Artificial Intelligence Techniques for Bioinformatics

A. Narayanan, E.C. Keedwell* and B. Olsson***

*School of Engineering and Computer Sciences, University of Exeter, Exeter EX4 4QF, UK.
**Department of Computer Science, University of Skövde, Box 408, 54128 Skövde, Sweden.

Abstract

This review article aims to provide an overview of the ways in which techniques from artificial intelligence can be usefully employed in bioinformatics, both for modelling biological data and for making new discoveries. The paper covers three techniques: symbolic machine learning approaches (nearest-neighbour and identification tree techniques); artificial neural networks; and genetic algorithms. Each technique is introduced and then supported with examples taken from the bioinformatics literature. These examples include folding prediction, viral protease cleavage prediction, classification, multiple sequence alignment and microarray gene expression analysis.

Author contact details:

A.Narayanan, hard mail address as above. **A.Narayanan@ex.ac.uk**.. Tel: (+UK)1392 264064. Fax: (+UK)1392 264067.

E.C. Keedwell, hard mail address as above. **E.C.Keedwell@ex.ac.uk**. Tel: same as above.

B. Olsson, hard mail address as above. **bjorn.olsson@ida.his.se**. Tel: (+Sweden) 500-44 83 16. Fax: (+Sweden) 500-44 83 99.

Keywords: Artificial intelligence; machine learning; bioinformatics; neural networks; genetic algorithms; data mining.

Please note that this article has been accepted for publication in Applied Bioinformatics.

Artificial Intelligence Techniques for Bioinformatics

A. Narayanan[*], E.C. Keedwell[*] and B. Olsson[**]

[*]School of Engineering and Computer Sciences, University of Exeter, Exeter EX4 4QF, UK.
[**]Department of Computer Science, University of Skövde, Box 408, 54128 Skövde, Sweden.

Abstract

This review article aims to provide an overview of the ways in which techniques from artificial intelligence can be usefully employed in bioinformatics, both for modelling biological data and for making new discoveries. The paper covers three techniques: symbolic machine learning approaches (nearest-neighbour and identification tree techniques); artificial neural networks; and genetic algorithms. Each technique is introduced and then supported with examples taken from the bioinformatics literature. These examples include folding prediction, viral protease cleavage prediction, classification, multiple sequence alignment and microarray gene expression analysis.

1. Introduction

There is growing interest in the application of artificial intelligence (AI) techniques in bioinformatics. In particular, there is an appreciation that many of the problems in bioinformatics need a new way of being addressed given either the intractability of current approaches or the lack of an informed and intelligent way to exploit biological data. For an instance of the latter, there is an urgent need to identify new methods for extracting gene and protein networks from the rapidly proliferating gene expression and proteomic datasets. There is currently very little understanding of how standard techniques of analysing gene expression and proteomic data, such as clustering, correlation identification and self organising maps, can contribute directly to the 'reverse engineering' of gene networks or metabolic pathways. Instead, such techniques aid in the first analysis of datasets which are typically very large (thousands of genes can be measured on one microarray), requiring further biological knowledge from human experts for the identification of appropriate and relevant causal relationships between two or more genes or proteins. There is a need to merge biological knowledge with computational techniques for extracting relevant and appropriate genes from the thousands of genes measured. For an instance of the former, predicting the way a protein folds from first principles may well be feasible given some algorithms for protein sequences of 20 or so amino acids, but once the sequences become biologically plausible (200 or 300 amino acids and more) current protein folding algorithms which work on first principles rapidly become intractable.

On the other hand, artificial intelligence is an area of computer science that has been around since the 1950s, specialising in dealing with problems considered intractable by computer scientists through the use of heuristics and probabilistic approaches. AI approaches excel when dealing with problems where there is no requirement for 'the absolutely provably correct or best' answer (a 'strong' constraint) but where, rather, the requirement is for an answer which is better than one currently known or which is acceptable within certain defined constraints (a 'weak' constraint). Given that many problems in bioinformatics do not have a strong constraints, there is plenty of scope for the application of AI techniques to a number of bioinformatics problems.

What is interesting is that, despite the apparent suitability of AI techniques for bioinformatics problems, there is actually very little published on such applications when one considers the vast and increasing amount of published papers in bioinformatics. The aim of this review paper is not to identify all previous work in bioinformatics that has involved AI techniques. This is because there will be some AI techniques which have not yet been applied to bioinformatics problems and a review which covers previous work would therefore be narrow. Instead, the aim of this paper is to introduce bioinformatics researchers to three particular AI techniques, two of which may be known (neural networks and symbolic machine learning) and one of which may not be so well known (genetic algorithms). One of the intriguing aspects of the last technique is the rather philosophically appealing idea of applying techniques from AI which have been influenced by developments in our understanding of biological phenomena to biological problems themselves.

The paper consists of three parts. First, classical symbolic machine learning techniques will be introduced (nearest neighbour and identification tree approaches), including examples of bioinformatics applications in secondary protein structure prediction and viral protease cleavability. Next, neural networks will be introduced, with a number of applications described. Third, genetic algorithms will be covered, with applications in multiple sequence alignment and RNA folding prediction.

2. Symbolic machine learning

2.1 Nearest neighbour approaches

We first introduce decision trees. A decision tree has the following properties: each node is connected to a set of possible answers; each non-leaf node is connected to a test which splits its set of possible answers into subsets corresponding to different test results; and each branch carries a particular test result's subset to another node.

2.1.1 Introduction

To see how decision trees are useful for nearest neighbour calculations, consider 8 blocks of known width, height and colour (Winston, 1992). A new block then appears of known size but unknown colour. On the basis of existing information, can we make an informed guess as to what the colour of the new block is? To answer this question, we need to assume a *consistency heuristic*, as follows. Find the most similar case, as measured by known properties, for which the property is known; then guess that the unknown property is the same as the known property. This is the basis of all *nearest neighbour* calculations. Although such nearest neighbour calculations can be performed by keeping all samples in memory and calculating the nearest neighbour of a new sample only when required by comparing the new sample with previously stored samples, there are advantages in storing the information about how to calculate nearest neighbours in the form of a decision tree, as will be seen later.

For our example problem above, we first need to calculate, for the 8 blocks of known size and colour, a decision space using width and height only (since these are the known properties of the ninth block), where each of the 8 blocks is located as a point within its own unique region of space. Once the 8 blocks have been assigned an unique region, we then calculate, for the ninth block of known width and height, a point in the same feature space. Then, depending on what the colour of its 'nearest neighbour' is in the region it occupies, we allocate the colour of the nearest neighbour to the ninth block. Notice that the problem consisted of asking for an 'informed guess', not a provably correct answer. That is, in many applications it is important to attribute a property of some sort to an object when the object's real property is not known. Rather than having to leave the object out of consideration, an attribution of a property to the object with unknown property may be useful and desirable, even if it cannot be proved that the attributed property is the real property. At least, with nearest neighbour calculations, there is some systematicity (the consistency heuristic) in the way that an unknown property is attributed.

So, for our problem above, we shall divide up the 8 blocks in advance of nearest neighbour calculation (i.e. before we calculate the nearest neighbour of the ninth block). To do this, we divide the 8 blocks by height followed by width, then height, width ... until only one block remains in each set. We ensure that we divide so that an equal number of cases falls on either side. The eight blocks with known colour are first placed on the feature space, using their width and height measures as co-ordinates (Figure 1 (a)). The ninth object (width 1, height 4) is also located in this feature space, but it may not be clear what its nearest neighbour is (i.e. which region of space it occupies), that is, the object could be orange or red. To decide which, the 8 blocks of known size are first divided into two equal subsets using, say, the height attribute. The tallest of the shorter subset has height 2 and the shortest of the taller subset has height 5. The midpoint 3.5 is therefore chosen as the dividing line. The next step is to use the second attribute, width, to separate each of the two subsets from Figure 1(b) into further subsets (Figure 1 (c)). For the shorter subset, the wider of the two narrow blocks is 2 and the narrower of the two wide

blocks is 4. The midpoint is therefore 3. For the taller subset, a similar form of reasoning leads to the midpoint being 3.5. Note that the two subsets have different midpoints. Since each block does not occupy its own region of space yet, we return to height (since there are only two known attributes for each object) and split each of the four subsets once more (Figure 1 (d)). For the two taller subsets, the midpoints are both, coincidentally, 5.5 (between 5 and 6). For the two shorter subsets, the midpoints are both, coincidentally, 1.5. Each block now has its own region of space.

Once we have divided up the cases, we can then generate a decision tree, using the mid-points discovered as test nodes (Figure 1 (e)) in the order in which they were found. Once we have the tree, we can then trace a path down the tree for the ninth block, following the appropriate paths depending on the outcome of each test, and allocate a colour to this block (orange). While this conclusion may have been reached through a visual inspection of the feature space alone (Figure 1 (a)), in most cases we will be dealing with a multi-dimensional feature space, so some systematic method for calculating nearest neighbours will be required which takes into account many more than two dimensions. Also, nearest neighbour techniques provide only approximations as to what the missing values may be. Any data entered into a database system, or any inferences drawn from data obtained from nearest neighbour calculations, should always be flagged to ensure that these approximations are not taken to have the same status as 'facts'. New information may come in later which requires these approximations to be deleted and replaced with real data.

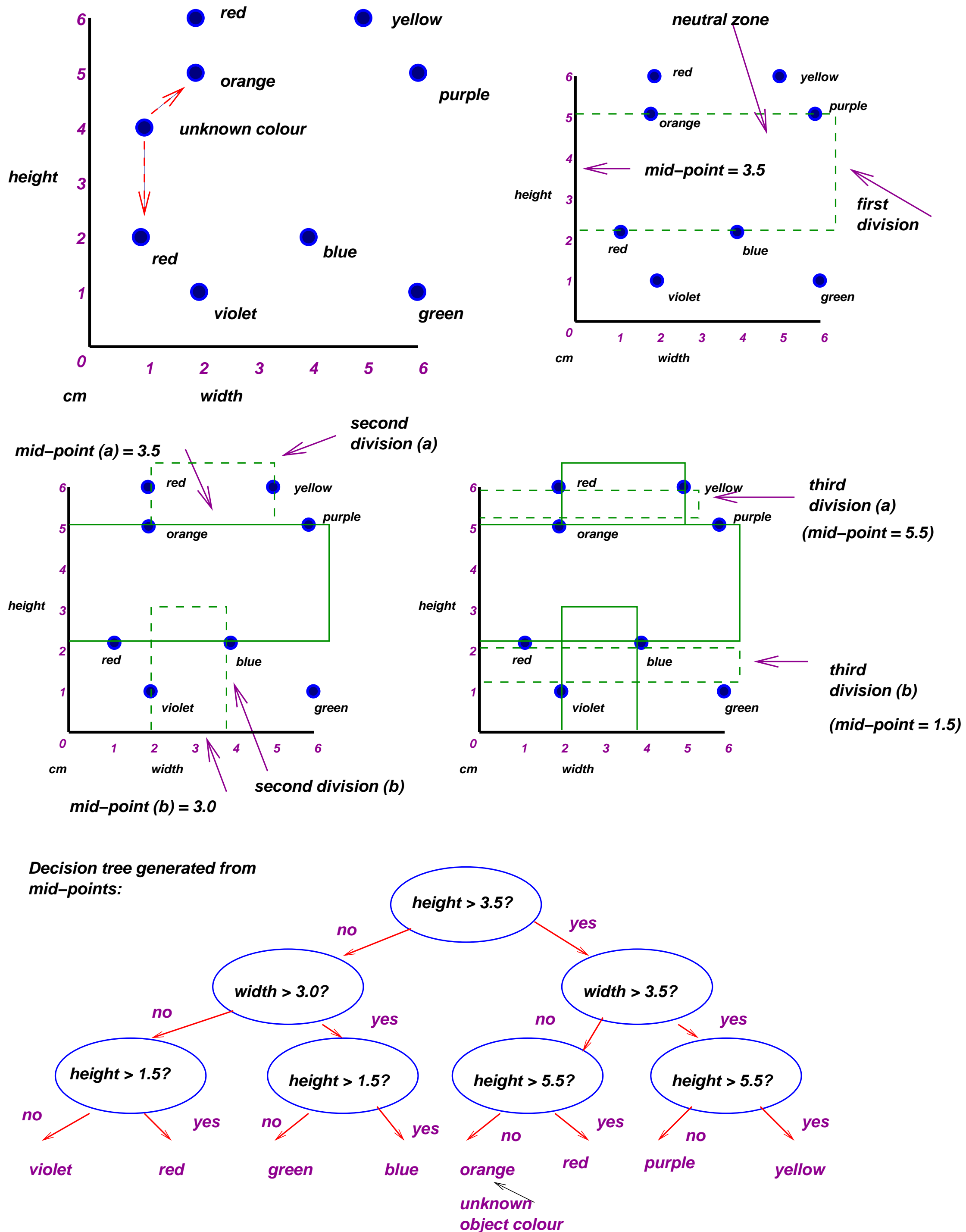


Figure 1: Nearest neighbour calculations and the resulting decision tree. See text for an explanation of (a), (b), (c), (d) and (e). Once the decision tree is constructed (e), we can use the known attributes of the object with unknown colour to allocate a colour to this object based on its nearest neighbour, which is orange. In general, a decision tree with branching factor 2 and depth d will have 2^d leaves; if n objects are to be identified, d must be large enough to ensure $2^d \geq n$. This example is adapted from Winston (1992).

We could of course leave calculating the nearest neighbour of a new sample until the new sample enters the system. But if there are thousands of samples and hundreds of attributes, calculating the nearest neighbour each time a new sample arrives with unknown properties may prove to be a bottleneck for database entry. Nearest neighbour calculations can be used to infer the missing values of database attributes, for instance, and if we are dealing with real-time databases the overheads in calculating the nearest neighbour of new records with missing values could be a real problem. Instead, it is more efficient to compute decision trees for a number of attributes which are known to be 'noisy' in advance of new records so that entering a new record is delayed only by the time it takes to traverse the relevant decision tree (as opposed to calculating the nearest neighbour from scratch). Also, once the decision tree is computed, the information as to why a new sample is categorised with a nearest neighbour is readily available in the decision tree. Analysing such trees can shed new light on the domain in question. A decision tree therefore represents knowledge about nearest neighbours.

One variation of nearest neighbour approaches is to use *k-nearest neighbour* classification. The process involves identifying, for a new object with unknown property P, the k closest objects based on some measure (usually Euclidean) between known properties of the new object and all other objects. Then depending on the p values of these other objects for P, assign to the new object the property p possessed by the majority of these other objects for P.

2.1.2 Nearest neighbour example in bioinformatics

Typically, bioinformatics researchers want to find the most similar biosequence to another biosequence, and such biosequences can contain hundreds and possibly thousands of 'attributes', i.e. positions, which are candidates for helping to identify similarities between biosequences. There will typically be many more attributes than sequences and therefore the choice of specific attributes to use as tests will be completely arbitrary and random. For this reason, nearest neighbour calculations usually take into account all the information in all positions before attributing a missing value. Some examples in the domains of clustering and secondary structure prediction will make this clear.

2.1.2.1 Clustering

Clustering follows the principles of nearest neighbour calculations but attempts to look at all the attributes (positions) of biosequences rather than just one attribute (position) for identifying similarities. This is achieved typically by averaging the amount of similarity between two biosequences across all attributes. For example, imagine that we have a table of information concerning four organisms with five characteristics (see Table 1).[1]

---

[1] This example is adapted from Prescott, L.M., Harley, J. P. and Klein, D. A. (1998) *Microbiology*. McGraw Hill, 1998. Some of the material is reviewed on-line at http://www2.ntu.ac.uk/life/sh/modules/hlx202/Lectures/Lectured.htm.

		Characteristics				
		1	2	3	4	5
Organism	A	+	+	-	-	-
	B	+	-	-	-	-
	C	-	-	+	-	-
	D	-	-	+	+	-

Table 1: A table describing four organisms and the characteristics they share or don't share. A '+' in a column signifies that the organism possesses that characteristic, whereas a '-' in a column signifies that the organism doesn't possess that characteristic.

Given this table, can we calculate how similar each organism is to every other organism? The nearest neighbour approach described earlier would work through the attributes ('characteristics') one at a time. For short biosequences this may be feasible, but for biosequences with hundreds of attributes (e.g. DNA bases) this is not desirable, since we could probably classify all the samples with just the first few attributes. The remaining attributes are not used, which somehow doesn't seem to take all the information into account when attempting to find sequences which are most similar to each other.

Clustering can be demonstrated in the following way. The first step is to calculate a simple matching coefficient for every pair of organisms in the table across all attributes. For instance, the matching coefficient for A and B is the number of identical characteristics divided by the total number of characteristics, i.e. 4/5 = 0.8 (1+0+1+1+1=4/5=0.8). Similarly, the matching coefficients for every other pair are as follows: A and C = 0.4 (0+0+0+1+1 = 2/5 = 0.4); A and D = 0.2 (0+0+0+0+1 = 1/5 = 0.2); B and C = 0.6 (0+1+0+1+1 = 3/5 = 0.6); B and D = 0.4 (0+1+0+0+1 = 2/5 = 0.4); and C and D = 0.8 (1+1+1+0+1 = 4/5 = 0.8).

We then find the first highest matching coefficient to form the first 'cluster' of similar bacteria. Since we have two candidates here (AB and CD both have 0.8), we randomly choose one cluster to start the process: AB. The steps are then repeated, using AB as one 'organism' and taking partial matches into account. So, for instance, the average matching coefficient for AB and C = 0.5 (0+0.5+0+1+1 = 2.5/5 = 0.5), where the 0.5 second match within the parentheses refers to C sharing its second feature with B but not A. The matching coefficients for AB and D = 0.3 (0+0.5+0+0+1 = 1.5/5 = 0.3) and for C and D = 0.8 (as before). Since C and D have the highest cooefficient, they form the second cluster.

