CLASSIFICATION OF PROTEIN STRUCTURES

A FRAMEWORK FOR AND TEST OF CLASSIFICATION OF PROTEIN STRUCTURES BASED ON TOPS DIAGRAMS

Department of Informatics, University of Bergen

Cand. Scient. Thesis

Narve Sætre

December 1999

CONTENTS

1. **Introduction** 1
   - 1.1. BACKGROUND 1
   - 1.2. AIMS 2
   - 1.3. METHODS 2
   - 1.4. IMPLEMENTATION 3
   - 1.5. ORGANISATION OF THE THESIS 3
2. **Biological background** 4
   - 2.1. THE BIOLOGICAL MACROMOLECULES 4
     - *2.1.1. DNA and RNA 4*
     - *2.1.2. Amino acids 5*
     - *2.1.3. Proteins 7*
   - 2.2. PROTEIN STRUCTURES 7
     - *2.2.1. Structures at different levels 7*
     - *2.2.2. The protein folding problem 10*
     - *2.2.3. 3d-models and visualisation 11*
   - 2.3. PROTEIN CLASSIFICATION 11
   - 2.4. CATH 12
     - *2.4.1. General 12*
     - *2.4.2. Classification Levels 13*
3. **TOPS** 14
   - 3.1. THE TOPS SYSTEM 14
     - *3.1.1. Cartoons 14*
     - *3.1.2. Diagrams 15*
   - 3.2. DETAILED DESCRIPTION 16
     - *3.2.1. Formal Definition 16*
     - *3.2.2. Fixed Structures 17*
     - *3.2.3. TOPS Patterns 17*
     - *3.2.4. The TOPS Database 19*
4. **Artificial Neural Networks** 21
   - 4.1. MATHEMATICAL CONCEPTS AND NOTATION 21
   - 4.2. INTRODUCTION TO ARTIFICIAL NEURAL NETWORKS 22
     - *4.2.1. History and definition 22*
     - *4.2.2. The Basic Neurone 22*
     - *4.2.3. The learning process 24*
   - 4.3. NEURAL NETWORK ARCHITECTURES 24
     - *4.3.1. General 24*
     - *4.3.2. Feed-forward networks 26*
     - *4.3.3. Recurrent networks 26*
   - 4.4. LEARNING ALGORITHMS 26
     - *4.4.1. General 26*
     - *4.4.2. Supervised learning 27*
     - *4.4.3. Unsupervised learning 27*
   - 4.5. IMPORTANT EXAMPLES OF NEURAL NETWORKS 28
     - *4.5.1. Perceptrons, Least-Mean-Square and Error Back-Propagation 28*
     - *4.5.2. Systems based on competitive learning 29*
     - *4.5.3. Modular networks and VLSI 31*

5. The Self-Organising Feature Map **32**
5.1. SOFM ARCHITECTURE 32
5.2. SOFM ALGORITHM 33
5.2.1. General *33*
5.2.2. Initialisation *34*
5.2.3. Neighbourhood functions *34*
5.2.4. Updating functions *35*
6. Classification and Clustering **36**
6.1. INTRODUCTION AND NOTATION 36
6.2. VECTOR QUANTIZATION AND THE VORONOI TESSELLATION 37
6.3. LEARNING VECTOR QUANTIZATION 38
6.3.1. Introduction *38*
6.3.2. Initialisation of the LVQ *38*
6.3.3. Updating *39*
6.3.4. Discussion of the algorithms *39*
6.4. CLUSTERING 40
6.4.1. Overview *40*
6.4.2. Clustering methods and a generic clustering process *40*
6.5. HIERARCHICAL CLUSTERING 41
6.5.1. General *41*
6.5.2. A simple hierarchical clustering algorithm *41*
6.6. THE ISODATA ALGORITHM 42
6.7. COMPARING CLASSIFICATIONS 43
6.7.1. General *43*
6.7.2. The rand statistic and similar indices *44*
6.7.3. Probability distributions and Monte Carlo estimations *45*
7. Description of the Classification System **46**
7.1. OVERVIEW 46
7.2. THE TOPS PACKAGE 47
7.2.1. Overview *47*
7.2.2. Internal abstract representations *48*
7.2.3. Extracting quantitative information – the Quantifier *49*
7.2.4. Selections *50*
7.2.5. Statistics *50*
7.2.6. Data files *50*
7.3. THE SOFM PACKAGE 51
7.4. THE LVQ PACKAGE 51
7.5. THE CLUSTER PACKAGE 52
7.5.1. Overview *52*
7.5.2. Hierarchical clustering *52*
7.5.3. Distance measures *53*
7.5.4. The Isodata class *53*
7.5.5. Comparing classifications *54*
8. Results **55**
8.1. OVERVIEW 55
8.1.1. Selections and general procedures *55*
8.1.2. Quantifiers *56*
8.2. TOPOLOGICAL MAPS 57
8.3. LVQ CLASSIFICATIONS 61
8.4. CLUSTERING 65
9. Summary and Discussion **68**
9.1. OVERVIEW 68
9.2. DISCUSSION 68
9.3. FURTHER WORK 69
10. References **70**
Appendix A: Programs **72**

FIGURES

Figure 2-1: Generic amino acid 5

Figure 2-2: Amino acid chain 7

Figure 2-3: Alpha helix and beta sheet 8

Figure 2-4: Primary, secondary, tertiary and quaternary structures 9

Figure 3-1: Rasmol picture and corresponding TOPS cartoon 15

Figure 3-2: TOPS diagram 16

Figure 3-3: TOPS pattern 18

Figure 3-4: Linearised TOPS pattern and matching linearised diagram 18

Figure 4-1: A basic neurone 23

Figure 4-2: Simplified figure of a neurone 24

Figure 4-3: Architectural view of a neural network 25

Figure 4-4: Single-layered lattice network 30

Figure 4-5: Input patterns as points on a surface 30

Figure 5-1: Simple SOFM network 33

Figure 5-2: Topological map 34

Figure 6-1: Voronoi tessellation 37

Figure 7-1: Overview of the classification system 47

Figure 7-2: The TOPS package 48

Figure 8-1: Topological map of all domains 58

Figure 8-2: Topological maps of mainly-beta domains 59

Figure 8-3: Topological maps of sandwich topologies 60

Figure A-1: Central data structure 72

Figure A-2: Modules in the TOPS package 73

TABLES

Table 2-1: The genetic code table 6
Table 8-1: Supervised classification of all classes 61
Table 8-2: Supervised classification of mainly-beta diagrams........ 62
Table 8-3: Supervised classification of sandwich diagrams........ 64
Table 8-4: Unsupervised classification of all diagrams 65
Table 8-5: Unsupervised classification of mainly-beta architectures........ 66
Table 8-6: Unsupervised classification of barrel topologies 66
Table 8-7: Monte-Carlo approximation of Hubert Γ statistic 67

1. INTRODUCTION

1.1. Background

Understanding and using proteins is a vital area of research within the ever-more important fields of biology and biotechnology. Despite considerable efforts, predicting the full, three-dimensional structure of a given protein based on its components remains a persistent challenge to biochemistry. Although a variety of approaches display appreciable progress, this *protein folding problem* – as it is standardly referred to – seems unlikely to be resolved in the near future.

The proteins consist of chains of amino acids, simple organic acids that are connected to each other in long chains. The process of determining which amino acids a protein consists of is called *sequencing*. As new proteins are sequenced and analysed and their description added to central databases, several problems arise. One is the amount of information – already databases contain information on several thousands of more or less related proteins. Another and perhaps more serious problem is the organisation of all this information. As of today, there is no universally accepted classification of proteins into different categories and subcategories. Proper categorisations are important and will become even more so, for various reasons.

This problem is complicated by the fact that proteins often are divided into several distinct structures, connected by chains of unstructured amino acids. Often these structures repeat in the different proteins, and might serve the same purpose. For this reason, attention lately has been focused not only on proteins as a whole but also on the different *protein structures*.

There are several ways of classifying protein structures, these will be discussed in section 2.3. A new system that has received some attention lately is the CATH classification system. In this system, protein structures are classified at five different levels: C*lass*, *Architecture*, *Topology*, *Homologous super-family* and *family*. Methods for automatically classifying structures are being developed, although at some levels classification is still done semi-automatically or manually.

Another important tool is visualisations of the three-dimensional structure of proteins. The structure is essentially three-dimensional, and no two-dimensional maps can show the full structure. However, due to the recurring elements (beta-strands and alpha helices) and recurring patterns of organisations of these, two-dimensional maps can still be used to show the basic, three-dimensional organisation of secondary-level structures. TOPS is a system that uses a formal representation of a graph structure to construct two-dimensional maps – *cartoons* – suitable for this purpose. Moreover, this system can also be used for comparisons, accelerating searches and identifying recurring patterns. Algorithms for converting structural information from current databases to TOPS cartoons have already been developed.

TOPS cartoons are stored as formally represented directed graphs, using a special version of the constraint-programming language *clp*. This makes their representations very small, and thus easily searchable. In addition, *templates* – generalisations of structures – can be defined, making fast pattern matching and pattern searching possible.

1.2. Aims

In this thesis, a framework for the classification of protein structures based on their TOPS representation will be presented. Various classifying methods, both *unsupervised* and *supervised*, will be discussed and implemented. As a basis for comparison, the CATH system will be used; its classifications will be the target to which classifications and maps will be compared. Other classifications systems could be included as well.

At a more detailed level, the central part is a set of *functions* that, given a TOPS diagram, create a numerical vector. The vectors thus created will then be used as basis for several classification methods:

- Unsupervised learning, or *self-organising*, will be used to create *maps* of the structures. These maps will show the similarities between the different entries by clustering similar entries near each other, geometrical distances indicating differences between structures and clusters of structures.
- A supervised classification technique will be used for a more precise classification of all diagrams into categories.
- Unsupervised classification or *clustering* will be used to create classifications and to reveal patterns in the vectors.

The maps will be manually compared to the classification done by CATH. If successful, structures grouped together in the same category in CATH should be placed near each other in the map. A method for selecting TOPS diagrams according to their corresponding classification in CATH will be implemented. Methods for automatically comparing different classifications will also be implemented, and will be used to compare the obtained classifications to a "correct" target classification. The "correct" classifications in this thesis will be based on CATH, but other targets are possible, based on other existing classification systems. These approaches will be tried at different levels of categorisation in the CATH hierarchy, and in several categories at each level.

1.3. Methods

The methods used will mainly be based on *Artificial Neural Networks* (ANNs) or similar approaches. ANNs have been used successfully in many cases where algorithmic solutions are hard to find or undesirable for other reasons. Unsupervised learning, used properly, often enables the network to find patterns in complex data. However, when it comes to categorisation relative to a given (known) system or classification, supervised methods generally perform better and are better suited. Although the training phase of a neural network may be long, running data through a network after training is usually very efficient.

The first task is to extract numerical information from the given TOPS cartoons. This functionality will be implemented in a software package, referred to as the TOPS package henceforward.

To create the map of protein structures (or rather, of TOPS cartoons), we will use a Self-Organising Feature Map (SOFM), also known as a SOM or a Kohonen network, after its inventor Teuvo Kohonen. This network architecture will be described in detail in a later chapter.

Learning Vector Quantization (LVQ) is a supervised classification technique, and will be used as the main classification method. Based on the Vector Quantization method, LVQ is also invented by Kohonen and has been applied successfully to several different areas within pattern classification.

We will also experiment with standard clustering techniques. A measure of the distance between two given diagrams will be implemented; this measure will be based on the vectors created by the rules mentioned earlier. Standard hierarchical clustering and the isodata (k-means) algorithm will then be used to perform unsupervised clustering of TOPS diagrams.

After the final categorisation process, the methods will be tested on protein structures *not* used in training the network or in the categorisation process. Success will depend on these structures being classified according to the classification done by CATH, in an efficient and generalizable manner.

1.4. Implementation

Java[1] has been chosen for programming language, and the choice of programming language deserves further explanation. Using Java in scientific work is rather uncommon, mainly because of its inefficiency compared to other structured or object-oriented languages like C++. It is also a recent language, with which the scientific community may not be accustomed. However, programming in Java usually leads to rapid development cycles, and for the tasks at hand its efficiency is sufficient. The built-in serialisation capability of Java will be used to provide object persistence. Other bonuses by using Java are cross-platform compatibility and ease of maintenance and distribution.

For the SOFM and LVQ tasks, the software packages developed by Kohonen and his co-workers will be used. The packages are called SOM_PAK and LVQ_PAK, respectively, and are more or less the reference implementation for both methods.

