# APPLICATION OF ANT COLONY OPTIMIZATION IN MULTIPLE SEQUENCE ALIGNMENT

**SUBMITTED IN PARTIAL FULFILMENT OF THE REQUIREMENT FOR THE DEGREE OF**

**BACHELOR OF TECHNOLOGY**

**(Bio Informatics)**

**(SESSION 2007-2011)**

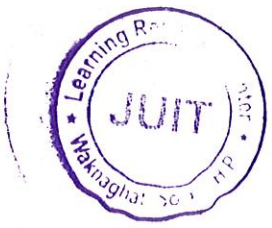

Submitted by

Varun Puri 071508

Abhinav Jain 071511


Under the Supervision of

Dr. Satish Chandra



**JAYPEE UNIVERSITY OF INFORMATION TECHNOLOGY**

**WAKNNAGHAT**

**SOLAN, HIMACHAL PRADESH**

**INDIA**

**JAYPEE UNIVERSITY OF INFORMATION TECHNOLOGY**

**WAKNNAGHAT**

**SOLAN, HIMACHAL PRADESH**

**Date: 25 MAY 2011**

***CERTIFICATE***

This is to certify that the thesis entitled Application Of Ant Colony Optimization In Multiple Sequence Alignment submitted by Varun Puri & Abhinav Jain in the partial fulfillment of the award of degree of Bachelor of Technology BioInformatics by JAYPEE UNIVERSITY OF INFORMATION TECHNOLOGY, WAKNAGHAT.

Signatures in full of Supervisor: Schandra

Names in Capital block letters: DR. SATISH CHANDRA

Designation: ASSISTANT PROFESSOR

## Acknowledgements

We take this opportunity to express our profound sense of gratitude and respect to all those who helped us throughout the project.

This report acknowledges to the intense driving and technical competence of the entire individual that have contributed to it. It would have been almost impossible to produce fruitful results without the support of those people. We extend thanks and gratitude to our Project Guide Dr. Satish Chandra who helped us at every step. He spent his valuable time from his busy schedule to train us and provided his active and sincere support for our daily activities.

We owe our heartiest thanks to Brig. (Retd.) S.P.Ghrera (H.O.D CSE/I.T Department) who've always inspired confidence in us to take initiative. He has always been motivating and encouraging.

We wish to express our deep sense of gratitude to Dr R S Chauhan (H.O.D, BI& BT Deptt.), for his guidance and all their valuable assistance in the project work.

Varun Puri(071508)

Abhinav Jain( 071511)

**B.Tech (BI)**

# Contents

## ABSTRACT

Ant colonies consist of large numbers of individuals, yet seem to be self-organising. This self-organisation is in part, due to interactions between individuals in the form of pheromone trails. This behaviour is encapsulated in a number of systems collectively known as 'Ant Colony Optimisation'. AntAlign is an implementation of Ant Colony Optimisation applied to the problem of multiple sequence alignment. Multiple sequence alignment is the process of aligning amino acid sequences to determine their homology. Along with the description of the algorithm itself is a study of its design and a number of examples of the operation of AntAlign. Also included are a critical evaluation of the system and a discussion of the further developments of AntAlign. Finally is a discussion of other areas of bioinformatics in which systems inspired by Ant Colony Optimisation could be applied.

# *1. Introduction*

## 1.1 The Problem Domain

Various Genome projects as well as individual laboratories are generating more and more protein sequences every day. However, our understanding of what these sequences mean is not increasing at the same rate. Currently the only reliable method for inferring the purpose or function of these sequences is by comparing them to other fully characterised proteins. The major issue with this is that many sequences may show only partial or very distant homology to each other. One method used to detect these distant homologies is multiple sequence alignment, which is detailed in a subsequent section. Although there are already several established method to align multiple sequences none are perfect and to this end there is still plenty of scope for new lines of investigation.

The aim of this project is to investigate the potential application of an 'Ant Colony Optimisation' (ACO) algorithm to the problem of multiple protein sequence alignment. This is to be achieved by the implementation of an ACO system modified for use with protein sequences followed by the evaluation of this system.

## 1.2 Bioinformatics

The 'Human Genome Project' which officially began in 1990 marked the beginning of a new era in biology. The aim of the project is to map the entire human genome, the complete genetic code of the human species. This project required a move away from the traditional use of computing within biology. The laboratory-based research of the human genome project began to create huge amounts of information. Far more than traditional systems could make use of. Computers now became essential not only in the use of this information but also in the determination of the sequence itself. The history of computing within the biological field stretches much further back than the 'Human Genome Project' itself but the project did

make computing in a biology a research field in its own right. This field is known as 'Bioinformatics' and is very much an applied science.

Bioinformatics makes use of the ability of computers to manipulate large quantities of information and from this information infer relationships and patterns that were beyond the scope of traditional methods. The term bioinformatics covers a wide range of fields mostly within the genomic and proteomic areas.

A particularly large area of bioinformatics lies within the use of databanks and the tools for searching them. There are several types of biological databanks:

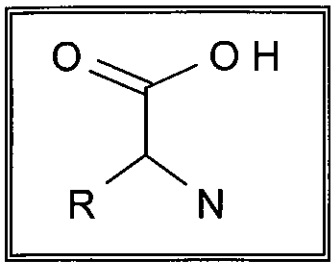
- **Primary archives** – these maintain data such as DNA and protein sequences and annotations associated with them. An example of this kind of database would be SWISS-PROT which holds annotated amino acid sequences. There are also more specialised archives that hold three-dimensional structures of proteins such as the Protein Data Bank (PDB).
- **Derived archives** – these maintain collections of data taken from primary archives.
- **Specialist archives** – these maintain more specialised information such as protein motifs which are similar to regular expressions.

The primary use of these databases is searching for homologous proteins and defining protein families. However, they can also be used for many other tasks when combined with other tools.

Other areas within the bioinformatics field include prediction of protein three-dimensional structure both by homology using databases like PDB and *ab initio* methods. Bioinformatic tools also include those that generate phylogenetic trees and produce multiple sequence alignments to investigate evolutionary relationships.

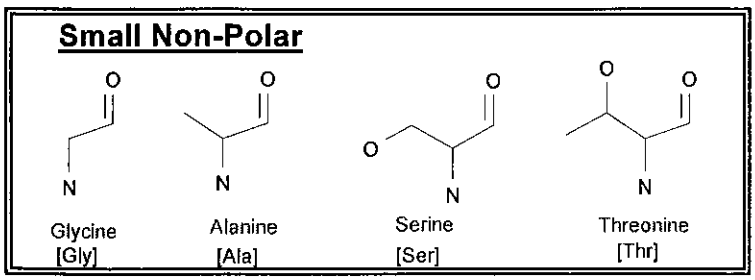
## 1.3 Amino acids

Amino acids are like building blocks; they are used in various permutations to create a wide range of different proteins. There are 20 basic amino acids (and 1 imino acid) that make up most proteins. There are also a number of non-standard amino acids that are present in a small proportion of proteins.

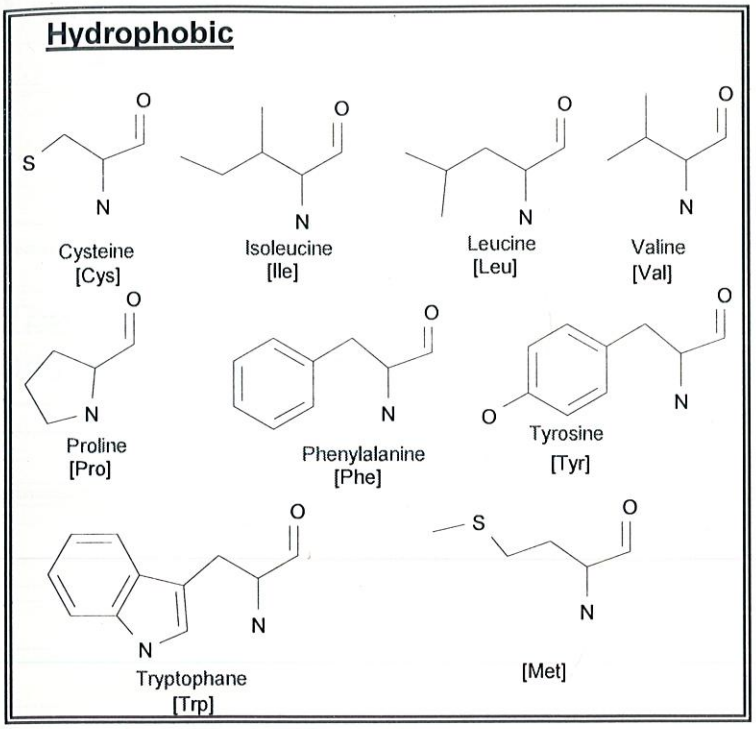



*The structure of a generic amino acid. The R represents a variable group, which is different in each amino acid and therefore has different properties.*

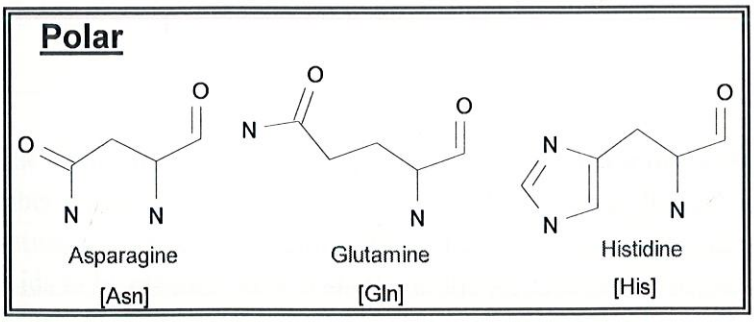
Figure (above) shows the generic structure of an amino acid. The different properties of the various amino acids are a result of the variable group designated as R. Each amino acid has a different R group, with different properties. These properties include its acid/base characteristics as well as size and hydrophobicity. The amino acids can be roughly divided into five groups based on their properties. These groups and the amino acids that are in them can be seen below in figure



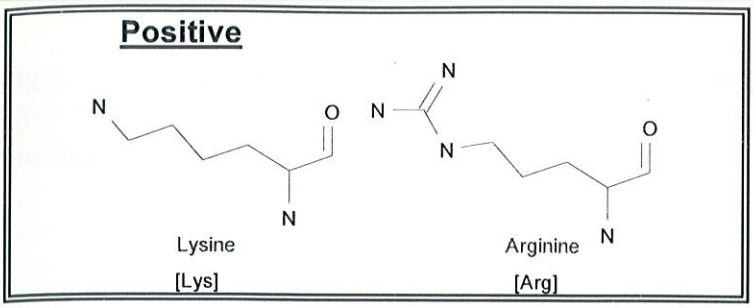

*The small non-polar amino acids. This group of amino acids is uncharged and does not display any significant polarisation.*



