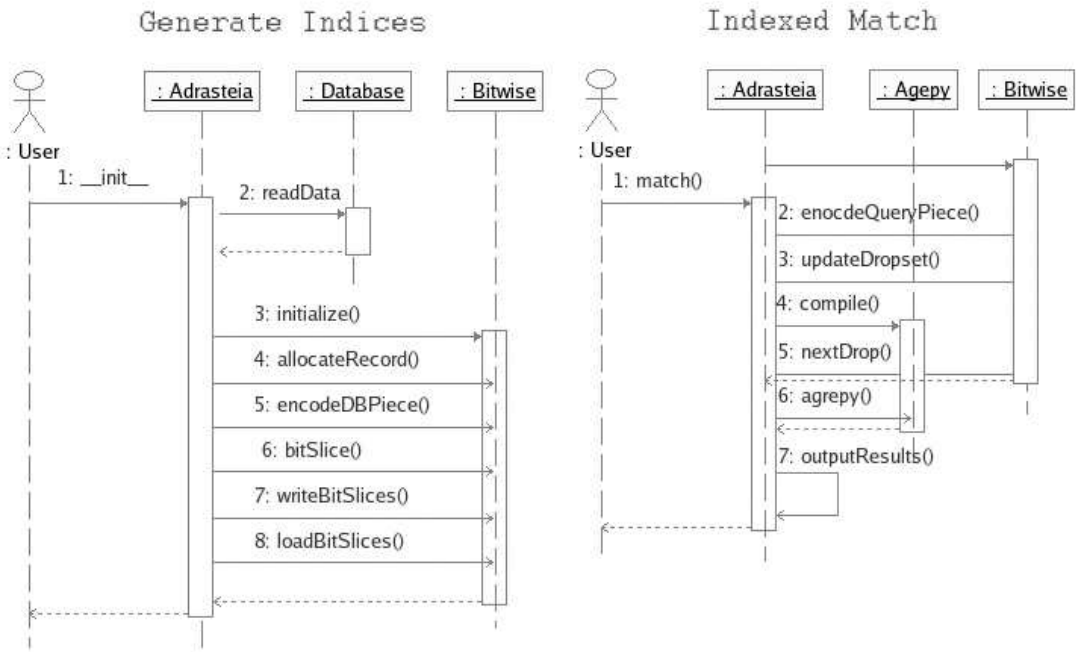

Figure 3.2: Sequence of events for indexing and matching with indices

the negative of the score. This boosts records containing the most queries to the top of the results, allowing the user to input multiple queries and quickly see if they are related in any records.

3.6 Program Structure

The program was structured as a layer of python code for interacting with the user and parsing the database, interacting with the agrepy utility, the database and a core of code called Bitwise which is in the C programming language. The python code acts as a single object, with methods available to the user, and the C code as another. While the structure does not follow a strictly object oriented design pattern, the two sections can be treated as objects with a set of well defined and modular methods.

The structure of the Adrasteia program is best shown in the sequence of actions needed to complete two of the required use cases for the system. The sequence diagram for indexing and matching a database of one record is shown in Figure 3.2. This implements the method developed in the previous sections.

3.7 Program Correctness

In order to construct a set of experiments to test the research aims, it is necessary to have a measure of whether the program is performing as specified. Following the agile methodology, a lot of the testing was done during development. Needed functionality was identified, and developed as a minimal group of functions working together. A black box abstraction was used to run informal unit tests for common inputs to each functional group.

The system testing then takes a more formal approach. Two test suites were developed, one that partitions input data into equivalence classes and tests program response to all classes, and one that tests performance for various inputs and databases. They are both included as part of the Adrasteia package, in order that the end user can check the program is functioning correctly. The successful completion of these tests indicates the program is performing as expected, and that the defined functions are being carried out correctly during experiments with the program.

CHAPTER 4

Time Performance

4.1 Speed Testing

4.1.1 Methodology

The time taken by Adrasteia to find queries in a biological database of proteins in FASTA format needs to be examined. The database used in this section is SWISS-PROT [5] version 50.7, it has 232322 sequences (record bodies) and an uncompressed size of 89 MB. This section contains an examination of how the speed of the program is affected by:

- Maximum allowed errors.
- Lengths of queries.
- Number and grouping of queries.
- The effects of the parameters of the indexing scheme.

These factors are all interdependent, for example a large number of errors may affect large queries more than small ones. An examination of interesting relationships between these factors is deferred until the analysis section.

In order to verify performance results and partially mitigate the effects of computer architectures on the results, testing was done on three different machines. All machines were running the Linux operating system (2.6 kernel).

MACHINE 1 is an Intel(R) Pentium(R)III Laptop, 32 bit 1.133GHz (M) processor with 32Kb L1 cache and 512 Kb L2 cache, 256Mb of RAM.

MACHINE 2 is a Intel(R) Pentium(R)IV, 32 bit 2.4GHz (2.4C - Hyper threading off) processor with 8Kb L1 cache and 512 Kb L2 cache, 512Mb of RAM.

MACHINE 3 is a AMD(R) Opteron(R), 64 bit 1.8GHz Dual-Core (165) processor with 2 x 64Kb L1 cache and 2 x 1Mb L2 cache, 1024Mb of RAM, (using 64 bit Linux kernel).

All machines had swap space disabled. Timing was done using an OS call and where possible a number of runs were done and the average result given. All code was compiled by python 2.4.3 and gcc with default parameters, apart from the -fPIC flag.

Another important factor is the selection of appropriate pseudorandom sequences for the queries. These tests take the length of the random sequence from a discretized log-normal distribution within given size bounds, and then generate the sequences from i.i.d processes using a mixture of Dirichlet densities.

Unless specified otherwise, all trials are averaged over 5 runs, with a maximum edit distance of 1, using a single input query of length 24 or 25, and a 2-3-4-5-6 filtering scheme with 1024 word length. To avoid printing out large numbers of results in many cases the p-value cut-off was set to 0, so that although results are found in the normal manner none of them qualify to be returned to the user.

4.1.2 Generating and using indices with different word lengths

MACHINE 1			MACHINE 2		MACHINE 3	
Word Length	**Time (mins)**	**Size (MB)**	**Time (mins)**	**Size (MB)**	**Time (mins)**	**Size (MB)**
32	92.24	6.3	45.69	6.3	20.41	5.5
64	92.39	9.9	44.82	9.9	20.74	9.2
128	92.69	17.3	43.73	17.3	20.59	16.5
256	93.28	32	44.25	32	20.99	31.3
512	94.14	61.4	45.01	61.4	21.50	60.7
768	95.22	90.8	46.17	90.8	22.05	90.1
1024	95.84	120.3	46.98	120.3	22.40	119.5
1280			46.14	149.7	22.76	148.9
1536			46.67	179.1	23.17	178.4
2048			47.31	237.9	24.24	237.2
4096					28.04	472.2

Table 4.1: Time and space required to generate indices

Word length refers to the length of the code words used by the indexing scheme of the program (these are broken into segments of length equal to the word length of the machine itself). Each code word encodes one attribute of a record or query, and so the longer the code word is the more likely that a given attribute will have a unique code, meaning that less falsely matching records have to be examined. The flipside is that longer code words take longer to generate in

both the pre-processing and matching stages, and take up more disk and memory space. The times to pre-process the database using various word lengths, as well as the space requirements of the corresponding indices, are given in Table 4.1.

Machine 1 was not able to generate the three largest indices in the test set due to them being too large to load into RAM, and Machine 2 was not able to generate the largest. As can be seen, the amount of space taken by the index for each word size is significant, as the entire index should be held in main memory. The time taken to generate the longer indices is not that much more than that taken to generate the shorter ones, but it is apparent that generation is time consuming, especially on the slower machines.

The time performance of each of these indices for matching, given a maximum error rate of 1, 2 or 4 (Figure 4.1), can then be examined. Disregarding the time for generating the indices, the benefits of reduced matching with the larger indices can be seen to be outweighing the extra processing time for creating the query code word and matching it against the indices.

4.1.3 Multiple ungrouped queries with and without indexing

The time taken by the direct match algorithm (linear search of a database) is compared to a similar measure of performance for the indexing algorithm (running only on areas of the database containing matches) for ungrouped multiple queries (Figure 4.2). The comparison shows the decrease in performance associated with adding increasing numbers of ungrouped peptide fragments to a query.

4.1.4 Multiple grouped queries

The effects of adding more peptide fragments to a query in an indexed search when the fragments are grouped is shown in Figure 4.3. The times taken for the grouped search decrease as more queries are added, due to each query adding more information to the filtering algorithm and so reducing the number of records marked by the algorithm for searching.

4.1.5 Effects of query length with and without indexing

The times taken to match a single query of various lengths for direct match and indexed match are shown in Figure 4.4. Both direct match and indexed match are faster on longer queries as they are favoured by the matching algorithm. For a given error indexed matching has the additional benifit that for longer queries