1.5. Organisation of the thesis

The emerging field of bioinformatics, or computational biology, is inherently cross-disciplinary. This thesis, again, is cross-disciplinary within bioinformatics, and correspondingly needs a thorough background. Chapters 2 and 3 cover the necessary biological background, including TOPS cartoons and their formal representation. Chapter 4 introduces the terms and definitions necessary to understand the theory behind neural networks. Building on the notations and concepts presented there, chapter 5 will then describe the Self-Organising Feature Map algorithm while chapter 6 discusses pattern classification and the Learning Vector Quantization method.

Having introduced the foundation on which this thesis is based, chapter 7 will describe the classification framework in detail, and glue the different parts together. In chapter 8, the results of applying this classification system to the current TOPS database are compared to the CATH classification system. Chapter 9, drawing conclusions and proposing directions for future work, concludes the thesis.

[1] Java Development Kit 1.2, available at http://java.sun.com

2. BIOLOGICAL BACKGROUND

In this chapter a brief summary of the biological aspect of the thesis is given. First, a description of the important biological molecules is presented in section 2.1**.** We then have a closer look into proteins and their compositions, structures and functions, in section 2.2**.** Section 2.3 discusses classification of proteins and protein structures, and lists the most important systems and methods. In addition, the protein-folding problem is explained. Finally, in section 2.4, the classification system named CATH is described in detail. Later in the thesis, CATH will be used as a basis for comparisons of classifications.

Most of the material on proteins and the biological macromolecules presented in this chapter is based on "Proteins: Structures and Molecular Properties" by Thomas E. Creighton [1].

2.1. The Biological macromolecules

2.1.1. DNA and RNA

We find the *Deoxyribo-Nucleic Acids* (*DNA)* in all cells of all known life forms. The DNA is macromolecules that store the genetic information of an individual. They consist of a double chain of smaller molecules: Adenine, guanine, cytosine and thymine; these are called *bases* and usually abbreviated as A, C, G or T. When we consider DNA molecules, the bases are connected to each other in long chains. The carbon atoms of the bases thus form a central *backbone*, and the side chains of these bases are usually denoted *nucleotides*. The two chains interact with each other and form the famous *double helix spiral*, discovered by Crick and Watson in 1953.

The reason DNA is important is that it is used as a blueprint for creating *proteins*. Proteins are essential in virtually all operations cells perform; for instance they transport oxygen, digest food, repair damaged cells and act as muscles. We will say more about proteins in paragraph 2.1.3, for now just note that they consist of chains of *amino acids*.

The DNA inside cells are found in *chromosomes*. Humans have 23 pairs of chromosomes, but the number varies from species to species. Not all parts of the DNA molecule are important – large sections are never used in the protein synthesis and thus serve no clear biological function; these sections are usually called *junk DNA*[1].

A contiguous sequence of nucleotides that are actually used in the protein synthesis – *expressed* in biological terms – is called a *gene*. Not all genes are expressed in all cells; which genes *are* expressed in a cell is determined by what *type* of cell it is. The processes that make different cells express different genes are quite complex and still not completely understood.

---

[1] Theories put forth by various researchers suggest that junk DNA have a function in accelerating mutational effects – ie, it acts as a *mutagen*.

The letters A, C, G and T are often called the *alphabet* of the DNA, as the genetic information is coded in those four letters. The information is extracted through complex chemical reactions, but basically three letters are grouped together at a time, and that group, called a *codon*, then either identifies an amino acid, or marks the start or the end of a *sequence*. A gene may then be considered *a contiguous stretch of DNA that codes for a protein*. This picture is not entirely correct[1], but then in biology few simplifications are, as the complexity of biology gives little room for simplifications.

The DNA molecule is not used directly in creating proteins; it is first transformed into another kind of molecule called ribonucleic acid or RNA. The RNA is much like DNA, despite some differences; the most important chemical differences are that RNA (usually) does not form double helixes, and that thymine is replaced with uracil (abbreviated U). The process that creates RNA from DNA is called *transcription*. During this process, only regions of DNA that code for genes are used. Thus, the RNA contains the genetic information of the DNA but none of the "junk". The RNA is then used to create proteins in a process called *translation*. For details about the creation of proteins, the *protein synthesis*, see [1].

2.1.2. Amino acids

There are more than 100 known amino acids. Twenty of these are used in the construction of proteins and only those are considered here. An isolated amino acid consists of a central carbon atom, called *the alpha carbon* or C_α, to which[2] is connected a hydrogen atom, an amino group, a carboxyl group and a *side chain*, see fig. 2-1. The side chains are what makes distinct amino acids different. When chained together, the amino acids are called *residues*, because the chaining process is a *condensation*, producing water molecules and residues.

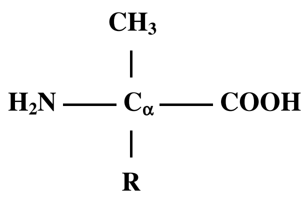

Figure 2-1: Generic amino acid

Schematic depiction of a generic amino acid. In the simplest case, alanin, the side chain R is simply a hydrogen atom.

---

[1] For instance, some genes do not code for proteins, but for RNA molecules.

[2] This does not apply to all amino acids – in some of them, the side chain and the carboxyl group are connected to different carbon atoms. It does, however, apply to 19 of the 20 amino acids that form proteins. A special case is *prolin,* where the side chain is also bonded to the nitrogen atom.

As mentioned, during the translation each codon codes for a specific amino acid. This three letter coding is universal – with very few exceptions, all organisms follow the same mapping from codons to amino acids. This mapping, often called *the genetic code*, is given in table 2-1.

Different side chains have different chemical properties, all of which are important in the formation of protein structures. A sample property is the degree of *hydrophobicity* – hydrophobic side chains generally occur in the core of a protein, while hydrophilic side chains usually are found on the surface. These properties are important when trying to predict protein structures, or when trying to define a measure of *distance* between amino acid sequences.

First position	**Second position**				Third position
	U	C	A	G	
U	Phe Phe Leu Leu	**Ser** **Ser** **Ser** **Ser**	Tyr Tyr Terminate Terminate	Cys Cys Terminate Trp	U C A G
C	Leu Leu Leu Leu	Pro Pro Pro Pro	**His** **His** **Gln** **Gln**	Arg Arg Arg Arg	U C A G
A	Ile Ile Ile Met	Thr Thr Thr Thr	Asn Asn Lys Lys	Ser Ser Arg Arg	U C A G
G	Val Val Val Val	Ala Ala Ala Ala	Asp Asp Glu Glu	Gly Gly Gly Gly	U C A G

Table 2-1: The genetic code table

The genetic code table shows what a combination of three amino acids – a codon – codes for in the translation process. Three-letter shortcuts for amino acids are used. Note the pattern in the redundancy – this explains how simple mutations in the DNA sequence often lead to no changes in the proteins, because several three-letter combinations code for the same amino acid. Met *is special, as AUG also signals the start of an expressed gene sequence, while UAA, UGA or UAG signals the end. Since the genetic code applies to RNA, T is replaced with U for uracil (see main text for explanation).*

2.1.3. Proteins

Chemically, proteins are polypeptide chains of amino acids, see fig. 2-2. Sometimes two or more chains combine to form a single protein. The number of residues varies from less than hundred to several thousand, with three hundred as a typical value. The sequence —N— C_α —CO— is repeated for each residue, producing the *main chain* or the *backbone*[1] of the protein, to which the side chains of the amino acids are attached. As can be seen, the backbone has two ends that are easily distinguishable, and it can thus be considered *directed*. The convention is that a protein starts at the amino group (*N-terminal*) and end at the carboxyl group (*C-terminal*).

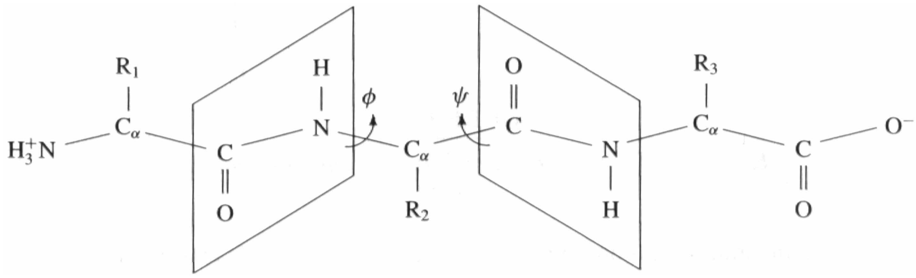

Figure 2-2: Amino acid chain

Amino acids appear in polypeptide chains, forming a central backbone of C_α - and N-atoms. The atoms can rotate with respect to each other – within certain restrictions – thus the backbone is able to form complex structures. From [2].

2.2. Protein structures

2.2.1. Structures at different levels

Proteins generally form a compact and complex three-dimensional structure. The sequence of amino acids that comprise a protein is called its *primary structure*. The main chain folds around itself, creating a *secondary structure*. This secondary structure is formed by reactions between *nearby* backbone atoms. Due to the chemical structure of peptide bonds, the secondary structure consists of repetitions of basic building blocks called *secondary structure elements* (*SSE*s). Only a few SSEs are likely to form, and only two of them are common enough to be of interest when describing protein structures. These are called *alpha helices* (hereafter *helices*) and *beta strands* or *extended elements* (hereafter *strands*). Between SSEs, the backbone has a sequence of non-connected residues. Such a sequence is called a *loop*, a *turn* or a *coil*.

Helices are spirals formed by the backbone binding to itself, and can be described by the average number of residues per turn, normally 3.6. A "standard helix" has about four to forty residues, but variants differ both in number of residues per turn and total number of residues.

---

[1] Not to be confused with the backbone of the DNA molecules.

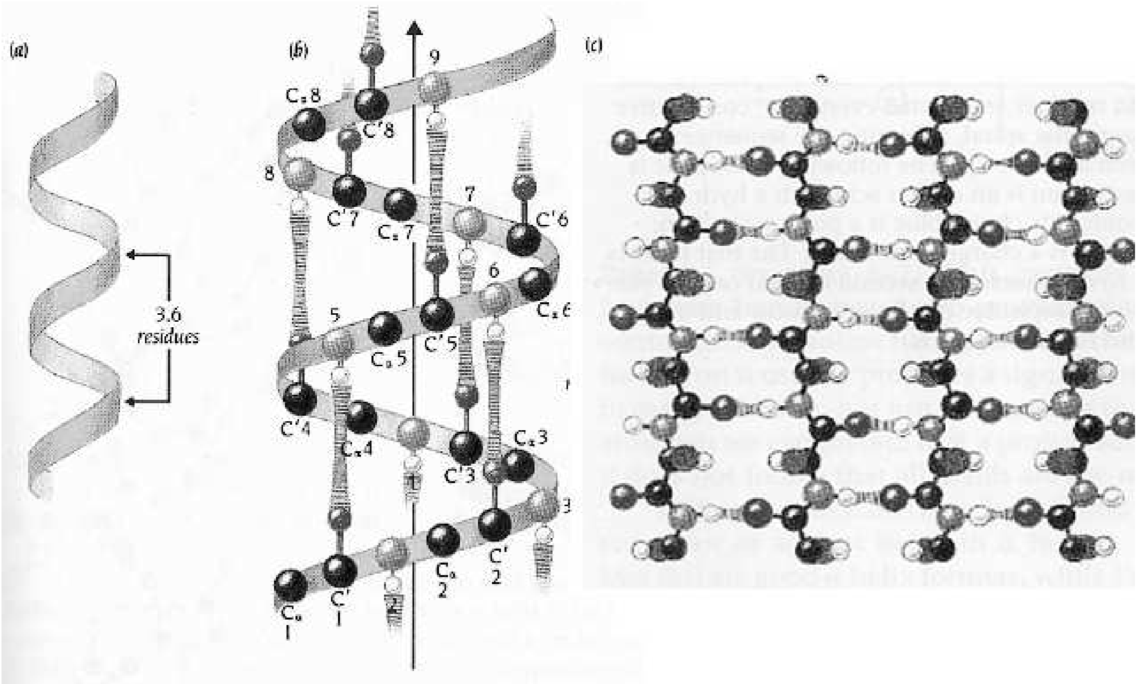

Figure 2-3: Alpha helix and beta sheet

A part of the protein backbone forming an alpha helix is here shown if four different ways. (a) Alpha helices are spiral structures with an average of 3.6 residues per turn. (b) A closer look at the alpha helix, side chains left out for clarity. Hydrogen bonds, shown red and striated, form between nitrogen and oxygen atoms. (c) An anti parallel beta sheet consisting of four beta strands. Note the alternating directions: The first strand to the left has direction top-down. Hydrogen bonds are depicted as striated bonds between oxygen and (the hydrogen atom connected to) nitrogen. Adapted from [3].