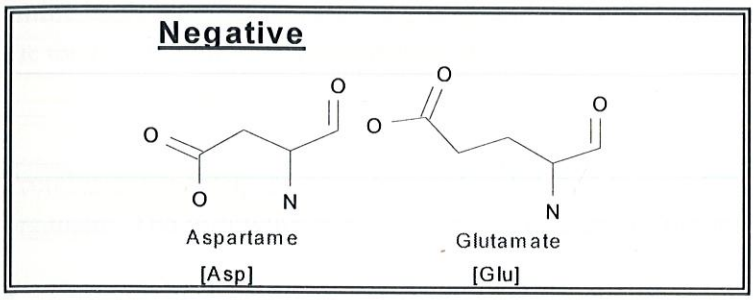

*The hydrophobic amino acids. These amino acids being hydrophobic do not interact with polar solvent such as water favourably.*




*The polar amino acids. This group of amino acids interacts most favourable with polar solvent such as water*

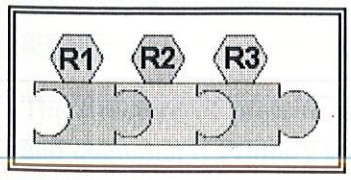



*The positive amino acids. These two amino acids have positively charged R groups.*




*The negative amino acids. These two amino acids have negatively charged R groups.*

The basic structure of the amino acids (i.e. the parts not in the R group) is able to bond with other amino acids. These bonds are called 'peptide bonds' and form a strong chemical link between the two amino acids. These bonds can be made end-to-end allowing strings of amino acids to be created. This is shown in the stylised figure below.




An abstract representation of the construction of a protein from amino acids building blocks

It is from these sequences of amino acids that proteins are formed. A protein may contain one of more amino acid sequences. Amino acid sequences may also allow for the binding of other 'prosthetic' groups. An example of this is haemoglobin that contains 4 separate amino acid sequences each of which binds a single prosthetic haeme group to bind oxygen. This further

highlights the flexibility of the amino acid system. Not only can they be used to create an almost infinite number of structures but they can extend their function by binding various prosthetic groups

## 1.4 Proteins

As seen in the previous section, proteins are made from one or more linear sequences of amino acids and that this allows great flexibility in the construction of proteins. In this section the enormous importance of proteins within biological systems is highlighted.

Proteins are a group of complex molecules that have a huge range of application within an organism. The following examples show just some of the roles that proteins fill.

- **Control of Chemical Reaction:** Almost all of the chemical reactions that occur in organisms are controlled a particular class of proteins known as enzymes. In addition to this, many reactions simply cannot occur without the interaction of an enzyme.
- **Protection:** The immune system is capable of dealing with a wide range of pathogens. Its adaptability is in part the result of the high flexibility of proteins.
- **Support:** Some proteins have very high tensile strength. The mechanical strength of bone and skin are largely due to a single protein family called collagen.
- **Transport:** A large number of proteins are situated in the cell membrane. Their role is to control the transport of small molecules such as dissolved ions and even other proteins.

The obvious flexibility that the examples above display is due to the ability of proteins to assume many different structures with many different properties. As mentioned earlier this is also extended by the ability of amino acid sequences to be created to bind various prosthetic groups.

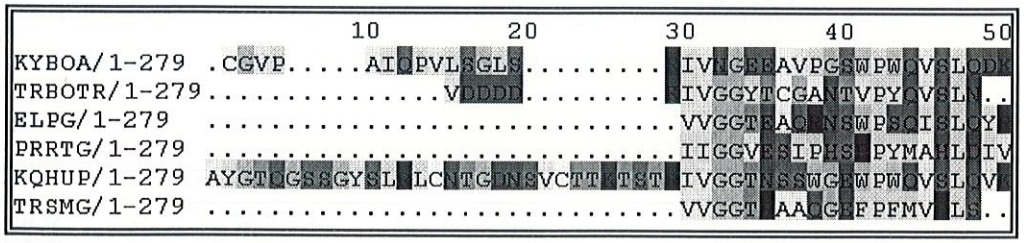
The structure of proteins is considered at four levels. These are primary, secondary, tertiary and quaternary. Primary structure refers to the amino acids sequence of the protein, that is the combination of the basic amino acids and occasionally other non-standard amino acids that make up the protein. Secondary structure is concerned with the regular patterns of localised three-dimensional structure within the amino acid sequence. Tertiary structure refers to the overall three-dimensional of all amino acids in the protein. Quaternary structure only occurs

if the final protein is made up from two or more amino acid sequences and defines how the different sequences interact.

The importance of the different levels of structure is that the structure of a level is defined by the structure of the one before it. Although methods to predict with 100% accuracy the complete three-dimensional structure of a protein from its primary structure do not yet exist, it is still possible to infer a certain amount of structural information. This can be achieved by comparing uncharacterised amino acid sequence to those for which the structure is already known. For this to be possible it is necessary to be able to locate these related proteins. This problem is one of the concerns of bioinformatics.

## 1.5 Multiple Sequence Alignment

A single amino acid sequence contains a wealth of information. This information dictates not only the final structure of the protein but also its action. However, elucidating this information from a single sequence is not a trivial problem, our understanding of proteins is not yet complete enough to fully characterise the protein from its primary amino acid sequence alone. It is possible to infer some characteristics with the 'multiple sequence alignments' of related proteins. By aligning several related proteins it is easier to detect areas within the proteins that are homologous or at least share partial homology.




Multiple sequence alignment of 6 serine proteases.

The figure (above) shows a partial segment of the multiple sequence alignment of 6 serine protease protein sequences. A single letter represents each amino acid and the colours represent the properties of the amino acid. At position 34 and 43 the columns form a continuous block of colour and all the sequences have the same amino acid at these points. This represents an area of homology between the different sequences. This information can be put to use in several ways. For example the PSI-BLAST algorithm uses multiple sequence alignments to produce a protein motif or profile. This motif represents the aligned proteins in a similar way to a regular expression. For example given the following alignment:

```
GIVCQDY
      .||.|.|

      -IVPQGG
```

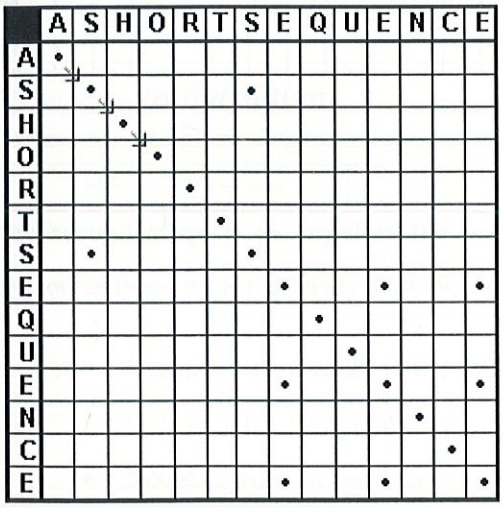
*The production of a motif from an alignment*

This motif can then be used to search sequence databases with the aim to discover other more distantly related sequences. It also allows the inference of potential function on uncharacterised sequences by matching their motifs to those of previously characterised proteins. The simplest of techniques for multiple sequence alignment is to do it by hand. To aid this, the amino acids are divided into five groups, each of which is assigned a colour. The figure 1.5.3 (below) shows these groups and one such colour coding.

| Colour | Amino Acid type | Amino Acids |
|---|---|---|
| Yellow | Small nonpolar | Gly, Ala, Ser, Thr |
| Green | Hydrophobic | Cys, Val, Ile, Leu, Pro, Phe, Tyr, Met, Trp |
| Magenta | Polar | Asn, Gln, His |
| Red | Negatively charged | Asp, Glu |
| Blue | Positively charged | Lys, Arg |

*An amino acids colour coding scheme*

Using this colour scheming system it is much easier to visually align the sequences by forming columns of solid colour. The complexity of multiple sequence alignments means that in many cases a computer is required to achieve the alignment in a reasonable time frame. The majority of multiple sequence alignment algorithms start by aligning the two amino acid sequences with the highest similarity. This process is called pairwise alignment. The consensus of this alignment (often known as a cluster) is then aligned with the next amino acid sequence. This process continues until all sequences have been aligned. The three general approaches to pairwise alignment are the segment method as well as global and local alignment. In segment alignment methods all parts of the two sequences which overlap one another are divided into segments of a fixed length. The segments of one sequence are then compared with all the segments in the other sequence for the best match. A technique known as the dotplot is often used for this. A computer is usually used to generate Dotplots for large segments, whereas they can be manually created for shorter segments. This is the case for figure (below).




An example dotplot created manually. The red arrows indicate the direction in which the segment alignment travels.

In global alignment the aim is to compare the aligned protein sequences over their entire length. This approach is useful if the sequences are expected to share homology over their entire length. Algorithms for global alignment aim to produce the best alignment by minimising the number of gaps that need to be introduced and maximising regions of similarity. An example of this is the Needle-Wunsch algorithm, which proceeds in 3 stages:

1. Initialisation
2. Scoring

3. Alignment

For example if we wanted to align the two sequence HEEFFCEHFFE and HHEFCHE we would start by creating a grid. Then we need to decide on a scoring scheme. The simplest is 1 for a match, 0 for a mismatch and 0 for the insertion of a gap. The initial state of the grid would therefore look like the following:

| | | H | E | E | F | F | C | E | H | F | F | E |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | 0 | | | | | | | | | | | |
| H | 0 | | | | | | | | | | | |
| E | 0 | | | | | | | | | | | |
| F | 0 | | | | | | | | | | | |
| C | 0 | | | | | | | | | | | |
| H | 0 | | | | | | | | | | | |
| E | 0 | | | | | | | | | | | |

*Initialisation*

The second stage is to populate the matrix by scoring each position. This is achieved by sequentially going through all the cells and determining the highest value from the following:

- Gap penalty + value to the left
- Gap penalty + value above
- Score of current position + value of cell diagonally up and left

The highest value of the three calculations shown above is used as the value for the current cell.