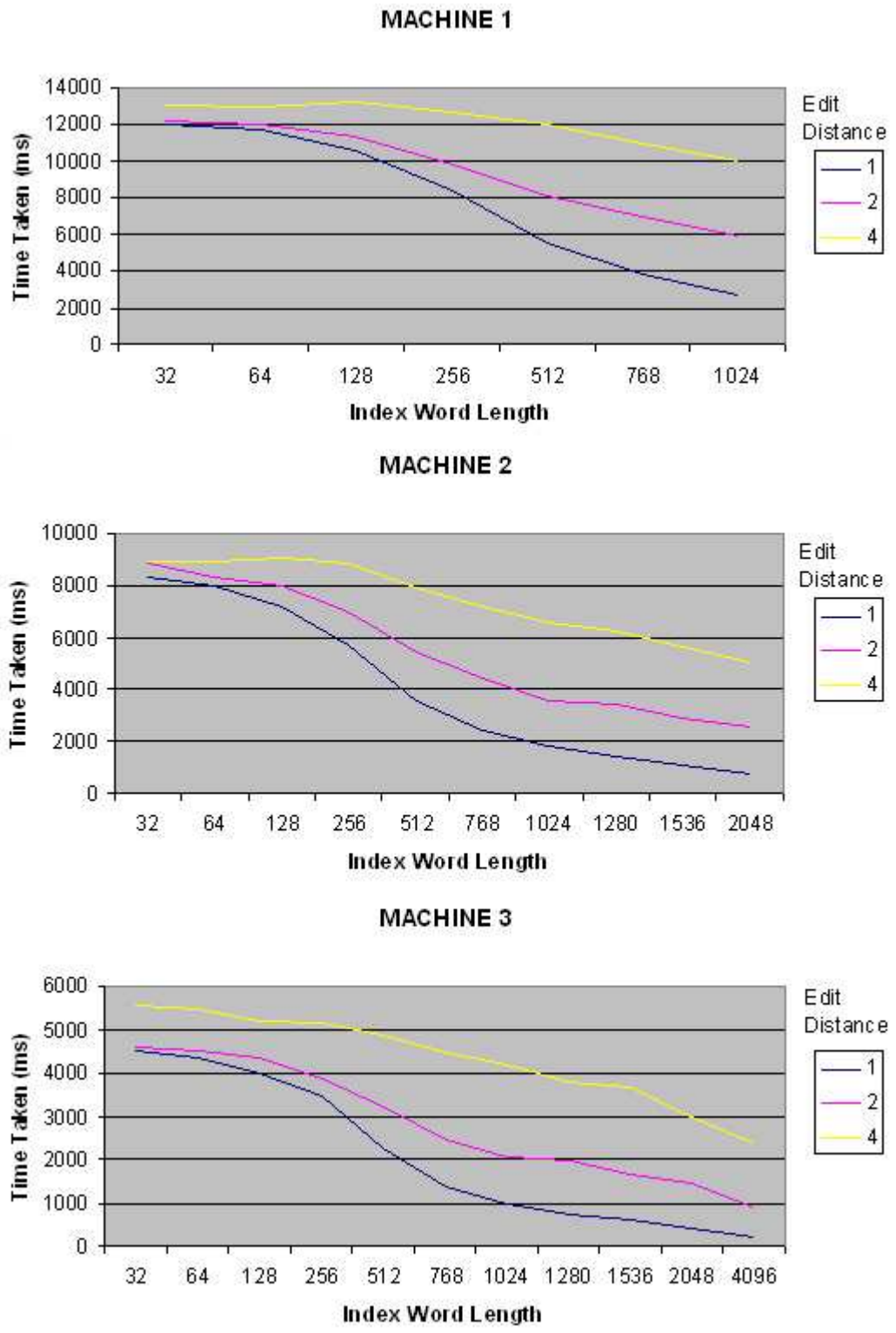

Figure 4.1: Effects of index word length on match time for different max errors

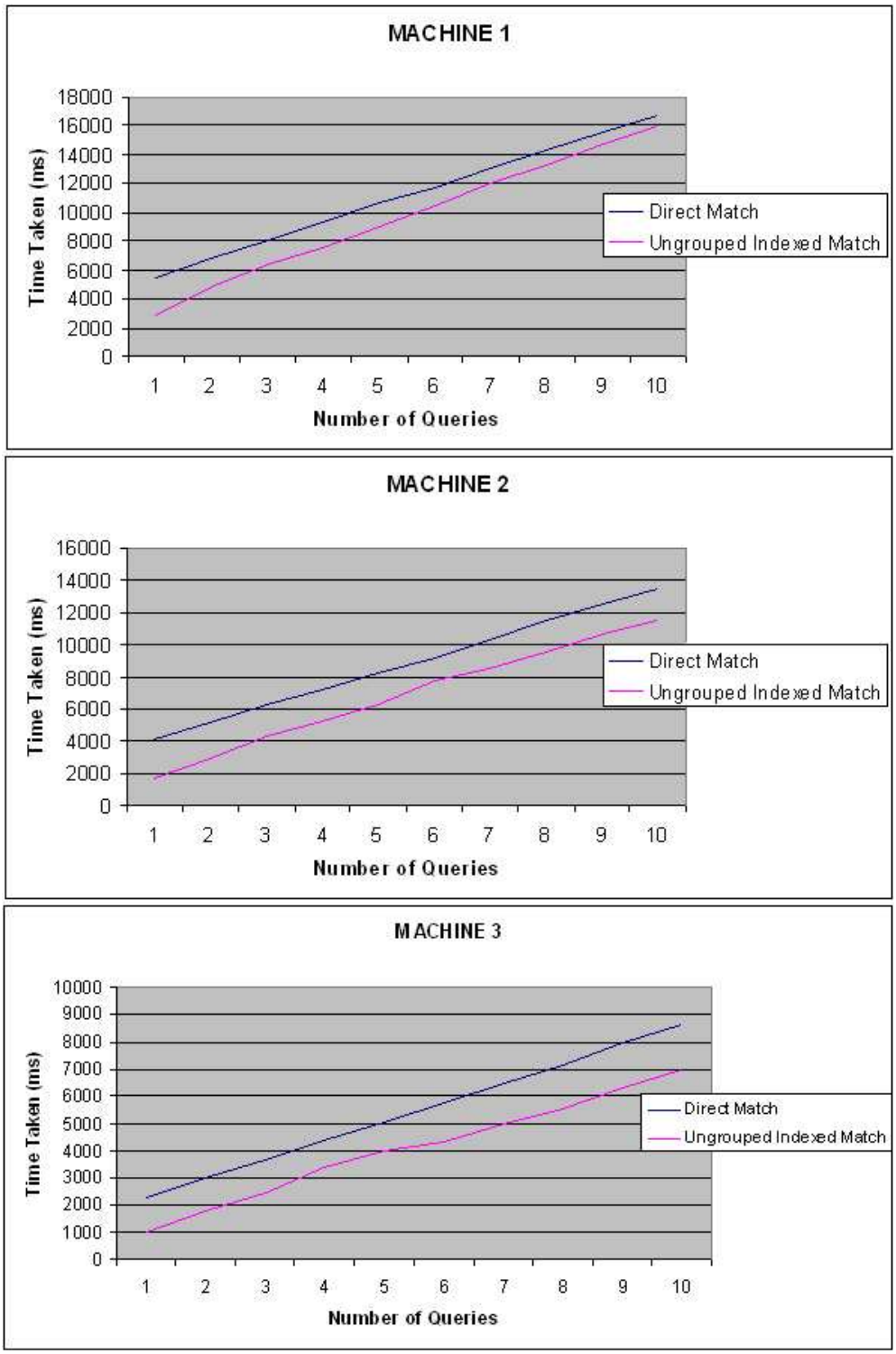

Figure 4.2: Match times for multiple ungrouped queries

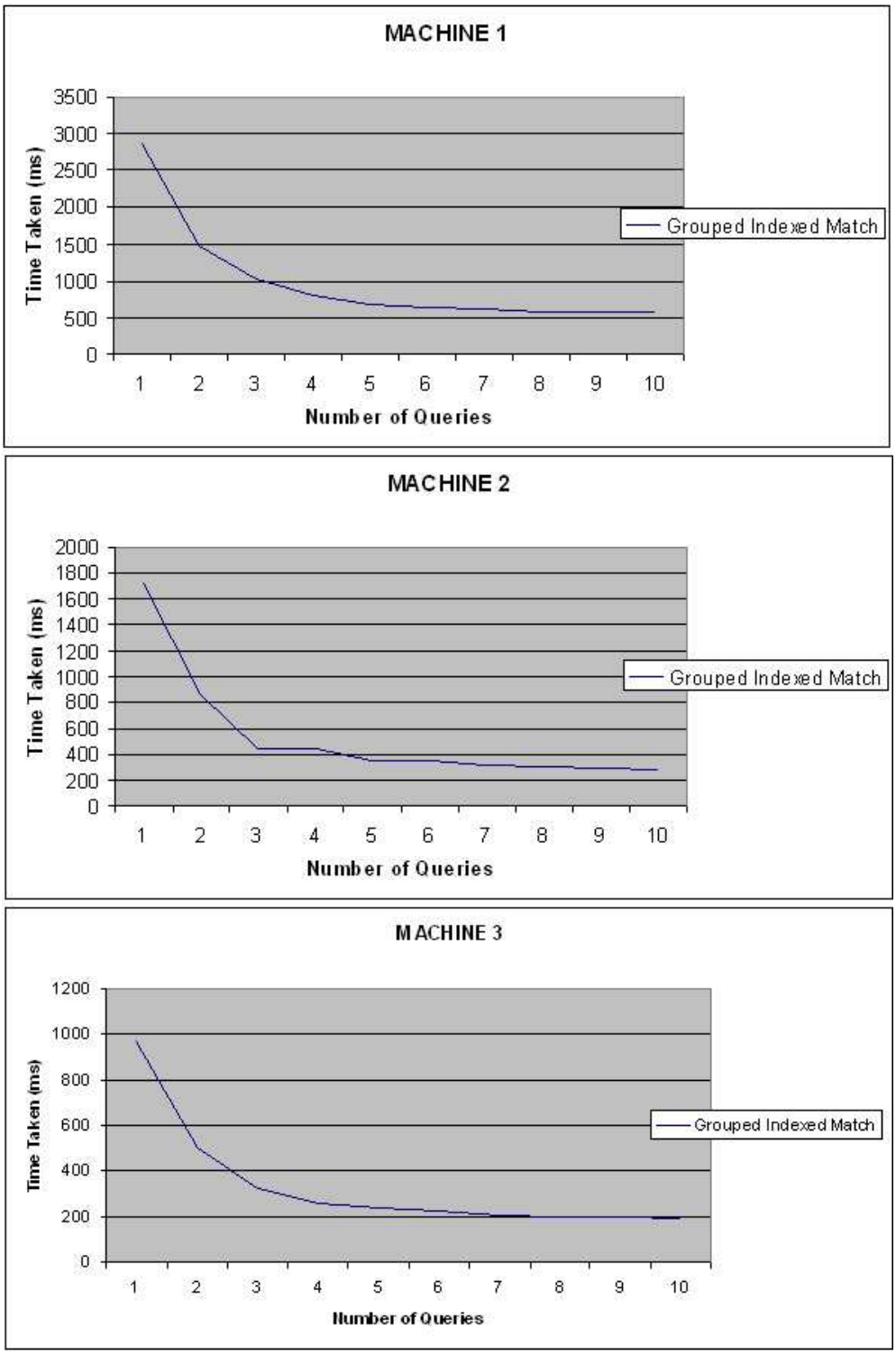

Figure 4.3: Match times for multiple grouped queries

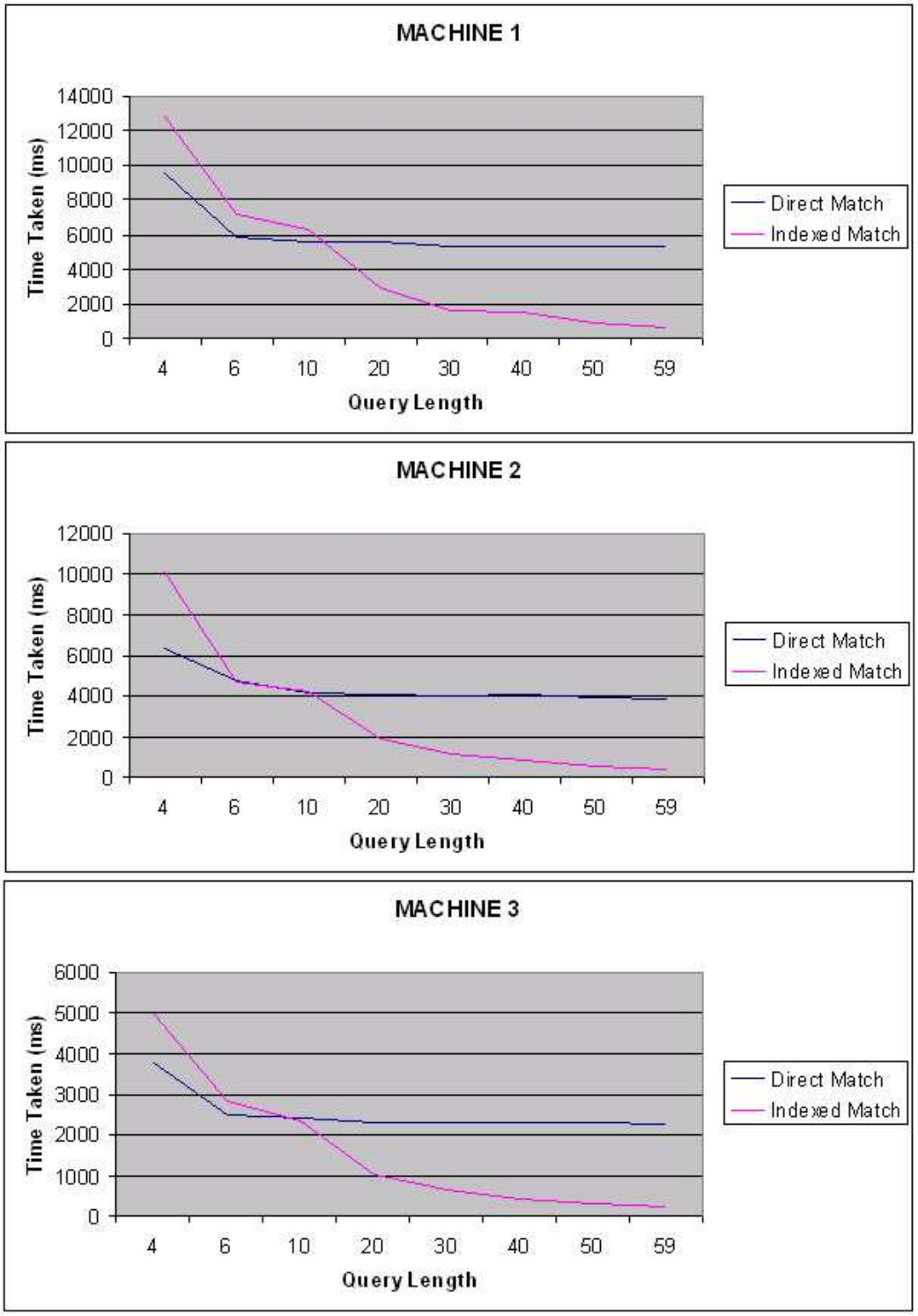

Figure 4.4: Performance with various sized queries

more records can be discarded, which explains why the indexed match is seen to be outperforming the direct match for queries above roughly ten aa in length.

4.1.6 Effects of maximum error rate with and without indexing

The times taken for direct match and indexing match with various maximum edit distances for a single query is compared in Figure 4.5. Both direct and indexed match are slowed greatly by large error rates, both suffering because of the matching having to take into account more possibilities. Indexed matching has the additional penalty of having to consider more records, as there is less certainty about the attributes for matching each record in the filtering stage. As a result the break even point for using indexed matching is three errors, beyond this direct matching is faster.

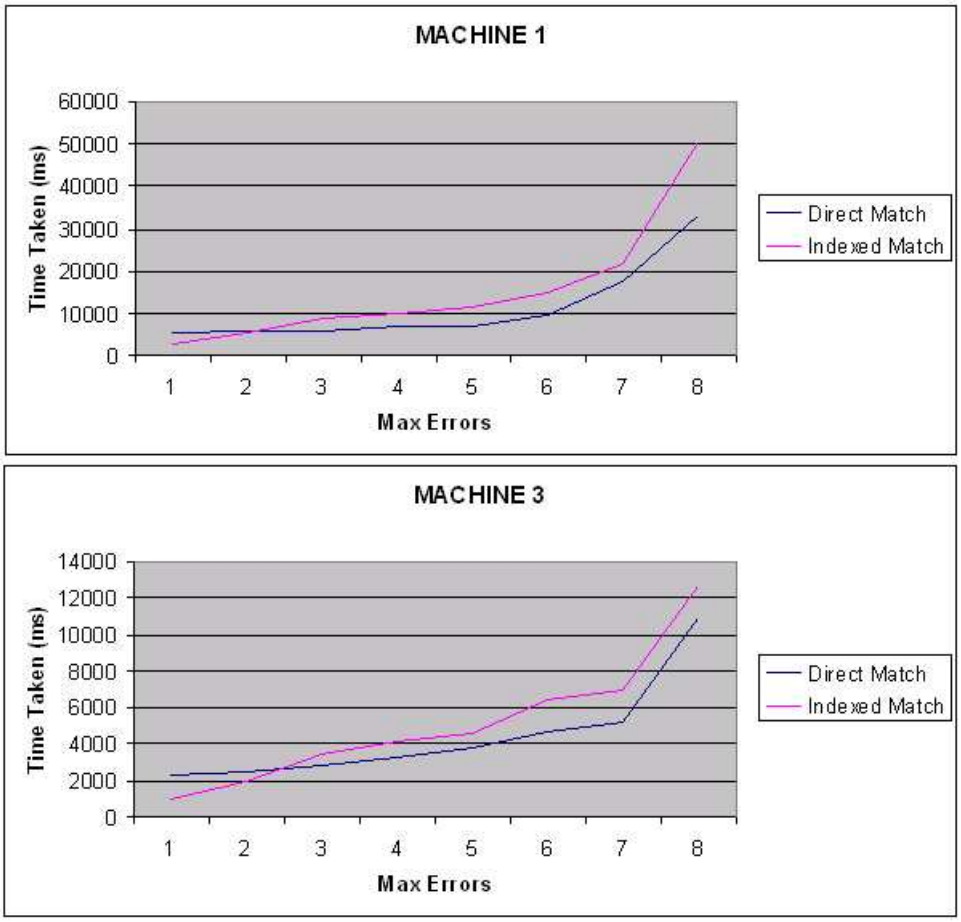

Figure 4.5: Performance with various maximum error rates

4.1.7 Filtering schemes with different segment sizes

The above results use the 2-3-4-5-6 partitioning scheme for the filtering, that is the database is pre-processed by breaking each record into segments of 2 then 3 then 4 then 5 then 6, and storing the results in index files. In the matching stage the query is partitioned into a number of blocks (depending on the error rate), and each block is composed, as best as possible, of segments of size 2 or 3 or 4 or 5 or 6.