The other important SSE, beta strands, can be described as a special type of helix with 2.0 residues per turn. Strands by themselves are not stable, so strands are always found connected together in a *beta sheet*. We say that a beta strand has a direction: It “points” along the backbone (i.e., the start of a strand is the end that is nearest to the N-terminal of the backbone). The strands comprising a sheet can all be pointing the same way (called a *parallel* sheet), they can point the opposite way of their neighbour(s) (*anti parallel* sheet) or they can be of a *mixed* nature. Human proteins show a strong bias towards parallel sheets. See fig. 2-3 for a simplified depiction of the two secondary structure elements.

The protein is folded further, and the relative positioning of the SSEs are called its *tertiary structure*. The tertiary structure is more complex than the secondary, but some regularity can be found. For instance, we find simple combinations of SSEs occurring in different proteins and several times in the same protein. These combinations are often called *motifs* or sometimes *super-secondary structures*; examples are helix-loop-helix and helix-sheet-helix. Some motifs have a clear biological function, others do not. These super-secondary structures are formed by hydrogen-bindings between different SSEs that not necessarily are found near each other on the backbone. Tertiary structure prediction is complicated by the fact that the tertiary structure is often formed by connections between atoms far from each other along the backbone.

Proteins tend to form one or more compact structures that to a certain extent are self-contained. These complex structures, referred to as *domains*, usually (but not always) consist of only one or at most a few segments of the polypeptide backbone, and are linked to the other domains by a single stretch of the backbone. There is no consistent definition of domains, and researchers may divide a protein into domains differently. Paragraph 2.4.1 mentions some automated methods for domain assignment. Domains tend to be stable if cut loose from the rest of the backbone, and are able to fold back to its initial configuration if stretched. This is often used as a definition, i.e., a domain is a sub-component of a protein that is self-contained, stable and able to fold by itself. It should be emphasised, though, that this definition is not stringent, as some domains are unstable and/or unable to fold if left to them self. Note that some domains have a clear biological function – contains an *active site* – while others do not. For notational purposes, each protein is considered to consist of one or more domains and a set of domains make up a protein.

Whenever the protein consists of more than one chain of amino acids, the relative placements of these chains are called the proteins *quaternary structure*. Remember that a domain may involve several stretches of a chain; for instance the protein *1ppn* consists of two chains, and the first chain contains two domains. The first domain consists of residues 1 to 11 and 111 to 212, while the second consists of residues 12 to 109. Some residues are not in any domain at all (e.g. 110 in the previous example), these (stretches of) residues are called *fragments*.

One interesting property of protein structures is that, despite their complex folding, the protein backbone does not form a well-defined knot, i.e. if stretched the backbone always becomes a linear structure[1]. The reason for this is unclear.

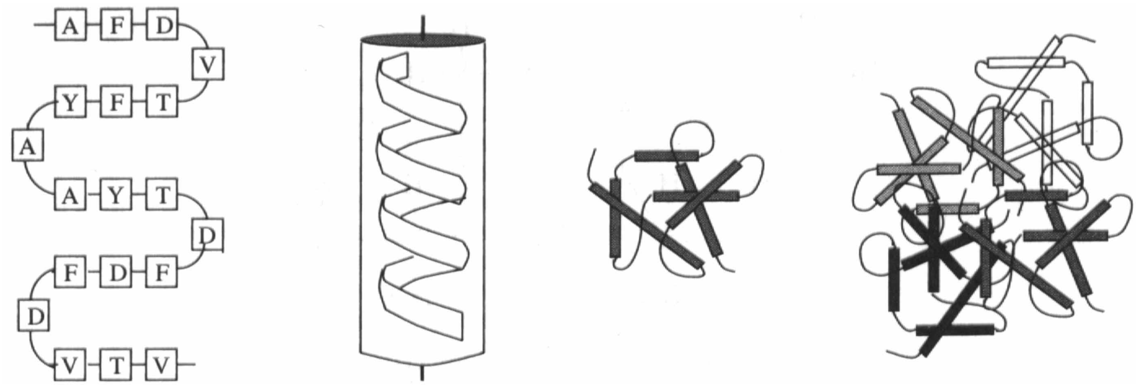

Figure 2-4: Primary, secondary, tertiary and quaternary structures

This diagram shows protein structures at the four levels, from amino acid sequences (primary structure) to complex, self-contained domains *(quaternary structure). From [2].*

[1] Exceptions exist, but these usually involve only small "knots" at the extreme ends of the backbone.

2.2.2. The protein folding problem

Experimental evidence shows that a protein that is unfolded in vitro immediately will fold back to its original three-dimensional structure[1]. This means that given the amino acid sequence of a protein – its primary structure – the full structure is uniquely determined. The structure of a protein determines its function to a large degree, by deciding which substances the protein can bind to. Finding the primary structure of a protein is easy, either by reading the corresponding DNA/RNA sequence and looking up the genetic table, or by sequencing the protein directly. Deciding the secondary structure of a protein is harder, and tertiary structure harder still.

Since the early 1950s, scientists have been working on this task: *How can one determine the exact structure of a given protein?* There are two kinds of methods in use: Those that try to find the structure directly, by observational means, and those that try to predict the complete structure given the primary structure, i.e. by theoretical means. Unfortunately, none of these methods has been perfected and they are still being improved upon.

Of observational methods, two are currently in use: X-ray crystallography and Nuclear magnetic Resonance (NMR). X-ray crystallography has been used since approx. 1910 to determine the position of atoms in crystal structures, and can be applied to proteins as well. Although the technique has improved, it is still complicated, as is evidenced by it sometimes being called an art, rather than science. Some proteins – especially those with highly hydrophobic components – have never been crystallised, and so cannot be studied except by NMR.

NMR works by sending radio waves through substances placed in a magnetic field. The waves affect the spin of the atomic nuclei of the substances, and when the radio wave is turned off, the nuclei releases pulses of energy. These pulses can be used to determine the relative positioning of atoms (and thus of the molecules). NMR is also known as Magnetic Resonance Imaging (MRI) in its medical applications.

Unfortunately, none of these two methods work perfectly. They can only be applied to some proteins, and the accuracy is not always high enough. In addition, it is a costly and time-demanding process.

There are two reasons for wanting to be able to predict the tertiary structure based only on the primary: One is that sequencing proteins, or translating from DNA/RNA to amino acid sequences, is so simple and efficient. Additionally, this ability would enable one to easily model new, synthetic proteins *in silico,* and, in long terms, design proteins with desired properties. For these reasons, lots of research has been devoted to solving this problem, referred to as the *protein folding problem*.

There are several ways to approach this problem: One is simulating the chemical reactions taking place between atoms and molecules, and select among the possible configurations. Another approach, *threading*, is to compare the primary sequence to other proteins with known structures, and to guess a tertiary structure based on the structure of those proteins, and possibly on additional data.

In determining secondary structure, several methods have been tried, including neural networks and pattern-matching methods. Currently the best methods have approximately 70% success rate [4] , which is good enough for some purposes but not completely satisfactory. When it comes to tertiary and quaternary structure, progress has been even less so.

---

[1] Some proteins require auxiliary molecules – e.g. iron atoms in the case of haemoglobin – to fold correctly.

The largest database of known structures is the *Protein Data Bank* (*PDB*)[1] and the latest release (16th November 1999) had 11065 entries. PDB, being the largest and most recognised database of determined structures, is used as a basis for several other databases focusing on structures, including, CATH.

2.2.3. 3d-models and visualisation

In medical and biological sciences the function of proteins are an important field of study. To see relationships between proteins, functions, active sites etc. the scientists need to be able to study the full three-dimensional structure of a protein. Several ways of representing structures have been developed. Some of them are purely ways of visualising a complex structure, others are designed to permit database searches and automated comparisons based on structural topology and recurring elements. Molscript/Ribbons/Chime are examples of the first. The representations operate on several levels of abstraction; some work at molecular level (displaying nucleotides), and some at secondary or super-secondary level (displaying only SSEs or only motifs). In general, the higher the level of abstraction, the easier it is to comprehend the full structure of a protein. On the other hand, some information is lost. Which representation to use depends on the task at hand.

Note that in order to support automatic topology-based searching and comparison, *invariance* of some kind is important; it is desirable for the representation to be one-to-one, i.e. each structure is represented by only one diagram. Most representations are not so – i.e., several different diagrams can represent the same structure.

Perhaps the most common representation is *Molscript*, which represents SSEs by arrows and cylinders. Molscript is viewed with a suitable program such as *Rasmol*, which displays the structure on a computer screen and allows the user to zoom and rotate the protein at will. Another representation is TOPS cartoons, which display SSEs and their relationships in two dimensions, but also imply their three-dimensional positioning. Lately TOPS has been formalised, and its representation can now be used for automated comparisons and searching as well. TOPS will be discussed further in chapter 3.

2.3. Protein classification

Out of the approximately 30.000 proteins found in humans, only few have been adequately described. Many of them exhibit large similarities, both in structure and function, and are naturally viewed as members of the same group. In other cases, small differences in the primary structure lead to large differences in tertiary structure and in function. More and more proteins are being sequenced and described in detail, as part of different research projects. To handle all this information, biologists and computational biologists have devised ways of classifying the proteins into distinct classes or groups.

There are many approaches to the classification of proteins. One consideration is exactly what to classify; Full (possible multi-chained) proteins, single chains, or single domains. Another is which properties to include: Should the focus be on biological function, chemical properties, similarities in primary structure, or on something else? Small changes in choice of algorithms lead to major changes in the final categories.

[1] Maintained by *Research Collaboratory for Structural Bioinformatics*, formerly *by Brookhaven National* Laboratory, this database is available at http://www.rcsb.gov.

Currently there is no universal, commonly accepted classification. Different research areas use different systems, and this will likely remain so. Two of the most common structural classification systems in computational molecular biology are *Structural Classification of Proteins* (*SCOP)* and CATH.

SCOP is a hierarchical structure-based classification of all proteins in PDB. Not only complete proteins are classified, sometimes individual domains are treated as classification units. CATH will be described further in section 2.4.

Both systems are currently semi-automated. Ideally, a classification should be completely automated and based upon primary structure only. Amongst other, this would enable researches to design new proteins more efficiently, performing fewer actual experiments. While the protein folding problem remains unsolved, this seems rather utopian. However, automated classification for *determined* protein structures should be possible, and work is in progress in this area.

2.4. CATH

2.4.1. General

CATH is a hierarchically organised database of protein *domain* structures, sponsored and maintained by Biomolecular Structure and Modelling Unit, University College, London. Data are taken from the PDB database, and only non-empty domains solved to a resolution better than 3.0 Angstroms are considered, together with NMR structures. The name derives from the initial letters of the four major classification levels: *Class*, *Architecture*, *Topology* and *Homologous superfamilies*.

The latest version of CATH (version 1.6) has 7703 PDB entries, which includes 13103 chains and 18577 distinct domains. CATH can be accessed and searched via Internet (http://bsm.bioc.ucl.ac.uk/cath) and the entire database can be downloaded.

The entries in the PDB database say nothing about domains or other structural units. To separate the protein chain(s) into distinct domains, a consensus method is used: The results of three different algorithms for domain recognition (DETECTIVE, PUU and DOMAK) are compared. If the algorithms all yield the same domain assignment, those are classified. If the algorithms disagree, manual inspection is needed to determine a suitable domain assignment. Currently approximately 53% of the proteins have domains assigned automatically.

CATH uses different classification methods at the different levels. Generally, a score and an overlap ratio are computed using global comparison algorithms, and if the domain matches one or more domains in a given category, it is added to that category. To avoid over-population, some categories use different cut-offs than the other categories at the same level. The cut-offs have been determined empirically, on a smaller number of structures, and are currently under reconsideration. This will probably affect the categories of future releases of CATH.

2.4.2. Classification Levels

At the top level, each domain belongs to one out of four *classes*. The class is determined according to the secondary structure composition. Using the method of *Michie et a l*[5] it is determined automatically for approx. 90% of the structures - the remaining domains are grouped manually.

The classes are:

- **Mainly alpha** Domains with mainly alpha helixes and few beta strands.
- **Mainly beta** Domains with mainly beta strands and few alpha helixes
- **Alpha and beta** Domains with both beta strands and alpha helixes
- **Other** Domains with few secondary structure elements or irregular structures (i.e., those domains that do not belong to any of the previous classes)

Each class is further subdivided into *architectures*; for instance, the *Mainly beta* class is divided into 18 architectures. Determination of architecture is achieved through manual inspection of the overall shape of the domain.

The next level is *Topology*, or *fold family*. At this level the grouping is achieved automatically using SSAP scores [6], and the focus is on the connectivity of the SSEs.

The last major level is *Homologous superfamily*. Here domains that are thought to share a common ancestor are grouped together automatically. SSAP scores and sequence identity are considered, along with domain function.

Additionally, each superfamily is further divided into *sequence families*, based on sequence identities. Domains clustered in the same group at this level have at least 35% sequence similarity, as well as the same overall domain structure.