Finally, we calculate the average matching coefficients for the new 'clusters' of organism taking AB as one organism and CD as another organism = 0.4 (0+0.5+0+0.5+1 = 2/5 = 0.4), again taking partial matches into account. We can then construct a similarity tree using these coefficients, as follows:

Note that the tree is 'on its side' and that the individual organisms and clusters of organisms are joined using the average matching coefficients. The tree indicates that the nearest neighbour of A is B, and *vice versa*, and that the nearest neighbour of C is D, and *vice versa* (0.8 average coefficient value in both cases). These two sets of neighbours are joined at 0.4. Each organism is located in its own leaf node in the tree. This form of hierarchical clustering is known as UPGMA (Unweighted Pair Group Method with Arithmetic Mean) (Michener and Sokal, 1957; Sneath and Sokal, 1973).[2] Real clustering examples will consist of biosequences with possibly hundreds of characteristics (attributes, positions), but the above examples demonstrates the principle of how each sequence can be linked to other sequences on the basis of similarity across all attributes (positions).

---

[2] UPGMA2, a FORTRAN program for clustering, is available from http://wwwvet.murdoch.edu.au/vetschl/imgad/UPGMAMan.htm.

2.1.2.2 Secondary structure prediction

Another form of nearest neighbour calculation in bioinformatics can be found in SIMPA (Levin, Robson and Garnier, 1986; Levin, 1997), a tool for predicting secondary protein structures. A simplified form of their approach is presented here. Table 2 contains a ‘similarity matrix’ which describes a score for the replacement of one amino acid by another.

G	2																			
P	0	3																		
D	0	0	2																	
E	0	-1	1	2																
A	0	-1	0	1	2															
N	0	0	1	0	0	3														
Q	0	0	0	1	0	1	2													
S	0	0	0	0	1	0	0	2												
T	0	0	0	0	0	0	0	0	2											
K	0	0	0	0	0	1	0	0	0	2										
R	0	0	0	0	0	0	0	0	0	1	2									
H	0	0	0	0	0	0	0	0	0	0	0	2								
V	-1	-1	-1	-1	0	-1	-1	-1	0	-1	-1	-1	2							
I	-1	-1	-1	-1	0	-1	-1	-1	0	-1	-1	-1	1	2						
M	-1	-1	-1	-1	0	-1	-1	-1	0	-1	-1	-1	0	0	2					
C	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2				
L	-1	-1	-1	-1	0	-1	-1	-1	0	-1	-1	-1	1	0	2	0	2			
F	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	0	1	0	-1	0	2		
Y	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	0	0	0	0	-1	0	1	2	
W	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	0	-1	0	0	0	-1	0	0	0	2
	G	**P**	**D**	**E**	**A**	**N**	**Q**	**S**	**T**	**K**	**R**	**H**	**V**	**I**	**M**	**C**	**L**	**F**	**Y**	**W**

Table 2: A secondary structure similarity matrix (adapted from Levin, Robson and Garnier (1986)). The values in the table reflect the importance of the substitution of one amino acid by another in the formation of secondary structure.

Imagine we have the short amino acid sequence STNGIYW, the secondary structure of which is unknown. Imagine also that we have three homologues, with secondary structures, as follows:

h	***h***	***s***	***s***	***s***	***s***	***c***	***h***	***s***	***c***	***c***	***c***	***c***	***c***	***h***	***h***	***s***	***s***	***c***	***c***	***c***
A	T	S	L	V	F	W	S	T	S	G	V	V	W	S	C	N	G	A	F	W

For example, the sequence ATSLVFW has the secondary structure ***h h s s s s c***, where ***h*** stands for ‘helix’, ***s*** for ‘sheet’ and ***c*** for ‘coil’. (Levin, Robson and Garnier (1986) in fact use eight secondary structure conformations rather than three.) The question now is whether we can predict the secondary structure of STNGIYW from our knowledge of the structure of these three homologues and their similarity to the sequence with unknown structure. SIMPA’s method is as follows. First, we complete a *conformation matrix* where we first compare the test sequence’s residues against the residues of the three homologues, residue by residue, and calculate an overall score for the test sequence. That is, for the test strand STNGIYW, we first calculate how far away ‘S’ (the start residue in the test strand) is from the start residues in the homologues, how far the second residue in the test strand is from the second residues in the homologues, etc, and sum up for each strand. To help in the construction of this conformation matrix, the distance values in Table 2 are used.

So, for example, the similarity between STNGIYW and ATSLVFW = 1+2+0-1+1+1+2 = 6. That is, the similarity between S and A = 1, T and T = 2, N and S = 0, G and L = -1, I and V = 1, Y and F = 1, and W and W = 2. We calculate the scores for STNGIYW and the other two homologues also, giving us 9 for STSGVVW (2+2+0+2+1+0+2) and 9 (2+0+2+2+0+1+2) for SCNGAFW. Next, we allocate these scores in a conformation prediction table for each residue:

	h	***s***	***c***
Residue 1	6+9+9		
Residue 2	6+9		9
Residue 3		6+9	9
Residue 4		6+9	9
Residue 5		6	9+9
Residue 6		6	9+9
Residue 7			6+9+9

That is, for residue 1, all three homologues have the ***h*** conformation for their first residue, so the three scores are inserted under the ***h*** column for this residue. However, for the second residue, the first and the third of the homologues have ***h*** whereas the second homologue has ***c***. The scores for the first and third homologues are inserted under the ***h*** column and the score for the second homologue is inserted under the ***c*** column. This process is continued for all the residues. Then, for each residue in the test strand STNGIYW, the conformation with the maximum score is allocated to that residue. Therefore, the test strand STNGIYW has predicted conformation ***hhssccc***. It should however be pointed out that SIMPA (Levin, 1997) uses a more complex version of this method, adopting, first, a threshold value of 7 before a score can be inserted into the conformation prediction table, and second, a 'moving window' whereby the calculations are repeated again, this time moving one residue along both the test and homologue strands. That is, with longer strands, the calculations are first made for residues 1 to 7 and predictions made, then for 2 to 8 with predictions made or revised, then for 3 to 9, and so on. Their algorithm therefore makes a comparison between every sequence of seven residues, moving along one residue at a time. Finally, additional weightings can be placed on the scores in the secondary structure similarity matrix to help guide the prediction process in different ways.

SIMPA is a form of nearest neighbour technique since it essentially attributes a conformation to a residue in the new sample on the basis of nearest neighbour residues with known conformations in homologues. In general, a nearest neighbour procedure predicts a class *c* on the basis of the nearest neighbour *n* to a query object *q. A* variation of this nearest neighbour procedure is *k*-nearest neighbours, whereby a number *k* of *n* nearest neighbours are used to determine the class *c*. For example, in the case of SIMPA, if two entries in a row have roughly the same value, SIMPA can return this information to the user so that the user can decide on the most plausible structure for that residue. Another application of the nearest neighbour procedure in bioinformatics can be found in Ankerst *et al.* (1999), where structural classification of proteins based on their 3D structures using a nearest neighbour approach is described.

2.2 Symbolic machine learning and classification

Nearest neighbour approaches do not try to rank the attributes or features used for classifying objects. Instead, all features and attributes are considered of equal weight. The *identification tree* approach, as described by Quinlan (1987) and Winston (1992), differs from the decision tree approach of nearest neighbour calculations by attempting to identify which subset of features from the feature set are the most important for classification.

2.2.1 Introduction

Consider the data of information in Table 3 (Winston, 1992).

Name	*Hair colour*	*Height*	*Weight*	*Lotion used*	*Result*
Sarah	Blond	average	light	no	sunburned
Dana	Blond	tall	average	yes	not sunburned
Alex	Brown	short	average	yes	not sunburned
Annie	Blond	short	average	no	sunburned
Emily	Red	average	heavy	no	sunburned
Pete	Brown	tall	heavy	no	not sunburned
John	Brown	average	average	no	not sunburned
Katie	Blond	short	light	yes	not sunburned

Table 3: A dataset of individuals describing who is and who is not sunburned. The table is adapted from Winston (1992).

The task now is to determine which of the attributes contribute towards someone being sunburned or not. Nearest neighbour approaches, with their use of decision trees, do not rank attributes, features or properties. However, *identification trees* do. First, we need to introduce a disorder formula and associated log values to rank attributes in terms of their influence on who is and who isn't sunburned.

$$\text{Average disorder} = \sum_b \left(\frac{n_b}{n_t} \right) \times \left(\sum_c - \frac{n_{bc}}{n_b} \log_2 \frac{n_{bc}}{n_b} \right)$$

where n_b is the number of samples in branch b, n_t is the total number of samples in all branches, and n_{bc} is the total of samples in branch b of class c. The idea is to divide samples into subsets in which as many of the samples have the same classification as possible (as homogeneous subsets as possible). The higher the disorder value, the less homogeneous the classification. We now work through each attribute in turn, identifying which of the samples fall within the branches (attribute values) of that attribute, and signify into which class each of the samples falls (Figure 2).

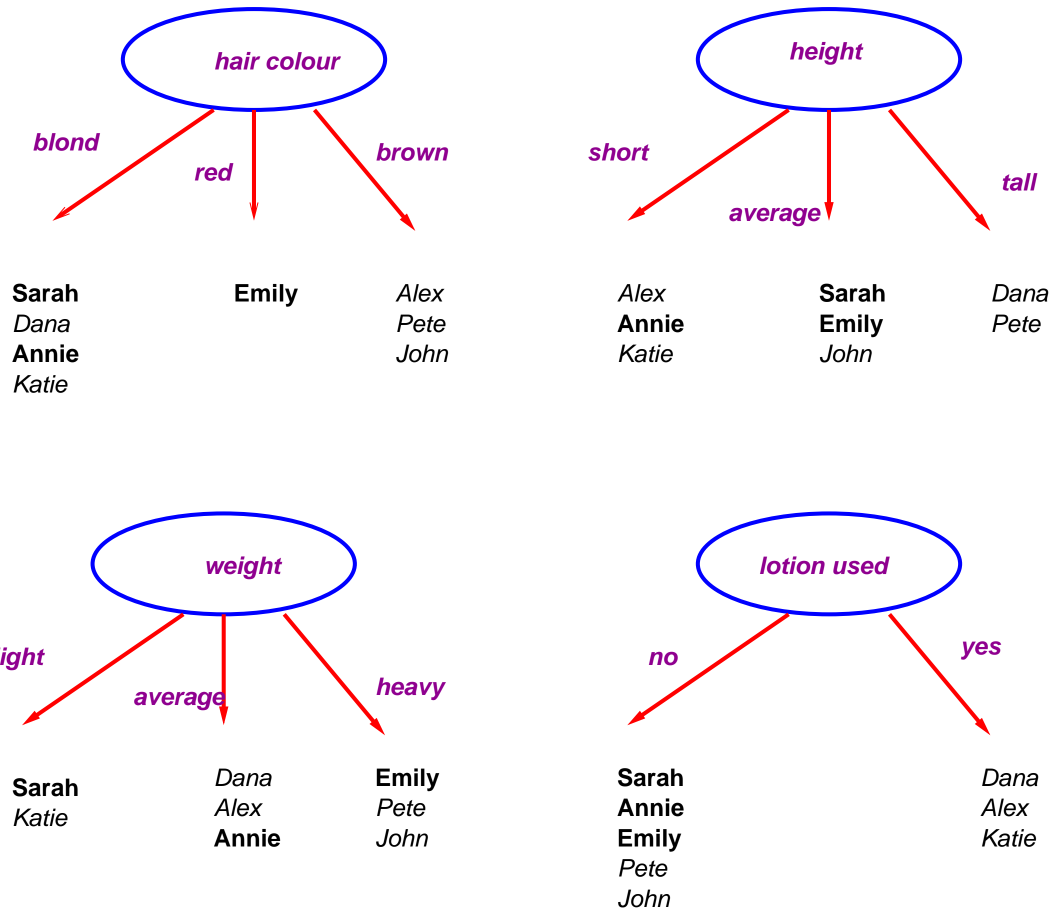


Figure 2: The first allocation of the eight individuals under each attribute value for each attribute. Names in bold indicate who is sunburned.

The average disorder for each of the attributes can now be calculated, using the disorder formula introduced above:

Average disorder for hair colour =

$$\frac{4}{8}\left(-\frac{2}{4}\log_2\frac{2}{4}-\frac{2}{4}\log_2\frac{2}{4}\right)+\frac{1}{8}\left(-\frac{1}{1}\log_2\frac{1}{1}-\frac{0}{1}\log_2\frac{0}{1}\right)+\frac{3}{8}\left(-\frac{0}{3}\log_2\frac{0}{3}-\frac{3}{3}\log_2\frac{3}{3}\right)=0.5$$

That is, 4 out of the 8 samples fall in the first branch (blond hair), of which 2 out of 4 are sunburned and two out of four are not sunburned, plus 1 out of 8 samples falls in the second branch (red hair), of which 1 out of 1 is sunburned and 0 out of 1 is not sunburned, plus 3 out of 8 samples fall under the third branch (brown hair), of which none of the three is sunburned and 3 out of 3 are sunburned. Putting this altogether gives us:

$$\frac{4}{8}(0.5+0.5)+\frac{1}{8}(0+0)+\frac{3}{8}(0+0)=0.5$$

For height, weight and lotion, respectively, we have the following:

$$\frac{3}{8}\left(-\frac{1}{3}\log_2\frac{1}{3}-\frac{2}{3}\log_2\frac{2}{3}\right)+\frac{3}{8}\left(-\frac{2}{3}\log_2\frac{2}{3}-\frac{1}{3}\log_2\frac{1}{3}\right)+\frac{2}{8}\left(-\frac{0}{2}\log_2\frac{0}{2}-\frac{2}{2}\log_2\frac{2}{2}\right)$$
$$=\frac{3}{8}\left(0.528+0.39\right)+\frac{3}{8}\left(0.39+0.528\right)+0=0.69$$

$$\frac{2}{8}\left(-\frac{1}{2}\log_2\frac{1}{2}-\frac{1}{2}\log_2\frac{1}{2}\right)+\frac{3}{8}\left(-\frac{1}{3}\log_2\frac{1}{3}-\frac{2}{3}\log_2\frac{2}{3}\right)+\frac{3}{8}\left(-\frac{1}{3}\log_2\frac{1}{3}-\frac{2}{3}\log_2\frac{2}{3}\right)$$
$$=\frac{2}{8}\left(0.5+05.\right)+\frac{3}{8}\left(0.528+0.39\right)+\frac{3}{8}\left(0.528+0.39\right)=0.94$$

$$\frac{5}{8}\left(-\frac{3}{5}\log_2\frac{3}{5}-\frac{2}{5}\log_2\frac{2}{5}\right)+\frac{3}{8}\left(-\frac{0}{3}\log_2\frac{0}{3}-\frac{3}{3}\log_2\frac{3}{3}\right)=\frac{5}{8}\left(0.442+0.529\right)+0=0.61$$

Given the range of homogeneity values for hair colour (0.5), height (0.69), weight (0.94) and lotion used (0.61), the best determinant of whether someone being sunburned or not is the attribute with the lowest value: hair colour (0.5). That concludes the first step. The next step is to look at those branches of hair colour for which there is a confused classification (the blond hair value) and to determine, for those samples only, which of the three remaining attributes is best for determining whether blond-haired people are sunburned or not. Red haired and brown haired samples are left alone, since these samples fall within a branch in which there is no mixed classification. So we repeat the procedure for the subpopulation of blond haired people only (Sarah, Dana, Annie and Katie – Figure 3).

Figure 3: Once blond haired people have been identified as falling within an inhomogeneous branch of hair colour, the task is to determine which of the remaining three attributes can classify just these samples.

By repeating the procedure for height, weight and lotion used, we get the following homogeneity values, respectively:

$$\frac{2}{4}\left(-\frac{1}{2}\log_2\frac{1}{2}-\frac{1}{2}\log_2\frac{1}{2}\right)+\frac{1}{4}\left(-\frac{1}{1}\log_2\frac{1}{1}-\frac{0}{1}\log_2\frac{0}{1}\right)+\frac{1}{4}\left(-\frac{0}{1}\log_2\frac{0}{1}-\frac{1}{1}\log_2\frac{1}{1}\right)=0.5$$

$$\frac{2}{4}\left(-\frac{1}{2}\log_2\frac{1}{2}-\frac{1}{2}\log_2\frac{1}{2}\right)+\frac{2}{4}\left(-\frac{1}{2}\log_2\frac{1}{2}-\frac{1}{2}\log_2\frac{1}{2}\right)+\frac{0}{4}\left(-\frac{0}{0}\log_2\frac{0}{0}-\frac{0}{0}\log_2\frac{0}{0}\right)=1.0$$

$$\frac{2}{4}\left(-\frac{2}{2}\log_2\frac{2}{2}-\frac{0}{2}\log_2\frac{0}{2}\right)+\frac{2}{4}\left(-\frac{0}{2}\log_2\frac{0}{2}-\frac{2}{2}\log_2\frac{2}{2}\right)=0$$

That is, lotion used is the next best attribute given its perfect classification. This attribute becomes the second level of test once we have checked hair colour (Figure 4).

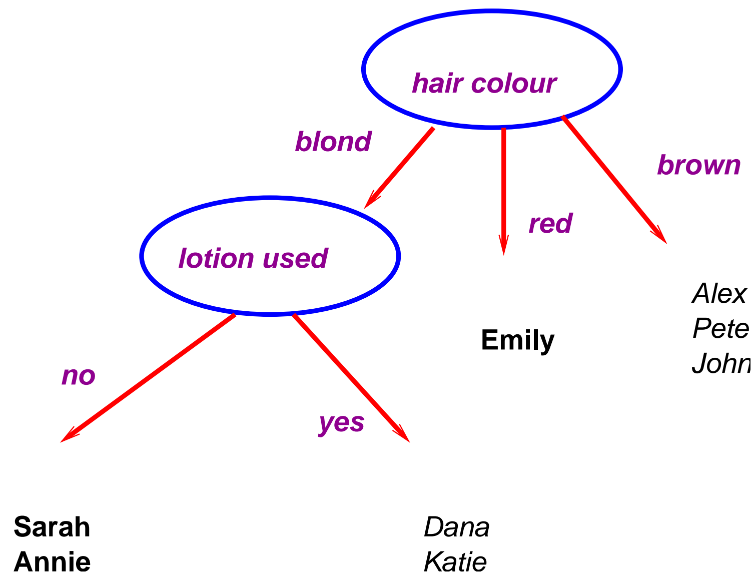


Figure 4: We now construct the identification tree for determining which tests are to be applied, in which order, to determine whether someone is sunburned or not. Note that the leaves of the tree contain individuals of the same class.