The aim of our partitioning scheme is to use the fewest pieces that best match the largest number of blocks. The fewer pieces used, the quicker the pre-processing and the less memory wasted in holding the records for matching. However more pieces allow segments to better fit in blocks, and the larger the segment the more records can be thrown away with it. For example, a block of size 5 would be broken into a 3 and a 2 in a 2-3 scheme, meaning that any records that had in them either the 3 letters or the 2 letters would be returned. In a 2-3-4-5-6 scheme the 5 letter block could be fitted with a single segment of size 5, meaning that only records that had the entire 5 letters would be returned.

MACHINE 1			MACHINE 2			MACHINE 3	
	Time (ms)			Time (ms)			Time (ms)
2,3	2365253		2,3	1093370		2,3	550020
2,3,4,5,6	5750212		2,3,4,5,6	2818773		2,3,4,5,6	1343950
			2,3,4,5,6,7,8,9,10,11	5566892		2,3,4,5,6,7,8,9,10,11	2601950

Table 4.2: Time and space required to generate partitions

A 2-3-4-5-6 scheme will be compared to a more minimal 2-3 scheme, and with a more extensive 2-3-4-5-6-7-8-9-10-11 scheme. Table 4.2 shows the pre-processing times for creating a 1024 word length index for each of these schemes, and Figure 4.6 compares the times taken to match a query of various lengths with each of the schemes. Machine 1 was not able to compute and use indices from the 2-3-4-5-6-7-8-9-10-11 as there was not enough space in memory. It is clear that the 2-3 scheme is not only faster than 2-3-4-5-6 at creating the indices, but also provides faster matching in the tested cases, while the 2-3-4-5-6-7-8-9-10-11 scheme is slower in both.

To see if 2-3 is a faster scheme than 2-3-4-5-6 it is also worth examining how it performs at different error rates. The results of a trial are given in Figure 4.7. From this it would seem that 2-3 does marginally outperform 2-3-4-5-6 except for very high error rates. As high error rates are not likely to be specified often in the use of this program (small queries will generate extremely high numbers of matches for larger error rates), the 2-3 scheme is the more appropriate for use