This would therefore proceed as shown below in figure

| | H | E | E | F | F | C | E | H | F | F | E |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| H | 0 | 1 | 1 | 1 | | | | | | | | |
| E | 0 | 1 | 2 | 2 | | | | | | | | |
| F | 0 | 1 | 2 | 2 | | | | | | | | |
| C | 0 | 1 | 2 | 2 | | | | | | | | |
| H | 0 | 1 | 2 | 2 | | | | | | | | |
| E | 0 | 1 | 2 | 3 | | | | | | | | |

*Scoring*

These calculations would continue until the matrix was filled as shown in figure

| | | H | E | E | F | F | C | E | H | F | F | E |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| H | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| E | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 2 | 2 | 2 |
| F | 0 | 1 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 3 |
| C | 0 | 1 | 2 | 2 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |
| H | 0 | 1 | 2 | 2 | 3 | 3 | 4 | 4 | 4 | 4 | 4 | 4 |
| E | 0 | 1 | 2 | 3 | 3 | 3 | 4 | 4 | 5 | 5 | 5 | 5 |
| | 0 | 1 | 2 | 3 | 3 | 3 | 4 | 5 | 5 | 5 | 5 | 6 |

*Scoring completed*

Then the final stage or alignment (also known as traceback) can begin. For this we start in the bottom right cell which has a value in this case of 6. The algorithm then attempts to determine the direct predecessor of the current cell. In this case all three predecessors have a value of 5 and since we have set the gap penalty to 0 neither the cell above or to the left of the current cell could have been it predecessor since they would have to have had the same value. Therefore the cell diagonal to the current cell must have been it predecessor. The algorithm then moves its focus to the predecessors cell and the process repeats until the origin it reached. It is possible that for any traceback step there could be more than one possible

predecessor. If this should occur then there is more than one possible alignment. This example has two possible alignments shown in figures below:

| | | H | E | E | F | F | C | E | H | F | F | E |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| H | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 2 | 2 | 2 |
| E | 0 | 1 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 3 |
| F | 0 | 1 | 2 | 2 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |
| C | 0 | 1 | 2 | 2 | 3 | 3 | 4 | 4 | 4 | 4 | 4 | 4 |
| H | 0 | 1 | 2 | 2 | 3 | 3 | 4 | 4 | 5 | 5 | 5 | 5 |
| | 0 | 1 | 2 | 3 | 3 | 3 | 4 | 5 | 5 | 5 | 5 | 6 |

*Alignment solution 1*

| | | H | E | E | F | F | C | E | H | F | F | E |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| H | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 2 | 2 | 2 |
| E | 0 | 1 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 3 |
| F | 0 | 1 | 2 | 2 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |
| C | 0 | 1 | 2 | 2 | 3 | 3 | 4 | 4 | 4 | 4 | 4 | 4 |
| H | 0 | 1 | 2 | 2 | 3 | 3 | 4 | 4 | 5 | 5 | 5 | 5 |

| E | 0 | 1 | 2 | 3 | 3 | 3 | 4 | 5 | 5 | 5 | 5 | 6 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|

*Alignment solution 2*

Figures above correspond to the following alignments respectively:

```
HEEFFCEHFFE
H_EEFFCEHFFE
```

*The resulting alignments*

This approach would not successfully align two sequences that contained sections of similarity at large distances apart in the different sequences. This kind of problem is quite normal in biological sequences as insertion and deletion mutations push or pull sections of the sequences out of alignment with one another. This problem requires local alignment. With local alignments techniques the sequences are searched for areas of localised similarity. The final alignment is based on these regions of local similarity. The Smith-Waterman algorithm is the most well known method for achieving this. It constitutes a modification of Needle-Wunsch algorithm to allow better alignments of localised segments. These consist of small changes in each stage of the alignment process:

1. **Initialisation** – The first column and row are populated with the value 0 to allow the sequences to slide over each other without penalty. Note that by selecting a gap penalty of zero in the original Needle-Wunsch algorithm the same effect can be seen.
2. **Scoring** – The Smith-Waterman algorithm allows the additional choice of stopping the alignment of the current section.
3. **Alignment** – The Smith-Waterman algorithm starts traceback at the cell containing the optimal value (largest number) wherever it appear in the matrix which in the case of the example above is the bottom right and continues until the region of similarity ends. The Needle-Wunsch algorithm always starts in the bottom right irrespective of where the optimal value is located.

The majority of multiple sequence alignment systems such as CLUSTAL-W used today are based on successive application of pair-wise alignment. [8] The exception to this are those based on the Hidden Markov Model (HMM) which defines a type of finite state machine. The advantage of the HMM is that it give more flexibility when assigning scores and gap penalties since in real proteins these are subject to location within the sequence not just amino acid type or merely their introduction .

## 1.6 Ant Colony Optimisation

Ant Colony optimisation or ACO is a form of swarm intelligence optimisation algorithm inspired by the foraging behaviour of ants. An ants nest can contain thousands of individuals yet they all seem to work together to achieve their goals. Studies of ant behaviour have shown that their actions are largely self-organising, they achieve the level of co-operation shown without the aid of direct supervision.

When ants forage for food initially they have no idea were to look for it. They cover the area in a random fashion until they find something of worth which they carry it back to the nest. In performing this action they lay a trail behind them. This trail consists of a pheromone, which attracts other ants of the same nest. The net effect of this is that when one ant finds some food it leaves a trail to direct others to the same place. As more and more ants follow the same path the trails intensity increases and in turn attracts more ants. This is an example of positive re-enforcement. This alone is not enough to solve any difficult problems. The additional feature of this system, which allows for a more complex behaviour is that the pheromone trails, evaporates with time. The advantages of this are best described by an example.

Swarm intelligence and indeed ACO are general-purpose algorithms and are not designed to solve any specific problem. They are most useful in situations were no algorithm yet exists that can efficiently solve a problem without having to try every solution to determine which is the best. An example of this type of problem is the travelling salesman problem (TSP) in which a travelling salesman has to visit a number of different cities in the shortest time possible without visiting any city more than once. All routes have to be tested exhaustively before the shortest route can be determined. Under optimum conditions ACO algorithms have out performed several other algorithms when applied to the TSP. [13] ACO has also been applied to other problems such as routing of packets over networks where it slightly out performed 'link state', the current standard. [14,15 & 16]
ACO algorithms are not guaranteed to produce the most optimised solution but rather a 'best guess' solution that may or may not be the optimum.

In this project we will be attempting to apply ACO to the problem of multiple sequence alignment which was discussed in the previous section.

# 2. Planning and Approaches

## 2.1 Discussion of Approaches

### 2.1.1 Choice of Language

The implementation of an ACO algorithm requires a great deal of mathematical calculation and although ACO algorithms are essentially relatively simple their implementation is not. Therefore the choice of which language to use must be a balance between its efficiency and its ease of use. This would suggest the use of a high-level language. The two most well known high-level languages are C/C++ and Java ™. C/C++ is a very efficient language and is of a sufficiently high-level to enable rapid development. Java ™ does not have the reputation for efficiency that C/C++ has especially when a graphical user interface is concerned. The implementation of this algorithm does not require a complex user interface. A simple console base interface is sufficient. Java ™ is a modern Object-Oriented language and its performance is close to that of C/C++ when the user interface is not a concern. Familiarity with the language is also an important factor. I have little experience with C/C++ whereas my ability to programme in the Java ™ language is quite strong. For this reason the language I have chosen is Java ™ since it is the one with which I am the most familiar and its efficiency in the non-graphical area is similar to that of C/C++. Portability is not a requirement of the problem domain but is also a benefit of choosing Java

### 2.1.2 Data Representation

The way in which data used by the system is represented is an important factor when deciding on what approach should be taken to solving the problem domain. The choice of language also greatly effects this. This algorithm will be dealing with a number of protein sequences that are traditionally represented as a string of characters, each characters representing a single amino acid. Therefore one option would be to simply represent the

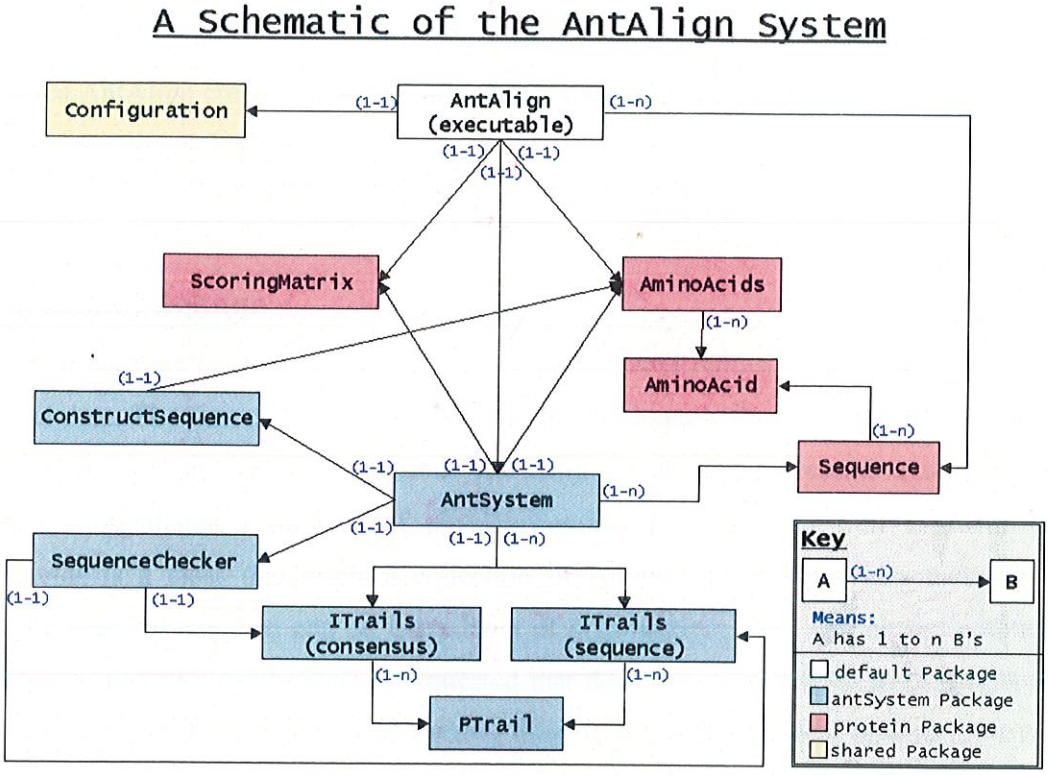
protein sequences used by this system as strings. Java's support of string manipulation is good, especially in the newly released JDK 1.4, which provides additional functions for dealing with regular expressions. This approach would also reduce the amount of code that was needed since separate classes would not need to be created for all the data that needs representing. This approach may also boost efficiency since most operation would involve simple string manipulation.