Given the full identification tree, we can then derive rules by following all paths from the root to the leaf nodes, as follows:

(a) If a person's hair colour is brown, then the person is not sunburned.
(b) If a person's hair colour is red, then the person is sunburned.
(c) If a person's hair colour is blond and that person has used sun tan lotion, then the person is not sunburned.
(d) If a person's hair colour is blond and that person has not used sun tan lotion, then the person is sunburned.

Winston (1992) describes further optimisation of this rule set through the removal of redundant antecedents and whole rules, but we must be careful not to throw away useful information in such optimisation procedures. If we now receive the information that Betty is brown haired, of tall height, has light weight and has not used lotion, we can follow the rules and conclude, on the basis of Betty's hair colour alone (brown haired), that she is not sunburned. The construction of the identification tree has allowed the determination of which attributes are the most important for classification, namely, just two in the above case.

The above process underlies the best known classification techniques in symbolic machine learning: ID3, C4.5[3] and See5/C5.0 (Quinlan, 1993; 2002).[4] One variant of machine learning is splitting the samples into two sets: the training set, and the test set. The training set is first used to extract identification trees and rules from the samples in that set. Then the test set, which consists of samples not previously seen by the machine learning system, is input to the extracted trees and rules to see whether the samples are correctly classified (the classification information will be available for each sample in the test set also to allow for a comparison between what the system

[3] See http://yoda.cis.temple.edu:8080/UGAIWWW/lectures/C45/index.html for an overview of the differences between ID3 and C4.5.
[4] See http://www.rulequest.com/see5-comparison.html for a comparison between C4.5 and See5/C5.0.

predicts and the actual class of each sample). Typically, 10% of all samples are removed to form a test set, and the remainder used for training. A common problem with machine learning is 'overfitting', where the system converges on a set of rules or identification trees that are highly accurate in classifying samples in the training set but perform much worse when presented with samples in the test set. To allow for this, a process of 'cross-validation' is sometimes used,[5] where a number of runs of the machine learning system are made on all samples, with each run choosing a random 10% of samples to generate a test set. This may then produce a different set of rules after each run, since the rules found are based on a different 90% of samples for each run. As the rule sets build up after each run, later test cases can be classified using not just the rules extracted from the training cases in that run but from all the previous rule sets from earlier runs. A form of 'voting' takes place between the different rule sets, so that test cases are predicted to fall into a class depending on a majority vote between different rule sets. One would expect to see the predictive ability of the system improve after each run as alternative rule sets are found. Other approaches to overfitting include pruning the rule set and stopping the addition of further nodes to the tree at a certain point (see Mitchell (1997) for further details).

2.2.2 Application in viral protease cleavage prediction

Symbolic machine learning and classification have many uses in bioinformatics. One particular application is in the area of viral protease cleavage prediction (Narayanan *et al.*, 2002). Intact HIV and Hepatitis C (HCV) virions are endocytosed (inserted into a cell) via specific cellular receptors on human cells. For 'retroviruses' in general, a single stranded RNA sequence (typically between 8-12 kilobases and containing at least 9 genes, including genes for producing core protein precursors (gag), envelope proteins (env) and pol (reverse transcriptase, integrase and protease)) is then transcribed into double stranded DNA by the reverse transcriptase enzyme and, if latent, can be integrated with the host genome by the integrase enzyme. In some cases the DNA provirus (originally reverse transcribed from RNA or single stranded DNA, or simply the original double stranded inserted viral DNA) is unexpressed, whereas in other cases it is transcribed into messenger RNA (mRNA) and translated into a protein chain (viral polyproteins), giving rise to new viral molecules which then reassemble to form complete virions that are then released by budding from the cell membrane, thereby destroying the cell as well as looking for other similar cells to infect.

Protease is the third enzyme typically accompanying viral DNA or RNA into the cell, although protease can also self-cleave naturally from the viral polyprotein if it isn't introduced through endocytosis. It cleaves the precursor viral polyproteins when they emerge from the ribosomes of the host cell as one long sequence. This cleavage step is essential in the final maturation step of HIV and HCV. That is, protease is responsible for the post-translation processing of the viral gag and gag-pol polyproteins to yield the structural proteins and enzymes of the virus for further infection (Figure 5). If viral protease action can be inhibited by drugs, viral replication can be stopped. Protease can be regarded as a 'scissors' enzyme which cuts long pieces of polyproteins into smaller chains that make new viral particles.

While this is a gross oversimplification of what typically happens in human viral infection, it nevertheless demonstrates at a general level the behaviour of viruses that use protease in the final stages of reproduction. Viral protease recognises and binds to n-residue amino acid sequences in the polyprotein, where n can vary typically between 8 and 15 from virus to virus, and then

[5] See http://kiew.cs.uni-dortmund.de:8001/mlnet/instances/81d91eaae465a82b29 (Machine Learning Network Online Information Service) for basic definitions of overfitting and methods for dealing with this problem.

cleaves the polyprotein wherever it finds these n-residue amino acid sequences (Figure 6). A viral polyprotein can be cleaved 8 or 9 times or more by one or more proteases. This n-residue regions act as recognition sites for the viral protease, which must have a substructure within it that binds to these recognition sites before cleaving each recognition site. These n-residue regions vary in content within the same polyprotein, yet the protease manages to cleave despite this variation. For HIV (both HIV-1 and HIV-2), there is a growing body of evidence that suggests that cleavage typically takes place between phenylalanine (F) and proline (P) amino acids at various recognition sites in the polyprotein, but it is not known how close or how far these amino acids have to be for cleavage to take place. The polyprotein is typically several thousand amino acids long, and these recognition sites can occur at any point in this sequence.

Cai and Chou (1998) reported on a neural network approach to the problem of predicting which 8-amino acid substrates were cleaved by the HIV protease. They identified 299 samples in the HIV literature, of which 60 were positive (cleaved) and 239 were negative (not cleaved). Most of these samples were identified *in vitro*, with artificially produced oligopeptide sequences of eight amino acids each being placed in a test tube with the HIV protease and observing what happens. They reported that their ANN produced a 92% accurate prediction on cases not used to train the neural network. However, because of the form of representation used for presenting the samples to the ANN, the reasons for the ANN classifying the samples the way it did were not clear. (More details of their experiments will be presented in Section 3 below.) Narayanan *et al.* (2002) repeated the learning experiment, but this time used the symbolic learning tool See5,[6] which embodies a learning algorithm very similar to that described in the sunburned case above. The samples were presented to See5 in the following form: *position1, position2, position3, position4, position5, position6, position7, position8, class.* For example, here are some examples of the dataset:

Q,M,I,F,E,E,H,G,1
R,Q,V,L,F,L,E,K,1
K,V,F,G,R,C,E,L,0
V,F,G,R,C,E,L,A,0

where the letters are taken from the standard amino acid alphabet, and ‘1’ signifies cleavage (between positions 4 and 5) and ‘0’ non-cleavage. The first amino acid of each sequence is the *position 1* attribute value, the next is the *position 2* attribute value, and so on. The final value is the class information. In addition to the samples used by Cai and Chou, an extra 63 substrate samples were also found in the HIV literature, giving a total of 362 sequences, of which 114 were positive (cleaved) and 248 negative (non-cleaved). After putting all 362 samples into See5, the following rules were obtained:[7]

(a) If position 4 is Phenylalanine then cleavage (35/5)
(b) If position 4 is Leucine then cleavage (38/9)
(c) If position 4 is Serine then non cleavage (26/1)
(d) If position 4 is Tyrosine and position 5 is Proline then cleavage (32/5)
(e) If position 6 is Glutamate then cleavage (44/8)

---

[6] See5 (scaled-down version) is available free from http://www.rulequest.com.
[7] See5 has the option of converting an identification tree into a rule set through the process of tracing paths from the root node to each terminal node, as described previously. However, See5 also optimises the converted rule set through the removal of redundant antecedents to deal with overfitting.

That is, See5 identified the relative importance of phenylalanine and proline, but not together. The figures in brackets at the end of each rule specify the number of true positives and false positives. For example, for rule (a), 35 out of the 114 positive (cleaved) samples were correctly classified by this rule, but 5 samples which fell into the non-cleaved class were also, falsely, classified as cleaved. Rules (a), (b) and (d) between them cover 105 of the 114 positive cases through the identification of position 4 (immediately to the left of the cleavage site). The relative importance of glutamate (position 6, rule (e)) had not been previously identified, to our knowledge.

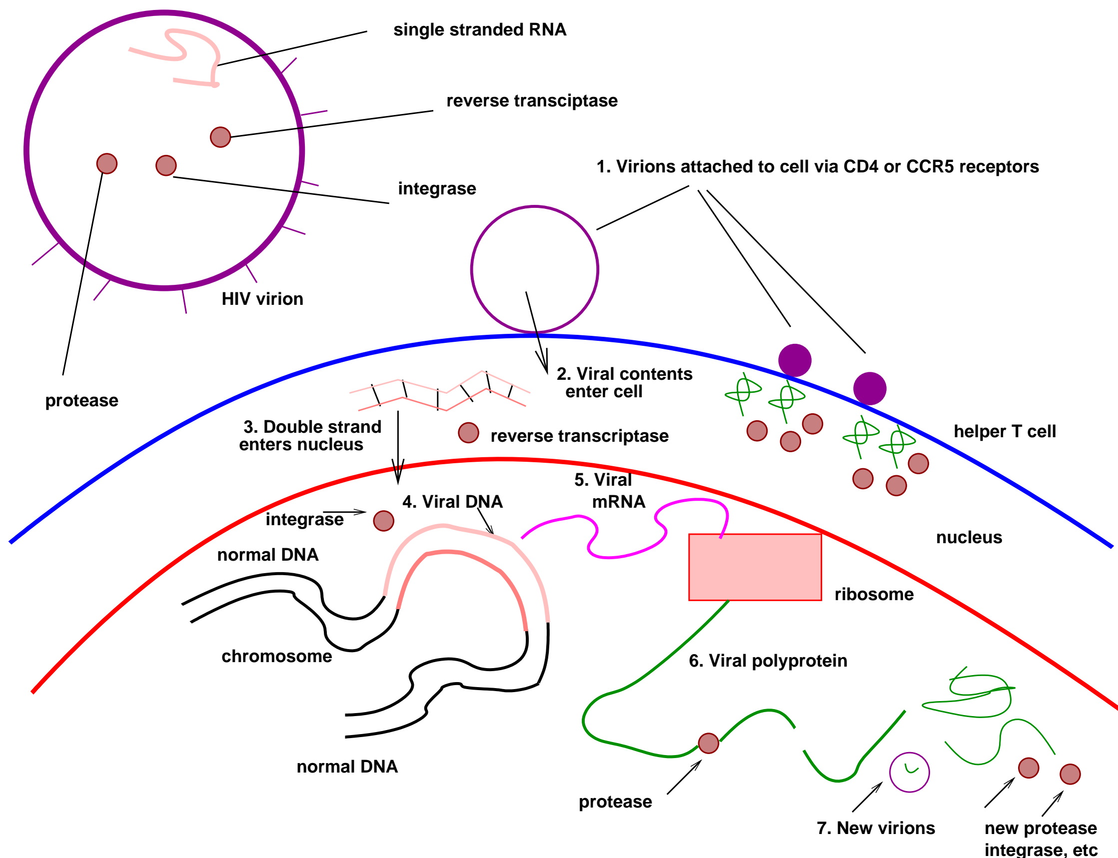


Figure 5: An overview of how HIV works. The HIV virion (upper left) consists of single stranded RNA and three enzymes. After attachment to helper T cells (step 1), the viral contents are injected into the cell (2), where the reverse transcriptase enzyme makes, from the viral single strand RNA, a double stranded copy which enters the nucleus (3). The integrase enzyme inserts the viral double strand into the cell's chromosomes (4), allowing the transcription of viral mRNA through normal nucleus mechanisms (5). After translation, the third injected enzyme, viral protease, cuts the viral polyprotein (6) which enables new HIV virions to be assembled for further infection after budding from the cell. Adapted from Narayanan *et al.* (2002).

A similar approach was adopted for Hepatitis C (HCV) protease NS3 (Narayanan *et al.* 2002). This protease appears to bind to a 10-amino acid substrate in the viral polyprotein and cleaves between position 6 and position 7. After identifying in the HCV literature 168 oligopeptide sequences that were cleaved, a further 752 negative sequences were obtained from published work as well as by applying a moving window of 10 amino acids between cleavage sites. That is,

by examining the HCV viral polyprotein and ignoring the known cleavage sites, we generated a large number of negative 10-amino acid substrates on the basis of ‘If we don’t have the evidence that NS3 cleaves at this point, then we’ll assume that it doesn’t.’ When all 920 sequences of 10-amino acid substrates were input into See5, the following rules were produced:

(a) If position 6 is Cysteine then cleavage (133/27)
(b) If position 6 is Threonine and position 4 is Valine then cleavage (28/5)
(c) If position 6 is Cysteine and position 7 is Serine then cleavage (100/33)
(d) If position 1 is Aspartate then cleavage (122/41)
(e) If position 10 is Tyrosine then cleavage (98/22)
(f) If position 10 is Leucine then cleavage (70/27)

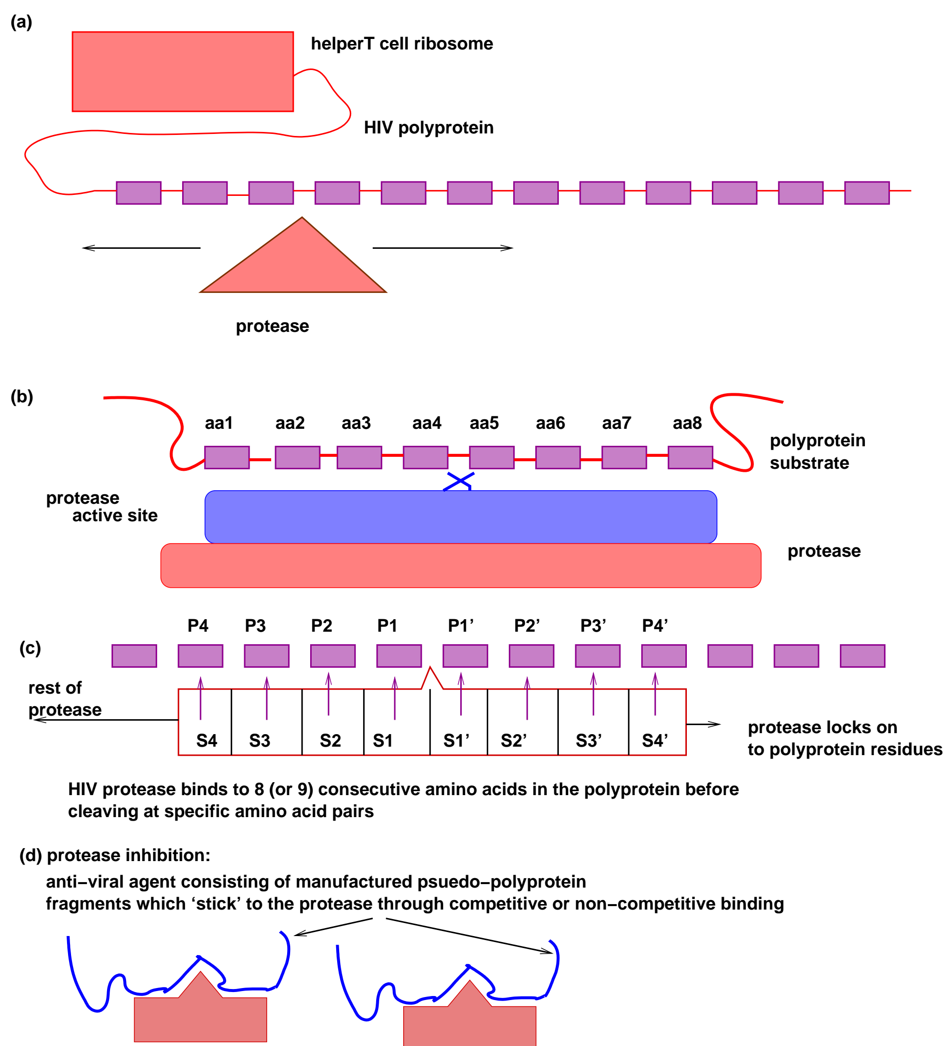


Figure 6: An overview of how HIV protease works. See the text for an explanation of this figure. Future drug design (d) could focus on ‘competitive’ binding (the agent binds to the active site of the protease) or ‘non-competitive’ binding (the agent binds to some other part of the protease and distorts the protease structure). Adapted from Narayanan *et al.* (2002).

As far as we are aware, this was the first time that any knowledge concerning HCV protease substrates had been identified in this form. Interestingly, position 6 and 7 (on either side of the cleavage point) figure in the first three rules. However, the relative importance of positions 1 and 10 had, to our knowledge, not been previously reported. Although there are still a number of false positives, these results provide some indication as to where to start looking for competitive protease inhibitors in the laboratory. Overall, See5 returned an accuracy figure of 85% for HIV data and 91% for HCV data using cross-validation (see Table 7). More interestingly, See5 obtained these results with no biochemical or biophysical information present in the patterns concerning substrates. See5 adopted a purely symbolic pattern matching approach to the problem. Further research will indicate whether the accuracy figures reported above can be improved through the introduction of biochemical and biophysical properties of amino acids and substrates.

2.4 Conclusion

Symbolic machine learning techniques have not been extensively used in bioinformatics given the sheer volume of papers in the area, although techniques based on linear programming have been used for breast cancer diagnosis (Wohlberg *et al.*, 1994; Street *et al.*, 1995), and more recently gene expression data (see Section 3 below) has started to be mined using decision tree approaches (e.g. Brazma *et al.*, 1998; Brazma, 1999). A welcome development recently has been BIOKDD01 (Workshop on Data Mining in Bioinformatics[8]) which took place in August 2001 at San Francisco,[9] which provides evidence of growing interest in the application of machine learning techniques to traditionally hard bioinformatics problems. The need for better informed analysis of biological data is now well understood (Altman, 2001), and several useful web tutorials now exist (e.g. Brazma and Jonassen, 1998; Kumar and Joshi, 1999) to help bioinformatics researchers understand the basic techniques. A good formal and historical introduction to the area of data mining is provided by Holsheimer and Siebes (1991). Nearest neighbour approaches have been used for clustering in microarray data analysis (Ralescu and Tembe, 2000), supplementing their continued use in secondary structure prediction (e.g. Yi and Lander, 1993; Salamov and Solovyev, 1995).