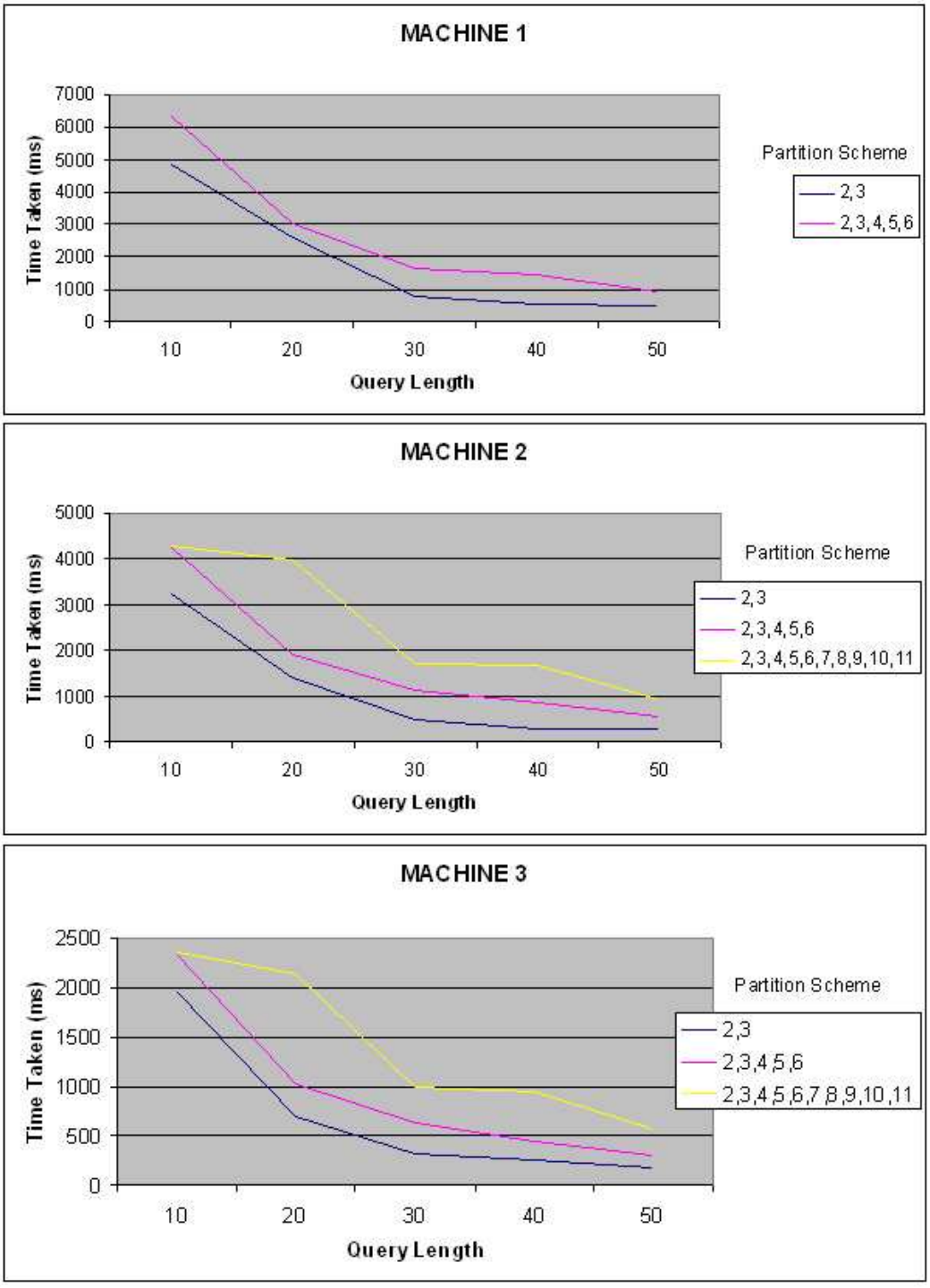

Figure 4.6: Match performance with different partition schemes

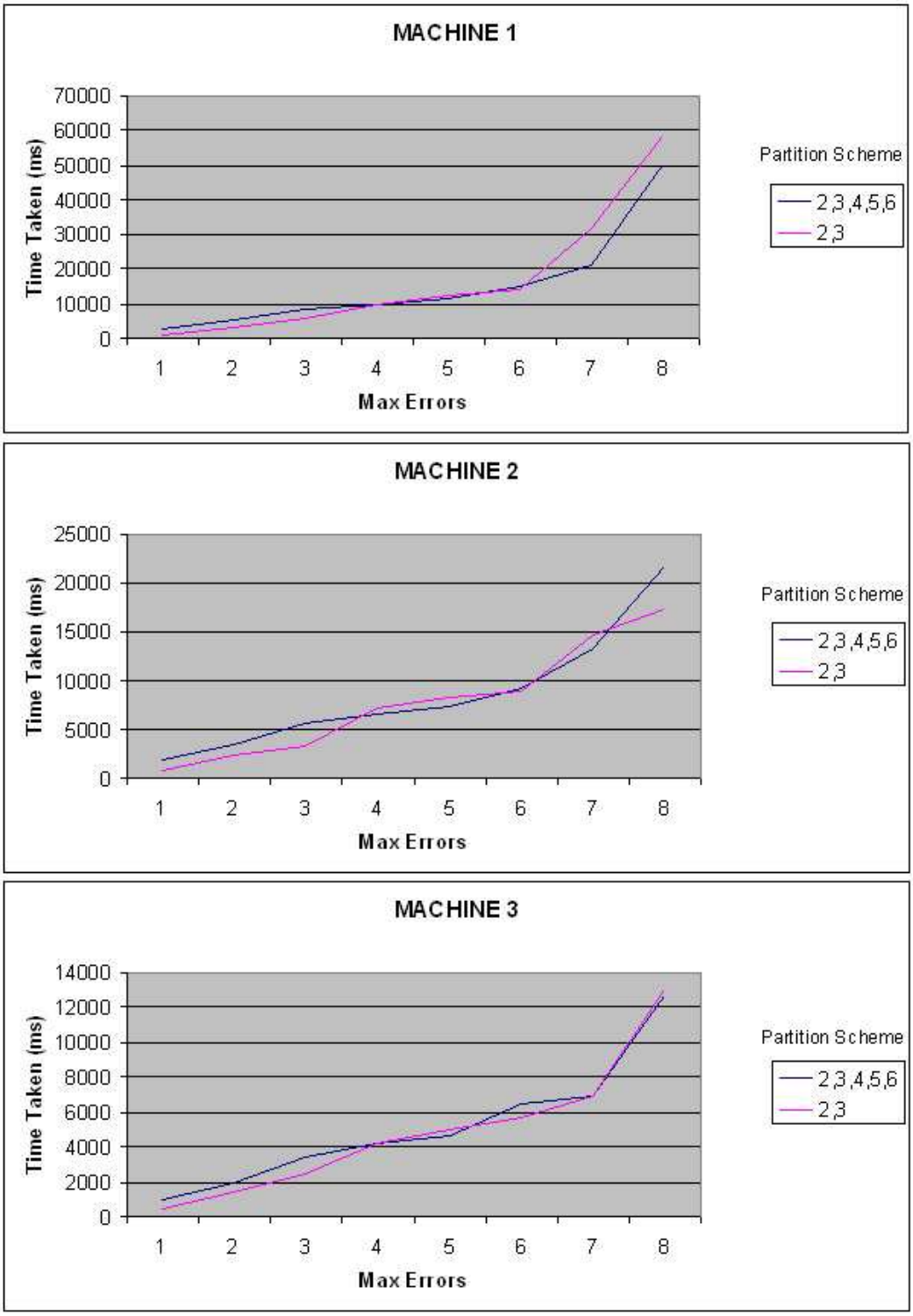

Figure 4.7: Effects of errors on two partition schemes

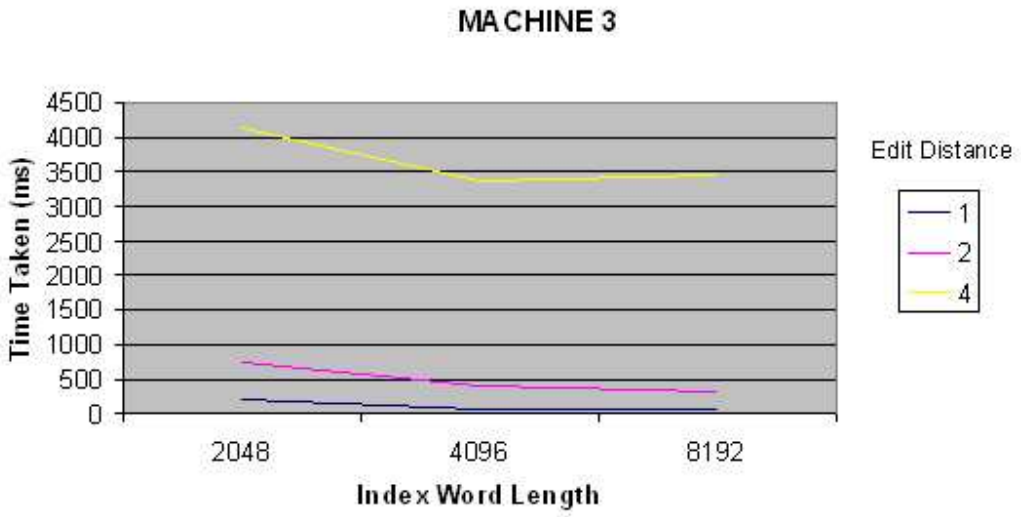

Figure 4.8: Performance of 2-3 partitioning scheme for large word lengths

in this program.

Finally, the small size of the indices generated by the 2-3 scheme allows larger word lengths to be used for indexing. As was seen in Figure 4.1, the larger the word length the faster the match, however due to the limited space in memory it was not possible to take this beyond 1024 word length for machine 1, 2048 word length for machine 2 and 4096 word length for machine 3. At 4096 word length matching was still faster than at 2048 word length, so it is worth examining how fast machine 3 can perform matching given larger indices. The results of using the 2-3 scheme with word lengths of 2048, 4096 and 8192 with error rates 1,2 and 4 are shown in Figure 4.8.

While the 2-3 scheme provides faster matching for the 1 and 2 error rates, by 4 it is performing at about the same level as the 2-3-4-5-6 scheme. The 2-3 scheme with a word length of 8192 provides the fastest matching of a query with the low error rates that are expected in this type of program, however as the 4096 word length indices use roughly 200Mb rather than the 400Mb of the 8192 word length indices, it is probably more practical to set the default word length of the indices to 2048 or 4096.

4.2 Benchmarking

4.2.1 Methodology

The performance of Adrasteia on a large 1.1GB protein database (TREMBL [5], 3182016 sequences) is tested and compared to a local version of the BLAST program for the same database. Testing was done entirely on Machine 3, due to the constraints of available memory, and swap space was turned on if needed.

A 2-3 index of wordlength 256 was used for the Adrasteia program. While the previous section reveals that a 256 word length is not particularly fast, it should still be slightly better than direct matching. The reason a larger index could not be used is that swapping should be avoided at all costs, and the program itself may use a fairly large amount of memory in addition to the space taken by the indices.

4.2.2 Indexing

The first step is to generate the indices for both programs. For the 1.1GB database, the processing time of BLAST was 20 mins, and Adrasteia took 2 hours. The size of the BLAST indices was 1.4GB, while Adrasteia's indices were 224MB.

4.2.3 Single Query Speed

The time taken to match a single query of various lengths is given in Figure 4.9 for the two programs. BLAST is faster for very small queries as it has to do much less work matching them, whereas the performance of Adrasteia improves for longer queries due to the indexing scheme being able to discard many more records from consideration.