The other option is to adopt a more object-oriented (OO) approach where each item is represented by a separate specialised object. This would require an increase in the amount of code and the classes will also need to be carefully designed to ensure that they are kept streamlined and processing is efficient. An OO approach does have advantages such as easier code reuse and it will also make the system more organised and maintainable. If this approach is used a number of classes will have to be created to represent objects such as amino acids and protein sequences. These classes have any number of additional uses outside of the scope of this investigation. Therefore this approach is far more flexible and allows the code to be easily extended or used in other applications. Java ™ is an OO programming language and is therefore better suited to this approach.

With these issues in mind it would seems that an OO approach to the design would be more appropriate. This project is aimed at implementing an algorithm within a relatively short amount of time. There is no reason however, to not take into account the possibility of further development of the system or at least reuse of some of it components. The code will therefore be divided into a number of packages that can not only be reused as a whole but also individually.

# 3. Development

## 3.1 System Summary




A schematic overview of the AntAlign System showing the relationships between .

Figure (above) is a simple schematic of the AntAlign system that shows the relationships between the various classes. It does not show classes that are not instantiated such as the factory classes or those objects of a transient nature. As can be seen from this figure there are 3 packages plus the default package. These are protein, antSystem and shared. I will provide a brief explanation of the purpose of each package and the classes within it.

## The default Package

This package contains only one class, AntAlign that forms the executable portion of the system. It can been seen that it is associated with an instance of the Configuration class which enable it to locate all the data files required by the system.

The AntAlign class also handles the loading of the data files and facilitates parsing with a number of factory classes, namely ScoringMatrixFactory, AminoAcidsFactory and SequenceFactory. Once all the data files have been loaded and the sequences have been constructed AntAlign creates an AntSystem object and passes control to it along with the objects and data it has loaded.

## The protein Package

The protein package contains all the classes associated with representing proteins and other biological data. The first collection of classes in this package is those associated with amino acid and protein sequence representation. The AminoAcid class represents a single amino acid and encapsulates a number of physicochemical properties as well as identifiers. AminoAcids is a class that holds a collection of AminoAcid and provides methods for retrieving them by name to add an extra level of organisation. This also allows additional checking. If an AminoAcid object is requested that does not exist and InvalidAAReference exception is thrown. The AminoAcidsFactory class is never instantiated, instead it provides static methods to load amino acid data from persistent storage and use it to create a fully populated AminoAcids object. Should an error occur during the parsing of the amino file a new instance of AminoParseException is thrown with details of the error encountered. The Sequence class represents a protein amino acid sequence and is essentially a collection of references to AminoAcid objects along with a number of methods for accessing and manipulating the sequence. The SequenceFactory class provides static methods used by the constructor of the Sequence class to create the contents of the Sequence object. The SequenceFactory object throws a new InvalidSequenceException if illegal characters are found in the sequence.

The protein package also contains a number of classes involved in creating and representing scoring matrices. The ScoringMatrix class produces an object the can be used to score amino acid substitutions. The ScoringMatrixFactory provides static methods to load scoring matrix data from persistent storage and use it to create a ScoringMatrix object. Should an error occur during the parsing of the matrix file a new instance of MatrixParseException is thrown containing details of the error encountered.

## The antSystem Package

The antSystem package contains all the classes associated with implementing the ACO algorithm. Referring back to figure it can be seen that the class AntSystem forms the hub of the AntAlign system. It is this object that is created by the executable portion of the program to carry out the actual alignment. Encapsulated within this object is the ACO algorithm devised for this system.

The algorithm is based around a number of classes used to represent the agents and their pheromone trails. The first of these is the Ant class, which forms the agents within the system. As the Ant objects pass through the system they leave behind pheromone trails which are embodied by the class PTrail. PTrail contains data pertaining to the trails strength and position. These objects are used in part to determine the paths taken by subsequent Ants. The interface ITrails defines the behaviour of classes designed to maintains PTrail objects. In this system this role is taken by the NaiveTrails. It provides methods to add and add to PTrails as well as methods for matching PTrails and applying evaporation to existing PTrails. The MatchedTrail objected is used by the getMatches() method of NaiveTrails to return information about a PTrail that has been matched to a specified sequence.

The class SequenceChecker provides the self-regulating element of the AntAlign system. The method it provides is invoked every 10 cycles to check through existing pheromone trails to ensure that no rogue trails have entered the alignment. Rogue trails are defined as those that contains sequences which are not within the allowed drift parameter and would result in the output sequence being scrambled when compared with the sequence used to generate it.

The final two classes in this package are Construct Sequence and pAmino. The Construct Sequence object provides a method, which take an object implementing the ITrails interface and use it to reconstruct a Sequence from the PTrail objects it holds. The PAmino class is used during the construction process to hold data regarding the strength of amino acid placements at particular positions.

### The shared Package

This package is used to hold classes, which are used as utility classes within the system.

The Configuration class is used to load and parse Microsoft Windows style in files. It is used by the Ant Align classes to read in locations of data files as well as runtime variables. It provides read-only access to the information contained in the ini file loaded.

The Output Formatter class provides static methods used by the Ant System object to format data contained in Sequence objects into HTML/CSS code for placement in the output file.

The Invalid Argument Exception class provides a general-purpose object thrown by several other classes when an invalid argument is passed to a method.

## 3.2 The Design Process

### 3.2.1 The Design Approach

A bottom up approach to the design was adopted. This was mainly because the implementation of the actual ACO algorithm is quite complex and relied heavily on the way in which the lower level objects have been constructed. I was essential that the objects that represent the data such as amino acids and sequences be designed carefully not only to successfully encapsulate the data they represent but also to minimise that amount of memory they occupy.

### 3.2.2 The Representation of Amino Acids and Protein Sequences

The representation of amino acids within the system is quite straightforward. Amino acids each have a name and a number of physicochemical properties, which make them unique. The naming of amino acids can be done in three different ways. They have a full name such as 'Lysine', and short three-letter name such as 'Lys' and a single letter representation such as 'K'. All of these are easily represented with Strings and char data types. The only additional requirement is that the short name be only three letters long. The difficulty comes in representing the other properties of the amino acids. The values of these properties are take from real world observation so are real numbers and should therefore be represented by float or double data types. However, initially I intended to base the scoring of alignments on the properties of the amino acids. For this integers would allow much faster calculation times but at the expense of precision. It was decided that the double data type would be used to represent the physicochemical properties of hydropathy, pKa and weight since the values could be converted to integers for calculations should speed of execution become an issue.

It was later decided not to use the amino acids properties directly for scoring alignments but to use a more simplified scheme, which is explained in a later section. As a result of this decision there was no longer a need for the additional properties of the amino acids to be included in the system. The additional data was kept in place with a view to code re-use and system development.

Protein sequences, as shown are made up from a string of amino acids. Therefore the most logical method to represent them within the system is as a collection of amino acid objects. The most used format of representing a protein sequence is by their single letter names. Therefore the inputted sequences are going to have to be converted from a string of characters into a collection of amino acid objects. The simplest way to achieve this is to use the letter name to look up the properties associated with that amino acid and use it to construct an Amino Acid object. This is wasteful since a single amino acid type could be repeated any number of time in a given sequence, the repetition becomes even more

pronounced when you consider that a huge number of ants may be generated. Therefore it was decided that the system should create a single set of Amino Acid objects to which all other occurrences of the amino acid are referenced. This should save a great deal of memory.

### 3.2.3 Persistence of Amino Acids Data

It was realised that there was little merit in hard coding the values associated with amino acids into the system, as this does not provide a very flexible set of classes. So some type of persistent data store was required. Since the data that needs to be stored is very simple a flat text file was decided upon and the format of this file was to be slightly modified "comma separated values" (CSV) list. The file is split into two sections. The first section delimited by the [Def] and [/Def] tags contains a list of the references used to obtain the data for the amino acids. The second section delimited by the [Data] and [/Data] tags contains the actual comma separated values.

As a result of having a data file it was also necessary to create a parser for the file to handle reading in the data, validating it and using it to construct the Amino Acid objects. For this the class Amino Acids Factory was written.

### 3.2.4 The Scoring Mechanism

Initially I intended to base the scoring mechanism on characteristics of the individual amino acids. With further research into current methods used to score alignments I found that there already exist a number of simplified mechanisms. The simplest scoring method is to assign each match the value of 1 and all mismatches a 0. This is a very fast scoring mechanism but not very useful since it does not take into account the relative importance of the different amino acids. For example the existence of a cysteine in one sequence and its absence in another can make a profound difference in the structure and function of a protein but in this scoring mechanism it would have only a very minor effect. Other scoring mechanisms work on a similar basis but attribute different scoring to different amino acid pairs. The two most widely recognised substitution matrices are PAM250 and BLOSUM62. PAM250 was based on the relative probability of substitution within a number of related proteins. The number refers to the how related the sequences used to generate the matrix were. For example the

PAM250 matrix was generated using sequences which showed only 20% identity to one another. The values used in the matrix are expressed as *log-odds* of observed mutation frequency over expected mutation rate. The values calculated are then multiplied 10 to avoid decimal places. Logarithmic values are used to allow the summing of results rather than multiplication since log (x) + log (y) ≡ log (x * y). BLOSUM62 was devised by taking blocks of aligned proteins without gaps from many different protein families. These blocks were then analysed to determine the frequency of the substitution of each amino acid with every other amino acid. The aim of the BLOSUM matrix was to produce a set of values that would better identify distant relationships. As with the PAM matrix the results are expressed as log-odds * 10. The number part of the BLOSUM matrix name refers to the threshold used. The value 62 for example refers to a threshold value of 62% identity. That is to say blocks with identities in excess of 62% were replaced prior to substitution frequency determination. This was done to eliminate the over-weighting of closely related sequences.

### 3.2.5 Persistence of Scoring Matrix data

There is some debate as to which scoring matrices are best in which situation. Therefore it seemed logical to allow the user to choose which matrix they want to use for a particularalignment. This suggested some form of persistence was required. Initially some form of XML mark-up was considered. The relative simplicity of the data lent itself more readily to a simpler format. The format chosen was similar to that chosen for the amino acid data, a modified CSV file. An example of the format can be found in the appendix in section . It is split into two sections, a definition section and a matrix section. The definition section is delimited by [Def] and [/Def] tags. In this case there is more in the definition section than just references. It also contains two other fields. A 'MatrixName' field, which is used to put a name to the and an 'AAOrder' field, which contains a list of single letter amino acid names in the order they appear in the matrix. The AAOrder field is used to associate the numeric data in the Matrix section with the amino acid it relates to. The matrix section is delimited by the [Matrix] and [/Matrix] tags. The contents of this section are a list of simple comma separated values. They are ordered in such a way that the first row contains data associated with the first amino acid named in the AAOrder field in the definition section and so on. This also works in the other direction, the first value on a line is associated with the first amino acid

named in the AAOrder field and the second value with the second amino acid named and so on.