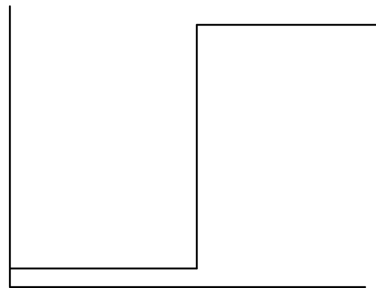
---

[8] Data mining is a subarea of AI and deals with finding underlying descriptive relationships in typically large amounts of data, where these relationships are presented in an intelligible form (typically, rules).
[9] Details of the Workshop (presented in conjunction with the 7th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining), together with papers presented at the Workshop, can be found at http://www.cs.rpi.edu/~zaki/BIOKDD01/.

3. Neural Networks in Bioinformatics

3.1 Introduction

Artificial Neural Networks (ANNs) were originally conceived in the 1950s and are computational models of human brain function (see Rumelhart, McClelland *et al.* (1986) for a history of ANN development as well as an introduction to ANNs). They are made up of layers of processing units (akin to neurons in the human brain) and connections between them, collectively known as weights. For the sake of exposition, only the most basic neural network architecture, consisting of just an input layer of neurons and an output layer of neurons, will be considered first. Artificial neural networks (ANNs) are simulated by software packages which can be run on an average PC.[10] A processing unit is based on the neuron in the human brain and is analogous to a switch. Put simply, the unit receives incoming activation either from a dataset or activation from a previous layer and makes a decision whether to propagate the activation. Units contain an activation function which performs this calculation and the simplest of these is the step or threshold function (Figure 7).

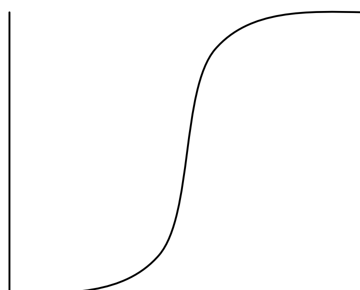


If (activation > threshold) output = 1 Else output = 0

Figure 7: Step or threshold function with pseudo-code equation. Such a step function forces the unit containing this function to dichotomise its input into one of two outputs: 1 and 0. This was the basis for the perceptron learning theorem (see Rumelhart, McClelland *et al.* (1986) for further details).

However, the most common activation function used is the sigmoid function (Figure 8). This is a strictly increasing function which exhibits smoothness and asymptotic properties. Sigmoid functions are differentiable. According to Rumelhart, McClelland *et al.* (1986), the use of such sigmoid activation functions in multilayer perceptron networks with backpropagation contributes to stability in neural network learning.

[10] See, for instance, Stuttgart Neural Network Simulator (SSNS), downloadable for free from http://www-ra.informatik.uni-uebingen.de/SNNS.



$$output = \frac{1}{1 + e^{-\alpha\ activation}}$$

Figure 8 : Sigmoid activation function and equation. Displayed here is the *logistic function*, where α is the slope parameter which can be varied to obtain different slopes.

Units are connected by weights which propagate signals from one unit to the next (usually from layer to layer). These connections are variable in nature and determine the strength of the activation which is passed from one unit to the next. The modification of these weights is the primary way in which the neural network learns the data. This is accomplished in supervised learning by a method known as backpropagation where the output from the neural network is compared with the desired output from the data and the error from this is used to change the weights to minimise the error

The task of the ANN is to modify the weights between the input nodes and the output node (or output nodes if there is more than one output node in the output layer) through repeated presentation of the samples with desired output. The process through which this happens is, first, *feedforward*, whereby the input values of a sample are multiplied by initially random weights connecting the input nodes to the output node, second, comparing the output node value with the desired (target) class value (typically 0 or 1) of that sample, and third *back-propagating* an error adjustment to the weights so that the next time the sample is presented, the actual output is closer to the desired output. This is repeated for all samples in the data set and results in one *epoch* (the presentation of all samples once). Then the process is repeated for a second epoch, a third epoch, etc, until the feed-forward back-propagating (FFBP) ANN manages to reduce the output error for all samples to an acceptable low value. At that point, training is stopped and, if needed, a test phase can begin, whereby samples not previously seen by the trained ANN are then fed into the ANN, the weights 'clamped' (i.e. no further adjustment can be made to the weights), and the output node value for each unseen sample observed. If the class of the unseen samples is known, then the output node values can be compared with the known class values to determine the *validity* of the network. Network validity can be measured in terms of how many unseen samples were falsely classified as positive by the trained ANN when they are in fact negative ('false positive' rate) and vice versa ('false negative' rate). If the class of an unseen sample is not known, then the output node values make a *prediction* as to the class of output into which the sample falls. Such predictions may need to be tested empirically.

More formally and very generally, the training phase of an ANN starts by allocating random weights w_1, w_2, … w_n to the connections between the n input units and the output units. Second, we feed in the first pattern p of bits $x_1(p)$, $x_2(p)$ … $x_n(p)$ to the network and compute an activation value for the output units given p: $O(p)=\sum_{i=1}^{n} x_i(p)w_i(p)$. That is, each input value is multiplied by the weight connecting its input node to the output nodes, and all weighted values are then summed to give us a value for the output nodes. Third, we compare the output value for the pattern with the desired output value and update each weight prior to the input of the next pattern p': $w_i(p') = w_i(p) + \Delta w_i(p)$, where $\Delta w_i(p)$ is the weight correction for pattern p calculated as follows: $\Delta w_i(p) = x_i(p) \times e(p)$, where $e(p)= O_D(p) - O(p)$, where in turn $O_D(p)$ is the desired output for the pattern and $O(p)$ is the actual output. This is carried out for every pattern in the dataset (usually with shuffled, or random, ordering). At that point we have one *epoch*. The process is then repeated from the second step above for a second epoch, and a third, etc. Typically, an ANN is said to have *converged* or *learned* when the sum of squared errors (SSE) on the output nodes for all patterns in one epoch is sufficiently small (typically, 0.001 or below). The equations above constitute the *delta learning rule* which can be used to train single-layer networks. A slightly more complex set of equations exists for learning in ANNs with more than one layer and making use of the sigmoid function described earlier. These more advanced equations are not required here, but can be found in Rumelhart *et al* 1986.

While many different types of neural network exist, they can be generally distinguished by the type of learning involved. There are two basic types of learning: supervised and unsupervised learning. In supervised learning, the required behaviour of the neural network is known (as described above). For instance, the input data might be the share prices of 30 companies, and the output may be the value of the FTSE 100 index. With this type of problem, past information about the companies' share price and the FTSE 100 can be used to train the network. New prices can then be given to the neural network and the FTSE 100 predicted. With unsupervised learning, the required output is not known, and the neural network must make some decisions about the data without being explicitly trained. Generally unsupervised ANNs are used for finding interesting clusters within the data. All the decisions made about those features within the data are found by the neural network. Figure 9 illustrates the architecture for a *two*-layer supervised ANN consisting of an input layer, a 'hidden' layer and an output layer.

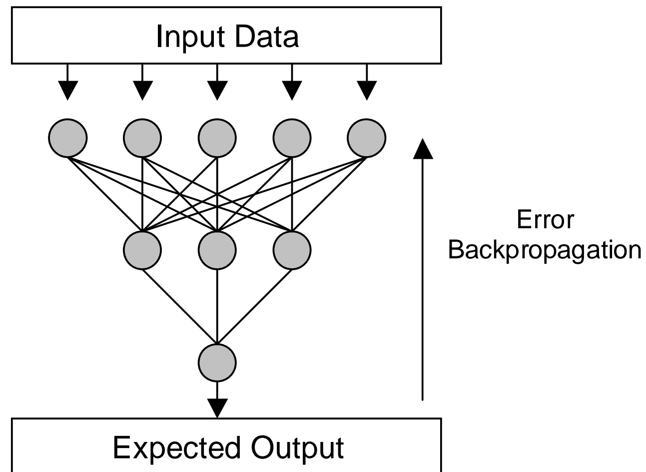


Figure 9: Supervised learning. Signals are propagated forward through the network from input to output. An error between the current and required outputs is computed at the output layer (consisting of just one node in our example). This error is backpropagated through the network, changing weight values. The ANN architecture is called 'supervised' because the error between the expected output and actual output is used to back-propagate changes to the weights between the layers of the network.

Unsupervised neural networks have been frequently used in bioinformatics as they are a well-tested method for clustering data (see, for example, Azuaje (2001)). The most common technique used is the self-organising-feature-map (SOM or SOFM; Kohonen, 1982), and this learning algorithm consists of units and layers arranged in a different manner to the feedforward backpropagation networks described above. The units are arranged in a matrix formation known as a map, and every input unit is connected to every unit in this map. These map units then form the output of the neural network (Figure 10).

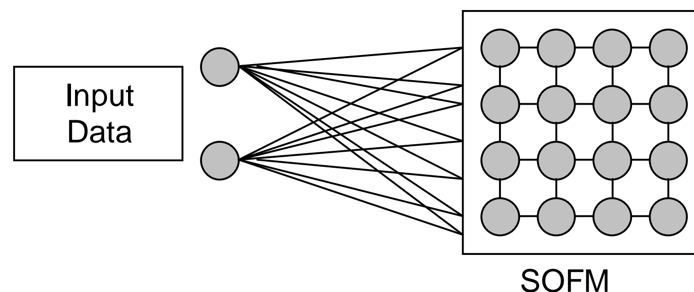


Figure 10: Self Organising Feature Map (Kohonen, 1982). This uses a map of interconnected neurons and an unsupervised learning algorithm which uses a neighbourhood of units to find common features in the data and therefore develop clustering behaviour.

The SOFM relies on the notion that similar individuals in the input data will possess similar feature characteristics. The weights are trained to group those individual records together which possess similar features. It is this automated clustering behaviour which is of interest to bioinformatics researchers.

Numerous advantages of ANNs have been identified in the AI literature. Neural networks can perform with better accuracy than equivalent symbolic techniques (for instance decision trees) on the same data. Also, while identification tree approaches such as See5 can identify dominant factors the importance of which can be represented by their positions high up in the tree, ANNs may be able to detect non-dominant relationships (i.e. relationships involving several factors, each of which by itself may not be dominant) among the attributes. However, there are also a number of disadvantages. Data often has to be pre-processed to conform to the requirements of the input nodes of ANNs (e.g. normalised and converted into binary form). Training times can be long in comparison with symbolic techniques. Finally, and perhaps most importantly, solutions are encoded in the weights and therefore are not as immediately obvious as the rules and trees produced by symbolic approaches.

3.2 Applications of ANNs in temporal data analysis

3.2.1 Example 1: gene expression data modelling and analysis

Microarray (gene chip) data is created by measuring gene expression values (or "activation" values) either over a number of timesteps, often whilst subjecting the organism to some stimulus, or over a number of individuals (one timestep) who fall into different classes, or both. The measurements gained from this study often span thousands of genes (fields, attributes) and only tens of timesteps (records) for temporal gene expression data or tens of individuals for non-temporal gene expression data. The goal of modelling and analysing temporal gene expression data is to determine the pattern of excitations and inhibitions between genes which make up a gene regulatory network for that organism. A gene regulatory network can be inferred by observing the interactions between genes from one timestep to the next. Finding an optimal or even approximate gene regulatory network can lead to a better understanding of the regulatory processes that are occurring in the organism and therefore represents a significant step towards understanding the phenotype of the organism. Drugs can be designed, for instance, to interfere with or promote gene expression at certain points in the network. For non-temporal data, the goal is to identify, from the thousands of genes measured for an individual, the handful of genes that contribute to that individual falling within a specific class (e.g. diseased or normal) and thereby use those genes as a basis for predicting the class of an individual in future diagnosis through limited measurement. The identification of gene regulatory patterns within microarray data is certainly non-trivial, even for non-temporal data. Surprisingly, there have been few attempts at using standard feedforward backpropagation ANNs in these areas. One of the reasons for this may be that it has not been clear how to represent such data to a FFBP network. There are two aspects to representation: architectural representation, and data representation. The former deals with issues of layers and learning rules, while the latter deals with data input.

With regard to the latter issue, real-world microarray data is stored in a number of different formats and at different levels of detail, with no uniform standards as yet agreed for reporting such data. One method for overcoming this is to use Boolean networks which approximate real-world gene expression data, in that gene expression values are restricted to 'on' or excitation, and 'off' or inhibition, states. Another interpretation of two-valued gene networks is the 'presence' or 'absence' of specific genes, given their values as recorded on the gene chip. For example, the gene expression databases for multiple myeloma (an incurable cancer involving immunoglobulin secreting plasma cells) contain, in addition to real values extracted through the Affymetrix process,[11] absolute values of gene expression involving Absent, Present and Marginal. Fortunately, Marginal values occur an insignificant number of times in the datasets we worked on, thereby leading to a database with two absolute values: Absent (represented to our neural networks by 0), and Present (1).[12] While the myeloma data is not time-dependent (i.e. the data represents the gene expressions of 74 cancer samples (individuals) and 31 normal donors (individuals) at only one time point), it illustrates that the principle of representing gene expression by 1 and 0 is now well accepted in the gene expression research community, due to the Affymetrix process. For temporal data, genes that are on at one timestep can subsequently turn other genes on or off in the next timestep. Any gene that receives no "excitation" in a timestep will switch itself off or can be in a neutral state. These networks therefore model the interactions between genes in a controlled and intelligible manner. The problems associated with extracting networks from what might be considered Boolean data are still not trivial. Gene expression data, including Boolean networks, are problematic because of their dimensionality.

---

[11] Details of the Affymetrix process can be obtained from www.affymetrix.com.
[12] This myeloma data can be obtained from http://lambertlab.uams.edu/publicdata.htm.

They have a large number of attributes (genes) and few examples (time samples): a typical set for the yeast genome, for example, can consist of 6,000 genes (attributes) and less than 10 timesteps (records or samples).

With regard to the issue of architectural representation, we use the simplest form of FFBP ANN: there are no hidden layers and the activation function used is the simplest of all – the step function. Since there has been no previous work on the application of supervised FFBP ANNs in this area, the first task is to determine the simplest type of FFBP network possible for dealing with artificially produced temporal gene expression data represented in Boolean form, where rules have been used to generate the data (forward engineering the data) and to see whether it is possible, after successful training of the FFBP ANNs, to reverse engineer back to the original rules by looking at the weight relationships between input and output layers of the ANNs alone. In the real world, of course, it will not be possible to determine the accuracy of the reverse engineered rules in purely computational terms, because it is not known what the true causal relationships between genes actually consist of. Nevertheless, the experiments reported here demonstrate that reverse engineering with trained FFBP ANNs leads back to the original rule set which produced the data in the first place and thereby lends confidence to the view that neural networks can play a significant part in reverse engineering from gene expression data.

The FFBP ANN is single layered and fully connected between input and output layer, with the activation function, the step function, shown here:

Equation (1)

$$\left.\begin{array}{l}\sum w_{ij}x_i > 0.5 \\ \sum w_{ij}x_i < 0.5\end{array}\right\} \frac{1}{0}$$

where w_{ij} is the weighted value connecting node i to node j and x_i is activation value for node i. There are no bias terms. The architecture learns the input and output connections by virtue of the backpropagation algorithm for the step function which is quite simple:

Equation (2)

$$w_i(t+1) = w_i(t) \pm \eta x_i(t)$$

where $w_i\ (t + 1)$ is the weight value at time t+1 and ηx_i is the learning rate associated with x_i at time t. Due to the fact that the inputs will be either 0 or 1, there must be some graduation to allow the algorithm to learn weights around the threshold (here 0.5) area. Therefore the term η must be included to direct backpropagation in smaller steps towards an optimal solution. The term has been introduced in the experiments below as 1/20 and means that the weights progress towards the optimum in steps of 0.05. The learning rate (the size of change to weights) can be adjusted through a *momentum* term (not shown in Equation (2)) that can be 0, in which case the backpropagation algorithm operates without momentum (the learning rate is constant all the way through learning), or it can be set to accelerate descent in a specific direction through larger learning rates, or it can be set to stabilize the network through increasingly smaller learning rates (Rumelhart, McCleland *et al.*, 1986).

The training is iterative with regard to the output. That is, each output gene is individually and separately trained by the network before progression to the next output gene. Training consists of presenting pairs of data (input and target) to the network as normal. The pairs, however, consist of gene expression values from two consecutive timesteps. Gene values for Time=*t* are input to

the algorithm and the target consists of gene values at Time =*t*+1 (Figure 11). Once training has been completed over all output genes individually, testing consists of re-presenting all input Time=*t* and comparing it against the target Time=*t*+1 to compute a percentage accuracy for all outputs. That is, while training only optimized one individual output gene at a time until all output genes were sequentially trained, testing allows us to determine what the overall output pattern for a specific input pattern consists of after successful training of individual output genes. This testing regime therefore tests whether the FFBP ANN has indeed captured a full gene network in terms of the effect that activation values of input genes have in parallel on activation values of output genes.

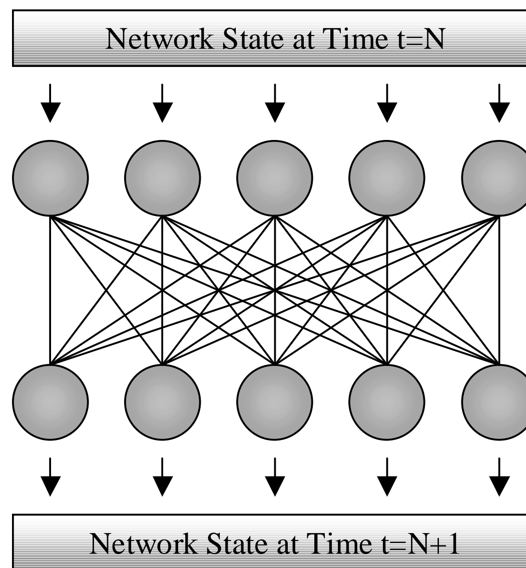


Figure 11: Gene expression values at Time T=N are input to the network and expression values at Time t=N+1 are used as training targets for the neural network

The experiments below have been run on artificial gene expression data 'forward engineered' by a program which creates temporal Boolean data based on 'Liang Networks' (D'Haeseleer *et al.*, 1999). A Liang network consists of rules which are made of AND and OR operators distributed randomly in a number of random fixed length logical expressions, but to a specific *k* value which determines the maximum number of genes which can affect other genes in the next timestep. We used only two *k* values. If *k*=2, then between two and four genes can be affected at Time=*t*1 by genes at Time=*t*0. When k=3, anything between three and nine genes are involved in the next time step. The experiments below have been undertaken on artificial data with 10 genes derived from Boolean rules with *k* values of 2 and 3. The error represents the percentage difference between the output and the target of the Boolean network.