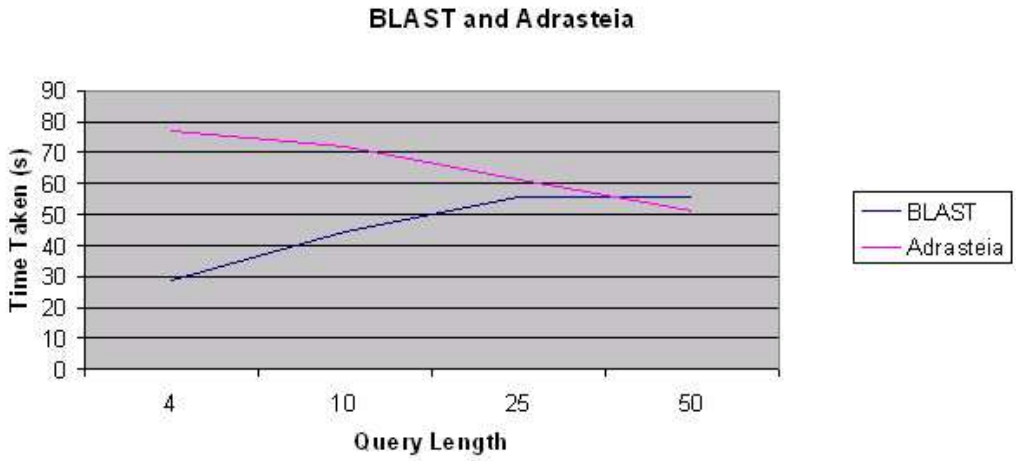

Figure 4.9: Comparison of performance of BLAST and Adrasteia for different query lengths

CHAPTER 5

Result Quality

5.1 Methodology

In this section, the accuracy of results from Adrasteia are compared with those of BLAST for different types of queries, using the TREMBL database from the previous section. Simply measuring the number of results is not a sufficient measure of accuracy as with a good scoring scheme less relevant results can be removed from the result set. As such, we qualitatively examine accuracy as based on the ability of a search tool to return an appropriate set of good matches.

There are a number of things to consider in examining these results. Firstly both programs have a method of cutting off results below a given threshold. BLAST allows the user to set an E-value cut-off, specifying the maximum expected value for returned results, and Adrasteia has a cut-off for the maximum p-value of returned results. The default value of 10 for the E-value in BLAST, and 0.05 for the p-value in Adrasteia, are used throughout this section.

In the testing, where possible BLAST is used with the values recommended by the online 'search for short nearly exact sequences' page [7]. The most important result of this is that we use PAM30 rather than the default BLOSUM62 for the scoring matrix, although we also state the values from BLOSUM62 in some cases for comparison.

A problem with drawing comparisons between the two programs is that the number of errors is restricted to a single value in Adrasteia, whereas in comparison BLAST looks for likely areas and then allows gap opening to various levels. In order to make the result sets more equivalent, tests in Adrasteia were done using an edit distance as high as possible, resulting in larger numbers of distant matches in the result set.

Finally, multiple queries are supported in protein search BLAST only in a basic batch fashion. Blastn and Megablast do support group searching and weighting of results by groups, but only work for nucleotide databases, and also cannot

combine this with ungapped searches, and so do not work well for small queries.

5.2 Result set for a length 40 query

The results obtained from a query of length 40, generated by deleting one letter from a section of record Q50214, are compared. The results for Adrasteia are given in Table 5.1, and for BLAST in Table 5.2. There were 798 results for BLAST, and 522 for Adrasteia, so the results in the tables have been shortened. It is worth noting that the original record (Q50214) showed up much higher in the BLAST result set than in the Adrasteia result set. The higher confidence given by Adrasteia to its top matches is a consequence of the likelihoods being modelled by a normal distribution of short matches, resulting in long consecutive regions being considered very likely. As these confidence values are internally generated the results need to be examined cautiously, the fact that Q50214 is not given a high confidence value by Adrasteia is probably more indicative of result accuracy than the listed p-values. BLAST has a greater number of significant matches, as the query is still long enough to suit the distribution model it uses, and the ranking will be reasonable as long as a scoring matrix appropriate for short sequences is used.

5.3 Result set for a length 15 query

The input for this section is a query of length 15, taken from the record O93734 and with the last characters AALQAT deleted and substituted to give TLQQT with 3 edit operations. The 46 results for Adrasteia are given in truncated form in Table 5.3, and the 5 results for BLAST in Table 5.4. Both tools returned 093734 as a top match, but for short queries it would be expected that there would be a much greater range of candidate records than has been reported by BLAST here. It is worth considering if we can get better results from BLAST by using different search parameters. The 16 results obtained by BLAST when the filtering is turned off and the E-value increased are given in truncated form in Table 5.5. For comparison, the three results of this same search using the default BLOSUM62 scoring matrix are given in Table 5.6. These results are very unlikely, partly because the BLOSUM62 matrix is intended for longer, more divergent sequences.

In order of significance, these are the 522 matches
found for the pattern ['TALAGNGNAGLSAWYLSMYLHKEAHGRLGFFGFDLQDQCG']

Record	In Text	Raw Score	P-Value
Q6SCJ0	AIATGNGNAGLSAWYLSMYL HKEAWGRLGFFGYDLQDQC	31	7.340792e-182
Q50908	AIATGHGNAGLSAWYLSMYL HKEAHGRLGFFGYDLQDQC	31	7.340792e-182
Q75NC7	SIATGNGNAGLSAWYLSMYL HKEAHGRLGFFGYDLQDPV	31	7.340792e-182
Q50257	AIATGNGNAGLSGWYLSMYL HKEAHGRLGFFGYDLQDQC	31	7.340792e-182
Q0Z8P7	TAAAATGNANAGLSAWYLSM YLHKEAWGRLGFFGYDLQD	27	2.587001e-135
.....			
Q98LV6	ITALDQV	03	4.763615e-02
Q6CGA2	ALAGAGGGALSMVVTYPLIT LSTRAQTESM	04	4.830220e-02
Q6BN91	SGNTGNGNYSAEFDDIFETF PIDELFKNLN	04	4.830220e-02
Q9CAJ9	AGNGYSSPRDSSFPSSFTKF NSALTAGLLN	04	4.830220e-02
Q9LCB7	ANGNAEAFVPALERALPCMR HRGPDDAGTW	04	4.830220e-02
Q8NNK1	ANGNAEAFVPALERALPCMR HRGPDDAGTW	04	4.830220e-02

Table 5.1: Adrasteia - Truncated result set for a 40aa query with edit distance 3

```
                                                          Score    E
Sequences producing significant alignments:              (bits) Value

Q49114_9EURY (Q49114) Methyl-coenzyme M reductase...      100   5e-21
Q50214_METMT (Q50214) Methyl-coenzyme M reductase...      100   5e-21
Q2ABQ9_9ARCH (Q2ABQ9) Methyl-coenzyme M reductase...       98   4e-20
Q2ABQ6_9ARCH (Q2ABQ6) Methyl-coenzyme M reductase...       98   4e-20
Q2ABQ8_9ARCH (Q2ABQ8) Methyl-coenzyme M reductase...       98   4e-20
Q2ABR0_9ARCH (Q2ABR0) Methyl-coenzyme M reductase...       98   4e-20
Q15BI6_9ARCH (Q15BI6) Methyl coenzyme M reductase...       98   4e-20
Q15BI7_9ARCH (Q15BI7) Methyl coenzyme M reductase...       98   4e-20
Q5EGK1_9EURY (Q5EGK1) McrA (Fragment)                      98   4e-20
Q6VVF6_9ARCH (Q6VVF6) Methyl-coenzyme M reductase...       98   4e-20
Q9C4K1_9EURY (Q9C4K1) Methyl-coenzyme M reductase...       98   4e-20
Q2Z0F2_9ARCH (Q2Z0F2) Methyl-coenzyme M reductase...       98   4e-20
Q75ND0_9ARCH (Q75ND0) Methyl-coenzyme M reductase...       98   4e-20
Q5EGJ7_9EURY (Q5EGJ7) McrA (Fragment)                      98   4e-20
Q50908_9EURY (Q50908) Methyl-coenzyme M reductase...       98   4e-20
Q49KB3_9ARCH (Q49KB3) Methyl coenzyme M reductase...       98   4e-20
.............................................................
Q70CP0_9EURY (Q70CP0) Methyl-coenzyme M reductase...       75   3e-13
Q70CN8_9EURY (Q70CN8) Methyl-coenzyme M reductase...       75   3e-13
Q8NKL0_9EURY (Q8NKL0) Methyl-coenzyme M reductase...       75   3e-13
.............................................................
Matrix: PAM30
Gap Penalties: Existence: 9, Extension: 1
Number of Sequences: 3182016
Number of Hits to DB: 15,428,189
Number of extensions: 70979
Number of successful extensions: 3665
Number of sequences better than 10.0: 798
Number of HSP's gapped: 3653
Number of HSP's successfully gapped: 798
Length of query: 40
Length of database: 1,030,253,293
Length adjustment: 28
Effective length of query: 12
.............................................................
```

Table 5.2: BLAST - Truncated result set for a 40aa query

In order of significance, these are the 46 matches
found for the pattern ['QSAQAWAWMGTLQQT']

Record	In Text	Raw Score	P-Value
O93734	QSAQAWAWMGAALQ	10	9.405188e-18
O93735	QSAQAWAWMGAALE	10	9.405188e-18
Q0Y8W3	QSAQAWAWMGAALQ	10	9.405188e-18
Q4K7F0	QSAQAEAWT	06	1.254688e-06
Q4WKZ8	IRATHGALMGTLQQ	06	4.627578e-06
Q9LLB8	QASAQAWAAMVTEMQ	06	6.042407e-06
Q8WTI7	QSAQAGTGYQ	05	1.588562e-04
...	...	...	...
Q75GD1	QAAATDWMGRLQV	04	8.738978e-03
Q34V82	QSASAMAWRTLFE	04	8.738978e-03
Q1YPB5	VSKQAWAEWMTHQ	04	8.738978e-03
Q87X85	SSAQADAMATIQQ	04	8.738978e-03
Q6RKF4	QGAQWARMGTELIQ	04	9.851458e-03
Q4PBG2	QSQAHLAAASTLQQ	04	9.851458e-03
Q92X62	QSGFAWAWLGAALQA	04	1.157312e-02

Table 5.3: Adrasteia's result set for a length 15 query with edit distance 6

```
                                                                 Score    E
Sequences producing significant alignments:                      (bits) Value

Q0Y8W3_9EURY (Q0Y8W3) Luciferase-like                              41   0.004
O93734_9EURY (O93734) F420-dependent alcohol dehydro...            41   0.004
O93735_9EURY (O93735) F420-dependent alcohol dehydro...            38   0.028
Q3ZB04_MOUSE (Q3ZB04) Acr protein                                  32   2.3
Q3ZB06_MOUSE (Q3ZB06) Acr protein (Fragment)                       32   2.3

>Q0Y8W3_9EURY (Q0Y8W3) Luciferase-like
          Length = 330

 Score = 41.4 bits (90), Expect = 0.004
 Identities = 13/16 (81%), Positives = 13/16 (81%), Gaps = 1/16 (6%)

Query: 1  QSAQAWAWMG-TLQQT 15
          QSAQAWAWMG  LQ T
Sbjct: 49 QSAQAWAWMGAALQAT 64
.....................................................................
Matrix: PAM30
Gap Penalties: Existence: 9, Extension: 1
Number of Sequences: 3182016
Number of Hits to DB: 6,916,253
Number of extensions: 13244
Number of successful extensions: 2198
Number of sequences better than 10.0: 5
Number of HSP's gapped: 2198
Number of HSP's successfully gapped: 5
Length of query: 15
Length of database: 1,030,253,293
Length adjustment: 5
Effective length of query: 10
Effective length of database: 1,014,343,213
Effective search space: 10143432130
Effective search space used: 10143432130
Neighboring words threshold: 16
Window for multiple hits: 15
.....................................................................
```