To each group at each level is assigned a number as well as a more describing name. The classification of a given domain is then given as a set of numbers separated by dots, e.g. *1hxn* is classified as *2.110.10.10.1* (class no. 2, architecture no. 110 etc.). In similar manner *2.110.10* specifies the *Hemopexin* fold family in the *4-Propellor* architecture.

3. TOPS

This chapter describes TOPS, a system for representation of protein structures. Section 3.1 introduces TOPS cartoons, the simplest representations in the TOPS system, and TOPS diagrams, a specialisation of cartoons that display more information. A brief summary of the origin and status of TOPS is also given. In section 3.2 formal definitions are given and the diagrams are explained in details. The format of the database used as a basis for the classification system is then described.

Most of this chapter is based on articles published by David Gilbert and David Westhead et al [9]. The following description follows their notation.

3.1. The Tops System

3.1.1. Cartoons

TOPS Cartoons are a simplified way of representing protein structures. The TOPS system was originally used for hand-drawings [7], but an algorithm for automatic generation of cartoons was formulated by Flores et al in 1994 [8]. Tops cartoons have recently been formalised by Gilbert, Westhead et al [9]. They introduced the concept of diagrams, and created a database of all protein structures in PDB.

In TOPS cartoons, the structure of a protein is shown in two dimensions, and only SSEs are displayed. Each alpha helix is displayed as a circle, and each beta strand as a triangle. Length and structure of turns are discarded, as are the size (length) of SSEs. Loops are represented by lines connecting two SSEs. In addition, the N-terminals and C-terminals are displayed for each chain. Normally each domain is shown separately.

The direction of the SSEs relative to the protein fold are also shown, through the connecting loops: If the connection is drawn to the centre of a SSE it points downward into the plane; if the connection is drawn to the boundary of a SSE it points upward, out of the plane. For beta strands, direction is also shown more directly: An up triangle points upward and vice versa. See figure 3-1.

When constructing cartoons, domains must be assigned somehow. In the current database, this is done using domain assignments from CATH, described in section 2.4.1.

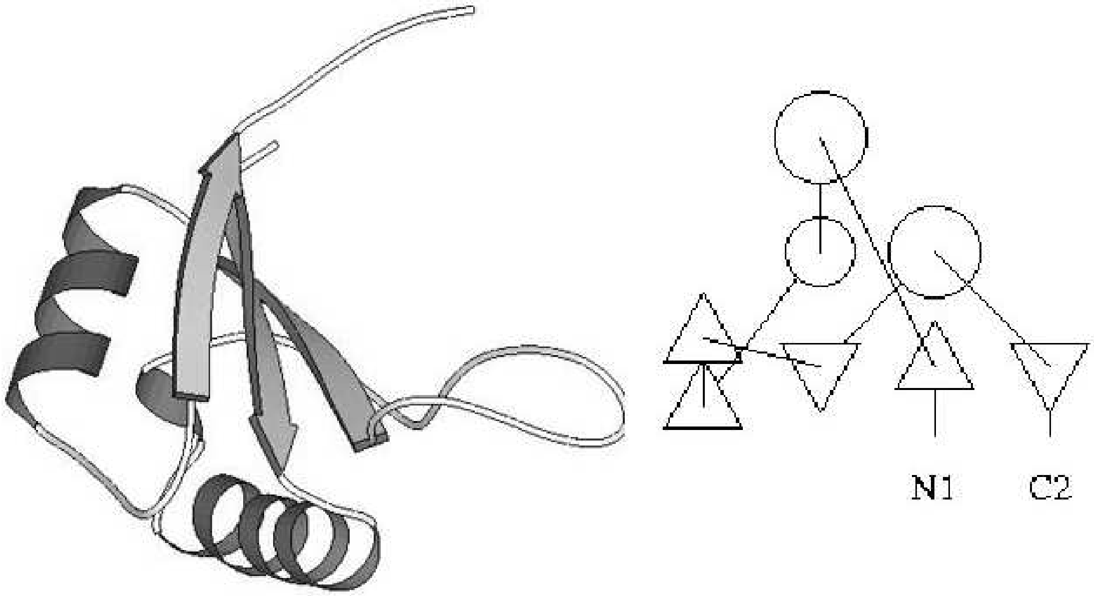

Figure 3-1: Rasmol picture and corresponding TOPS cartoon

Rasmol diagram and TOPS cartoon of the 2bop protein domain, consisting of five beta strands (directed arrows, not all of them are visible), connected in a sheet, and three alpha helices.

3.1.2. Diagrams

A *TOPS diagram* is a formalisation of a TOPS cartoon. A diagram contains more information than a cartoon – it also describes the nature of the hydrogen bindings between SSEs, and the chirality between some of the SSEs. This information is used to create cartoons, but is not showed explicitly except in diagrams.

Hydrogen bindings (H-bonds) can be either parallel or anti-parallel, shown by either A or P above a dotted line connecting two SSEs. Hydrogen bindings are implicitly assumed present between any two beta strands situated next to each other in a strand. Chiralities can be either right-handed or left-handed, shown by L or R above dotted connecting lines.

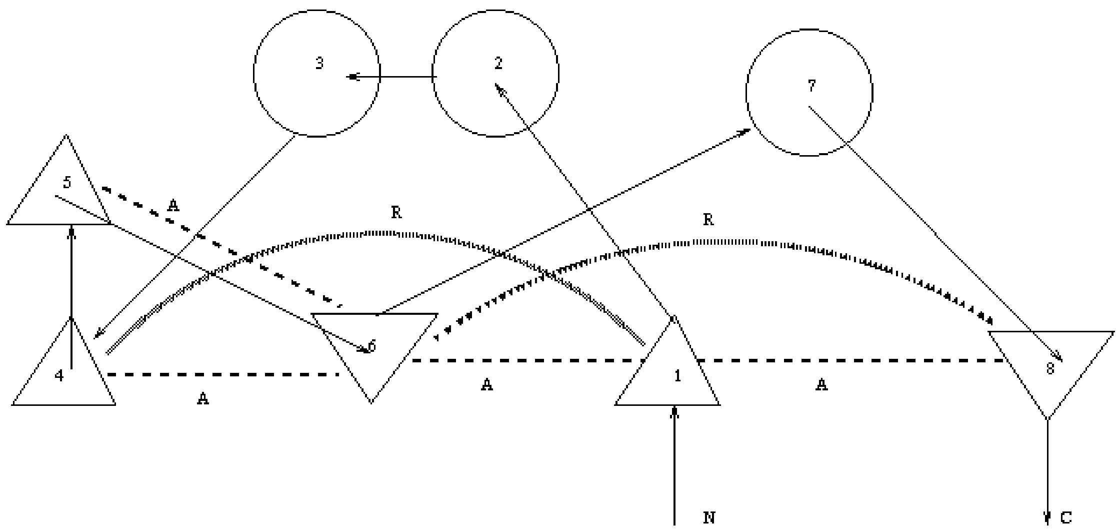

Figure 3-2: TOPS diagram

A TOPS diagram showing the same protein structure as the cartoon in figure 3-1, the 2bop1 *protein. Note the anti parallel hydrogen bindings between neighbouring beta sheets and the two right-handed chiralities present.*

3.2. Detailed description

3.2.1. Formal Definition

Formally a TOPS-diagram is a triple $D = (E, H, C)$, where $E = (SSE1, SSE2, \ldots, SSEn)$ is a sequence of secondary structure elements, $H \subseteq \{ (s_1, d, s_2) \mid s_1,s_2 \in E \wedge d \in \{A, P\} \}$ and $C \subseteq \{ (s_1, h, s_2) \mid s_1,s_2 \in E \wedge d \in \{L, R\} \}$, that is, H and C are sets of binary labelled relations over the SSEs.

E describes the SSEs. Each SSE is a beta strand (*e*), an alpha helix (*h*) or a terminal (*C* or *N*). To strands and helixes are associated a direction as well; +1 for upward and –1 for downward.

H represent hydrogen bonds – each relation represents a ladder of individual hydrogen bonds between adjacent strands in a sheet. *Direction* can be either parallel (*P*) or anti parallel (*A*).

C represent chiralities – *handedness* says whether it is right-handed (*L*) or left-handed (*R*).

The diagram *D* in fig. 3-2 could thus be represented as:

$D = (E, H, C)$

$E = (SSE0{=}N, SSE1{=}e(+1), SSE2{=}h(-1), SSE3{=}h(-1), SSE4{=}e(+1), SSE5{=}e(+1), SSE6{=}e(-1), SSE7{=}h(+1), SSE8{=}e(-1), SSE9{=}C)$

$H = \{ (SSE1, A, SSE6), (SSE1, A, SSE8), (SSE4, A, SSE6), (SSE5, A, SSE6) \}$

$C = \{ (SSE1, R, SSE4), (SSE6, R, SSE8) \}$

A TOPS-diagram is invariant under rotation, which is an important property. This can be seen most easily by "stretching" a diagram, as in fig. 3-4, to achieve a *linearised diagram.*

3.2.2. Fixed Structures

In TOPS cartoons the higher level structures – referred to as *fixed structures* - are shown indirectly. For instance, a *beta sheet* is shown by two or more beta strands lying next to each other. In a TOPS *diagram*, this is made more explicit by the fact that the H-bonds are also shown. Although not part of the formal definition of a TOPS diagram, these structures are important for the graphical presentation. Structural information of this kind is therefore stored in the TOPS database, as will be seen later.

Currently, there are six types of fixed structures in the TOPS database:

- **Sheets**, consisting of three or more strands. Note that sheets may be bifurcated, as in the case of 2bop.
- **Barrels**, in which a long beta sheet is curved to form a circular structure.
- **Curved sheets**, in which a beta sheet is curved but not enough so to form a barrel.
- **Vertically curved sheets**, a vertically curved but not closed beta sheet.
- **Sandwiches**, consisting of two sheets packed together.
- **Unknowns**, which covers unknown higher level structures.

Note that all these structures consist of beta strands only. Alpha helixes also tend to form structures but these are harder to find, and the alpha structure description in the current TOPS formalism is quite weak. This means that TOPS diagrams are more suitable for describing protein structures with high beta strand content. Work is in progress to improve description of structures containing helices.

3.2.3. TOPS Patterns

A *TOPS pattern* is a generalisation of a diagram. In addition to specifying the SSEs and their relationships, a pattern may permit *insertions* of a specified number of SSEs at certain specified positions. A diagram may then be considered a pattern with no allowed insertions. In this way, a TOPS pattern may describe the important properties of a group of structures, or a common motif, without being too specific about which SSEs are to be found and in which order.

A TOPS pattern is like a TOPS diagram except that to each pair of adjacent SSEs – i.e., for each directed arrow representing a loop – is associated two numbers; these numbers represent the minimum and maximum number of insertions allowed at this position, respectively. The range of the numbers is from 0 to **N**, where **N** is the maximum number of SSEs in any TOPS diagram – currently about 60. **(0, 0)** means, understandably, that no insertions are allowed at that point. See figure 3-3.

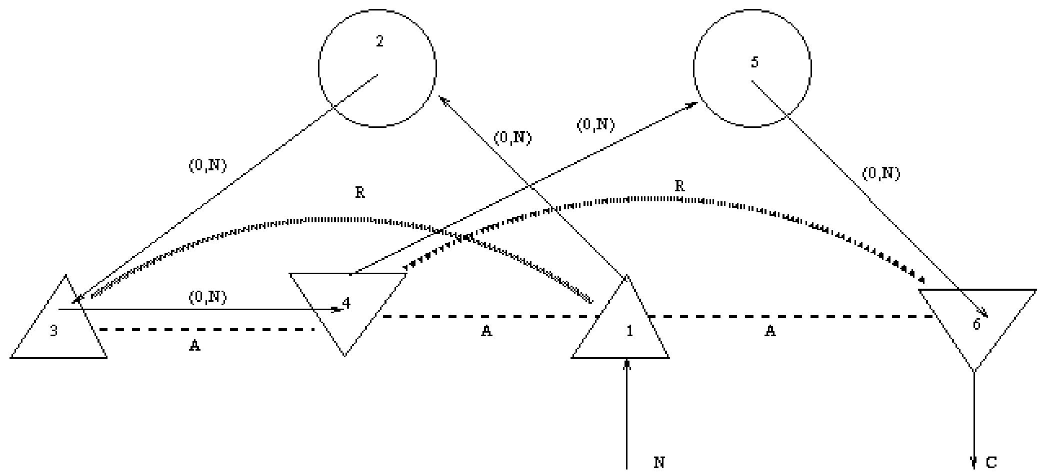

Figure 3-3: TOPS pattern

Sample tops pattern describing the plait motif. Note the similarity to the cartoon for 2bop1*, it is easy to see how inserting SSEs could yield 2bop1.*

A pattern may then *match* some diagrams – i.e., the diagrams can be obtained by inserting SSEs (zero or more) at allowed positions in the pattern. Note that several patterns may match the same diagram, a diagram may match several patterns and a diagram may match a pattern in several ways.