As can be seen in the toy example (Table 4), the activations at T=*t*0 represent the full logic table of the three genes. The activations at T=*t*1 represent the activations once the rules have been applied. For instance, the expression values 0 0 1 for gene 1, gene 2 and gene 3, respectively, at *t*0

become 0 1 1 for these three genes at *t*1 (fifth rows of Table 4). The ‘reverse engineering’ task in its basic form here is to use the information in the left and right hand parts of Table 4 (row by row) to model the data through an ANN. While this may be trivial for three genes (and eight rows), the problem quickly becomes non-trivial, especially when one considers Liang networks consisting of hundreds and possibly thousands of genes. The rules for producing (‘forward engineering’) the artificial gene expression data contained in Table 4 are as follows (in the format gene expression at *t*1 = gene expression at *t*0):

Rule 1: Gene 1’ = Gene 2;
Rule 2: Gene 2’ = Gene 1 OR Gene 3
Rule 3: Gene 3’ = (Gene 1 AND Gene 2) OR (Gene 2 AND Gene 3) OR (Gene 1 AND Gene 3)

Time = t0		
Gene 1	Gene 2	Gene 3
0	0	0
1	0	0
0	1	0
1	1	0
0	0	1
1	0	1
0	1	1
1	1	1

Time = t1		
Gene 1’	Gene 2’	Gene 3’
0	0	0
0	1	0
1	0	0
1	1	1
0	1	1
0	1	1
1	1	1
1	1	1

Table 4: A (toy) example of the type of boolean network used in experiments (a “Liang Network” (adapted from Liang *et al.,* 1998.)). The left-hand table represents the gene expression values at Time T=t0, and the right hand table, those at Time T=t1. Note that only three genes are involved here. In the actual forward engineering of data, 10 genes are involved.

For instance, looking at the fifth row of Table 4 again, since gene 2 is 0 at t0, gene 1's value at t1 is 0 (first rule); since gene 3's value is 1 at t0, gene 2's value is 1 at t1 (second rule), and so on. The FFBP ANNs on which these experiments are based are exactly the same except that the rules are randomly distributed and the networks involve 10 genes rather than 3. Altogether, six artificial datasets were produced, with different random rules. Three of the datasets had $k = 2$ and three had $k = 3$, which means that up to four genes and up to nine genes, respectively, can be affected at the next time step. Already, 2^{10} (1024) records are needed to specify every possible truth assignment for each dataset. If one were to increase this to 20 genes, this would require 2^{20} (1,048,576) records. Although Liang networks quickly become intractable to fully enumerate, the point here is to determine whether FFBP ANNs can successfully reverse engineer even constrained, forward engineered Liang networks. If they don't, there is very little chance that FFBP ANNs will be successful for large numbers of genes with sparse data (typically, a real microarray will have data for several hundred and perhaps several thousand genes, for anything between 3 and 50 timesteps) and where the original rules giving rise to the data are not known.

When the rules become more complex than above, then they can be expressed as a wiring diagram. This method of showing the interactions between genes can also be helpful in allowing biologists to

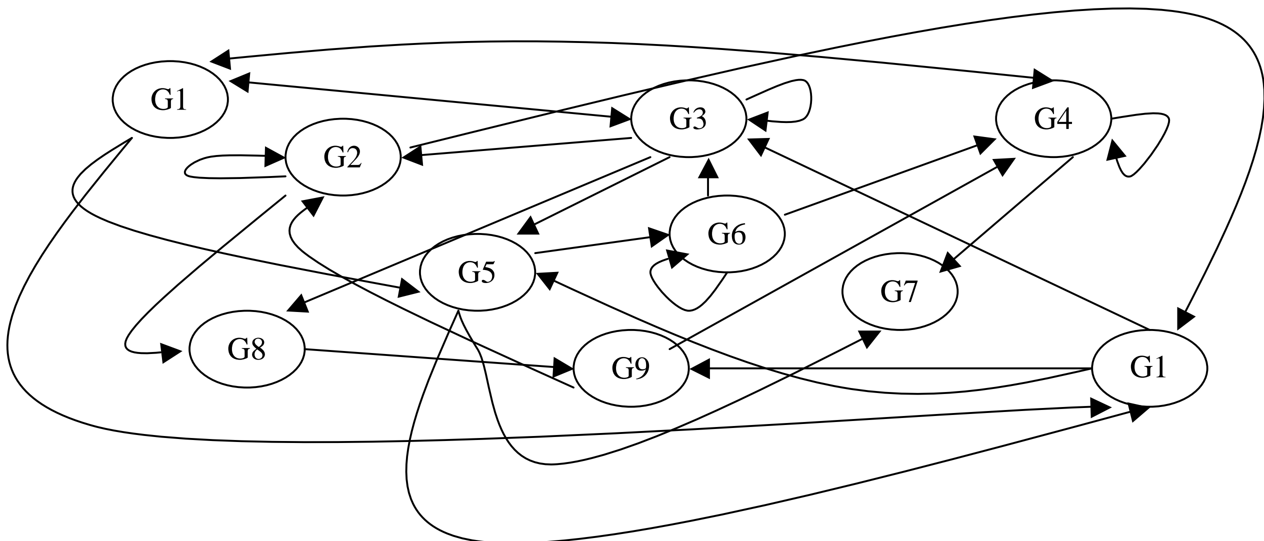


Figure 12: Gene regulatory "wiring diagram" for k=2 dataset number 3. As can be seen, self-connections are permitted and the total number of connections is quite large. Note that the in degree of each node is 4 or less.

see the overall gene regulatory network. The desired regulatory network for the k=2 dataset number 3 (see below) is shown in Figure 12. Such wiring diagrams are abstract in the sense that the arrows are neutral in terms of excitation ('switch on') and inhibition ('switch off'), signifying instead that one gene influences another. Also, if there is more than one arrow pointing to a gene, wiring diagrams do not specify whether an 'AND' or 'OR' relation is involved. Nevertheless, such wiring diagrams provide an immediate graphical representation of how genes are linked together and regulate each other (either through excitation or inhibition) each other.

Six separate step-function linear models were trained on six 10-gene Liang-type datasets (1024 records each), three with k=2 and three with k=3 (see Table 2). The goal for the experiments is to show that FFBP ANNs with a simple step function can model this type of data and embody the correction connections within their weights. The linear models seen here were all implemented in C++ using the equations above and the runs were made with a boundary of 0.1% error, or 100 epochs, whichever was

achieved sooner. With only 100 epochs, these runs were completed in very quick time on a modest PC, typically less than 10 seconds per network.

k	Data1	Data2	Data3
2	98.125%	100%	99.84%
3	100%	98.359%	99.375%

Table 5: Percentage error of runs of the step-function neural network over six "Liang Networks".

As can be seen (Table 5), the results are very close to the optimum for these networks. The average accuracy over the 6 datasets is 99.283%. That is, when all 1024 records are used for training (one output gene iteratively trained at a time) as well as for testing, almost perfect results are obtained. This represents extremely good accuracy over this size and number of datasets. The high k values (up to nine genes being allowed to be affected at the next time step) don't appear to affect the accuracy. Once the ANNs have demonstrated success at modeling the data, an important aspect is to reveal the knowledge and rules embodied in the networks to see whether the ANNs have indeed reverse engineered successfully back to the original rules which gave rise to the datasets in the first place. What is interesting is that the weights reveal a great deal about the actual distribution of the data – for instance, the most common weights seen are:

0.6-0.7 – Used for genes which are the sole contributors to output, or genes which are part of OR logic functions. Genes coupled with this weight are capable of turning on units by themselves.

0.05-0.2 – Used for genes which do not contribute to the output of that gene. These weights effectively nullify the contribution of that gene.

0.25-0.45 – Used for genes which are part of AND statements, where these weight values are not enough to trigger the gene directly but when coupled together with another gene will give good results.

These mathematical values were used successfully to extract rules linking genes in the input layer to genes in the output layer and so derive rules from the trained neural networks. The experiments therefore demonstrate that, with constrained Liang datasets, FFBP ANNs can model temporal Boolean gene expression data and successfully reverse engineer such data to derive rules which are compatible with the data and close in accuracy to the actual rules which underlie the data.

To test this claim further, we trained the model on fewer numbers of records to ensure the ANN was capable of generalising to new data. Table 6 shows the high level of accuracy (average 96.81%) which is maintained when training on only 100 randomly chosen records (less than 10% of the available data) and tested on the full enumeration of the Liang network.

k	Data1	Data2	Data3
2	96.67%	97.92%	97.98%
3	95.53%	95.28%	97.46%

Table 6: The percentage accuracy of the step-function neural network when trained on only 100 records and tested on the entire enumeration (1024 records).

Our ANNs are not learning to classify and are therefore not attempting to discriminate linearly between different class values on the basis of the data. Instead, our networks are learning how to predict the next microarray state (the state of data at the next subsequent timepoint, given data from the previous time point). Most genes display some background activation, but this is not enough to turn on a gene in the output. Also, not every rule in the discovered network is specified exactly as it is in the original ruleset. Nevertheless, the experiments reported here demonstrate the feasibility of using standard FFBP networks for modeling temporal gene expression data. As and when such data become widely available, our results indicate that ANNs offer a radically different way to analyse such data (see Keedwell *et al.*, 2002, for more details on these experiments). As can be seen from the results, the problem of modeling gene expression data and extracting candidate rules from trained networks looks soluble from an ANN viewpoint, using a relatively simple architectural representation.

3.2.2 Example 2: clustering temporal yeast data

As described before, SOMs find application in the domain of clustering, and work performed by Tamayo *et al.* (1999) shows that they are ideally suited to this task. Self-Organising-Maps will find different numbers of clusters depending on the numbers of nodes which are present in the map itself. Tamayo *et al.* found that with few nodes, no meaningful clusters were found, the excessive variation within clusters revealed that the genes being grouped were not similar enough for this task. As more nodes were added though, the clusters became more well formed and increasing the nodes beyond a limit did not increase the tightness of the clusters appreciably further. Before input into the SOM, the data had genes removed which did not change expression value significantly over the time space, and also normalised the data so that the waves of expression could be found rather than absolute values for genes. This type of preprocessing reduces the number of variables quite drastically and ensures that the SOM can find meaningful clusters within the data.

Tamayo *et al.* performed a set of experiments on data sourced from Cho *et al.* (1998) which takes gene expression values at 10 minute intervals from the yeast *Saccharomyces cerevisae* over two cell cycles (160 minutes). They measured 6,218 genes expression values for these timesteps which are reduced to 828 after removing those with little variation. Tamayo *et al.* ran the SOM on this data and arrived at 30 clusters which grouped the 828 genes. Several of the clusters found appear to show the periodicity of the cell cycle with stages G1-S-G2-M. Those clusters which exhibit peak values in these stages are compared with those which were determined by visual inspection by Cho *et al.*, the results were very similar. Tamayo *et al.* conclude by stating that the clusters derived by SOM were closely related to the stages of the cell cycle

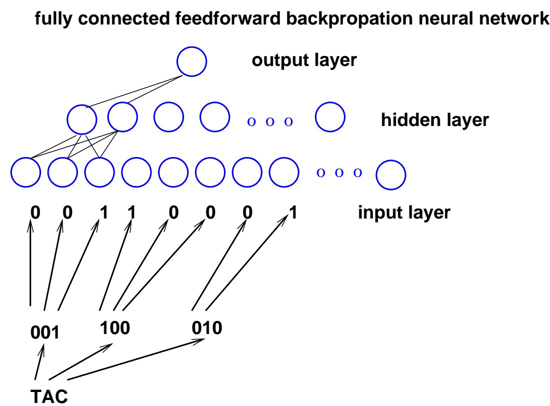
This result shows that the SOM is capable of finding meaningful biological patterns in gene expression data of thousands of genes. The SOM was used only on the data itself, and had no expert knowledge to guide it to the correct clusters beyond the number of units in the map, yet it managed to discover clusters which were similar to the stages of the cell cycle. This is a remarkable example of the learning power of neural networks and shows that unsupervised learning can yield important biological information from such datasets.

3.3 Applications of ANNs in non-temporal classificatory tasks

3.3.1 Example 1: HIV and HCV protease cleavability prediction

Yu-Dong Cai and Kuo-Chen Chou (Cai and Chou, 1998) produced a first attempt at using a neural network to predict HIV protease cleavability (see Section 2), with some success. Since there are 20 amino acids, each amino acid in the viral polyprotein sequence is represented by a unique 20-bit binary string consisting of nineteen 0s and one 1, where the position of the 1 identifies the amino acid. This method of representation is known as 'sparse coding'. An octapeptide is converted into an 8×20=160-bit binary string, as shown in Figure 13. This is the method adopted by Cai and Chou (1998). This representation method then leads to the use of a FFBP neural network with 160 input units (one for each bit of the input binary string), 8 hidden units, and one output unit. Training continued until the output node produced 0.9 or greater for positive samples and 0.1 or less for negative samples. The weights were then clamped and the test set run through the trained neural network. For the HCV dataset, given that there are 25 characters to encode (See Narayanan *et al.*, 2002 for further details), a 25-bit binary string is used to represent each of the 25 characters. Thus a dacapeptide is converted into a 10×25=250-bit binary string. Our FFBP neural network used 250 input units, 20 hidden units (after several trial runs which indicated that 20 hidden units produced good results), and one output unit, using the same training regime as for the HIV dataset. No physical properties are encoded at all.

Backpropagation with momentum is used in the training, where initialization weights are randomly selected between –1 and 1. In the training set, the output of positive sequences is 1 and the output of negative sequences is 0. In the test set, the sequences are predicted as positive if the output is greater than 0.5 and predicted as negative if the output is less than 0.5. 300 training cycles were sufficient for HIV training, while for HCV NS3 training 500 training cycles were used. In both cases, the sum of squares error (SSE) of the neural networks was below 0.001. Because neural networks are data sensitive, that is, different results are produced when different data is given to the model, ten different training and corresponding test sets were used, where each training and corresponding test set covered all the data. Neural networks can also be sensitive to the initial random allocation of weights. So three random initializations are given to each training data set, from which three prediction results are obtained for each test data set subsequently. There were therefore 30 different runs for each of the HIV and HCV datasets.



Alanine (A)	10000000000000000000
Cysteine (C)	01000000000000000000
Aspartate (D)	00100000000000000000
Glutamate (E)	00010000000000000000
Phenylalanine (F)	00001000000000000000
Glycine (G)	00000100000000000000
Histidine (H)	00000010000000000000
Isoleucine (I)	00000001000000000000
Lysine (K)	00000000100000000000
Leucine (L)	00000000010000000000
Methionine (M)	00000000001000000000
Asparagines (N)	00000000000100000000
Proline (P)	00000000000010000000
Glutamine (Q)	00000000000001000000
Arginine (R)	00000000000000100000
Serine (S)	00000000000000010000
Threonine (T)	00000000000000001000
Valine (V)	00000000000000000100
Tryptophan (W)	00000000000000000010
Tyrosine (Y)	00000000000000000001

Figure 13: Feedforward backpropagation neural network architecture and representation: Imagine we have the amino acid sequence 'TAC' (left half of diagram). Each amino acid is assigned a unique binary code (in this case, three bits), with only one bit of the three set to 1. Each bit is then input into the neural network. So, for instance,'TAC' has the binary representation '001100010'. Each sequence has its class (positive, negative), and this class value is used to train the network output (one output unit only). The network is fully connected (only a few connections are shown here between the input and hidden layer, and between the hidden layer and the output layer). The same, sparse coding principle has been applied to the HIV and HCV datasets, following Cai and Chou (1998), except that each amino acid has its own 20 bit representation for HIV sequences (right half of diagram) and 25 bit representation for HCV 'residues'. The ANN for HCV consisted of 250 input units (10 amino acids by 25 bits each), 20 hidden units and one output unit (0 for noncleavage, 1 for cleavage). The ANN for HIV consisted of 160 input units (8 amino acids by 20 bits each), 8 hidden units and one output unit.

Tables 7A and 7B show all the results of the FFBP neural networks for the HIV and HCVNS3 test (unseen) samples. The accuracy, sensitivity and specificity values (see Table 1 for an explanation of these terms) in each column of Table 1 are the mean of three runs for that training and test set combination. In the calculation of sensitivity and specificity value, a sequence is considered as correctly predicted only if it is correctly predicted all the three times, otherwise, it will be considered wrongly predicted. The accuracy of the FFBP neural network for HIV protease is 92.50%, very similar to that of previous work (92.06%) (Cai and Chou, 1998). For the HCV NS3 protease, the FFBP network produced high accuracy, 96.25%, and for the ten test sets, the standard deviation is 0.93%. This demonstrates that artificial neural networks provide powerful and reliable techniques for predicting HCV NS3 protease substrate cleavage sites.