Table 5.4: BLAST - Truncated result set for a length 15 query with default short search values

```
                                                                Score    E
Sequences producing significant alignments:                     (bits) Value

Q0Y8W3_9EURY (Q0Y8W3) Luciferase-like                             41   0.004
O93734_9EURY (O93734) F420-dependent alcohol dehydro...           41   0.004
O93735_9EURY (O93735) F420-dependent alcohol dehydro...           38   0.028
Q3ZB04_MOUSE (Q3ZB04) Acr protein                                 32   2.3
Q3ZB06_MOUSE (Q3ZB06) Acr protein (Fragment)                      32   2.3
Q60491_CAVPO (Q60491) Preproacrosin precursor (Fragment)          29     14
Q37I66_RHOPA (Q37I66) Hypothetical protein                        29     14
Q28609_RABIT (Q28609) Putative preprosperminogen precursor       29     14
Q0Z2Y7_9BURK (Q0Z2Y7) Hypothetical protein                        28     33
Q8UAN4_AGRT5 (Q8UAN4) ABC transporter, membrane spann...          28     44
Q2P236_XANOM (Q2P236) Hypothetical protein XOO2636                28     44
Q5GZ23_XANOR (Q5GZ23) Hypothetical protein                        28     44
Q8L125_9RHIZ (Q8L125) PalF                                        28     44
Q7NZ32_CHRVO (Q7NZ32) Hypothetical protein                        27     59
Q39KS4_BURS3 (Q39KS4) TPR repeat protein                          27     79
Q2BPH3_9GAMM (Q2BPH3) Amino acid/cation symporter                 27     79
.....................................................................
Matrix: PAM30
Number of extensions: 13244
Number of successful extensions: 2198
Number of sequences better than 100.0: 16
Number of HSP's gapped: 2198
Number of HSP's successfully gapped: 16
.....................................................................
```

Table 5.5: BLAST - Truncated result set for a length 15 query with no filtering and a high E-value

```
                                                                Score    E
Sequences producing significant alignments:                     (bits) Value

Q0Y8W3_9EURY (Q0Y8W3) Luciferase-like                             28     54
O93734_9EURY (O93734) F420-dependent alcohol dehydro...           28     54
O93735_9EURY (O93735) F420-dependent alcohol dehydro...           27     54
```

Table 5.6: BLAST - Truncated set for a length 15 using BLOSUM62 with no filtering and a high E-value

```
                                                          Score    E
Sequences producing significant alignments:              (bits) Value

Q49535_9EURY (Q49535) Methyl-coenzyme M reductase...      68   6e-11
Q57067_9EURY (Q57067) Methyl-coenzyme M reductase...      68   6e-11
Q50273_9EURY (Q50273) Methyl-coenzyme M reductase...      68   6e-11
Q50386_9EURY (Q50386) Methyl-coenzyme M reductase...      68   6e-11
.....................................................................
Q2I7U1_9ARCH (Q2I7U1) McrA (Fragment)                     49   3e-05
Q6ITY9_9EURY (Q6ITY9) Methyl-coenzyme M reductase...      49   3e-05
.....................................................................
Matrix: PAM30
Gap Penalties: Existence: 9, Extension: 1
Number of Hits to DB: 26,063,232Number of extensions: 126060
Number of successful extensions: 6063
Number of sequences better than 10.0: 710
Number of HSP's gapped: 6063
Number of HSP's successfully gapped: 957
```

Table 5.7: BLAST - Truncated set for a multiple queries

5.4 Multiple queries

An examination is made of how well each program finds 3 medium and small queries, all lifted from the same record (Q50215) with 1, 2 and 0 errors introduced. BLAST is run without filtering and with an E-value cut-off of 10. It does a simple batch search without considering relations between the queries, and returns 957 results, shown in Table 5.7.

Adrasteia is run with the standard 0.05 cut-off, and with the queries ungrouped and each assigned the highest edit distance possible. The 531 matches are shown in (Table 5.8). These results are reduced to the 20 best matches using grouped matching. There is a high degree of confidence in the best results, as unlike BLAST the relationship of queries is considered, and so multiple queries being present in a record boosts confidence in it being an accurate match.

In order of significance, these are the 531 matches
found containing the patterns in ['YMSYGVGFTQYSTAAYTNNI', 'ILRDNLY',
'CGATNTFSYQSDEGL']

Record	In Text	Raw Score	P-Value
Q50215	YMSGGVGFTQYSTAAYTNN and ILDDNL and CGATN TFSYQSDEG	35	2.161048e-91
Q50386	YMSGGVGFTQYSTAAYTNN and ILDDNL and CGATN TFSYQSDEG	35	2.161048e-91
Q50273	YMSGGVGFTQYSTAAYTNN and ILDDNL and CGATN TFSYQSDEG	35	2.161048e-91
Q57067	YMSGGVGFTQYSTAAYTNN and ILDDNL and CGATN TFSYQSDEG	35	2.161048e-91
Q49535	YMSGGVGFTQYSTAAYTNN and ILDDNL and CGATN TFSYQSDEG	35	2.161048e-91
Q50908	YMSGGVGFTQYATAAYTNN and ILDDNL and CGATN VFSYQSDEG	31	9.759336e-69
.....			
Q2ABR3	YMSGGVGFTQYASATYTDN	09	2.030083e-13
Q2ABQ4	YMSGGVGFTQYASATYTDN	09	2.030083e-13

Table 5.8: Adrasteia - Truncated set for multiple queries with edit distances 5, 2 and 3

CHAPTER 6

Analysis and Conclusions

6.1 Performance

As would be expected, the performance of Adrasteia is highly dependent on machine type, with the amount of RAM present being an especially important factor. Indexing time is significant, especially compared to BLAST, but can be reduced by using smaller word lengths and a 2-3 partitioning scheme. It is recommended that the 2-3 partitioning scheme is used, as it has better indexing performance, and better match performance at the more commonly encountered error levels. The longest word length possible should also be used in generating indices, subject to the indices fitting on disk and being able to fit into RAM without having to be paged to disk (this is severely detrimental to performance).

Given the creation of good indices, the fastest match times for single queries occur with longer queries and low error rates. However the program does not suffer greatly with smaller queries, and similarly there is only a single order of magnitude difference between the times for a single maximum error and 8 maximum errors, on average.

Comparing indexed and non-indexed searches raises the question of whether the extra indexing time is worth the performance increase. For a single query of medium length, the indexed matching is roughly twice as fast for low error rates, but is overtaken by direct matching for high error rates. As the query length increases the benefits of indexing become greater, so if the user expects only to be dealing with extremely short queries then it may not be worth setting up the indexing first. An indexing method such as the one employed in this program seems better suited to longer queries, rather than the very short queries that the agrep algorithm used in this program matches fastest.

The area where indexing is most beneficial is with grouped queries. With a good indexing scheme, a large group of queries can be several orders of magnitude faster than the same search performed directly. For example an average of 40x was observed with a 2-3-4-5-6 scheme with 1024 word length and ten 25aa queries, and

this could probably be increased with a better indexing scheme. Unfortunately in most cases this sort of scenario is unrealistic, if the researcher had ten 25aa queries and was interested in protein sequences in the database that contain all of the queries, then they already have 250aa worth of information about the target sequence, an unrealistic expectation. So while grouping gives a definite improvement in searching for more than one query, the huge performance benefits of large grouped queries are unlikely to be seen in practice.

The speed of the program is comparable to BLAST in most cases, exceeding the performance of that program for multiple searches, and being not much slower for most smaller word lengths.

6.2 Accuracy

The scoring scheme and calculation of probabilities used in the Adrasteia program is very simple compared with that in BLAST. None the less, in several cases it managed to return a more useful set of results for short queries, due to the fact that a lot of possibilities seem to be filtered out of the result set for BLAST. This is in a large part due to the scoring matrices used by BLAST being set up for sequences that have changed quite a lot over time. A lot of the testing that was done in this work was artificially constrained to closely related sequences mutated by hand by the author, and so caution is needed about drawing too many conclusions from the limited test data so far acquired.

An area where there is a more definite advantage in accuracy over BLAST is in the quality of multiple query searches. Because Adrasteia takes account of multiple queries being in a record when calculating p values, it returns the better records for a multiple query search first, while BLASTP treats them as a simple batch search and does not adjust the scores of records with more than one query in them.

6.3 Program Development Lessons Learnt

The agile method of development proved suitable for developing a program of this nature. While the requirements were implemented as planned, an important lesson learnt comes from the use of indexing in the program. The initial requirements were based on the assumption that indexing would be suitable for short queries; however during development it became apparent that indexing was more beneficial to longer queries. A better analysis of the risks and rewards of

pursuing an indexing strategy would have paid dividends in time free to pursue other functional aspects of the program.

Despite all this, the functionality was improved in an ordered, sequential manner, as may be glimpsed from the changelog included in Appendix C. In fact, it may have been possible to improve the method of development used by considering the process more as a waterfall model, considering that the stages of development were predictable and mostly sequential. In fact a combination of agile and traditional development methods would seem to allow the balance between incremental functionality and well defined completion of milestones needed in a project of this type.

6.4 Conclusions and Future Work

The performance and value of the result set are reasonable for most short single query searches. The program, especially in direct match mode, provides a quick and easy alternative to BLAST for short peptide queries, although better indexing performance and better scoring of results is needed for it to be truly competitive. In terms of performance, agrep appears to work best on short sequences, and the indexing on long sequences. Work is needed in integrating the filtering method with the agrep algorithm in order to boost program performance on short sequences. In terms of accuracy, adapting agrep to use more sophisticated scoring matrices may make a difference in the quality of results.

The program is most beneficial as a multiple matching tool, providing very fast grouped searches and a better result set than BLAST for a group of short queries. By grouping queries or setting an appropriate cut-off value, a user can input a series of short peptide fragments and quickly find out if there are any proteins in the database that contain most or all of the fragments. Further work is needed in improving the accuracy of ranking multiple results, and speeding up multiple matching by making on the fly calculations of the usefulness of each record based on a temporary p-value.

Satisfying our research aim of presenting and analysing an alternative method for short and multiple peptide searches has shown ways of improving on existing search methods, and demonstrated that future work may improve the performance and accuracy of these types of searches.