### 3.2.6 The Pheromone Trail classes

A number of classes were required to represent and manipulate the pheromone trails created by the Ant agents as they pass through the system. The first of these classes is PTrail, which is used to represent a pheromone trail. This class simply encapsulates details such as the strength of the trail as well as its position in the sequence. The difficulty with handling the pheromones is in manipulating them. Since a large number of trails are generated by the system it is necessary to hold them in some kind of data structure. The data structure apart from storing the PTrails has to be able to perform a few other functions. Firstly when an Ant is created the sequence it carries must be compared to those of trails already in existence to determine the possible paths the Ant will take. This look for partial matches as well as exact ones. The second function it must provide is the adding of PTrails. This is not as simple as just adding them to the data structure. The PTrail that is being added must first be compared to those already in the data structure for exact matches in terms of the sequence they represent and their offset. If a match is found then instead of adding a new PTrail the trail strength associated with the existing one is modified. Finally the data structure object also has the responsibility of applying evaporation to the trail strengths at the end of every cycle. These functions are essentially based around searching for objects. However, the data represented by the PTrails does not lend itself to any standard data structure optimised for searching such a binary tree. In the time frame available for the development of this system it was decided to provide a simple solution to the problem rather than attempt to develop a more optimised data structure. To aid in the addition of a more optimised data structure at a later date it was decided that an interface should be developed to define the methods associated with PTrail data structure. This would allow new objects to be easily inserted into the system without having to modify the code extensively. The interface is defined in ITrails, which all PTrail managers must implement. The ITrails interface defines five methods:

- **putPTrail(PTrail trail)** – Should search for PTrails that match trail and if one is found modifies its strength by the amount held in trail. If no matches are found it should simply add trail to the data structure
- **getMatches(Sequence sequence, int offset, int drift)** – Should return an Array List containing all PTrails that (partial) match the input parameters.
- **applyEvap(float evapFactor, int length)** – Should modify the strength associated with each PTrail by the factor evapFactor. The length parameter will be covered later.
- **getTrails()** – Should return an Array List containing all the PTrails maintained by the object.
- **Remove(PTrail trail)** – Removes the specified PTrail

The class that implements ITrails in this system is NaiveTrails. NaiveTrails holds all PTrails associated with it in an ArrayList. The partial matching process employed by NaiveTrails is shown below in pseudo-code:

The MatchedTrail class represents a PTrail that has be matched by the above process. It also hold the probability associated with the match.

NaiveTrails handles the putPTrail() method in a similar way except that it looks for exact matches and does not generate MatchedTrail.

```
LOOP through all PTrails
        IF (Ant Sequence contains the PTrail sequence) AND (PTrail offset = Ant offset
+/- drift)
        THEN created MatchedTrail object for PTrail and added to return ArrayList
        END IF
END LOOP
RETURN ArrayList of MatchedTrails
```

The applyEvap() method simply loops through all the PTrail objects held by the NaiveTrails object and modifies the strength of the sequences by the factor passed to it by AntSystem. The evaporation factor can be set in the ini file specified for the run. During initial testing a small flaw in this system was found. As lengths of sequences carried by the Ant objects increase, trails laid by Ants with short sequences associated with them get eliminated very quickly. To avoid this it was necessary to modify the method to prevent the application of evaporation to PTrail objects that had sequences shorter than those currently being created. Essentially this fixes short trails in place and prevented them from being eliminated so early that they had little to no effect on the next round of Ant objects.

### 3.2.7 The AntSystem Class

The AntSystem class constitutes the main part of the Ant Align system. It represents the actual ACO algorithm. There were several issues that arose during the design of this class that are specific to the problem of multiple sequence alignment and are therefore not mentioned in previous ACO papers.

Firstly there are a large number of variables within the system that a user may want to tailor to a particular alignment. These include the scoring matrix and amino acid data files that they wish to use along with several other values, which are used during the running of the alignment system. The number of parameters needed is quite large and therefore I decided that having them all required on command line would result in a very cumbersome system. I decided that an initialisation file would be best. This would mean only a single command line parameter would be required. The format chosen for this file is that of a Windows style ini file. The ini file is split into three sections. AntAlign is the first and contains the locations of the amino acid and scoring matrix data files. The second section is RunParams. This contains a list of the sequences that are to be aligned as well as the results filename. The final section is entitled Variables. This section contains the following variables used by the system:

| | |
|---|---|
| *Cycles* | The number of cycles through the algorithm to perform. |
| **PopPerTime** | The number of Ant to generate during each cycle for each sequence |
| **StartLen** | The starting length of the ants payload |
| **EndLen** | The ending length of the ants payload |
| **Evap** | The trail evaporation factor |
| **Intensity** | The trail intensity factor |
| **Drift** | The permitted drift of ants |
| **RndChance** | The chance of a random alignment |

*The variables used in the initialisation*

The above represents a slightly modified list of variables than was originally implemented. Initially StartLen and EndLen were represented by a rate of length increase variable. The starting length was set at 3 it the length was increased by the factor defined by the rate of increase. However, it soon became apparent that the length of the Ants payload could quickly exceed the length of the sequences used on long runs. Therefore the variable was replaced by StartLen and EndLen which are used by the system to determine the rate of increase in Ant payload size relative to the number of cycles that are to be performed. The rate is given by:

(EndLen - StartLen) / Cycles

This method ensure that the payload length never exceeds the length of the sequence providing the user never sets EndLen to be greater that then length of the sequence. During each cycle a number of ants equal to the PopPerTime variable are generated for each sequence. The length of the payload sequence is determined by:

(rate of payload increase * current cycle number) + StartLen

The offset the payload sequence is taken from is determined randomly to give a random set of ants and is given by:

Random( sequence size – current payload length)

Which generates a number between 0 and the size of the sequence – the current payload length.

In the case of most ACO's the problem the algorithm is attempting to solve is singular. That is the algorithm is attempting to find the most optimised conditions for a single task. Multiple sequence alignment by definition is not working on one set of data. It is working on several sets of data at a time i.e. the sequences that are to be aligned. The alignment does require a degree of communication between the alignments of the different proteins. During the design of this system two manners in which to achieve this cross talk were devised. The first method was to provide a single ITrails object for each sequence. Ants generated from a sequence can only be used to generate pheromone trails in the ITrails object that the sequence is associated with. However, they do have read access to the ITrails objects associated with all other sequences in the system for purposes of finding potential paths to travel. This provides the required cross-talk without compromising the ITrails associated with a sequence so they can still be used at the end of the alignment to generate an output sequence. This does mean that for every ant passing through the system all the ITrails objects have to be checked for matches. If there only two or three sequences this will not make much of an impact of the speed of the alignment but if a large number of sequences are being aligned the impact will be much more noticeable. The second option avoids the necessity to constantly check every ITrails object. In this implementation not only does each sequence have an associated ITrails object but an additional 'Consensus' ITrails object is also created for the alignment. Ants

generated by any of the sequences have both read and write access to the consensus ITrails object and only have write access to their ITrails object. This means the determination of possible paths to follow involves the consultation of a single ITrails object, which should have less of an impact on the speed of the alignment. An additional advantage of this implementation is that at the end of the alignment is possible to generate a consensus sequence. This could have particular use for the searching of sequence databases for related protein sequences. It could also be used to generate regular expressions (or motifs) that can be used to define the protein family the aligned sequences represent.

It was decided that option 2 provided not only the more efficient method but that the consensus data could also be useful itself. Given additional time to develop the system, implementing option 1 as well would have provided an interesting comparison.

The determining of the path an Ant should take is achieved in three parts by the AntSystem class. Firstly the sequence carried by the ant is used to generate a number of matches to the previous pheromone trails. This is handled by NaiveTrails. The second and third parts and handled by the AnySystem class itself. These are the generation of the probabilities associated with the matched trails and the determination of the path taken. The probability of an ant taking a particular path is generated based on the strength of that path compared to all the other paths found. It is given by:

(Trail strength / sum of all trail strengths) * range

Where range is 100 – the chance of a random alignment, which is defined by the user in the initialisation file.

The value generated by this is summed with those that have been generated before it. This gives a list of values between 0 and 100 (- random alignment chance). The next step is to determine which path the ant will take. This is achieved by generating a random number between 0-100. This value is then used compared to the list of values generated earlier to determine which path the ant should follow. If the value is not covered by any of the matched

trails then the ant will take a random path (within an area determined by the value of the drift variable).

A number of minor modifications were required to increase the efficiency of this approach. Firstly a number of the alignments generated negative intensities. This resulted in a loss of efficiency since these pheromone trails were still used during the trail matching process but are of no use for further alignments since they represent inappropriate alignments. The algorithm was modified to discard them prior to inserting them into the ITrails objects. This resulted in a noticeable increase in the speed of alignments. A modification was also made to the intensity calculation. Initially the intensity was given by simply consulting the scoring matrix and summing the scores of all substitutions made. This meant that longer sequences could potentially gain much higher trail intensity scores. Therefore the method was modified to take into account length.

It was noticed during initial test runs that on occasion the sequence generated for output did not match that of the sequence used to generate it. This was determined to be due to the way in which the system works. By splitting the protein sequence into a number of smaller sequences during ant creation there was a chance that the ant may not align itself correctly. It was originally hope that constraining the amount the ants were allowed to drift would avoided this. However, this did not seem to be the case so it was decided that some level of check was required. Checking every trail during every cycle through the system would result in a large increase in the time required for the alignment to complete so I decided that checking should only occur every 10 cycles. Since the system does not modify trails for sequences shorter than the current ant payload length it was decided that it was not necessary to re-check trails for sequences shorter than the current ant payload length every time the check routine is run. Therefore the checking routines in the class SequenceChecker check only those PTrails that are for sequences of the current ant payload length. Any rogue PTrails found that do not match up with the original sequence are removed from the system. This routine essentially provides the system with a way to compare the pheromone trails that are being created back to the sequence that they were created from. In doing so the system became to some extent self-regulating.