Table 7A

HIV protease		set1	set2	set3	set4	set5	Set6	set7	set8	set9	set10	mean	STD
	Accuracy (%)	88.6	95.4	91.3	92.2	91.8	90.3	95.0	89.0	95.9	95.4	92.50	2.76
	Sensitivity (%)	82.6	95.7	78.3	78.3	82.6	87.0	78.3	69.6	82.6	87.0	82.19	6.94
ANN	Specificity (%)	86.0	94.0	92.0	94.0	90.0	80.0	94.0	84.0	100	94.0	90.80	5.90
	Accuracy (%)	82.2	84.9	80.0	90.4	86.3	80.8	89.0	86.3	87.7	86.3	85.47	3.31
	Sensitivity (%)	82.6	69.6	60.9	87.0	70.0	82.6	73.9	73.9	78.3	73.9	75.27	7.65
See5	Specificity (%)	82.0	92.0	90.0	92.0	94.0	80.0	96.0	92.0	92.0	92.0	90.2	5.12

Table 7B

HCV NS3 protease		set1	set2	set3	set4	set5	Set6	set7	set8	set9	set10	mean	STD
	Accuracy (%)	96.0	96.0	95.3	95.5	97.3	98.0	94.9	96.7	96.4	96.4	96.25	0.93
	Sensitivity (%)	88.2	88.2	79.4	88.2	94.1	88.2	82.4	88.2	85.3	85.3	86.77	3.98
ANN	Specificity (%)	96.0	96.7	96.0	96.7	96.7	98.0	96.0	98.0	96.7	97.3	96.80	0.76
	Accuracy (%)	94.0	90.2	90.2	88.0	87.0	93.5	96.2	87.5	88.6	93.5	90.87	3.21
	Sensitivity (%)	70.6	61.8	70.6	61.8	52.9	67.6	82.4	47.1	47.1	76.5	63.84	12.01
See5	Specificity (%)	99.3	96.7	98.7	94.0	94.7	99.3	99.3	97.0	98.0	97.3	97.40	1.91

Table 7: The results of running ANNs and See5 on the 10 sets of test data. The upper table gives the overall performance on the HIV data, and the lower table on the HCV data. The means show that ANNs outperform See5 on both test data sets. Accuracy (AC), specificity (SP, or true negative rate) and sensitivity (SE, or true positive rate) are defined as follows: AC=(TP+TN)/(TP+TN+FP+FN); SE=TP/(FP+TP); SP=TN/(TN+FN), where TP=true positive, TN=true negative, FP=false positive, and FN=false negative.

We ran See5 on the same ten training and test datasets (just once, since See5 is deterministic) to see how a symbolic machine learning approach compared with the ANN approach and if rule-based knowledge could be derived (Table 7). Overall, See5 was not as accurate or sensitive as the ANN but did demonstrate comparable specificity (i.e. See5 identified good rules for non-cleavage). For the HIV data, See5's accuracy of 85.47% was not as good as the FFBP ANN's accuracy of 92.5%. Similarly, for the HCV data, the FFBP ANN's accuracy of 96.25% was better than See5's 90.87%. Also, the standard deviation for the ANNs was better than for See5, indicating fewer variations in performance across runs.

3.3.2 Example 2: Leukaemia Dataset

The leukaemia dataset from Golub *et al.* (1999) consists of 72 individuals and 7129 gene expression values is collected from each of them. Classifications consist of individuals suffering from a type of leukaemia, ALL or AML. Distinguishing between these two types of data is very important as they respond very differently to different types of therapy and therefore the survival rate may well be improved by the correct classification of this leukaemia. The outward symptoms are very similar, so a method which can differentiate between the two based on gene expression profiles could help patients of the disease. An approach by Su *et al.* (2002) uses several neural networks trained on different aspects of the data in conjunction to give a final classification. Su uses the original data and two fourier transforms of the data to train three separate "experts", a gating network then uses a majority vote to determine the final classification of these networks. This approach yields a 90% classification rate (therefore only 10% error) from the final network, which is in contrast to 75% for each expert.

An approach by Ryu and Cho (2002) has been to operate many different feature selection algorithms and a variety of neural networks on this leukaemia dataset. Feature selection is a method of reducing the number of features or attributes within the dataset which reduces the time taken to train any algorithm, but can be very important for neural networks. The standard neural network with a three layers and 5-15 hidden units performed very well, achieving an error rate of just 2.8% on the test dataset, when coupled with a Pearson rank correlation feature selection.

It is obvious from these approaches that neural methods are capable of classifying this data with very good accuracy. The problem with using neural networks in this manner is that whilst the network itself can be used a method for predicting classifications, the attributes which have been used to make the classifications cannot be easily determined. That is, the model can be used to predict the class, but we cannot easily find the genes which are responsible for that classification. This is especially the case where networks with hidden layers have been used (as in the previous experiments) because the hidden layer can find non-linear relationships between combinations of attributes and classes. This non-linear relationship cannot be easily described in the rule format that has been shown earlier.

3.4 Conclusions

What has been described here are some applications of neural networks to the area of bioinformatics, and in particular their potentially novel use in modelling gene expression data. Neural networks are flexible learning tools which have found applications in a huge variety of domains, and so it should come as no surprise that they have had some success in bioinformatics. Their learning properties and the ease with which they can be applied to new domains ensure that they will continue to be used in many other areas of bioinformatics. ANNs, as can be seen above, are good tools for predicting either the next temporal state of a gene network or the class of an individual or sample. See5 and other symbolic machine learning techniques provide output in the form of decision trees or rules. Doctors using future palm-held computers to help diagnose whether a patient in their surgery is suffering from a disease or not may be perfectly happy with the palm-held computer making an initial prediction concerning a particular disease, given the patient's gene expression profile, without necessarily needing a detailed reason for the prediction. Such palm-held devices can therefore be programmed with ANNs. Referral to expert consultants for more detailed analysis can then be recommended. Drug companies designing new drugs for combating disease may need detailed reasons (in the form of causal or classificatory rules identifying dominant factors) as to why a disease develops and progresses the way it does, at the level of individual genes contributing to the disease. They may be happier with symbolic approaches. Our results indicate that, head-to-head on the same data, ANNs are 'softer' in their approach to modelling the data and also more accurate in many cases with regard to prediction. See5 and symbolic approaches, on the other hand, produce symbolic knowledge and reasons. It is important to use the right tools for the task. While ANNs have been used reasonably successfully in protein folding prediction (see Rost and Sander, 1993; Wu, 1997; Shepherd, 1999), the problem is that biologists also want to know *why* the protein folds in the predicted manner. When only prediction and initial classification are required, however, ANNs have proved extremely successful and useful.

4. Evolutionary Computation

4.1 Introduction

Evolutionary computation is based on the idea of using an (artificial) evolutionary process in order to evolve solutions to complex optimization problems. The advantage of such an approach is that it makes it possible to rapidly find acceptable (although not provably optimal) solutions in very large search spaces. Evolutionary algorithms can also easily be applied in a parallel manner - the population of evolving candidate solutions can be distributed over an array of computer processors working in parallel, which greatly speeds up the time used to find acceptable solutions to very large problems.

From the perspective of bioinformatics, an intriguing aspect of evolutionary algorithms is that they are based on the use biological metaphors. Instead of designing or searching for the solution to our problem we are trying to *evolve* solutions. Just as in natural evolution, the two key ingredients of this process are random variation and selection biased towards better-performing solutions. The art of using evolutionary algorithms therefore lies in how candidate solutions should be represented in such a way that random changes ("mutations" and "crossovers") can be applied, in how to balance the amount of variation against the amount of selection pressure, and how to evaluate the performance of candidate solutions.

In overview, an evolutionary algorithm works by creating a pool ("population") of candidate solutions, testing the quality ("fitness") of each candidate solution, and thereafter creating a new pool of candidate solutions by randomly mixing properties ("genes") from the best candidate solutions from the current population. Figure 14 shows an overview of this process. In the toy example used in the figure, we are trying to evolve a character string containing the correct spelling of the word "sequence" using an extremely small population of only six candidate solutions. At generation time g the population contains six sub-optimal solutions, and we can calculate the fitness of each simply by counting the number of characters occurring in the correct position. Correct characters are marked as bold face underlined letters, and the fitness value is given in the box immediately to the right of each candidate solution.

The algorithm now proceeds by selecting "parents" from which to create new "offspring" solutions for generation $g + 1$. We want fitter parents to get lots of offspring and parents of lower fitness to get fewer offspring. A basic standard method to create this bias is roulette wheel selection, which can be described as using a virtual roulette wheel with the same number of slots as the population size. The slots are sized in proportion to fitness, so that candidate solution number 2 in our example ("SEQVENCA") receives the largest slot, occupying 25% of the whole roulette wheel, since its fitness value of 6 constitutes 25% of the population's total summed fitness of 24. Selecting parents is now performed simply by "spinning" the wheel twice, which in the example happens to result in candidates number 2 and 6 being selected.

After selection of parents, new candidates can be generated by applying "genetic" variation. This is often done by a combination of crossover and mutation operators. In our example the crossover operator cuts the two parent strings at the same randomly chosen position, which happens to fall between the sixth and seventh character (indicated by small arrows). Having thus cut both parent "chromosomes" we proceed to create two offspring by splicing together the left part of the first parent with the right part of the second parent, and the right part of the first parent with the left part of the second parent. We thereafter apply mutation by changing randomly chosen characters in the two offspring to new (also randomly chosen) characters. In our example, the fourth character of the first offspring (V) is mutated into W, and the fifth character of the second offspring (A) is mutated into E.. This concludes the process of generating two new offspring, and these are subsequently placed into the population for generation g+1. For evolutionary algorithms using non-overlapping generations, we would replace the whole current population by new offspring before proceeding to the next generation.

For algorithms using overlapping generations, we would simply let the two new offspring replace two current low-fitness candidates, while keeping the rest of the population unchanged.

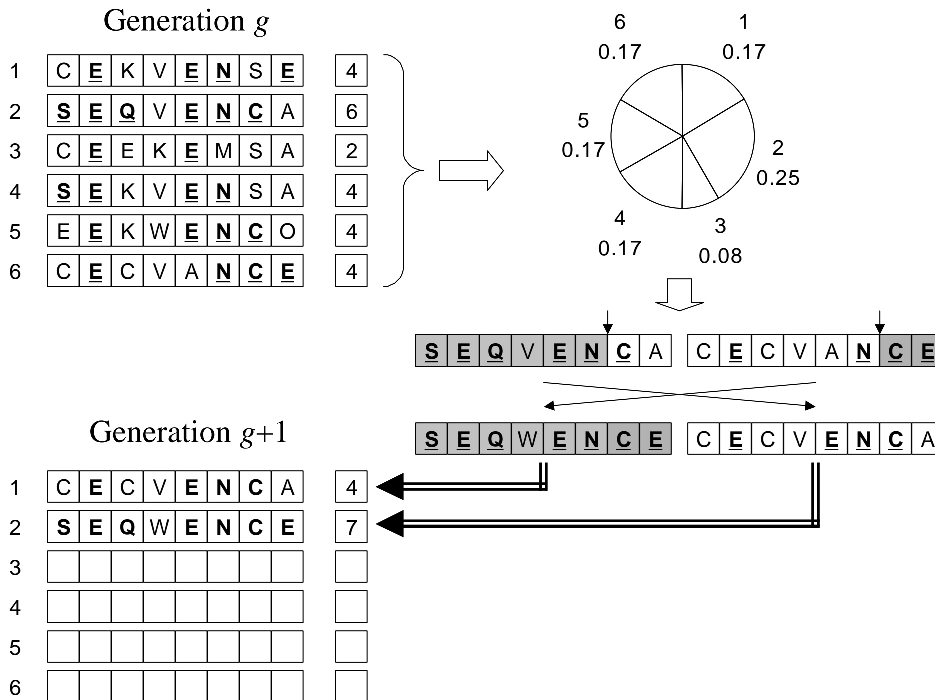


Figure 14: Overview of an evolutionary algorithm. See the text for explanations.

Evolutionary algorithms have a long history, spanning from the earliest proposals of simulated evolution in the 1950s (Box, 1957; Friedman, 1959), over the first ideas for how to use simulated evolution to solve optimization problems (Bremermann, 1962; Rechenberg, 1965; Fogel *et al*., 1966; Holland, 1973), to the present-day plethora of different types of evolutionary algorithms and application. Introductions to the area can be found in Coley (1998), Michalewicz and Fogel (1999), Mitchell (1998), Michalewicz (1996), or Haupt and Haupt (1998). Of special interest for bioinformatics researches will also be the upcoming overview of evolutionary computation in bioinformatics by Fogel and Corne (2002).

Evolutionary algorithms has been a very active research area over several decades, which has resulted in a large number of variations to the general technique. One of major variations is *genetic algorithms* (Holland, 1975; Goldberg, 1989; Vose, 1999), which is based on the use of binary strings as universal representation format for chromosomes. This has the advantages that standard mutation and crossover operators can be designed and used regardless of the application, and that stringent mathematical analysis of the behaviour of the algorithm can be developed. Due to the popularity of genetic algorithms during 1980s and 1990s, it is not uncommon to use the term "genetic algorithm" even when referring to other types of evolutionary algorithms, such as *evolutionary programming*, *evolution strategies*, *genetic programming*, and others. The insistence on the use of binary representations in genetic algorithms does, however, have drawbacks. One of the major drawbacks is that it requires the use of a mapping function from the solution space ("phenotype" space) to the chromosome space ("genotype space"). In many applications it is more natural to use other forms of representations. In our introductory example above, for instance, it was obvious how spellings could be represented naturally as character strings and how suitable crossover and mutation operators could easily be designed. Therefore, it has become increasingly popular to use whichever representation suits the application at hand best, and design genetic operators that facilitates the evolutionary process in the given optimisation problem. The two examples described in the next two sections both use this approach. For an in-depth analysis of the pros and cons of different representations, see Rothlauf (2002).

In recent years, the variation of evolutionary algorithms called genetic programming has become particularly popular. Genetic programming evolves computer programs by using a representation designed in such a way that each individual in the population represents a complete computer program. For overviews of this promising and rapidly growing sub-field of evolutionary algorithms see Banzhaf *et al*. (1998), Poli and Langdon (2002), Koza (1999), or Langdon (2000).

4.2 The SAGA System for Multiple Sequence Alignment by Genetic Algorithm

Multiple alignment is an example of a classical bioinformatics problem where the search space is too large for exact algorithms to be possible. For pairwise alignment, algorithms such as Needleman-Wunsch and Smith-Waterman, which use a dynamic programming approach, do guarantee finding the optimal solution. For multiple alignment, however, exact algorithms based on dynamic programming cannot handle more than five or six sequences, since the solution space increases exponentially with the number of sequences. As a result, all multiple alignment algorithms currently in use depend on different kinds of heuristics.

The most common current solution to the multiple alignment problem is to use a progressive approach (Feng and Doolittle, 1987; Taylor, 1988; Barton and Sternberg, 1994) where more and more sequences are added progressively to a small initial alignment. The most well known example is Clustal (Thompson et al., 1994), which adds progressively more sequences to the alignment according to the branching order of a phylogenetic guide-tree. The heuristic reasoning in Clustal is that first aligning the most similar sequences to each other lowers the risk of mistakes being made in the beginning of the alignment process. This is important, since a progressive alignment algorithm has very little (if any) possibility of correcting mistakes made earlier in the alignment process. By using the progressive alignment strategy in Clustal an efficient implementation is possible, where hundreds of sequences can be aligned in reasonable time. The drawback of the approach, on the other hand, is that the resulting alignment may not be the optimal solution since any mistake during initial alignment steps will result in the final alignment being stuck on a local optimum.

SAGA, Sequence Alignment by Genetic Algorithm (Notredame and Higgins, 1996), deals with the multiple alignment problem from a different perspective. The viewpoint taken in SAGA is that we should design an objective function, i.e. a measure of the overall quality of a multiple alignment, and then attempt to optimise this objective function using a genetic algorithm. An advantage of this reasoning is that we can separate the two parts of the multiple alignment problem - that of designing a biologically valid objective function and that of designing an efficient optimisation algorithm. Once we have found a good objective function, we can apply different optimisation algorithms to it, and once we have found a good optimisation algorithm, we can apply different objective functions to it. For the purpose of this article, we will focus on a description of how a genetic algorithm is used in SAGA. The authors of SAGA have also designed a novel objective function named COFFEE, which has successfully been used with the SAGA algorithm as well as with other multiple alignment algorithms (for a description of COFFEE, see (Notredame et al., 1998)).

SAGA starts the optimisation process by creating an initial population of random alignments. These random alignments are created by simply shifting each sequence to the right by a randomly chosen offset and padding it with gap characters until it has the same length as the number of columns in the alignment. In each generation, SAGA will evaluate all alignments that currently make up the population, and assign each alignment an expected number of offspring depending on its score according to the objective function. Obviously, alignments of high quality should on average have more offspring than alignments of poor quality. The expected number of offspring is calculated according to one of the standard genetic algorithm methods called remainder stochastic sampling without replacement (Mitchell, 1998). In the context of SAGA, this method will typically assign each alignment between zero and two expected offspring.

When moving from generation to generation by creating a new population from the previous one, SAGA uses a so-called overlapping generations technique where only part of the population is replaced by new alignments. SAGA does this by ranking the population according to fitness (alignment quality) and letting the 50% (typically) of highest quality get copied into the new population, and thereafter filling up the remaining 50% of the new population by selecting parents according to fitness and modifying them by applying genetic operators. When selecting parents for these remaining 50% of the new population SAGA applies roulette wheel selection (as described in the precious section).

SAGA uses a total of 22 genetic operators, of which three are crossover operators and 19 are mutation operators. After choosing a parent by applying the roulette wheel selection, one of the 22 operators is chosen according to their probabilities (initially 1/22 for every operator). If the chosen operator happens to be a crossover, a second parent is again chosen by roulette wheel selection before the crossover operator is applied. The probabilities of the different operators are adjusted dynamically during the optimisation process, in such a way that operators that tend to produce improved alignments will be used more often in later generations.

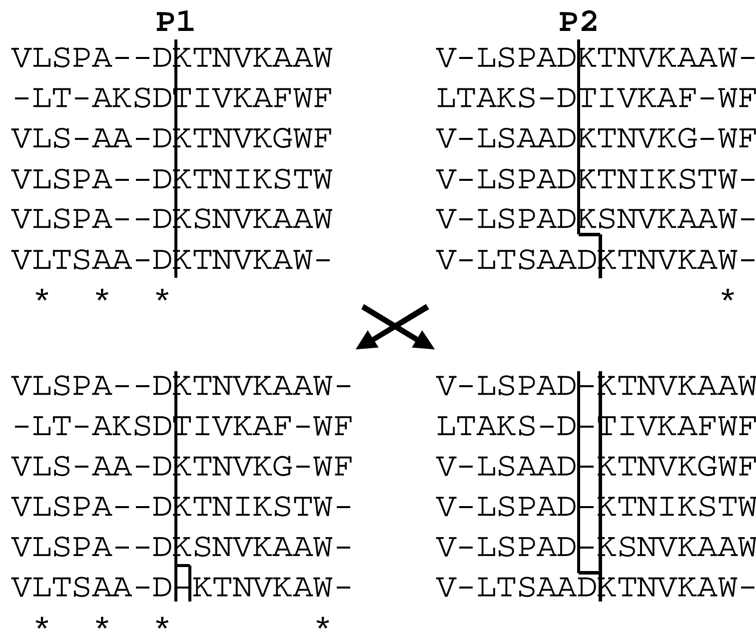


Figure 15: One-point crossover is applied by choosing a random crossover point P1 in the first parent alignment, and cutting the second parent alignment at such a point P2 that every sequence is cut at the same residue as with P1. Two offspring alignments are created by splicing together the left side of the first parent with the right side of the second parent, and the right side of the first parent with the left side of the second parent. Gap characters are inserted to make offspring alignments consistent. In the example shown here, a single gap is introduced in the left offspring alignment, while five gaps are needed to make the right offspring alignment consistent. Note that the left offspring alignment has four conserved columns (indicated by asterisks), three of which originate from the first parent and one from the second parent. Only the offspring alignment with highest fitness, i.e. best objective function score, is added to the new population, while the other one is discarded.