APPENDIX A

Original Honours Proposal

Short Protein Sequence Similarity Searches Using Agrep

Miles T. Hampson

A.1 Background

This project is the development of a computer program, which when given a short (between 6 and 40 amino acids in length) peptide sequence, can search one or more protein sequence databases and return the proteins that contain the query sequence. Time permitting, the program will be extended to allow the user to input more than one query peptide. Making no assumptions about the order of occurrence, the program will then search the database(s) for all proteins that contain one or more of the query peptides.

The genetic material of a living organism contains information allowing organic compounds called proteins to be put together from various component amino acids. A protein can be modelled as a string composed of symbols from an alphabet of twenty, each symbol in this alphabet being a representation of one of the twenty amino acids that make up a protein sequence. Proteins under 40 amino acids in length are the focus of this project, and for our purposes any protein satisfying this criterion will be called a peptide.

The user will be considered to have one or more peptide sequences, which can be put into the computer as a sequence of symbols (a string), and for which the user would like to know the locations of any approximate matches in some database(s). An approximate match allows a given number of symbols to be inserted, deleted (indel) or mutated between the input string (the query) and the subject string (the database), thus allowing for differences between the two due to evolutionary and other factors. This may return a number of possible matches, and so a weighting may be given via a scoring matrix for the likelihood of indel

and mutation, or of the likelihood of the indel or mutation of each amino acid, thus allowing the most likely approximate matches to be given high alignment scores and shown to the user.

The query and subject sequences can be considered to be collections of independent identically-distributed (i.i.d.) random variables (that is the likelihood of a given symbol at one position in the string is the same as at any other position in the string, and a symbol being present in one position does not affect likelihood of other symbols being present). The likelihood of getting a given symbol (at any position, as all positions have identical probabilities for each symbol) can be modelled as a probability distribution (a mapping of a probability to each outcome, in this case a given symbol occurring at this position), the shape of the distribution depending on the scoring scheme used for each symbol.

This information can then be used to create another probability distribution, this time mapping matches between query and subject strings to alignments between the two. In other words this is a distribution that maps any random query string to an alignment score. From this we can return additional information when we match a user specified string in the database, namely a p-value, which, very roughly, is a value that indicates the likelihood that the match was obtained simply by chance.

A.2 Summary of Related Work

The Smith-Waterman algorithm [15] is a dynamic programming solution to the problem of local sequence alignment. When set up with a scoring system containing the probability of each possible indel or mutation event it will find the optimal local alignment.

The Basic Local Alignment Search Tool (BLAST) [2] uses heuristics to find good (but not necessarily the optimal) alignments based on an approximation of the Smith-Waterman algorithm, dramatically speeding up the search process.

Profile search systems such as ELM [4] and PRINTS [3] can take multiple query peptides and return proteins that contain most of the query peptides from a database. However the peptides must be given in the order they are found in the database protein.

A.3 Short Sequence Searches

The current method of choice for similarity searching with short proteins is a BLAST search with the settings adjusted appropriately. For example the short protein sequence search page at http://www.ncbi.nlm.nih.gov/BLAST/ is simply a normal BLAST search with the following adjustments:

- No composition-based statistics (for protein searches)
- No low complexity filter
- Higher Expect value
- Decreased word size
- PAM30 matrix

There are some problems with running Smith-Waterman or BLAST with small proteins, namely:

- Matching alignments produces too many gaps, and so the number of gaps needs to be constrained much more
- The extreme-value distribution model used to score local alignments does not handle shorter proteins very well, and so the evaluation of short matches is often not helpful.

A.4 Project Proposal

It is the primary aim of this project to investigate whether it is possible to provide more accurate match results for short protein similarity searches, compared to those returned by the Smith-Waterman and BLAST algorithms, by using an alternative method of evaluating matches. Time permitting, the investigation will be extended to see if it is feasible to take multiple unordered query peptides and find how close they are in the database.

This investigation will be carried out by developing a software suite with this functionality, and comparing the accuracy of results it generates against those of Smith-Waterman and BLAST for a set of short query peptides. In the second stage an evaluation of the speed and usefulness of searches of unordered short query peptides will be performed. The measures of accuracy and performance will be quantitative where possible, however qualitative results and remarks on

accuracy and the usefulness of any novel methods will also be provided to guide future work in this area.

A.5 Agrep

This project will use the agrep utility [17], specifically the Agrepy python port of the utility [16]. This utility uses a shift and add string matching algorithm to approximately match a sequence of input characters (a query string) against another reference string. Agrepy allows the user to specify the total number of indel and mutation events to tolerate, but does not allow the user set the individual numbers of each, or assign a weight to each for the probability of occurrence.

A.6 Functionality and Structure

The program will be developed as a layer that sits above the Agrepy code. It will have the following major responsibilities:

- Running multiple Agrepy searches over parts of the database, returning proteins that contain the query sequence in order of decreasing match. Where matches score equally, shorter target sequences are to be ranked above longer ones as the match will be more significant.
- Returning a p-value for each hit based on an appropriate probability model for short sequences.
- Possibly (depending on observed usefulness) incorporating a scoring matrix, such as PAM or BLOSUM, containing the probability of each possible indel or mutation event, and modifying Agrepy to use this information to produce more relevant results.
- Modifying the search based on user specified parameters such as sensitivity and word size
- Outputting the location and surrounding sequence for all matches in the database (will most likely need modification of Agrepy to do this, if extensive may need to build both agrepy and the control layer into a single utility).

- Time permitting, take multiple query peptides, in any order, and output their locations in the database and how close the matches were to each other. Possible optimize the search by searching near the location of the first match.

A.7 Milestones

- The first step will be the development of a basic layer that takes input strings from the user (or a text file) and feeds them directly to agrep, as well as taking a single reference string and giving it to agrep to match against. Then a set of tests will be run to determine the basic order of performance of the agrep algorithm in the situations that will be required for the project. While this stage is fairly trivial it allows time for examining the agrep code in more detail, as well as familiarity with the structure of the data that the algorithm will be working with. This will be used to help structure the revised proposal, and it is intended that this stage be finished before the end of April.

- The layer will then be extended to perform searches on multiple reference strings (to represent different databases or parts of databases), and to take and use additional user input parameters controlling the mechanics of the search (although not all options will be implemented at this stage). Testing should begin on appropriate distribution models to give good p-values. This should be finished before the exam study week.

- Over the winter holidays, before the end of July, work will proceed to the modifications of the Agrepy code, as the structure should be fairly well understood at this stage. User tuning options, obtaining information about the region of the match, distribution modelling and possibly the incorporation of a scoring system, will be completed. This may require a large amount of work over this period.

- To allow enough time to work on the draft dissertation, the last major coding stage will only be started if the project is not falling behind the schedule above. This stage involves changing the input to allow multiple entries, altering the control layer to run multiple searches and to link the match results together to report how close they were. This must be completed by the end of August.

- Enough performance testing will have been done at the completion of each of the above stages to allow the draft dissertation to be completed. Any

extra performance testing, as well as corrections to the code, will continue until the end of September in conjunction with the writing of the final dissertation. No major coding work will be undertaken during September or October to allow the necessary time for the seminar and poster.

APPENDIX B

Selected program segments

This section contains some of the more important segments from the Adrasteia code. For brevity they have been edited to show just the more relevant ideas in a section of code, and only critical comments have been included. Sections checking the validity of data have also generally been removed.

B.1 Typical values of program constants

```
#The maximum p-values to report (scores with p-values above this will
#not be reported back to the user)
PVAL_THRESHOLD = 0.05
#Length of the code words
WORD_WIDTH = 256
#Number of bits to set for the encoding of an attribute
NUM_SET_BITS = 1
#Each line of the database will be broken into pieces of these sizes
#for preprocessing.
PIECE_SIZES = [2,3]
#Table of piece sizes that most efficently fit each segment size
SEGMENTS = [[],[],[2],[3],[2,2],[3,2],[3,3],[3,2,2],[3,3,2],[3,3,3],\
              [3,3,2,2],[3,3,3,2],[3,3,3,3],[3,3,3,2,2],[3,3,3,3,2],\
              [3,3,3,3,3],[3,3,3,3,2,2],[3,3,3,3,3,2],[3,3,3,3,3,3],\
              [3,3,3,3,3,2,2],[3,3,3,3,3,3,2],[3,3,3,3,3,3,3],\
              [3,3,3,3,3,3,2,2],[3,3,3,3,3,3,3,2],\
              [3,3,3,3,3,3,3,3],[3,3,3,3,3,3,3,2,2],\
              [3,3,3,3,3,3,3,3,2],[3,3,3,3,3,3,3,3,3]]
#Length of a line of data in this database
LINE_LENGTH = 60
#For efficiency set a max bound on the size of any query protein
```

```
MAX_QUERY_LENGTH = 60
```

B.2 Building the indices

B.2.1 Parse database loop in Python

```
for i in xrange(num_pieces):
    piece_size = self.PIECE_SIZES[i]
    #db index pointer
    file_index_pointer=-1
    db_file.seek(0)
    #find the location of the beginning of the line
    position = db_file.tell()
    while 1:
        line=db_file.readline()
        if not line:
            break
        #if we have a database body section, index it, and
        #break it into pieces which are encoded and stored
        if line[0] != '>':
            length = len(line)-piece_size
            for j in xrange(length):
                #pass control to c code
                encodeDBPiece(line[j:j+piece_size])
        #Otherwise it is a header, so store its info
        else:
            file_index_pointer +=1
            header_index[file_index_pointer] = position
            allocateRecord()
        #before moving on to next line, get its starting position
        position = db_file.tell()
```

B.2.2 Encode database piece in C

```
void encodeDBPiece(char *piece)
{
int cur_bit;
unsigned long hash = 5381;
int i;
```

```
for (i=0;i<bit_string_segments;i++)
    result[i]=0;
while (i = *piece++)
    hash = ((hash << 5) + hash) + i;
srand(hash);
//Generate a series of integers from the seeded random number
//generator and use to create and store a code word for this
//piece. The code word is broken into a number of segments so
//it can be stored in the words of the current architecture.
for(i=0;i<NUMSETBITS;i++)
{
     cur_bit = STRINGWORDLENGTH * (rand() / (RAND_MAX + 1.0));
     while((1<<(cur_bit%WORDLENGTH))&(result[cur_bit/WORDLENGTH]))
        cur_bit = STRINGWORDLENGTH * (rand() / (RAND_MAX + 1.0));
     result[cur_bit/WORDLENGTH] |= (1<<(cur_bit%WORDLENGTH));
     bit_strings[(((bit_string_elms-1)%WORDLENGTH)
     *bit_string_segments)+cur_bit/WORDLENGTH] |=
     result[cur_bit/WORDLENGTH];
}
}
```