# APPENDICES

## 1 THE DEFAULT PACKAGE

### 1.1 ANT ALIGN:-

```
import java.io.*;
import java.util.*;
import shared.*;
import protein.*;
import antSystem.AntSystem;
public class AntAlign {
        private static Configuration config;
        private static AminoAcids aas;
        private static ScoringMatrix mat;
        private static AntSystem as;
        //runParams
        private static ArrayList sequences;
        private static File outputFile;
        //Variables
        private static Hashtable vars;
public static void main(String[] args) {
                if (args.length != 1){
                        System.out.println("Invalid Parameters");
                        useMsg();
                        System.exit(1);
                }else{
                        if (args[0].equals("/?") || args[0].equals("-?")){
                                useMsg();
                                System.exit(0);
```

```
        }else{
            try {
                config = new Configuration(args[0]);
            } catch(IOException e) {
                System.out.println(args[0] +" is not a valid config file");
                System.out.println(e.toString());
                useMsg();
                System.exit(1);
            }
            System.out.println("AntAlign 1.0");
            System.out.println("^^^^^^^^^^^^");
            System.out.println("Loading Datafiles...");
            loadDataFiles();
            System.out.println("Loading Sequences...");
            sequences = loadSequences();
            System.out.println("Loading Parameters...");
            vars = loadParams();
            outputFile = loadOutputFile();
            System.out.println("Creating AntSystem...");
            as = new AntSystem(aas, mat, sequences, vars, outputFile);
            System.out.println("Running...");
            as.run();
            System.out.println("Ant Alignment Complete");
        }
    }
}
```

```
    private static void useMsg(){
        System.out.println("useage: AntAlign <config file>");
    }
    private static void loadDataFiles(){
            String aaFile = config.getItem("AntAlign","AAFile");
        if (aaFile != null){
            try {
                aas = AminoAcidsFactory.createAminoAcids(new
File(aaFile));
            } catch(AminoParseException e) {
                System.out.println("Invalid AmioAcids file");
                System.exit(1);
            } catch(FileNotFoundException ex){
                System.out.println(aaFile + " could not be found");
                System.exit(1);
            }
        }else{
            System.out.println("Config does not contain location of AminoAcids
file");
            System.exit(1);
        }
        String matrixFile = config.getItem("AntAlign","ScoreMatrix");
        if (matrixFile != null){
            try {
                mat = ScoringMatrixFactory.createScoringMatrix(new
File(matrixFile));
```

```
                    } catch(MatrixParseException e) {
System.out.println("Invalid matrix file");
                        System.exit(1);
                    } catch(FileNotFoundException ex){
                        System.out.println(matrixFile + " could not be found");
                    }
                }else{
                    System.out.println("Config does not contain location of ScoringMatrix
file");
                    System.exit(1);
                }
        }
        private static ArrayList loadSequences(){
            ArrayList ret = new ArrayList();
            String sTemp = "";
            String dTemp = "";
            File fTemp;
            String seqTemp;
            BufferedReader sIn;
            int c = 0;
            while (dTemp != null){
                c++;
                dTemp = config.getItem("RunParams","S"+c);
                if (dTemp != null){
                    fTemp = new File(dTemp);
                    try {
                        sIn = new BufferedReader(new FileReader(fTemp));
```

```
                    seqTemp = "";
                    sTemp = "";
                    Sequence seq;
                    while(sTemp != null){
                        try {
                            sTemp = sIn.readLine();
                        } catch(IOException e) {
                            System.out.println("Error reading
"+fTemp);
                            System.exit(1);
                        }
                        if (sTemp != null){
                            seqTemp += sTemp.trim();
                        }
                    }
                    try {
                        seq = new Sequence(aas, seqTemp);
                        ret.add(seq);
                    } catch(InvalidSequenceException e) {
                        System.out.println(fTemp+" does not contain a
valid sequence");
                        System.exit(1);
                    }
                } catch(FileNotFoundException e) {
                    System.out.println(fTemp+" not found");
                    System.exit(1);
                }
```

```
                }
            }
            return ret;
        }
        private static Hashtable loadParams(){
            Hashtable ret = new Hashtable(6);
            try {
                //parse variables
                Integer cycles = new Integer(config.getItem("Variables","Cycles"));
                Integer popPerTime = new
Integer(config.getItem("Variables","PopPerTime"));
                Integer startLen = new
Integer(config.getItem("Variables","StartLen"));
                Integer endLen = new Integer(config.getItem("Variables","EndLen"));
                Float evap = new Float(config.getItem("Variables","Evap"));
                Integer intensity = new
Integer(config.getItem("Variables","Intensity"));
                Integer drift = new Integer(config.getItem("Variables","Drift"));
                Integer rndChance = new
Integer(config.getItem("Variables","RndChance"));
                //put variables in hastable
                ret.put("cycles",cycles);
                ret.put("popPerTime",popPerTime);
                ret.put("startLen",startLen);
                ret.put("endLen",endLen);
                ret.put("evap",evap);
                ret.put("intensity",intensity);
                ret.put("drift",drift);
```

```
                ret.put("rndChance", rndChance);
            } catch(NumberFormatException e) {
                System.err.println("Unable to parse params - Number Format error");
                e.printStackTrace(System.err);
                System.exit(1);
            }
            return ret;
        }
        private static File loadOutputFile(){
            File ret;
            String filePath = config.getItem("RunParams","Output");
            if (filePath == null){
                filePath="output.html";
            }
            ret = new File(filePath);
            return ret;
        }
}
```

ANT SYSTEM PACKAGE

ANT:-

```
package antSystem;
import protein.Sequence;
public class Ant {
        private Sequence cargo;
        private int offset;
```

```
public Ant(Sequence cargo, int offset) {
                this.cargo = cargo;
                this.offset = offset;
        }
public Sequence getCargo() {
                return cargo;
        }
        public int getOffset(){
                return offset;
        }
}
```

MATCHED TRAIL:-

```
package antSystem;
public class MatchedTrail {
         private int offset;
         private int strength;
         private int probability;
 public MatchedTrail(int offset, int strength){
                this.offset = offset;
                this.strength = strength;
                probability = 0;
         }
         public int getOffset(){
                return offset;
         }
         public int getStrength(){
```

```
            return strength;
        }
        public void setProbability(int prob){
            probability = prob;
        }
public int getProbability(){
            return probability;
        }
```

P TRAIL COMP:-

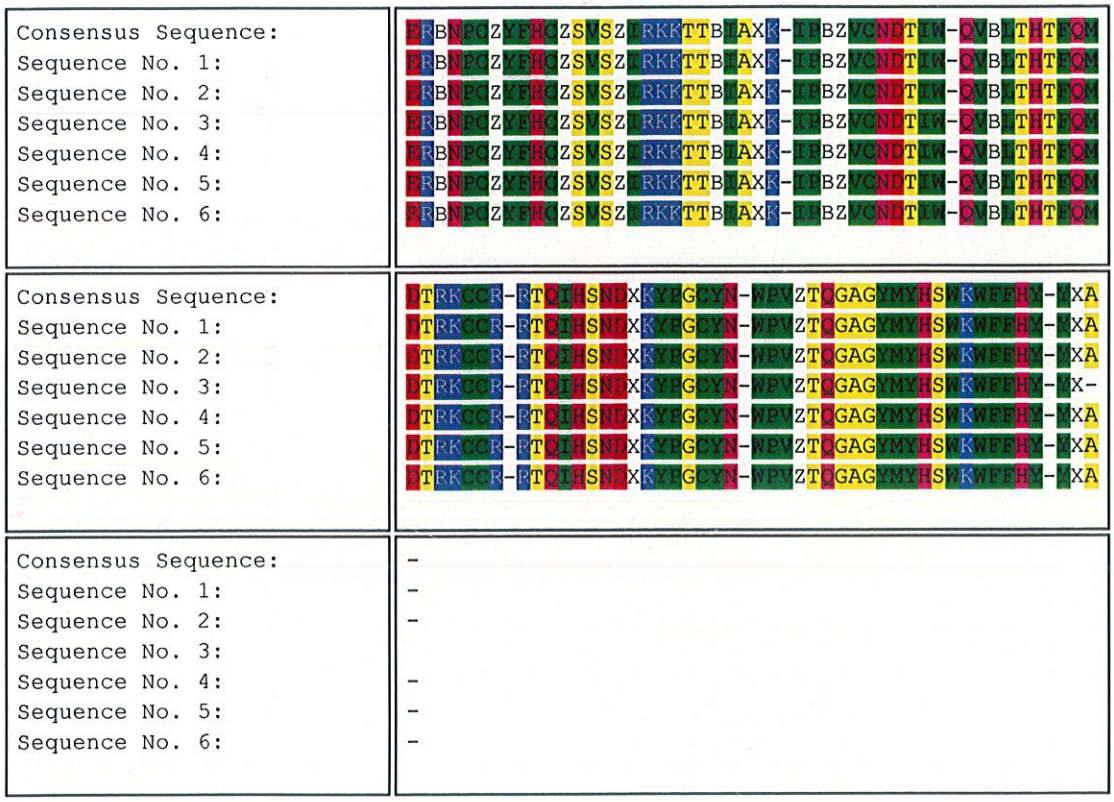
```
package antSystem;
import java.util.*;
public class PTrailComp implements Comparator{
        public int compare(Object o1, Object o2) throws ClassCastException{
            int ret = 0;
            PTrail pt1 = (PTrail)o1;
            PTrail pt2 = (PTrail)o2;
            int pt1Off = pt1.getOffset();
            int pt2Off = pt2.getOffset();
            ret = pt1Off - pt2Off;
            return ret;
        }

}
```

# RESULTS

## 5.1 Identical Sequences

The first test was performed with six identical sequences. The sequence used was test1. Figure shows the results:

**Alignment:**




**Stats:**

**Score** = 5.0 **Hits** = 24342 **Misses** = 35658

**Time Elapsed:** 305.77 secs

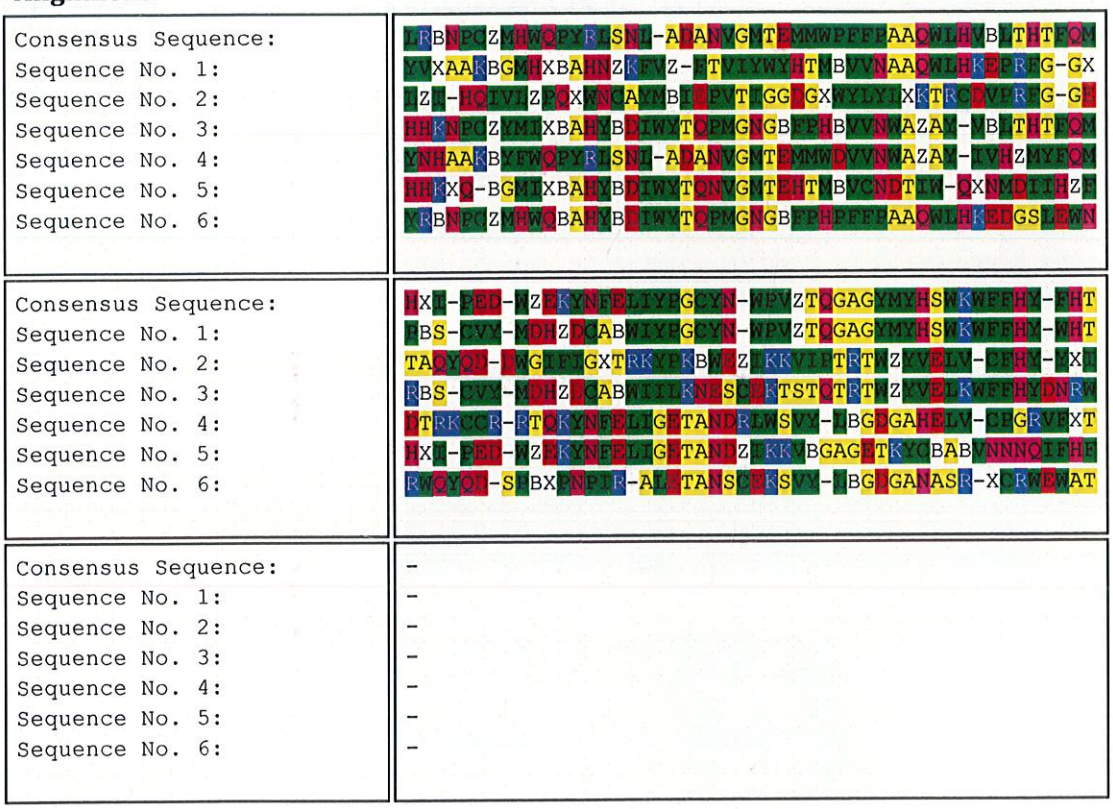
The alignment shown in figure 4.2.1.1 was performed with 6 copies of the same protein sequence (test1). The purpose of this test was to determine whether the system was working as expected and also to check that the system was capable of re-constructing the original sequence purely from the pheromone trails created. The results show clearly that the system is functioning as expected. It also shows that the sequences constructed are identical to the ones used as input except for sequences 3 in which the final amino acid is missing. The score associated with this alignment is 5.0, a positive score indicates that some similarity is present in the aligned sequences. The more positive the score

the higher the homology. A score of 5.0 indicates a very high. This test was essentially a proof of concept to ensure that the ACO algorithm used was capable of performing alignments.

### 5.2 Unrelated Sequences

The second test was performed with six different randomly generated sequences. The sequences used were test1 – test6. Figure below shows the results:

**Alignment:**




**Stats:**

**Score** = 0.0 **Hits** = 25348 **Misses** = 34652

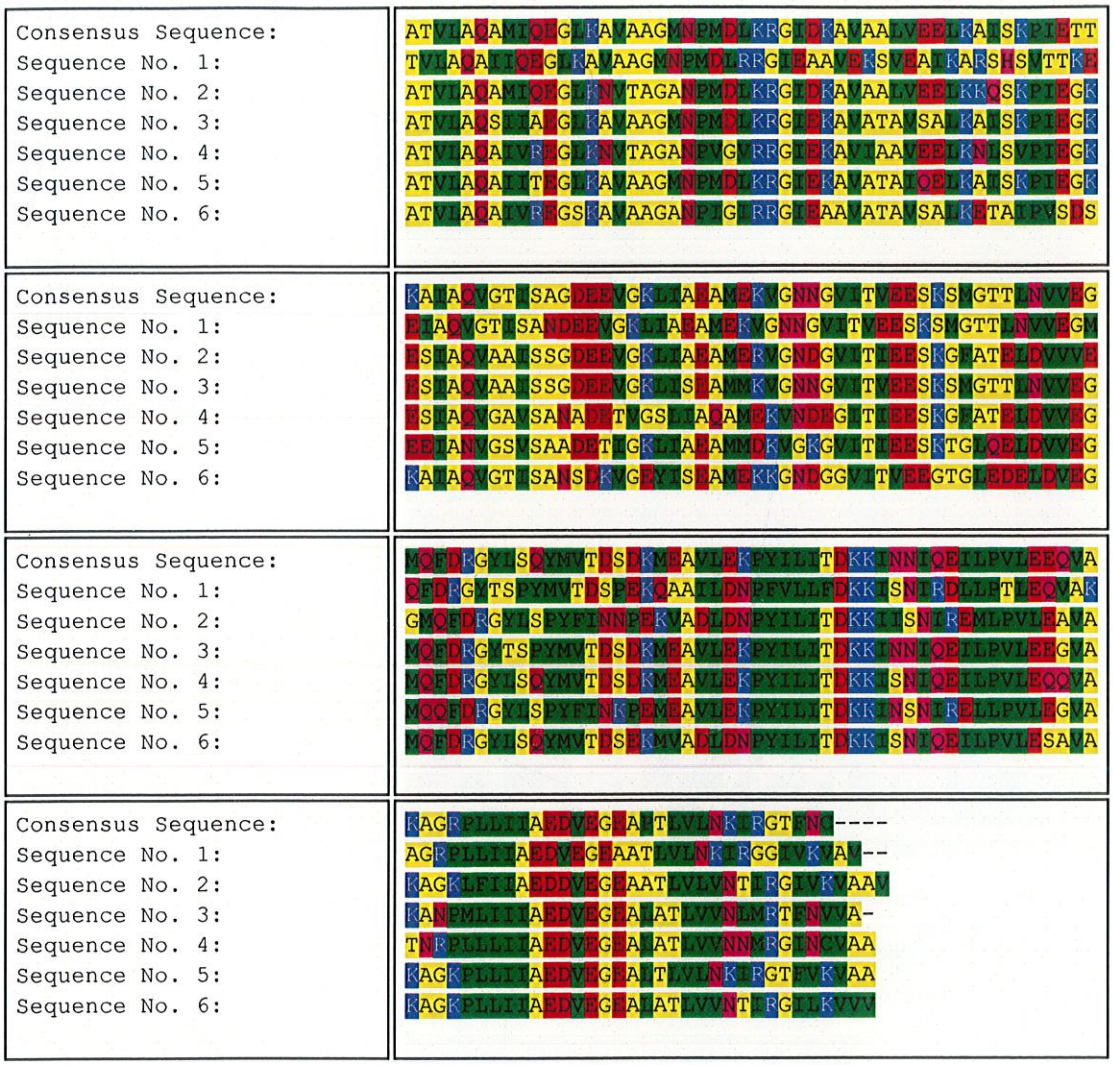
**Time Elapsed:** 835.36 secs

This test was designed to ensure that the system did not provide good alignments for sequences that bare little homology with each other. The results do produce a score of 0.0, which indicates that there was little to no homology between the sequences over their entire length. However, there does appear to be several areas of homology between pairs of sequences. On inspection of the original sequences, this homology does not seem to exist. It would seem that the despite the actions of the sequence checker the resulting sequences do not always match exactly the sequence from which they are derived. We would expect this to be increasingly more common as the sequences

being aligned share less homology. Since the sequences used in this test show little to no homology, these results should represent a worst case scenario.