Crossover operators are applied by combining parts of two parent alignments and thereby producing two offspring alignments. One of the crossover operators used in SAGA is illustrated in Figure 15, showing how each parent alignment is cut into two parts so that the parts can be recombined to produce to offspring alignments. The operator has been carefully designed in such a way that the order of amino acids is guaranteed to be preserved. Apart from the one-point crossover shown in Figure 15, SAGA also uses two variants of uniform crossover. These crossover operators are based on the idea of first identifying columns that are consistent between the two parent alignments, i.e. columns in which the same symbols from all sequences are aligned with each other. An offspring alignment is then created by first copying all the consistent columns into the new alignment, and then for each block

between two consistent columns choosing randomly which parent the block should be copied from. Just like one-point crossover, this procedure also guarantees that the order of amino acids in every sequence is preserved. An advantage of uniform crossover over one-point crossover is that it does not introduce any new gaps. As can be seen in Figure 16, applying one-point crossover results in *n* new gap characters in an alignment of *n* sequences.

Mutation operators are applied to a single parent alignment to produce a single offspring alignment. Of the 19 mutations operators used in SAGA, only the gap insertion operator will be described here, as an example. The gap insertion operator inserts a gap of the same length in every sequence. The positions at which the gaps are inserted are chosen in accordance with an estimated phylogenetic tree, in such a way that the sequences in one of the sub-trees will be realigned against the sequences in the other sub-tree.

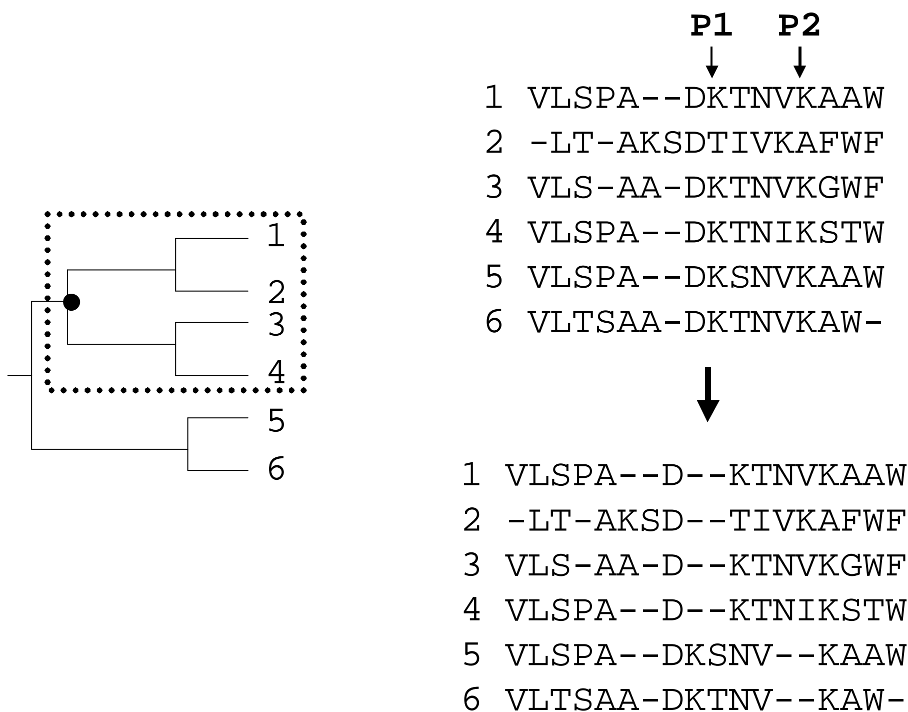


Figure 16: Gap insertion is made by choosing a random node (indicated by the dot) in the estimated phylogenetic tree, and dividing the sequences into two groups accordingly. A first gap insertion point P1 is randomly chosen, and a gap of randomly chosen length (here two) inserted at that point in every sequence belonging to the first group. A second gap insertion point P2 is chosen to fall within a specified maximum distance from P1 and a gap of the same length inserted at that point into every sequence from the second group.

As mentioned earlier, an advantage of the genetic algorithm approach is that the algorithm can be used with any objective function. In the original work on SAGA (Notredame and Higgins, 1996) two objective functions were used which were both based on weighted sums of pairs scoring with affine gap costs. Sums of pairs scoring assigns a score to each pair of aligned residues in every column of the alignment and calculates the overall score of the alignment by adding all of the pairwise scores. When weighting is used, each pairwise score is modified according to the weights of the sequences. Calculation of weights is done according to a phylogenetic tree, in such a way that unusual sequences receive higher weights. The two objective functions used by Notredame and Higgins (1996) in their evaluation of SAGA differed in whether weights were calculated as in the MSA program (Lipman et al., 1989) or as in ClustalW (Thompson et al., 1994). The results achieved by SAGA using the first objective function was compared with those found by MSA, and SAGA's results using the second objective function were compared with those found by ClustalW. In both cases, SAGA was found to consistently find alignments that scored as well or better than those found by the compared algorithm,

albeit at a higher computational cost. When assessing the biological relevance of the alignments by comparison with structural alignments, SAGA alignments consistently had either the same or better correspondence with the structural alignment.

The MSA program that SAGA was compared with works by first narrowing down the search space to a smaller region in which the optimal alignment is likely to be, and then using a strategy that guarantees finding the optimal alignment within this region. Despite this reduction of the search space, MSA is in practice limited to being used on only seven or eight sequences, at the most. MSA and SAGA were therefore compared on rather small sets of sequences. Using nine test sets of five to eight sequences each, it was found that the best alignment found by SAGA had the same sums of pairs score as the best MSA alignment in five cases and a slightly better score in four cases. In the eight test cases where PDB structures were available for the sequences, SAGA's best alignment had the same level of correspondence with the structural alignment as the best MSA alignment did in four cases and a better correspondence in four cases.

Although performing better than the near-optimal MSA for small sequence sets, SAGA is not quite as limited as MSA in terms of the number of sequences. In the comparisons with ClustalW, four sequence sets ranging in size from 10 to 32 sequences, were used. In this case, SAGA found slightly better scoring alignments (according to the sums of pairs score) than ClustalW in all four cases, and in three of the four cases the agreement with the structural alignment was better for the SAGA alignment than for the ClustalW alignment. For these larger sequence sets, however, running times of SAGA did become rather long, with CPU-time ranging from approximately 40 minutes to 11 hours, while none of the ClustalW runs took more than one minute. As pointed out by Notredame and Higgins (1996), however, it is likely that the running time of SAGA could be drastically reduced by seeding the initial population with heuristically derived alignments, rather than with random alignments. Since SAGA consistently found higher scoring alignments than ClustalW, it would, for example be possible to seed the initial population with alignments derived by Clustal and other algorithms, and thereby design an approach that is a hybrid between progressive alignment and genetic algorithm alignment.

4.3 Using Genetic Algorithms to Predict RNA Folding

A second example of a bioinformatics problem with a prohibitively huge search space is prediction of RNA folding. Since an RNA structure is largely formed by base pairs such as C/G, A/U, and G/U, an RNA sequence of *n* bases will have approximately $2^{n/2}$ possible folds. This means that for a 400 base sequence the number of possible folds (i.e. 2^{200}) exceeds the number of atoms in the known universe. Algorithms for prediction of RNA folding therefore either rely on heuristics to prune away large parts of the search space, or on the use of stochastic search algorithms such as simulated annealing or genetic algorithms in order to quickly located promising folds. This section will give a brief overview of the genetic algorithm based approach to RNA structure folding prediction developed by Shapiro and co-workers (Shapiro and Navetta, 1994; Shapiro and Wu, 1996, 1997; Wu and Shapiro, 1999, Shapiro *et al*., 2001). For ease of reference, the approach will here by referred to as the RNA-GA.

A folded RNA structure consists of a particular configuration of stems, strands and loops. A stem is a series of base pairs, of which the most stable ones are the canonical pairs C/G and A/U and the wobble pair G/U. At the ends of the stem there may be free strands or loops, the nucleotides of which are un-paired. Loops come in a number of varieties, such as bulge, hairpin, internal, and multi-branch loops, depending on the local configuration. Figure 17 shows examples of these four kinds of loops.

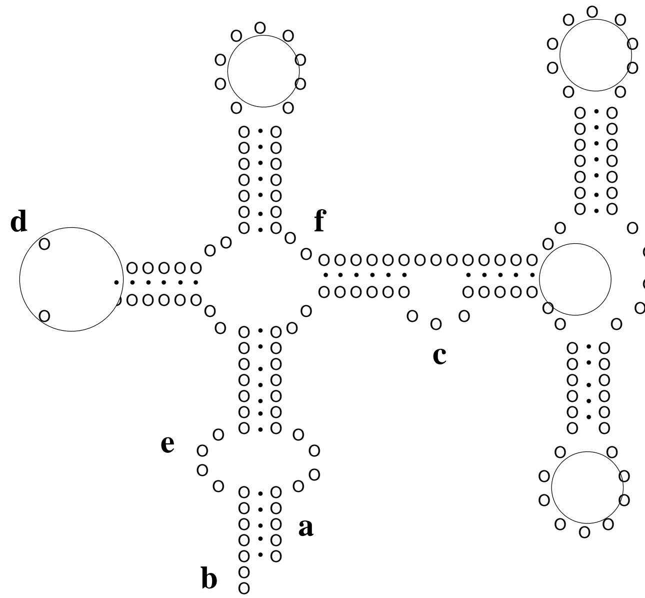


Figure 17: RNA structures may contain stems (a), free strands (b), bulge loops (c), hairpin loops (d), internal loops (e) and multi-branch loops (f). In addition, the bases of a loop may form pairs with bases outside the enclosing stem, which is referred to as a pseudoknot (not shown).

In order to apply a genetic algorithm we need an evaluation function computing the fitness of an RNA structure. For this purpose the RNA-GA computes the change in Gibbs free energy for the structure relative to the unfolded single-stranded RNA molecule. Stems will tend to stabilize the RNA structure, and therefore give negative energy, while loops tend to destabilize the structure, and therefore give positive energy. The energy of a stem depends both on the hydrogen bonds between pairs of bases and the stacking energies resulting from the stacking of base pairs adjacent to each other. Loop energies depend on the size and type of the loop.

The strategy used by the RNA-GA is to consider fully and partially zipped stems as building blocks for evolving RNA structures. A fully zipped stem is a stem that cannot be extended by adding any more complementary base pairs, while a partially zipped stem have some complementary but unpaired bases at the ends. (For examples, see Figure 18a, where the leftmost stem is fully zipped since it cannot be extended with the non-complementary G/A pair at the upper end, while the third stem from the left is partially zipped since it can be extended with the complementary U/A pairs at the ends.) At the beginning of a genetic algorithm run a stem pool is generated, containing all fully and partially zipped stems that can be generated from the given sequence. Since the stem pool can be very large, partially zipped stems are limited to contain two open complementary base pairs at each end of the stem. It should be noted that by randomly picking a number of non-overlapping stems from the stem pool, we can derive an RNA structure by filling in the loops according to the given sequence. The population of the initial genetic algorithm generation is therefore created by restricted random picking from the stem pool. The structures created this way do not have to be complete, i.e. they do not have to cover the full length of the sequence. After creating initial structures, each individual is evaluated by deriving the energy of the structure by applying a set of energy rules.

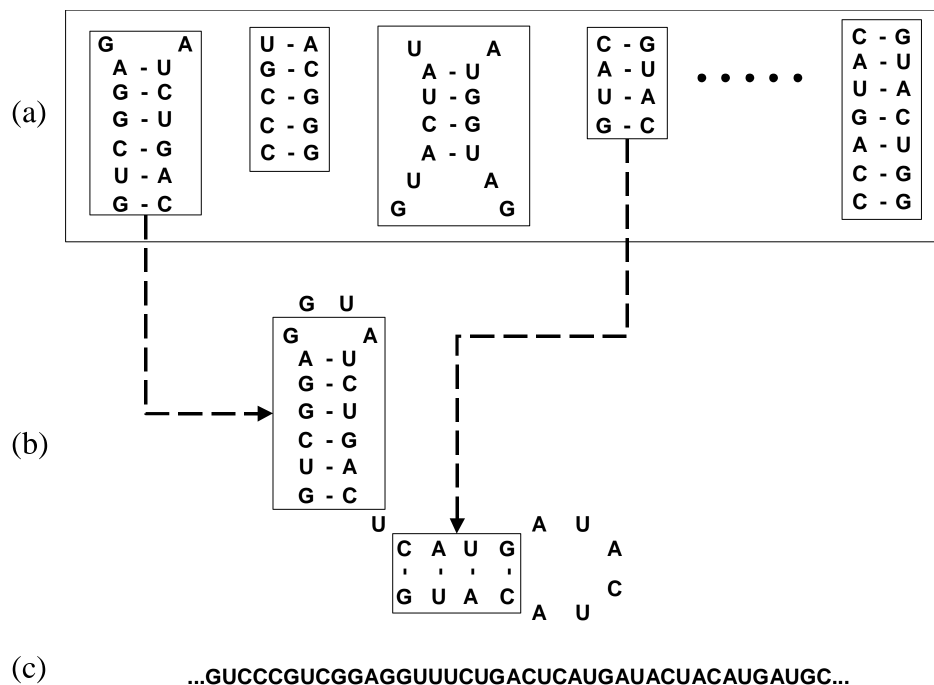


Figure 18: The genetic algorithm initially creates a stem pool (a) consisting of all fully and partially zipped stems that can be created from the given sequence. Structures (b) can be generated by randomly picking non-overlapping stems from the stem pool and connecting these by filling in the intermediate bases from the sequence. Underlined segments of the RNA sequence (c) correspond to the stems that were picked.

The set of genetic operators used in subsequent generations consist of one mutation and one crossover operator. To create offspring structures, two parent structures P_1 and P_2 are first chosen. Thereafter, two incomplete child structures C_1 and C_2 are initialised by randomly picking a number of stems from the stem pool, in just the same way as when creating the initial population. To complete the two child structures additional stems are distributed by crossover from P_1 and P_2 into C_1 and C_2. Finally, whichever of the two completed child structures has the lowest free energy is chosen to be the offspring, and replaces P_1 in the next generation.

Randomly picking a number of stems to be inserted into each child structure can be a very disruptive mutation operator. This can be seen by considering the structure in Figure 18. Applying mutation and crossover may mean initializing a new child structure by picking four randomly chosen stems from the stem pool, and completing it by adding some of the eight stems from the structure in Figure 18, plus some stems from another parent structure (P_2). The resulting structure may, obviously, bear very little resemblance to the original structure. Although this disruptiveness can be useful in the earlier generations of the run, when average folding quality tends to be low, but it will be very destructive in later generations when the average quality is higher. In order, therefore, to make the genetic operators progressively less disruptive during the run, the RNA-GA was extended (Wu and Shapiro, 1999) by the use of a filter in combination with a stochastic relaxation (annealing) algorithm that gradually decreases the probability of unfavourable changes being accepted.

The filter works by calculating the change in free energy of the resulting structure each time a new stem is added. If adding the stem decreases the free energy (i.e. makes the structure more stable), it is simply accepted. If adding the stem increases the free energy (i.e. destabilizes the structure), it has instead a particular probability of being accepted. This probability depends on the amount of change in free energy as well as on the generation count. As the genetic algorithm proceeds, the probability of accepting destabilizing stems is gradually lowered. This results in the algorithm eventually converging to a population with very little diversity, at which point the run is terminated.

An additional feature of the RNA-GA is that the population is distributed over a grid of processors working in parallel. This illustrates an important advantage of genetic algorithms, namely that they can easily be designed to take advantage of massively parallel computing architectures. The original implementation of the RNA-GA was developed for the MasPar MP-2 using an array of 16 384 processor elements executing instructions in parallel. Therefore, the population size was set to 16 384 RNA structures so that every processor element could carry out all operations (such as calculation of free energy - a very computation intensive part of the run) for a single structure and communicate only with immediate neighbour processors.

Figure 19 illustrates selection of parents in this parallel architecture, where the P_1 parent is located at the central one of the nine marked processors. Parent P_2 is chosen from the surrounding eight adjacent processors. The choice of P_2 is biased towards neighbour structures with better fitness values. After creation of child structures and selection of which child will become the offspring, P_1 is replaced by the offspring. The whole selection process is carried out simultaneously in parallel for all processors in each generation.

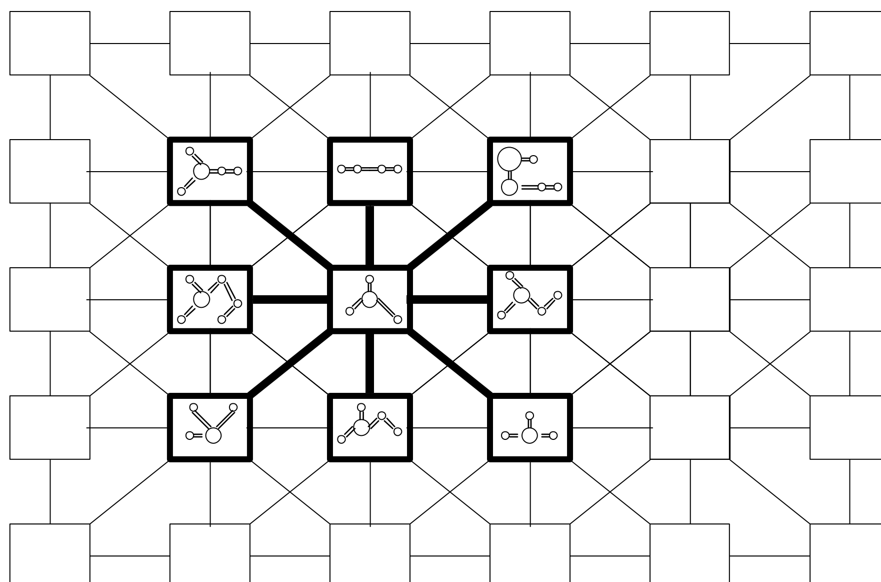


Figure 19: RNA folding prediction on a parallel computer architecture where each processor holds one structure and communicates only with its adjacent neighbours. Selection, mutation, and crossover are carried out in parallel in all 16 384 processors in each generation of the genetic algorithm. Adapted from Shapiro *et al.* (2001).