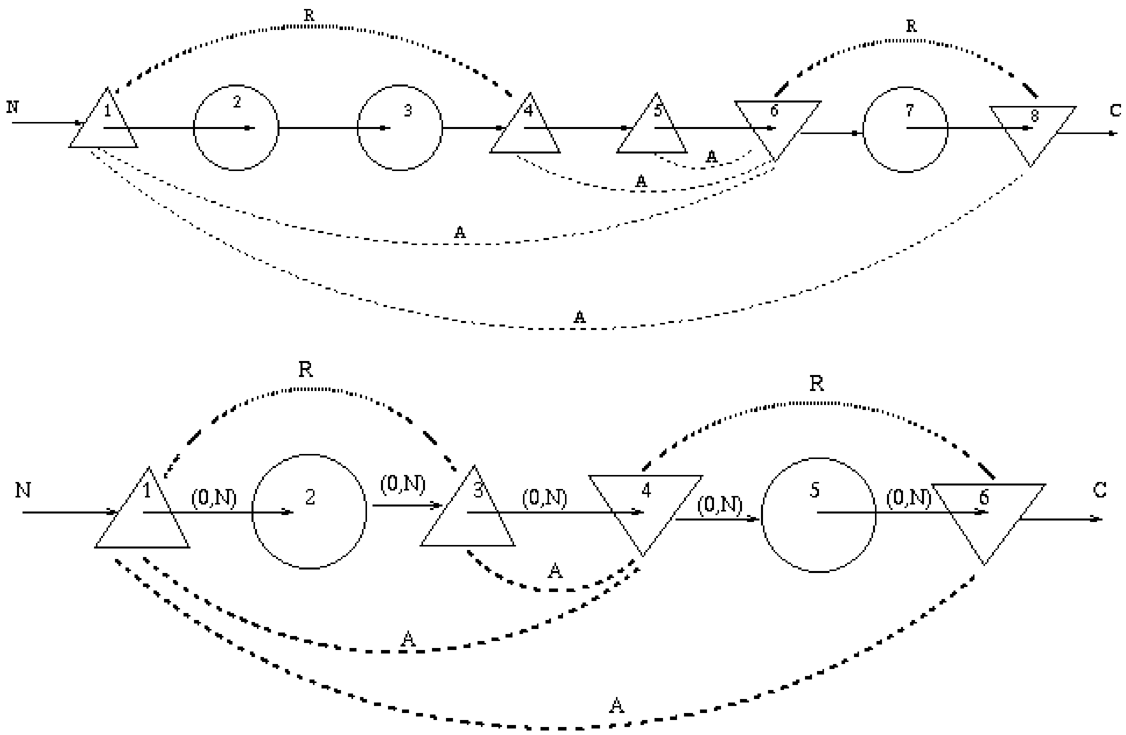

Figure 3-4: Linearised TOPS pattern and matching linearised diagram

A sample linearised TOPS pattern describing the plait motif, and a linearised TOPS diagram (2bop1) that matches this pattern.

When matching a pattern to a diagram, we obtain a *correspondence*, i.e. a sequence of matching pairs of SSEs, one for each SSE in the pattern, where first member is an SSE from the diagram and the second is an SSE from the pattern. We can describe the result of matching by the correspondence and also by a list of the total number of *inserts* between adjacent members of the SSE sequence in the diagram.

For example, there are two ways in which the plait motif can match the diagram for 2bop:

- Character matches: (1,1),(2,2),(4,3),(6,4),(7,5),(8,6). / Inserts: (0,1,1,0,0)
- Character matches: (1,1),(3,2),(4,3),(6,4),(7,5),(8,6). / Inserts: (1,0,1,0,0)

3.2.4. The TOPS Database

The TOPS database contains all structures in PDB. Each cartoon is represented as a separate entry, identified by its PDB domain name and chain id.

The format of an entry in the database is as follows:

```
TopsCartoon = c(Name, NodeList, H-bonds, FixedLists, Chirality)

Name = String[1]

NodeList = [N0,N1, ..., Nk]  where
   Nj (0 <= j <= k) = node(NodeNum,Type,Dir,Size,XY,Label) |
      NodeNum in 0..n
      Type in {N,C,E,H}
      Size = (PDBstart, PDBend)
      XY = (X,Y) | X , Y in +- Int
      Dir in {1, -1, 0},
      Label in {-, Nx, Cx}

H-bonds = { X -H(Z)- Y |
   X,Y in NodeNum,
   Z in {P,A}
}

Fixed = { FixedStructList |
   FixedStructList = FixedType( [X0,X1, ... Xi] )
   Xi (0 =< i =< k) in NodeNum
   FixedType in { SHEET, BARREL, CURVED_SHEET,
                  V_CURVED_SHEET, SANDWICH, UNKNOWN
   }
}

Chirality = { X -C-> Y | X,Y in NodeNum, C in {L,R} }
```

Note that this text representation contains additional information that is not part of the formal representation, for instance the co-ordinates.

---

[1] The PDB domain name, consisting of a four-character protein identification, one character identifying the chain and a digit identifying the domain.

An sample TOPS cartoon entry:

```
c('2bopA0',
 [N0,N1,N2,N3,N4,N5,N6,N7,N8,N9],
 [N1-h(-1)-N6,N1-h(-1)-N8,N4-h(-1)-N6,N5-h(-1)-N6],
 [s([N8,N1,N6,N5,N4])],
 [(N1,1,N4),(N6,1,N8)]  ):-
 N0=(0,n,1,(-9999,-9999),(275,-260),'N1'),
 N1=(1,e,1,(327,333),(275,-210),0),
 N2=(2,h,-1,(335,348),(225,-110),0),
 N3=(3,h,-1,(350,352),(225,-160),0),
 N4=(4,e,1,(355,356),(174,-227),0),
 N5=(5,e,1,(360,363),(174,-198),0),
 N6=(6,e,-1,(370,380),(225,-210),0),
 N7=(7,h,1,(383,392),(275,-160),0),
 N8=(8,e,-1,(400,405),(325,-210),0),
 N9=(9,c,1,(-9999,-9999),(325,-260),'C2').
```

4. ARTIFICIAL NEURAL NETWORKS

Later in the thesis we will use a special kind of artificial neural network, the Self-Organising Feature Map (SOFM), to perform unsupervised learning on TOPS diagrams. In order to understand SOFM, an outline of neural networks in general is needed. This will also ease the description of the other classification methods described later. Artificial neural networks are a huge research field within computer science, and this chapter will only cover the main outline, which should be sufficient to understand its basics and the notation used in this thesis. The material is heavily indebted to "*Neural Networks: A comprehensive foundation*" by Simon Haykin [10] and the reader is referred to his book for details and more information.

First, a brief summary of the mathematical notation is given in section 4.1. Section 4.2 defines neural networks and explains their origins. A model of the neurone, the building block of neural networks, is presented formally, and the learning process is briefly explained. Section 4.3 describes the architectures of neural networks, while section 4.4 explains and exemplifies the learning process in detail. A brief description of some of the important classes of artificial neural networks is given in section 4.5.

4.1. Mathematical concepts and notation

In general, we will use capital letters to denote sets and lower-case letters do denote entities in the corresponding sets, entries are usually vectors. Input spaces, that is, the space of possible values that the input to a neural network can have, is usually denoted *X*.

Neural networks and the classification systems described later operate on numerical vectors[1]. Unless otherwise stated, all vectors are assumed to be in Euclidean space, and additions and subtractions of vectors standard unweighted Euclidean addition, so that:

If $x = [x_1, x_2...] \in X$ and $y=[y_1, y_2...] \in X$, then $x+y = [x_1+y_1, x_2+y_2,...]$ is the vector sum of x and y.

A distance function $d(\cdot)$ is a function $d: (X, X) \to R$ that associates a real-valued distance to a pair of vectors. A standard distance is the (unweighted) Euclidean distance:

$$d_E(x, y) = \sqrt{\sum_i (x_i - y_i)^2} = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2 + \ldots}$$

Another distance measure is the standard dot product: $x \bullet y = \sum_i x_i y_i = x_1 y_1 + x_2 y_2 + \ldots$

We similarly define the *norm* of a vector x as $\|x\| = \sqrt{x \bullet x} = \sqrt{x_1^{\ 2} + x_1^{\ 2} + \ldots}$,

note that this implies that $d_E(x, y) = \|x - y\|$. We will use both notations interchangeably.

A distance function *d(x, y)* that is reflexive (i.e., $d(x, y) = d(y, x)$), 0 if and only if $x=y$ and that satisfies the triangle inequality $d(x, y) \leq d(y, x)$, is called a metric distance function.

[1] Generally, classification systems operate on objects, but a measure of distance between objects is needed; this distance is usually the distance between numerical representations of objects.

4.2. Introduction to Artificial Neural Networks

4.2.1. History and definition

The study of *artificial neural networks* has its roots in *neural modelling*. The University of Chicago had a group working on neural modelling as early as the late thirties. Their aim was to understand sensory perception in biological organisms, by modelling the neural processes taking place within the brain. Although neural modelling started out as a means to understand biological networks, the idea of using artificial neural networks to do computational tasks quickly spread. Von Neumann, generally considered the father of modern computing, addressed neural networks in one of his four famous speeches at the University of Illinois in 1949. Today ANNs are used in a variety of fields, for both purposes: To model biological neural processes and to solve practical real world problems.

Defining a neural network is not trivial, and different definitions serve different purposes. Based on the fundamental properties of all neural networks, the following general definition may be offered (adapted from [10]):

A neural network is a massively parallel distributed processor that has a natural propensity for storing experimental knowledge and making it available for use. Two of its main characteristics are:

1. *Knowledge is acquired by the network through a learning process.*
2. *Interneuron connection strengths known as synaptic weights are used to store knowledge.*

Clearly, this definition covers both biological neural networks (or brains) and ANNs. This is natural – they have several similarities, both in *properties* and in *structure*. Some of their most common, essential characteristics are:

- **Fault tolerance:** although individual neural cells / neurons may fail, this does not necessarily lead to malfunction or noticeable performance degradation in the system as a whole
- **Pattern discovery and matching** – the ability to discover important features and patterns in the input
- Both consist of a **high number of highly connected processing units** *[neurons]*.

Note that neural networks are essentially parallel. Yet most networks are never really implemented, but simulated in software running on an (non-parallel) computer, which is more cost efficient. This does not influence the main properties of neural networks – e.g. their ability to generalise – but it does decrease another of their benefits: The ability to run all the neurons in parallel, thus increasing the speed of large networks. Lately, however, advances in integrated circuit design have made it possible to produce thousands of neurons implemented on a single chip. One of the latest state-of-the-art neural networks consists of approximately 32000 neurons organised in modules, and is an experiment in designing an artificial brain – in this case, to control a robot cat.

4.2.2. The Basic Neurone

An artificial neural network consists of one or more *neurons*, organised in a certain structure. Based on the neural cells in biological brains, neurons used in ANNs operate in a similar though simplified manner. The basic neurone can be modelled in several equivalent ways, the upcoming description follows [10].

A neurone receives a number of inputs, each of which is associated with a certain *weight*. The neurone adds the products of the inputs and their corresponding weights, applies a *threshold* to this sum and optionally calls an *activation function* on the sum, to produce the final output value.

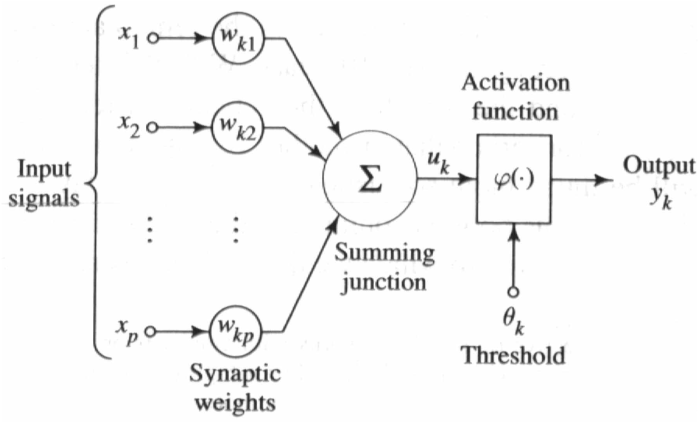

Figure 4-1: A basic neurone

A neurone k *receives* p *input values and has a fixed threshold value θ_i. After summing the values it applies the threshold and the activation function to get the output* y_k, *this forms part of the input of another neurone in the next layer. From [10]*

More formally, a neurone *k* can be modelled as in figure 4-1. Here x_i represents the numeric input number i to the neurone *k*, w_{ki} represents the weights of the corresponding connections and u_k is the sum of the input values times the weights, that is,

$$u_k = \sum_{i} w_{ki} x_i$$

In addition to the weight of the connection, each neurone is assigned a threshold level θ_k. To this threshold and the sum u_k the activation function $\varphi(\cdot)$ is applied, yielding y_k:

$$y_k = \varphi(u_k - \theta_k)$$

Being the output of the neurone, y_k is passed along to other neurons or forms part of the output of the neural network as a whole. The vector *[w_{k1}, w_{k2}, ...]* is referred to as the *synaptic (weight) vector*.