### 5.3 Real Data Alignment (Highly conserved)

**Alignment:**




**Stats:**

**Score** = 1.0 **Hits** = 21273 **Misses** = 38727

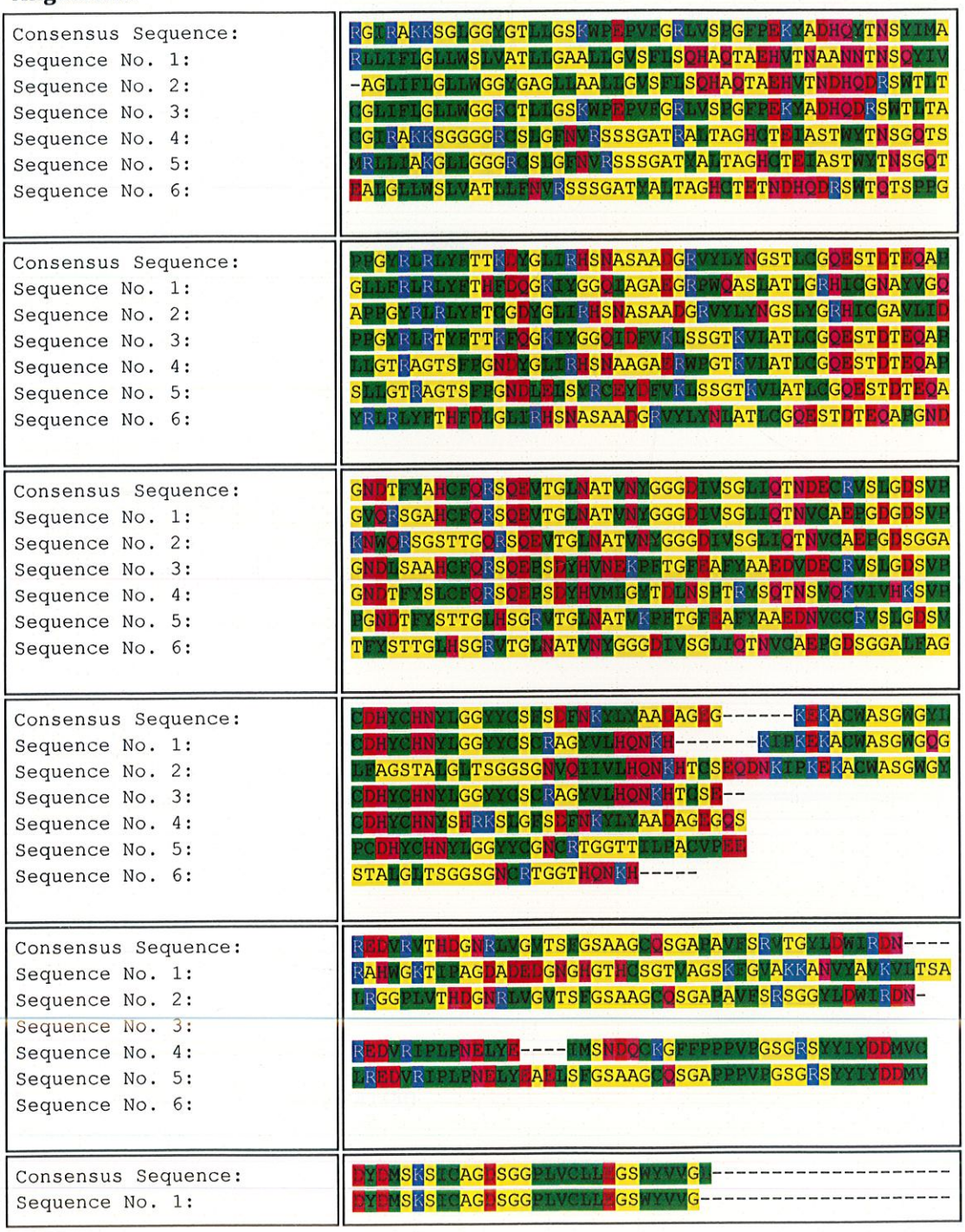
**Time Elapsed:** 1016.45 secs

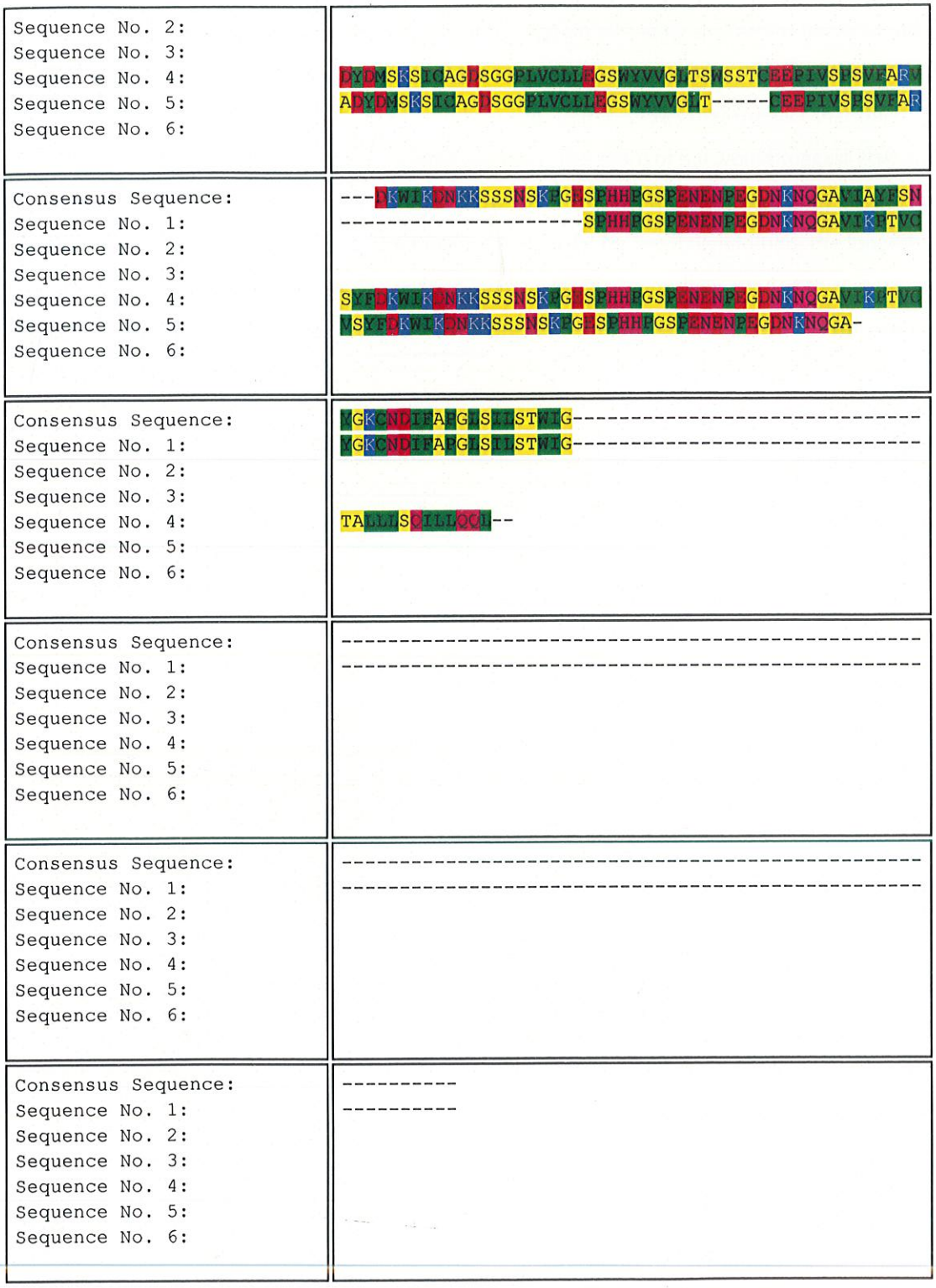
This test was designed to see how well the system performed when dealing with real data. The sequences used for this test are highly conserved and show substantial homology. The score associated with this test was 1.0, which show that some degree of homology was present between the sequences. This is also highlighted by the sequences themselves. There are several regions where it can be clearly seen that the sequences are either identical in terms of their amino acid

constitution or the properties of the amino acids at those points. Again, the sequences constructed from the pheromone trail are not identical to those from which the ants were generated. There also seems to be some shifting of sequences where with the insertion of a gap a better alignment may have been achieved.

### 5.4 Real Data Alignment (Not so highly conserved)

**Alignment:**






**Stats:**

**Score** = 0.0 **Hits** = 18570 **Misses** = 41430

**Time Elapsed:** 955.49 secs

This test was designed to examine the efficacy of the system at finding similarities between sparingly similar sequences. As can be seen it produced the most unexpected results. This was also the only test in which the input sequence differed greatly in size. Test-2 shows that the fact that the sequences were of differing sizes was handled correctly. The score of 0.0 would suggest little to no homology is present. However, the results do show some areas of localised homology. Again, the sequences generated are not exactly the same as those used for input. In addition, the longest sequence has not been completed. There are a large number of gaps at the end of the sequence suggesting that pheromone trails created for this area did not persist until the end of the run.

## 6. REFERENCES


1. Lesk, A. M., *Introduction to Bioinformatics*, Oxford University Press, Oxford, 2002
2. Attwood, T.K. and Parry-Smith, D.J., *Introduction to Bioinformatics*, Addison Wesley Longman ltd., Essex, 1999
3. Zubay, G.L., *Biochemistry ($4^{th}$ Ed.),* McGraw-Hill, IA (USA), 1998
4. Stryer, L., *Biochemistry ($4^{th}$ Ed.),* W.H.Freeman and Company, NY (USA), 1996
5. Voet, D. and Voet, J.G., *Biochemistry ($2^{nd}$ Ed.),* John Wiley and Sons Inc., NY (USA), 1995
6. Needlman, S.B., and Wunsch, C.D., A general method applicable to the search for similarities in the amino acid sequence of two proteins. , 1998, *Journal of Molecular Biology*, 48, pp. 443-453
7. Smith, T.F. and Waterman, M.S., Identification of common molecular sub-sequences., 1981, *Journal of Molecular Biology*, 147, pp. 195-197
8. Higgins D., Thompson J., Gibson T.Thompson J.D., Higgins D.G. and Gibson T.J., CLUSTAL W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice. , 1994, *Nucleic Acids Research*, 22, pp. 4673-4680.
9. Eddy S.R., Hidden Markov models, 1996, *Current Opinions in Structural Biology*, 6, 361-365.
10. Dorigo M., Di Caro, G. and Gambardella, L. M., Ant Algorithms for Discrete Optimization, 1999, *Artificial Life*, 5, (2), pp. 137-172.
11. Colorni A., Dorigo, M., Maffioli, F., Maniezzo, V., Righini, G. and Trubian, M., Heuristics from Nature for Hard Combinatorial Problems, 1996, *International Transactions in Operational Research*, 3, (1), pp. 1-21.
12. Dorigo M., Maniezzo, V. and Colorni, A. The Ant System: Optimization by a Colony of Cooperating Agents, 1996, *IEEE Transactions on Systems, Man, and Cybernetics-Part B,* 26, (1), pp. 29-41
13. Dorigo M. and Gambardella, L. M., Ant Colonies for the Travelling Salesman Problem, 1997, *BioSystems*, 43, pp. 73-81.
14. Di Caro, G. and Dorigo, M., Mobile Agents for Adaptive Routing, 1998, *Proceedings of the 31st Hawaii International Conference on System, IEEE Computer Society Press,* Los Alamitos, CA, pp. 74-83.
15. Di Caro G. and Dorigo M., AntNet: Distributed Stigmergetic Control for Communications Networks, 1998, *Journal of Artificial Intelligence Research (JAIR)*, 9, pp. 317-365.

16. Navarro Varela, G. and Sinclair, M.C., Ant Colony Optimisation for Virtual-Wavelength-Path Routing and Wavelength Allocation, 1999, *Proceedings of the Congress on Evolutionary Computation (CEC'99)*, Washington DC, USA.
17. Fitzgerald, R., Knoblock, T.B., Ruf, E., Steensgaard, B. and Tarditi, D., Marmot: An Optimizing Compiler for Java, 200, *Software - Practice and Experience*, 30, (3), pp. 199-232.
18. Dawson, R.M.C., Elliot, D.C., Elliot, W.H. and Jones, K.M., *Data for Biochemical Research (3rd ed.)*, pp. 1-31, Oxford Science Publications, 1986.
19. Engelman, D.M., Steitz, T.A. and Goldman, A., Identifying non-polar transbilayer helices in amino acid sequences of membrane proteins, 1986, *Annual Review Biophysics and Biophysical Chemistry*, 15, pp. 321-353.
20. Hopp, T.P. and Woods, K.R., Prediction of protein antigenic determinants from amino acid sequences, 1981, *Proceeding of the National Academy of Science. USA*, 78, (6), pp. 3824-3828.
21. Kyte, J. and Doolittle, R.F., A Simple Method for Displaying the Hydropathic Character of a Protein, 1982, *Journal of Molecular Biology*, 157, (6), pp. 105-142.
22. Altschul, S.F. Amino acid substitution matrices from an information theoretic perspective, 1991, *Journal of Molecular Biology,* 219, pp. 555-565.
23. Altschul, S.F., A protein alignment scoring system sensitive at all evolutionary distances, 1993, *Journal of Molecular Evolution,* 36, pp. 290-300.