4.4 Conclusions

The application of genetic algorithms to RNA folding is reasonably well understood (e.g. Shapiro and Navetta, 1994; Shapiro *et al.*, 2001). They have also been used in the optimisation of simulations of biochemical and genetic networks (Mendes and Hoops, 2001) and in the use of proteomic patterns in serum to identify ovarian cancer (Petricoin *et al.*, 2002). Also, Steffanini and Camissi (2000) used a genetic algorithm to extract salient features for classification from crop molecular profiles. The potential of GAs for predicting protein folding was recognised in the early 1990s (e.g. Dandekar and Argos, 1992; Patton *et al.*, 1993; Unger and Moult, 1993; Dandekar and Argos, 1996) and more recently by Krasnogor *et al.* (1999). A recent and potentially exciting development is the use of

genetic algorithms for drug design (Raymer, 2001). Also, there have been recent developments concerning the extraction of gene regulatory networks and models from gene expression profiles (Ando and Iba, 2001).

A growing list of evolutionary computational techniques used in bioinformatics is provided by Evoweb, the website of EvoNet (European Network of Excellence in Evolutionary Computing).[13] A number of tutorials now exist to help bioinformatics researchers understand the basic principles of genetic algorithms (e.g. eBioinfogen[14], Flying Circus[15]). One of the problems faced by researchers wishing to use genetic algorithms is the lack of appropriate, free and maintained software, unlike the availability of SNNS and See5. By and large, specific GA software has to be written for the problem at hand. While some GA developers' toolkits do exist on the web, they tend not to be maintained by a team of developers and programmers. It is to be hoped that this situation will be rectified soon, since this is probably the biggest obstacle to further applications of GAs in bioinformatics.

5. Conclusion

The application of AI techniques in bioinformatics is still at an early, unexplored stage. Yet, the examples provided in the paper demonstrate the tremendous potential of AI techniques to make useful and practical contributions to our knowledge in this domain. Freely available software helps bioinformatics researchers get their hands on the appropriate tools (e.g. See5 and SNNS). There is an initial overhead in getting to know how to use these tools, but the results can certainly be worth the effort. While the review covers the most popular AI techniques, it should not be forgotten that there are numerous other techniques which could have been covered also, such as Hidden Markov Models, Bayesian Networks and Probabilistic Reasoning, Fuzzy Logics, Temporal Logics, Spatial Logics (for protein folding, for instance), Pattern and Image Recognition Algorithms (for microarray and protein chip analysis), and so on. Nor have we covered Alife, that area of AI which allows groupings of individual elements, such as cells, into larger elements, such as tissues and organs, through communication only with neighbouring cells. Hidden Markov Models are an example of where an AI technique (HMMs were originally used for speech processing) has become a mainstream bioinformatics technique through the demonstration of practical examples of their use when dealing with biosequences. It can therefore be expected that, as researchers increasingly turn their attention to other AI techniques, these also will become part of mainstream bioinformatics in the near future.

Bioinformatics also provides AI researchers with an opportunity to test their techniques in a new domain and to modify these techniques in the context of the problem (as happened with HMMs, where the original speech processing HMM architecture was modified to make it more amenable to biosequence analysis). The difference between previous problem domains that AI has dealt with and bioinformatics is that data is growing at an incredibly fast rate, with our ability to analyse and model the data being stretched to its limit. We need to be better equipped to analyse the growing volume of data in an informed and intelligent manner, and that ultimately is what AI can do for us as bioniformaticians.

Acknowledgements

The authors are extremely grateful to the anonymous referees for the valuable and constructive remarks concerning the first draft of this paper.

[13] Evoweb can be accessed at http://evonet.dcs.napier.ac.uk/. Searching the site with 'bioinformatics' will return a number of hits to recent articles.
[14] http://www.ebioinfogen.com/other_tuto.htm.
[15] http://evonet.dcs.napier.ac.uk/evoweb/resources/flying_circus/index.html

References:

For Section 2 (Symbolic machine learning)

Altman, R. (2001) Challenges for intelligent systems in biology. *IEEE Intelligent Systems* November/December 2001: 14-18.

Ankerst, M., Kastenmüller, G., Kriegel, H.-P. and Seidl, T. (1999). Nearest neighbour classification in 3D protein databases. *Proceedings of the 7th International Conference on Intelligent Systems for Molecular Biology (ISMB'99),* AAAI Press.

Brazma, A. (1999). Mining the yeast genome expression and sequence data. Available from http://bioinformer.ebi.ac.uk/newsletter/archives/4/lead_article.html.

Brazma, A. and Jonassen, I. (1998). Sequence pattern discovery. Tutorial given at ISMB'98. Available from http://www.ebi.ac.uk/~brazma/tutorial3/index.htm.

Brazma, A., Jonassen, I., Vilo, J. and Ukkonen, E. (1998) Predicting gene regulatory elements from their expression data in the complete yeast genome. *Proceedings of the German Bioinformatics Conference, GCB'98*: 3pp.

Holsheimer, M. and Siebes, A. (1991). Data mining: the search for knowledge in databases. Differerent downloadable versions of this report are available via http://citeseer.nj.nec.com/holsheimer91data.html.

Kumar, V. and Joshi, M. (1999) Tutorial on high performance data mining. Available at http://www-users.cs.unm.edu/~mjoshi/hpdmtut/.

Levin, J. M., Robson, B. and Garnier, J. (1986). An algorithm for secondary structure determination in proteins based on sequence similarity. *Federation of European Biochemical Sciences (FEBS)* 4001, 205 (2), 303-308.

Michener, C. D. and Sokal, R. R. (1957) A quantitative approach to a problem in classification. *Evolution* 11, 130-162.

Mitchell, T. (1997) *Machine Learning.* McGraw Hill.

Narayanan, A., Wu, X. and Yang, Z. R. (2002). Mining viral protease data to extract cleavage knowledge. *Bioinformatics* 18, Supplement 1, S5-S13.

Quinlan, J. R. (1987) Simplifying decision trees. *International Journal of Man-Machine Studies*, 27, 221-234.

Quinlan,J. R. (1993) *C4.5: Programs for Machine Learning*. Morgan Kauffman.

Ralescu, A. and Tembe, W. (2000) A nearest neighbor clustering algorithm for gene expression data based on iterative sampling. Talk presented at the Critical Assessment of Microarray Data Analysis 2000 meeting. Abstract and presentation slides available from http://www.camda.duke.edu/CAMDA00/abstracts.asp (paper 26).

Salamov, A. A. and Solovyev, V. V. (1995) Prediction of protein secondary structure by combining nearest neighbor algorithms and multiple sequence alignments. *Journal of Molecular Biology* 247: 11-15.

Sneath, P. and Sokal, R. R. (1973) *Numerical Taxonomy.* W H Freeman and Co.

Street, W. N., Mangasarian, O. L. and Wolberg, W. H. (1995) An inductive learning approach to prognostic prediction. *Proceedings of the Twelfth International Conference on Machine Learning,* A. Prieditis and S. Russell, eds., pages 522-530, Morgan Kaufmann.

Winston, P. H. (1992). *Artificial Intelligence (3rd Edition).* Addison Wesley.

Wolberg, W. H., Street, W. N. and Mangasarian, O. L. (1994) Machine learning techniques to diagnose breast cancer from fine-needle aspirates. *Cancer Letters* 77: 163-171.

Yi, T.-M. and Lander, E. S. (1993) Protein secondary structure prediction using nearest neighbor methods. *Journal of Molecular Biology* 232: 1117-1129.

For Section 3 (Neural networks)

References

Azuaje, F. (2001) An unsupervised neural network approach for discovery of gene expression patterns in B-cell lymphoma. *Online Journal of Bioinformatics* 1: 26-41.

Cai,Y.D. and Chou,K.C. (1998) Artificial neural network model for predicting HIV protease cleavage sites in protein. *Advances in Engineering Software,* 29(2): 119-128.

Golub, T. R., Slonim, D. K, Tamayo, P., Huard, C., Gaasenbeek, M., Mesirov, J. P., Coller, H., Loh, M. L., Downing, J. R., Caligiuri, M. A., Bloomfield, C. D. and Lander, E. S. (1999) "Molecular Classification of Cancer: Class Discovery and Class Prediction by Gene Expression Monitoring" *Science* 286: 531-536

D'Haeseleer P., Liang S., Somogyi R., (1999) "Gene Expression Analysis and Modelling", *Proceedings of Pacific Symposium on Biocomputing, Hawaii, (PSB99).* Available from www.cgl.ucsf.edu/psb/psb99/genetutorial.pdf

Keedwell E., Narayanan A. and Savic, D.A. (2002) "Modelling Gene Regulatory Data Using Artificial Neural Networks" *Proceedings of the International Joint Conference on Neural Networks (IJCNN'02),* Honolulu, Hawaii: 183-188.

Kohonen, T. (1982) A simple paradigm for the self-organized formation of structured feature maps. In S. Amari and M. Arbib (Eds.) *Competition and Cooperation in Neural Nets*, Lecture Notes in Biomathematics, Springer Verlag.

Liang, S., Fuhrman, S., Somogyi, R. (1998) "REVEAL, A General Reverse Engineering Algorithm for Inference of Genetic Network Architectures" *Pacific Symposium on Biocomputing* 3: 18-29

Rost, B. and Sander, C. (1994) Prediction of protein secondary structure at better than 70% accuracy. *Journal of Molecular Biology* 232: 584-599.

Rumelhart, D.E., McClelland, J.L. and the PDP Research Group (1986) *Parallel Distributed Processing: Volume 1 Foundations* The Massachusetts Institute of Technology

Ryu, J., Sung-Bae, C., (2002) “Gene expression classification using optimal feature/classifier ensemble with negative correlation” *Proceedings of the International Joint Conference on Neural Networks (IJCNN’02), Honolulu, Hawaii*: 198-203,

Shepherd, A. (1999). Protein secondary structure prediction with neural networks: A tutorial. Available from http://www.biochem.ucl.ac.uk/~shepherd/sspred_tutorial/ss-index.html.

Su, T., Basu, M., Toure, A., (2002) “Multi-Domain Gating Network for Classification of Cancer Cells using Gene Expression Data” *Proceedings of the International Joint Conference on Neural Networks (IJCNN’02), Honolulu, Hawaii*: 286-289.

Wu, C. H. (1997). Artificial neural networks for molecular sequence analysis. *Computers and Chemistry* 21 (4): 237-256.

For Section 4 (Genetic algorithms)

Ando, S. and Iba, H. (2001) Inference of gene regulatory models by genetic algorithms. *Proceedings of the 2001 IEEE Congress on Evolutionary Computation*: 712-719.

Banzhaf, W., Nordin, P., Keller, R.E. and Francone, F.D. (1998). *Genetic Programming*: *An Introduction*. Morgan Kaufmann.

Barton, G.J. and Sternberg, M.J.E. (1987). A strategy for the rapid multiple alignment of protein sequences: Confidence levels from tertiary structure comparisons, *Journal of Molecular Biology*, 198(2): 327-337.

Box, G.E.P. (1957). Evolutionary operation: a method for increasing industrial productivity. *Appl. Stats*., 6: 81-101.

Coley, D.A. (1998). *An Introduction to Genetic Algorithms for Scientists and Engineers*. World Scientific, Singapore.

Bremermann, H.J. (1962). Optimization through evolution and recombination. In Yovits, M.C., Jacobi, G.T., and Goldstein, G.D. (eds.), *Self-Organizing Systems*, Spartan Books, Washington D.C.

Dandekar, T. and Argos, P. (1992). Potential of genetic algorithms in protein folding and protein engineering simulations. *Protein Engineering* 5: 637-645.

Dandekar, T. and Argos, P. (1996) Identifying the tertiary fold of small proteins with different topologies from sequence and secondary structure using the genetic algorithm and extended criteria specific for strand regions. *Journal of Molecular Biology* 256: 645-660.

Feng, D-F. and Doolittle, R.F. (1987). Progressive sequence alignment as a prerequisite to correct phylogenetic trees, *Journal of Molecular Evolution*, 25(4): 351-360.

Fogel, G.B. and Corne, D.W. (eds.) (2002). *Evolutionary Computation in Bioinformatics*. Morgan Kaufmann.

Fogel, L.J., Owens, A.J., and Walsh, M.J. (1966). *Artificial Intelligence through Simulated Evolution*. John Wiley, New York.

Friedman, G.J. (1959). Digital simulation of an evolutionary process. In *General Systems: Yearbook of the Society for General Systems Research*, Vol. 4, 171-184.

Goldberg, D.E. (1989). *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley.

Haupt, R.E. and Haupt, S.E. (1998). *Practical Genetic Algorithms*. Wiley-Interscience.

Holland, J.H. (1973). Genetic algorithms and the optimal allocation of trials. *SIAM J. Comp*., 2:88-105.

Holland, J.H. (1975). *Adaptation in Natural and Artificial Systems*. MIT Press.

Koza, J.R. (1999). *Genetic Programming III*: *Automatic Programming and Automatic Circuit Synthesis*. Morgan Kaufmann.

Krasnogor, N., Hart, W. E., Smith, J. and Pelta, D. A. (1999) Protein structure prediction with evolutionary algorithms. *GECCO-99: Proceedings of the Genetic and Evolutionary Computation Conference.* W. Banzhaf, J. Daida, A.E. Eiben, M.H. Garzon, V. Honavar, M. Jakaiela, and R.E. Smith, editors.

Langdon, W.B. (2000). *Genetic Programming and Data Structures*. Kluwer.

Lipman, D.J., Altschul, S.F. and Kececioglu, J.D. (1989). A tool for multiple sequence alignment, *Proceedings of the National. Academy of Science* 86(12): 4412-4415.

Mendes, P. and Hoops, S. (2001) Simulation of biochemical and genetic networks. Virginia Bioinformatics Institute. Available from http://www.sun.com/products-n-solutions/edu/events/archive/hpc/presentations/june01/MendesHPC.pdf.

Michalewicz, Z. and Fogel, D.B. (2002). *How to Solve It*: *Modern Heuristics*. Springer-Verlag, Berlin.

Michalewicz, Z. (1996). *Genetic Algorithms + Data Structures = Evolution Programs*. Springer-Verlag, Berlin.

Mitchell, M. (1996). *An Introduction to Genetic Algorithms*. MIT Press.

Notredame, C., Holm, L. and Higgins, D.G. (1998). COFFEE: an objective function for multiple sequence alignments, *Bioinformatics*, 14(5): 407-422.

Notredame, C. and Higgins, D.G. (1996). SAGA: Sequence alignment by genetic algorithm, *Nucleic Acids Res*., 24(8): 1515-1524.

Patton, A. L., Punch III, W. F. and Goodman, E. D. (1995) A standard GA approach to native protein conformation prediction. *Proceedings of the 6th International Conference on Genetic Algorithms*. L. J.Eshelman, ed., pp. 574-581.

Petricoin, E. F. III, Ardekani, A. M., Hitt, B. A., Levine, P.J., Fusaro, V.A., Steinberg, S.M., Mills, G.B., Simone, C., Fishman, D. A., Kohn, E. C. and Liotta, L. A. (2002) Use of proteomic patterns in serum to identify ovarian cancer. *Lancet* 359: 572-577. Available from http://image.thelancet.com/extras/1100web.pdf.

Poli, R. and Langdon, W.B. (2002). *Foundations of Genetic Programming*. Springer.

Raymer, M.L. (2001) Bioinformatics and drug design. *Information Technology Research Institute (ITRI), Wright State University, Ohio, 2001 Spring Workshop.* Powerpoint presentation available from http://www.cs.wright.edu/itri/EVENTS/workshop-Spr01-ppts/raymer.ppt.

Rechenberg, I. (1965). Cybernetic solution path of an experimental problem. Royal Aircraft Establishment, Farnborough, U.K., Library Translation No. 1122, August.

Rothlauf, F. (2002). *Representations for Genetic and Evolutionary Algorithms*. Springer.

Shapiro, B. A. and Navetta, J. (1994) A massively parallel genetic algorithm for RNA secondary structure prediction. *Journal of Supercomputing* 8: 195-207.

Shapiro, B.A. and Wu, J.C. (1996). An annealing mutation operator in the genetic algorithms for RNA folding, *Computer Applications in the Biosciences* 12: 171-180.

Shapiro, B.A. and Wu, J.C. (1997). Predicting RNA H-Type pseudoknots with the massively parallel genetic algorithm, *Computer Applications in the Biosciences* 13: 459-471.

Shapiro, B. A., Wu, J.-C., Bengali, D. and Potts, M. J. (2001) The massively parallel genetic algorithm for RNA folding: MIMD implementation and population variation. *Bioinformatics* 17(2): 137-148.

Stefanini, F. M. and Camussi, A. (2000) The reduction of large molecular profiles to informative components using a genetic algorithm. *Bioinformatics* 16 (10): 923-931.

Taylor, W.R. (1988). A Flexible Method to Align Large Numbers of Biological Sequences, *Journal of*

Molecular Evolution, 28: 161-169.

Thompson, J.D., Higgins, D.G. and Gibson, T.J. (1994). CLUSTAL W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice, *Nucleic Acids Researcg* 22: 4673-4680.

Unger, R. and Moult, J. (1993) Genetic algorithms for protein folding simulations. *Journal of Molecular Biology* 231: 75-81.

Vose, M.D. (1999). *The Simple Genetic Algorithm*: *Foundations and Theory*. MIT Press.

Wu, J.C. and Shapiro, B.A. (1999). A Boltzmann filter improves RNA folding pathway in a massively parallel genetic algorithm, *Journal of Biomolecular Structure and Dynamics* 17: 581-595.