The activation function can have different forms, the simplest being simply a threshold: $\varphi(x)=1$ if $x \geq 0$, 0 otherwise. The most common forms are sigmoid functions, that is, functions that are strictly increasing, smooth and asymptotic (when the argument approaches positive and negative infinity). A sample sigmoid function is the *logistic function*, defined by

$$\varphi(x) = \frac{1}{1 + e^{-ax}}$$

where a is the *slope parameter* (the slope at the origin is a/4). The logistic function is continuous and differentiable for all *x*, which is an important property in neural network theory.

When describing neural network architectures, a neurone may be simplified to a single node, as in figure 4-2.

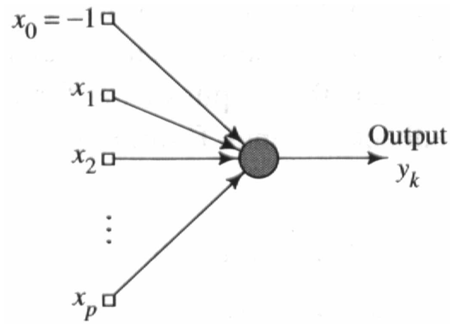

Figure 4-2: Simplified figure of a neurone

Each neurone is depicted as a single node in the graph, and connections between neurons as directed arrows. Source nodes are represented by square nodes. The threshold θ_k is represented as input x_0 with fixed value –1 and a synaptic weight vector $w_{k0}=\theta_k$. From [10].

4.2.3. The learning process

The learning process is at the heart of neural networks, and is what separates them from traditional rule-based computing. While normal programming requires knowledge to be built into the program itself, in ANNs merely the ability to *store* the knowledge is built in.

An ANN can be viewed as a black box input-output mapping: Given an input vector (the activation level of the source nodes) it produces an output vector (the output of the nodes in the output layer). The act of running an input vector through the network is called *running an iteration*, the time it takes a *time unit*.

Training a neural network consists of giving it input vectors and updating the weights. This updating process may take place after each iteration, or after a set of iterations, the last case is called *batch updating*. In some cases, learning never stops: The updating process never ceases, even when the network is doing whatever it is designed to do. This is called *on-line updating*. Normally, however, after a given number of iterations or when some desired or maximum performance is achieved, the learning is considered complete and no further updating takes place. This improves efficiency – by not having to run complicated algorithms to update weights – and prevents *overlearning*. Overlearning occurs whenever a network is trained for too long on the same input, thus losing its ability to generalise.

4.3. Neural Network Architectures

4.3.1. General

The organisation of neurons is called the *architecture* of the network. Normally the neurons are organised in one or more *layers* **–** non-layered architectures will not be considered here. Most networks have at least two layers, the *input layer* and the *output layer*. In addition, they have zero or more *hidden layers*. Note that the input layer is not counted when giving the number of layers of a network; thus, a network with one input-layer and one output-layer is called a *single-layer network*, as opposed to a *multi-layer network*. The output layer and the hidden layers are also called *computation layers*, and their neurones *computation nodes*, since it is here the computations are performed.

Using the simplified neurone model, the architecture of a graph may then be viewed as a directed, weighted graph, where nodes represent neurons, and directed, weighted links represent synaptic connections with the given weight. The input neurons, called *source nodes*, are represented by square nodes. Figure 4-3 shows a simple single-layered network.

Note that the architecture need not be constant throughout the entire life cycle of a network. A common approach, e.g., to accelerate the learning process, is to start out with few neurons, train them, and then add more neurons with weights that reflect the training of the original neurons. In this way a huge network can rapidly pass the initial training phase and proceed to the final training phases.

Haykin identifies four different classes of (layered) architectures:

- Single-layer feed-forward networks
- Multi-layer feed-forward networks
- Lattice networks
- Recurrent networks

The first three of these classes share enough common traits to be treated together as a single group – *feed-forward networks.*

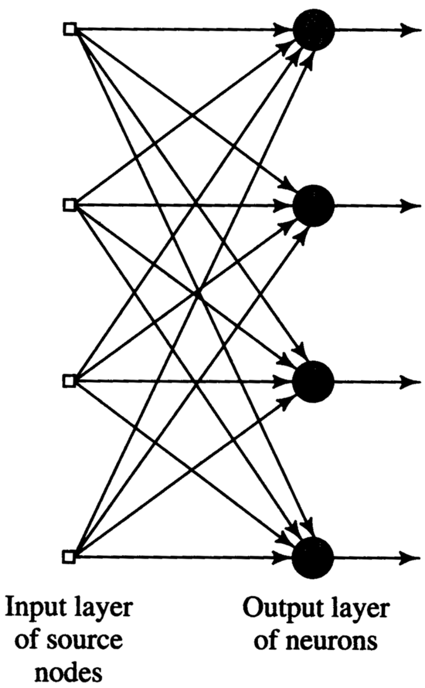

Figure 4-3: Architectural view of a neural network

A simple network consisting of an input layer of source nodes and an output layer of computation nodes. This network is feed-forward and fully connected (see main text).

4.3.2. Feed-forward networks

As the name implies, in feed-forward networks each layer projects onto the next layers" only, that is, no neuron connects to a neuron that is closer to the input nodes than itself[1]. In most cases, the nodes of a layer will project onto a single layer only, the "next layer".

If for each layer there is a connection from all of its nodes to all the nodes of the next layer, the network is said to be *fully connected*, if not it is *partially connected*. The network in figure 4-1 is fully connected.

If the output neurons are arranged in rows and columns; the computational nodes thus forming an *n*-dimensional array, the network is called a *lattice network*. A simple lattice network is depicted in figure 4-5.

4.3.3. Recurrent networks

A recurrent architecture contains at least one *feedback* loop – that is, there is a cycle in the architecture graph; a neurone or a layer projects onto a neurone or layer that is closer to the input layer. This involves a *unit-delay operator* – an element that delays the signal one time unit, thus letting the output of a neurone serve as the input of another neurone in the next iteration. If a neurone receives it's own output as input, it is called a *self-feedback loop*. Recurrent networks are not used in this thesis, and will not be described any further.

4.4. Learning algorithms

4.4.1. General

The weights of the inter-neural connections contain the information encoded in an ANN. Thus, how to change, or *update*, the weights are important; the set of rules that guide this updating is called a *learning algorithm*. When considering learning algorithms, an important thing to consider is how the network relates to the outside world, or, more precisely, to its model of its given environment. We may separate the different learning processes in three classes or *learning paradigms* [10]:

- **Supervised learning**, in which an external teacher gives feedback to the network
- **Reinforcement learning**, in which a "critic" evolves through a trial-and-error process
- **Self-organised / unsupervised learning**, in which no external teacher or critic is used

[1] Alternatively: The network, viewed as a directed graph, contains no cycles. This is valid for non-layered architectures as well.

For each paradigm different learning algorithms exist; which paradigm the network works under influences which algorithms can be used. There are four major types of algorithms:

- **Hebbian learning**, based on the following neurobiological principle: Whenever an axon repeatedly fires another axon, the connection between those neurons is strengthened.
- **Error-correction learning**, in which the differences between the actual and desired responses of neurons to events are minimised
- **Boltzman learning**, where random values are added to the synapses according to a specific probability function
- **Competitive learning**, where the neurons compete to be the one that is *active*

Often more than one algorithm is used in the same network. For instance, an unsupervised learning scheme is used in the initial training and then supervised learning is used to fine-tune the network in the final training phase. This is usually to make the computationally heavy tasks easier.

4.4.2. Supervised learning

The supervised network paradigm requires an external teacher that knows the solution or desired response to an input pattern. Normally this information comes in the form of a set of pairs of input vectors and their corresponding “correct” or desired output vectors. The learning algorithms adjust the connection weights so that, given the same input vectors, the output vectors will be closer to the desired responses. Which form this adjustment will take depends on the specific learning algorithm used.

The most common and well-known supervised learning algorithm is the *error back-propagation algorithm*, which will be briefly described in section 4.5.

Supervised learning may be viewed as a special kind of *line fitting*: The network functions as an input-output mapping, and we want the produced output to match the desired response as close as possible.

4.4.3. Unsupervised learning

Whereas the previous methods where operating within the supervised learning paradigm, other methods are used in the *unsupervised* paradigm. In self-organising systems the network must self discover important features in the input and learn to specialise in those features. This can be achieved in several ways, and so different models exist. These models can be quite different in origin, yet have similar properties. Haykin distinguishes between three main classes of self-organising systems:

- **Self-organising feature maps**, or the Kohonen model, described in chapter 5.
- Systems based on **Hebbian learning**
- Systems rooted in **information theory**

Hebbian learning is based on the principle that whenever two neurons fire simultaneously, their inter-neuron connection is strengthened, and vice versa – the principle of *positive feedback*. This principle has strong biological justifications, and algorithms based on it have been used extensively in the construction of ANNs and learning algorithms.

The information-theoretic models are based on Shannon’s information theory [16]. The input-output mapping performed by an unsupervised network can be treated like an *information preservation problem*: How do we maximise the mutual information content in the input and output vectors? This approach is often referred to as *infomax*. Solving this problem by mathematical means leads to some interesting algorithms, and this area is still open for further research.

4.5. Important examples of neural networks

4.5.1. Perceptrons, Least-Mean-Square and Error Back-Propagation

A perceptron is a simple network, in its original incarnation consisting of only one neurone. First described by Rosenblatt in 1958, it is the basis for several more advanced networks. The perceptron is used to classify patterns that are *linearly separable*, i.e. the pattern vectors can be separated in two distinct groups by a single hyper-plane. Though its usage is limited, the theory behind it has led to generalisations that are more powerful. The most apparent generalisation is to use more than one neurone, which allows the perceptron to classify patterns into more than two categories. Still, it requires the categories to be linearly separable.

Another limited yet important class of neural networks is those employing the *least-mean-square algorithm* (LMS). The LMS-algorithm, also known as the *delta rule* or the *Widrow-Hoff* rule, was first formulated by Widrow and Hoff in 1960 [17] for use in their *Adaline*, an adaptive pattern-classification machine. It operates on a single linear neuron, and has been applied in several fields, the most important of which is *adaptive signal processing*.

Both the original perceptrons and the LMS algorithm are needed to fully understand the *Multi-Layer Perceptrons* (*MLP*s) and their corresponding learning algorithms. An MLP is a multi-layer feed-forward network, and the first reported use of MLPs was the work of Widrow and his students on the Madaline in the early sixties. As the name implies, the Madaline was originally a more advanced version of the Adaline pattern-classifier, however due to inadequate learning algorithms its usage was similarly limited.

With the introduction of the *error back-propagation algorithm*, commonly known as *back-prop*, in 1985[1], the use of MLPs suddenly became widespread. It is probably correct to say that the introduction of the back-prop algorithm revolutionised the field of neural networks: The back-prop algorithm is computationally efficient yet able to solve a wide range of problems. Successful applications can be found in all fields in which neural networks have been applied; some examples are optical character recognition [22], speech recognition [23], and the opposite, i.e. pronouncing English text [24].

The back-prop algorithm may be considered a generalisation of the *least-mean-square algorithm*. The algorithm is conceptually simple. There are two major phases, the *forward pass* and the *backward pass*. In the forward pass, the input is fed to the first computational layer and propagated to the output layer the normal way, i.e. no updating is performed.

In the backward pass, the output vector y is compared to the desired response d, and an *error signal* $e = d - y$ for the output layer is computed. A *correction* is applied to the weights of the neurons in the output layer, to decrease this error. A new error signal is then computed for the neurons in the preceding layer, based on the previous error signal, and the weights are updated in a similar manner. The result is that the error is propagated backwards, giving the algorithm its name.

---

[1] A description of an algorithm similar to error back-propagation appeared in a thesis by Werbos as early as 1974 [18]. However, the term error back-propagation was not coined until 1985, when the algorithm was rediscovered and put to use.

4.5.2. Systems based on competitive learning

In competitive learning, the neurons compete among themselves for the right to be the one to be active (fired). Initially, all neurons in the competitive layer (usually the output layer) have randomised values[1]. When presenting the network with an input vector, one of the neurons "wins", according to some selective mechanism. This neurone, called a "winner-takes-all neurone", then gets its weights updated so that it becomes more likely to "win" if presented with similar input vectors. This process leads to specialisation – ie, each neurone learns to specialise on a set of similar patterns, thus enabling the network to function as a *feature detector*.