B.2.3 Bit Slice database pieces in C

```
void bitSlice()
{
    int j;
    int i;
    int bit_word_elms=((bit_string_elms-1)%WORDLENGTH)+1;
    if((bit_slices=realloc(bit_slices,sizeof(unsigned int)*
                ++record_segments*STRINGWORDLENGTH))==NULL)
    {
        printf("bit_slices out of memory\n");
        exit(1);
    }
    for (i=0;i<STRINGWORDLENGTH;i++)
        bit_slices[(record_segments-1)*STRINGWORDLENGTH+i]=0;
    //store bit strings in bit slice manner
    for(i=0;i<STRINGWORDLENGTH;i++)
        for(j=0;j<bit_word_elms;j++)
            if((1<<(i%WORDLENGTH)) &
```

```
                    bit_strings[(j*bit_string_segments)+
                    (i/WORDLENGTH)])
                bit_slices[((record_segments-1)*
                STRINGWORDLENGTH)+i] |= 1<<j;
    free(bit_strings);
}
```

B.3 Matching algorithm

B.3.1 Process query or queries in Python

```
for pattern in patterns_to_match:
    #create a pattern object for each query
    matchers.append(compile(pattern,len(pattern),max_errors[i]))
    #check the query length is within acceptable bounds
    ...
    #break the query into a number of segments equal to the
    #maximum number of errors for the query plus one
    num_segs = max_errors[i]+1
    segs_size=pattern_length/num_segs
    #for each segment encode all pieces
    createSegments(num_segs)
    completed=0
    for j in range(num_segs):
        #on the last one make size the leftover
        if j+1==num_segs:
            segs_size = pattern_length-(segs_size*(num_segs-1))
        for elm in segments[segs_size]:
            encodeQueryPiece(j,pattern[completed:completed+elm],\
            elm-2)
            completed+=elm
    for elm in groups:
        if i in elm:
            updateDropsetAnd()
        else:
            updateDropsetOr()
    i+=1
```

B.3.2 Dropset calculations in C

```
//Creates a set of the records that match when all segments of
//this query are or'ed with the previous dropset
void updateDropsetOr()
{
    ...
        for(i=0;i<record_segments;i++)
            dropset[i]=0;
    }
    for(i=0;i<record_segments;i++)
        for(j=0;j<secs;j+=record_segments)
            dropset[i] |= segments[i+j];
}

//Creates a set of the records that match when all segments of
//this query are and'ed with the previous dropset
void updateDropsetAnd()
{
    ...
        for(i=0;i<record_segments;i++)
            dropset[i]=-1;
    }
    for(i=0;i<record_segments;i++)
    {
        tmp = 0;
        for(j=0;j<secs;j+=record_segments)
            tmp |= segments[i+j];
        dropset[i] &= tmp;
    }
}

//returns the position of the next drop, starting from the position
//given in pos (counting from 1). Returns 0 if reached end of recs
int nextDrop(int pos)
{
    int i;
    for(i=pos-1;i<bit_string_elms;i++)
        if((1<<(i%WORDLENGTH))&dropset[i/WORDLENGTH])return i+1;
    return 0;
}
```

B.3.3 Search dropset for matches in Python

```
rec_num=nextDrop(1)
    while(rec_num):
        db_pos(index[rec_num-1])
        title =  db_file.readline().split()[1]
        rec = []
        line = db_file.readline()
        while(line and line[0]!='>'):
            rec.append(line[:-1])
            line = db_file.readline()
        record = ''.join(rec)
        begin_rec = 0
        for i in range(num_patterns):
            matches = agrepy(patterns_to_match[i],\
                      len(patterns_to_match[i]),\
                      record,len(record),1,matchers[i])
            if matches:
                if begin_rec==0:
                    begin_rec = 1
                    num_match_recs +=1
                results_record_headers.append(title)
                pattern_matches = []
                for elm in matches:
                    start=0
                    if elm[0]>0: start = elm[0]
                    pattern_matches.append(record[start:elm[1]])
                results_surrounding_text.append(pattern_matches)
        rec_num=nextDrop(rec_num+1)
    clearMatchInfo()
```

B.4 Scoring results

```
if(num_match_recs>0):
    #score matches and generate p values
    while i<total_pat:
        scr = 0
        pval = 1.0
        pat_num = 0
```

```
        this_header=results_record_headers[i]
        while i<total_pat and \
        results_record_headers[i]==this_header:
            #if there are multiple matches in this
            #record for this pattern then find the
            #most likely and use its score
            max_score=0
            max_elm=0
            max_len=1
            elmnt = 0
            for elm in results_surrounding_text[i]:
                len_sur_text = len(elm)
                this_score = score(len_sur_text,\
                patterns_to_match[pat_num],elm)
                if this_score>max_score:
                    max_score=this_score
                    max_len = len_sur_text
                    max_elm = elmnt
                elmnt += 1
            max_elms.append(max_elm)
            standardised_mean = max_score-means[max_len-1]
            if standardised_mean<0: standardised_mean = 0
            pval *= gaussian(standardised_mean)
            scr += max_score
            pat_num+=1
            i+=1
        cum_pat_num+=pat_num
        pat_nums.append(cum_pat_num)
        rslt_scores.append(scr)
        pvalues[j] =  pval
        j+=1
    #find ordering by pvals of results
    ordered = {}
    ordered = sorted(pvalues.items(),key=itemgetter(1))
    ordering = []
    cutoff = PVAL_THRESHOLD
    for elm in ordered:
        if elm[1]<=cutoff: ordering.append(elm[0])
        else: num_match_recs-=1
else:
 print "Sorry, could not find any results matching %s"%(patterns_to_match)
```

APPENDIX C

Program Changelog

```
Revision History:
 -Version 0.3.3 (current)
        - Changed p-value display format to scientific notation
        - Improved display format for header information
        - Updated Readme.txt

 -Version 0.3.2
        - Added a cut-off variable controlling the p-values of
          the results displayed to the user.
        - Changed the order of index creation so that everything
          is written to disk before an attempt is made to load
          indices (in case of failure).

 -Version 0.3.1
        -Resolved the occasional crashing on index creation bug
        -Improved the performance testing in the Measurement module
        -Fixed the error checking for the correct query length

    -Version 0.3.0
        -Implemented group matching feature
        -Added more detailed information to readme
        -Added more cases to testsuite and demonstration

    -Version 0.3.0 RC1
        -Implemented proper partition matching (as in agrep) for scws
        -Allows separate error rates for each query in a batch
        -Index data is now written to disk, so that only one
         indexing session needs to be done for a database even
         if the program is restarted.
```

```
-Version 0.2.0
    -Implemented better checking of input data.
    -Included a Test Suite.

-Version 0.2 RC2
    - Fixed bug where the lagrepy module of agrepy crashed
      on 64 bit architectures.

-Version 0.2 RC1
    -Implemented multiple query matching for both direct and
     preprocessed matches.
    -Removed dependencies on Numeric libraries, and slow
     recursive function for combinations

-Version 0.11
    -Changed directory structure slightly to fix compile errors
    -Fixed legal information
    -Removed table of means from calculation of p-values

-Version 0.1.0
    - Implemented p-values and sorting of results
    - Now returns information about area near match
    - Switch to direct match for potentially slow matches

-Version 0.09
    - Allowed specification of some more command line args
    - Implemented basic scoring of results

-Version 0.08
    - Changed dropset calculation to return next matching result
    - Rewrote init to parse every possible triplet of each record,
      so that there was less computation to be done in match

-Version 0.05
    - Implemented direct match
    - Implemented indexed match with SCWs calculated from all
      possible permutations of errors in the sequence partition
    - Implemented basic indexing
```

Bibliography

[1] Altschul, S. Amino acid substitution matrices from an information theoretic perspective. *Journal of Molecular Biology 219* (1991), 555–665.

[2] Altschul, S., Gish, W., Miller, W., Myers, E., and Lipman, D. Basic local alignment search tool. *Journal of Molecular Biology 215* (1990), 403–410.

[3] Attwood, T. K., Bradley, P., Flower, D., Gaulton, A., Maudling, N., Mitchell, A., Moulton, G., Nordle, A., Paine, K., Taylor, P., Uddin, A., and Zygouri, C. Prints and its automatic supplement, preprints. *Nucleic Acids Research 31* (2003), 400–402.

[4] Ausiello, G., Brannetti, B., Cameron, S., Chabanis-Davidson, S., Costantini, A., Gemnd, C., Linding, R., Martin, D., Mattingsdal, M., and Puntervoll, P. Elm server: A new resource for investigating short functional sites in modular eukaryotic proteins. *Nucleic Acids Research 31*, 13 (2003), 3625–3630.

[5] B., B., Bairoch, A., Apweiler, R., Blatter, M., Estreicher, A., Gasteiger, E., Martin, M., Michoud, K., O'Donovan, C., Phan, I., Pilbout, S., and Schneider, M. The swiss-prot protein knowledgebase and its supplement trembl. *Nucleic Acids Res 31* (2003), 365–370.

[6] Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R. C., Mellor, S., Schwaber, K., Sutherland, J., and Thomas, D. Manifesto for agile software development. 2001. Online. Internet. The Agile Alliance. October 2006. http://www.agilemanifesto.org/.

[7] Camacho, C., Coulouris, G., Dondoshansky, I., Zaretskaya, I., Papadopoulos, J., Bealer, J. Y. K., Madden, T., Raytselis, Y., and Merezhuk, Y. NCBI BLAST, 2006. Online. Internet. National Center for Biotechnology Information. October 2006. http://www.ncbi.nlm.nih.gov/blast/.

[8] Dayhoff, M., Schwartz, R., and Orcutt, B. *Atlas of Protein Sequence and Structure*. National Biomedical Research Foundation, 1978.

[9] Hyyro, H., Fredriksson, K., and Navarro, G. Increased bit-parallelism for approximate and multiple string matching. *ACM Journal of Experimental Algorithms* (to appear in 2006).

[10] Manber, U., and Wu, S. Glimpse: A tool to search through entire file systems. Usenix Winter 1994 Technical Conference, pp. 23–32.

[11] McEntyre, J., and Ostell, J., Eds. *The NCBI Handbook*. NLM/NCBI, 2002.

[12] Navarro, G. A guided tour to approximate string matching. *ACM Computing Surveys. 33*, 1 (2001), 31–88.

[13] Needleman, S., and Wunsch, C. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology 48*, 3 (1970), 443–453.

[14] Roberts, C. S. Partial-match retrieval via the method of superimposed codes. *Proceedings of the IEEE 67*, 12 (1979), 1624–1642.

[15] Smith, T., and Waterman, M. Identification of common molecular subsequences. *Journal of Molecular Biology 147* (1981), 195–197.

[16] Wise, M. Agrepy: Python port of agrep string matching with errors. 2002. Online. Internet. The University of Western Australia. October 2006. http://luggage.bcs.uwa.edu.au/ michaelw/pyagrep.html.

[17] Wu, S., and Manber, U. Fast text searching allowing errors. *Commun. ACM 35*, 10 (1992), 83–91.