Consider a simple fully connected feed-forward neural network with one input layer and one competitive output layer. Each of the weights is randomised so that the sum of all synaptic weights for a given node is *1*, that is, $\Sigma w_{ij}=1$ for all neurons j.

When presented with an input vector *x*, the winner is the neurone j that has the highest activation level v_j. The weights of the connections are then updated. The standard competitive-learning formula for this updating is

$$\Delta w_{ij} = \begin{cases} \eta(x_i - w_{ij}), & \text{if neurone j wins the competition} \\ 0, & \text{if neurone j loses the competition} \end{cases}$$

where η, the *learning-rate parameter*, usually is a function of time *t*. The overall effect of this rule is that of moving the synaptic weight vector w_j towards the input pattern *x*.

The following geometric analogy, adapted from [15] will serve to illustrate competitive learning:

Consider the simple network depicted in 4-5. All input patterns, as well as the synaptic weight vectors, have a fixed length *n*, and so may be viewed as a point on an *n*-dimensional hyper-surface, figure 4-6. It is assumed that the input patterns form clusters (on this hyper-surface), while the weight vectors initially are randomised (fig 4-6a). After successful training (fig 4-6b), the weight vectors have moved towards the centre of gravity of the clusters. I.e., as a neurone wins the competition, its synaptic weight vector is moved towards the input vector, and the neurone learns to specialise in the vectors in a cluster. Note that in the figure all vectors are constrained to have the same Euclidean length, so that they can be represented as points on an three-dimensional sphere.

[1] Different initialisation schemes exist and are usually more efficient – in effect they give the network a jump-start.

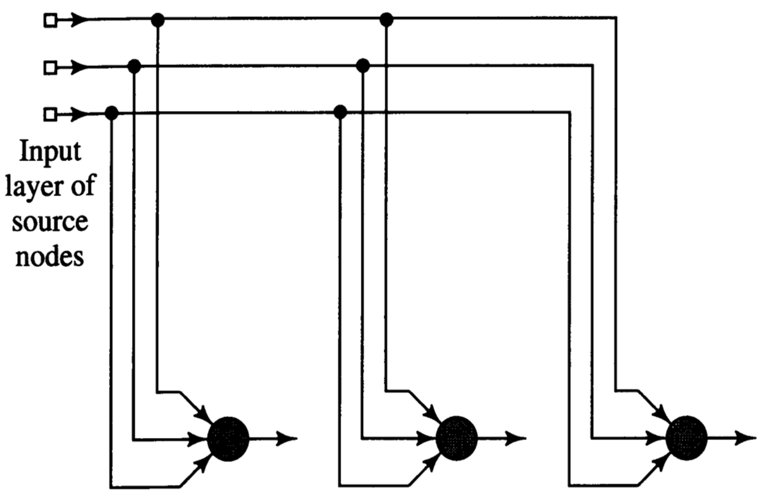

Figure 4-4: Single-layered lattice network

A simple feed-forward fully connected lattice network consisting of 3 input neurons and 3 output vectors organised in a 1-by-3 grid. From [10].

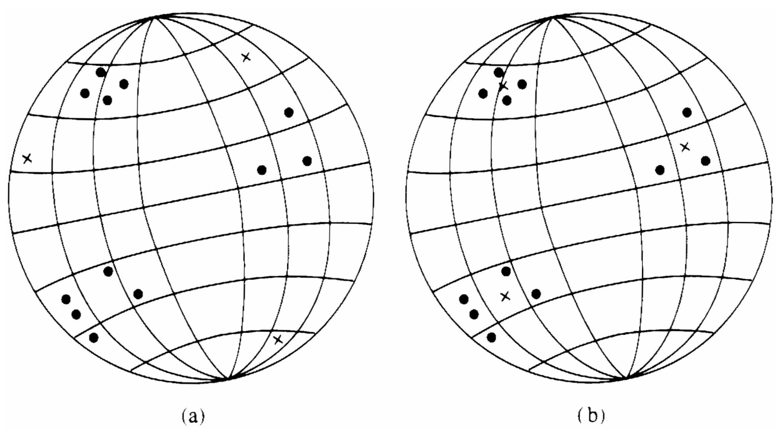

Figure 4-5: Input patterns as points on a surface

Input vectors to the network in figure 4-5 are here given as points on a three-dimensional sphere, while the synaptic weight vectors are represented as crosses. Figure a *is the initial values, with the input patterns forming natural clusters and the weight vectors placed randomly on the surface. Figure* b *shows the weight vectors after training: They have now moved in the direction of the centre of a cluster of input vectors. From [10].*

4.5.3. Modular networks and VLSI

Another important issue concerning neural networks is *modularity*, i.e. building networks that have other networks as their components. This is a complex undertaking, but one that shows great potential. By using more than one kind of architecture and more than one kind of learning algorithm, the benefits of several algorithms may be combined. For instance, an unsupervised learning scheme or local methods that converge swiftly could be used early on, to extract features and patterns in the input data. Following that, a supervised learning scheme could improve the performance further. This usually requires the use of an integrating unit, which directs the use of the networks. The integrating unit does not decide how to perform the different modes of learning, only which mode to use and which patterns to present to the network. Modularity is also important when building huge VLSI Circuits[1] - based neural networks. These consist of modules (chips) comprising several (in the range of a few to several thousand) units functioning as neurons, and it is necessary to treat them in a modular way to maintain efficiency.

[1] VLSI Circuits: Very Large Scale Integrated Circuits

5. THE SELF-ORGANISING FEATURE MAP

Self-organising feature maps (SOFMs) have been used extensively within bioinformatics, and will be used in this thesis as an important component of the classification system. We will here present a brief description, building on the theory presented in the previous chapter. The main sources of information are [12] and [19].

In section 5.1 the architecture will be described. Section 5.2 covers the details of the updating algorithm, and describes the topological maps that represents the main output from a SOFM.

5.1. SOFM Architecture

The *Self-Organising Feature Maps* (SOFMs), also known as *Self-Organising Maps* or *Kohonen Networks* after their inventor Teuvo Kohonen, were first described in 1981/1982 [21], although it is based on the *Learning Subspace Method*. Since its first appearance, the SOFM algorithm has been used widely, in such diverse fields as speech recognition [25], classification of biological sequences [27] and radar classification of sea ice [28]. The main reasons are its flexibility, ease of implementation and its efficiency as a *feature detector*. The SOFM also has neural justifications, which has led to it being used in neural modelling as well.

An SOFM is a single-layer fully connected feed-forward lattice network. The nodes in the competitive layer are organised in (usually) 2- or 3-dimensional grids. A modified version of the competitive learning algorithm selects winner(s) in this layer and updates weights accordingly. After training, the output layer can be used to produce a *topological map* of the input vectors. This map, described more fully in the next section, is the primary product of a SOFM network.

It should be stated at the outside that SOFMs are *not* designed for pattern classification by them selves. Their main applications are visualisation of complex data and reduction of the dimensionality of data. For pattern classification, a better approach is to use the SOFM algorithm in combination with a supervised learning scheme, e.g. Least Mean Square. Learning Vector Quantization (LVQ) is another method often used in combination with a dimensionality-reducing SOFM. Chapter 6 discusses pattern classification in general and LVQ in particular.

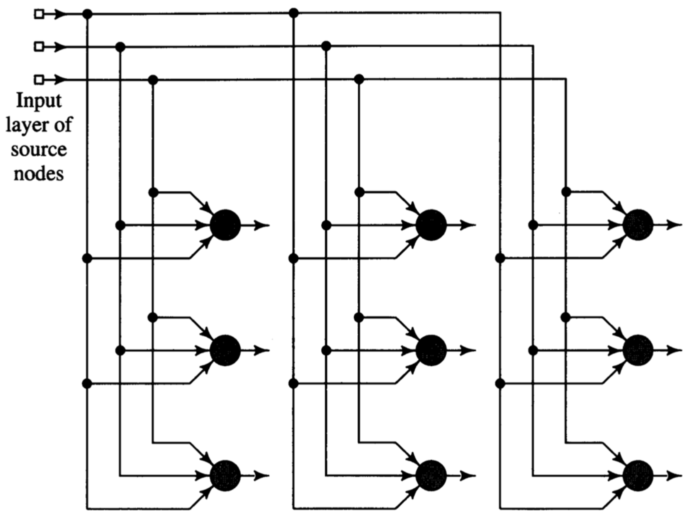

Figure 5-1: Simple SOFM network

A SOFM network is a single-layer fully connected feed-forward lattice network. In this case, the input layer has 3 input neurons while the output layer has 9 neurons organised in a 3-by-3 grid.

5.2. SOFM Algorithm

5.2.1. General

The main difference between SOFM and traditional competitive learning schemes is that in a SOFM, the winner neurone usually does not "take it all". Given an input vector, the data is fed through the input layer and to the competitive layer as normal, in particular a *winner* is chosen. A *winning area* around the winner is then chosen, according to a specific *neighbourhood function* (see beneath). The neurons inside this area are then all winners, and get their weights updated according to specific updating functions, not necessarily by the same amount.

By choosing a *neighbourhood area* or *neighbourhood kernel* and updating all neurons inside it, some properties different from normal competitive networks arise. Essentially, the SOFM preserves the *geometrical relationships* between different neurons and different classes of input data. This means that given two input vectors that are roughly similar, their winner neurons should be placed close in the output layer.

Remember that the output layer is organised in a lattice, usually two- or three-dimensional. It is then possible to construct a topological map of the neurons, with the same dimensions. The input vectors are placed on this map according to their winner neurons, i.e. if an input vector with label l causes neuron j to win, then the label l is placed on the position of neuron j. The result is a visualisation of the input data, where input vector (classes, labels) that share similarities are placed close to each other on the map (possibly on the same neuron) whereas different vectors are placed in distinct regions. Figure 5-1 shows a sample map.

A normal training proceeds by running all vectors in the input set through the network; this is called an *epoch*. The number of epochs necessary to achieve satisfactory performance is large; common values are 5000 to 20000.

5.2.2. Initialisation

The weights in the competitive layer could be initialised with random values, with small deviations from a centre (commonly 0.5). The network will then converge. A faster approach, however, is to initialise the network with samples from the input vector; this decreases the length of the initial training phase considerably. With unsuitable input values, however, this increases the risk of the network not reaching a optimum configuration; it is stuck in a local minimum.

5.2.3. Neighbourhood functions

The exact form of the neighbourhood function is not important. The requirements are that it starts out large (e.g. approximately half of the neurons) and decreases with time; in the final training phase, the area should be small, e.g. consisting of only the winner neuron. The simplest function is a square around the winner neurons, with length *2*k+1*, i.e. the winner neuron and its *k* nearest neurons, horizontally or vertically, are chosen. Other versions are hexagons or other approximations of circles.

The size and form of the neighbourhood is linked with the form of the updating function, considered next.

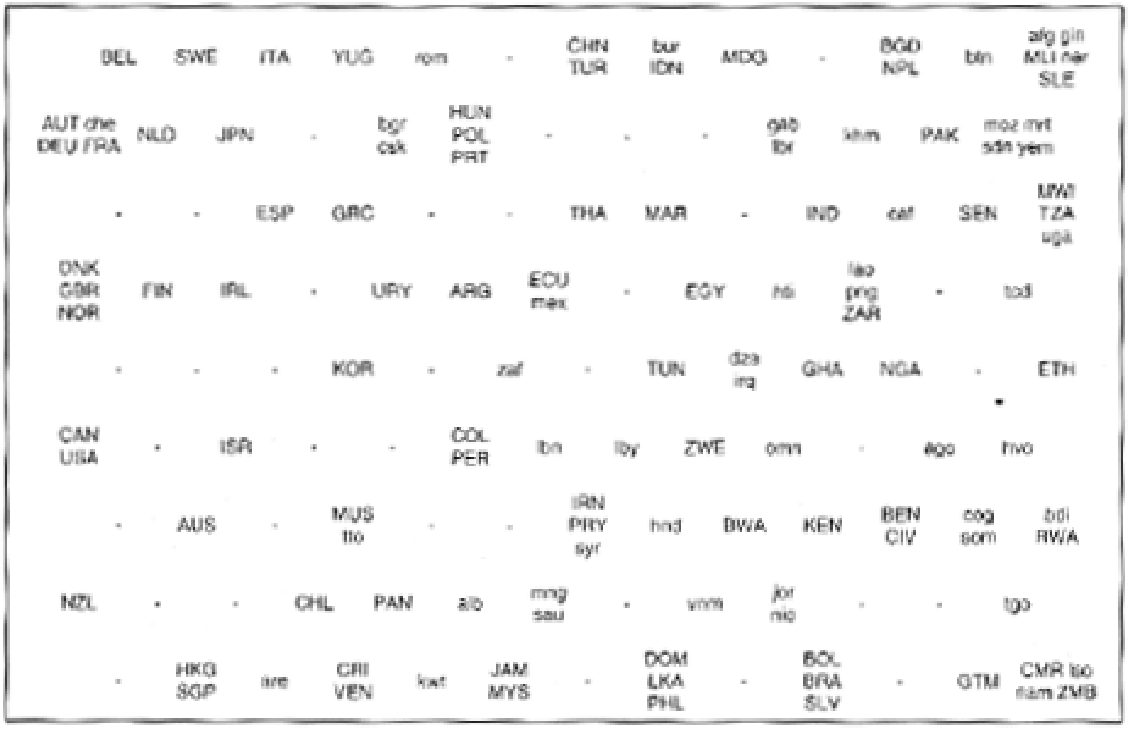

Figure 5-2: Topological map

This sample topological map shows the relationships between countries, as three-letter ids, based on statistical data on economy and politics. Data from the countries in capital letters were used in the training of the network, while countries in lower case only were used when creating the map. Note that countries that normally are considered similar, like the Nordic countries or Canada and the USA, are placed close together. From [12].

5.2.4. Updating functions

Many possible updating functions exist. Note that it is possible to apply the same formula to all the neurons in the winner area. Higher accuracy is usually achieved by adjusting for the distance to the neuron, according to a linear decrease or even a Gaussian function. The following formulas are taken from Kohonen:

First a winner is selected, the standard formula is based on Euclidean distance[1]:

$$c = i : \forall j : \| x - m_i \| \leq \| x - m_j \|$$

so that m_c is the winner neuron. The general updating formula is then

$$m_i(t+1) = m_i(t) + h_{ci}(t) \times [x(t) - m_i(t)]$$

where $h_{ci}(t)$ represents the neighbourhood function.

First, consider the simplest case where the neighbourhood function is a square around the winner neuron. Let $N_C(t)$ denote the neurons inside that region; note that the region is actually a function of time *and* neurone, as the neighbourhood area changes with time. We can then let

$$\begin{aligned} h_{ci}(t) &= \alpha(t) \quad \text{if } i \in N_C \\ h_{ci}(t) &= 0 \qquad \text{if } i \notin N_C \end{aligned}$$

where $\alpha(t)$ is the *learning-rate factor*. The value of $\alpha(t)$ should be in the range zero to one and decreasing with time *t*. This formula applies the same updating to all vectors within the winner area. While naive, it is still widely used and efficient for most purposes.

If we adjust linearly for distance to winner neurone, we get a standard neighbourhood formula:

$$h_{ci}(t) = \alpha(t) \times \frac{k(t)}{d_{ci}}$$

where d_{ci} is the distance (in neurons) from neurone *c* to neurone *i*, and *k(t)* is a linearly decreasing function of the time *t*. Other neighbourhood functions are based on Gauss functions or other more complex expressions.

[1] Other versions are based on dot products, which has stronger biological justifications.

6. CLASSIFICATION AND CLUSTERING

Classification of patterns into categories is an important task, having applications in biology, engineering and other fields. Many different methods exist, rooted in e.g. statistics, information theory and the theory of neural networks.

Most of the technical details of Learning Vector Quantization are from the documentation of the *LVQ_PAK* software package [20]. The general and statistical parts are based on [12], [13] and [14].

After introducing the concept of classes and class labels in section 6.1, this chapter will briefly introduce and explain some important concepts in section 6.2. These concepts will be used when describing the Learning Vector Quantization in section 6.3. Clustering in general and two specific clustering methods will be described in sections 6.4, 6.5 and 6.6. Some ways of measuring the quality of classifications are introduced in section 6.7.

6.1. Introduction and notation

In pattern classification, the aim is to classify a data set into a finite number of categories. This can be achieved in several ways; the method chosen depends to a large degree on the statistical properties of the input data set. In the general case, we have a set of vectors of n dimensions, the *input set*, from a set of possible values X, the *input space*. The task is to partition all (possible) input vectors into one of a (not necessarily fixed) number of categories or *classes*, that is, we want to associate a *class label* $c \in C$ to each input vector, where the *label* set C is the set of possible class labels. Essentially, what we want is a function $f\colon X \rightarrow C$, where X is the input set and C is the label set. Note that clustering sometimes is defined differently, see 6.4.

We can separate classification methods into two groups, supervised and unsupervised. Within these groups, several organisation schemes exist. Normally, clustering is considered an unsupervised classification method, while classification can be both unsupervised and supervised. Definitions of most of these terms vary, so, to clarify: The learning vector quantization is a supervised method, while the hierarchical and the isodata clustering methods, as described here, are unsupervised. The SOFM, which might be considered a classification method, is also unsupervised.

If the result of a classification method returns a set of sets of objects, the classification is called a *single-level* (or *flat*) classification, if it returns a hierarchy of sets it is called a *multi-level* (or *nested*) classification.

6.2. Vector Quantization and The Voronoi Tessellation

Consider an input space of n dimensions. We may partition the input space into *regions,* bordered by hyperplanes (of *n-1* dimensions), *"such that each partition contains a reference vector that is the "nearest neighbour" to any vector within the same partition"* [12]. See figure 6-1. This partition is a *Voronoi Tessellation*[1], the regions are called *Voronoi cells* and the sets of vectors in a region a *Voronoi set*. The set S of reference vectors is also called the *codebook* and a vector from the codebook (i.e., a reference vector) a *codebook vector*.

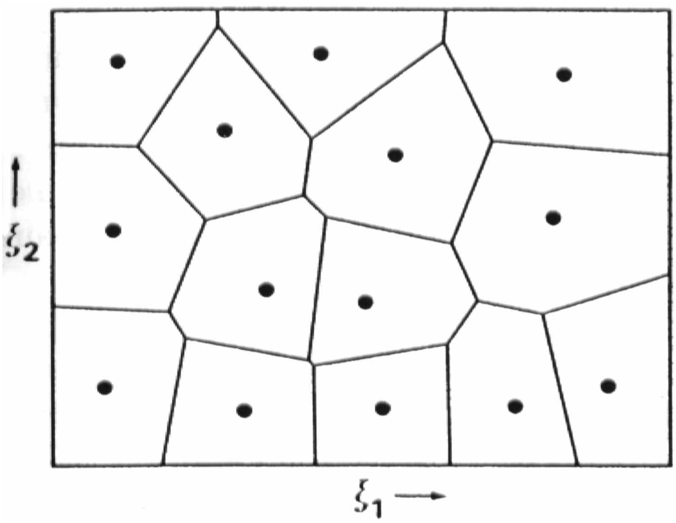

Figure 6-1: Voronoi tessellation

A Voronoi tessellation is a partition of a (hyper-) plane into polygons. All cells have reference vectors (marked by dots) that are the nearest reference vector to all points in their region. This Voronoi tessellation is two-dimensional and has fourteen cells and reference vectors. From [12].

Vector quantization is a *signal-approximation* method that uses these concepts. For a given input space, a set of codebook vectors is chosen. To approximate a given input vector v, we then find the codebook vector x that is closest[2] to v in the input space. Depending on the number of codebook vectors and the distribution of actual input vectors, this approximation can yield significant reduction of computing time / bandwidth / storage requirements, at the price of a certain distortion. This distortion, of course, may be too high a price to pay, depending on the input set and number and placement of the reference vectors.

---

[1] Tessellation: an arrangement of polygons without gaps or overlapping, esp. in a repeated pattern [Oxford]

[2] Closest in this context normally means Euclidean distance, although other measures are possible.

6.3. Learning Vector Quantization

6.3.1. Introduction

First described by Kohonen in 1986, the Learning Vector Quantization (LVQ) is a supervised learning scheme based on vector quantization and the Voronoi tessellation. Starting with a set of reference vectors, the idea is to use class information to move the reference vectors, thereby improving performance.

More specifically, consider an input space, a codebook S and a vector x picked randomly from the input set. Each codebook vector is associated with a single class label. Since this is a supervised method, there is, for each input vector, an associated class label. Let c denote the class label of the codebook vector w that is the nearest neighbour to x, similarly let d denote the class label associated with x. If $c=d$, the codebook vector w is moved in the direction of x; if not, w is moved away from x. The other codebook vectors are not modified. After a training period, consisting of a number of iterations of all vectors in the input space, *the codebook vectors will more closely approximate the input vectors.*

The codebook vectors may then be used both for representation and for classification. In the last case, the class of a given input vector is the class of its nearest reference vector.

The LVQ is a stochastic approximation algorithm, with certain convergence properties. Mathematical details can be found in [30].

6.3.2. Initialisation of the LVQ

Before learning starts, the LVQ has to be initialised; i.e., some initial reference vectors must be decided upon. The first decision is how many reference vectors to use. This depends on several variables, including, of course, the desired accuracy, the statistical properties of the input set and the amount of computational power available. A common value is a small number (less than 10) times the number of classes.

The second decision is how many reference vectors we need for each class. One possibility, the simplest, is to have the same number of reference vectors for each class. In most cases, this is adequate, even if the input samples are not evenly distributed. A sometimes better approach is to use proportional initialisation: If there are i vectors in the training set (input set), a class having d input vectors should be represented by approximately ed/i codebook vectors, where e is the total number of vectors in the codebook.

Having decided upon the number of reference vectors for each class, initial values must be assigned. Sample vectors from the input set can be used, *assuming they are classified correctly*. Failing this assumption, the reference vectors will not be representative and convergence is not guaranteed. Simple methods that check this assumption exist; Kohonen mentions the “K-nearest-neighbour”-method, which first performs tentative classifications and then removes or replaces codebook vectors that are incorrectly set.

6.3.3. Updating

Let $C = \{m_i\}$ be the codebook, initialised as described in the previous section, and consider an input vector x. Let

$$c = i \mid \forall j : \| x - m_i \| \leq \| x - m_j \|$$

so that m_c is the nearest reference vector to x. The algorithms update m_c (and possibly other vectors as well) according to specific update functions. Several functions exist, the most common are called LVQ1, LVQ2.1, LVQ3 and Optimised LVQ1 (OLVQ1) in Kohonens notation. LVQ1 is the basis for all algorithms, and will be presented beneath. LVQ3 is a modification of LVQ2.1 that generally perform better and is more stable, while OLVQ1 is a modified version of LVQ1 that converges faster. The details of the other algorithms can be found in [20].

Both the philosophy and the mathematics behind LVQ1 are simple. Let $m_c(t)$ denote the codebook vector $m_c(t)$ at time t, similarly let $\alpha(t)$ denote some time-dependant function called the *learning parameter* (see below). $x(t)$ represents the input vector. We then set

$$m_c(t+1) = m_c(t) - \alpha(t) * [x(t) - m_c(t)], \quad \text{if x and } m_c \text{ belong to different classes}$$
$$m_c(t+1) = m_c(t) + \alpha(t) * [x(t) - m_c(t)], \quad \text{if x and } m_c \text{ belong to the same class}$$

All other codebook vectors are left unmodified. The effect is thus, as stated earlier, that of moving the codebook vector that is closest to the input vector closer to the input vector $x(t)$ if the classifications agrees, away from the input vector if the classifications disagree. (This can most easily be seen by letting $\alpha(t)$ equal 1, in which case $m_c(t+1)$ will equal $x(t)$ if and only if $x(t)$ and m_c belong to the same class.)

6.3.4. Discussion of the algorithms

It should be noted that all of these algorithms, despite their differences, yield approximately the same accuracy. The OLVQ1 converges fast, and is recommended in the initial phase. Sometimes running OLVQ1 is enough to achieve satisfactory performance, but in order to improve accuracy one of the other algorithms should be run. Kohonen recommendations are running OLVQ1 for 30-50 times *k* iterations, followed by LVQ3 (or some other algorithm) for 50-200 times *k* iterations, where *k* is the number of codebook vectors. These recommendations are necessarily based on experience – no mathematically proven or optimal guidelines exist[1].

The learning parameter α must be in the range zero to one, and should decrease monotonically over time. Kohonens recommendation is a start value of 0.1 and linear decrease towards zero.

[1] Nor are possible, as the arbitrary statistical distribution of the input prevents mathematical analysis.

6.4. Clustering

6.4.1. Overview

Related to the classification systems described above is *clustering*. Consider a set of n objects: The task of a clustering algorithm is then to divide these objects into a number of *clusters*, where each cluster is a set of objects. The number of clusters need not be predetermined or even fixed, and the result is not necessarily a *partition*, i.e., in general each object may be found in zero, one or more clusters[1]. In addition, there are no class labels (or only arbitrary ones). These last properties distinguish a *clustering* process from a *classification* process. Of course, if we impose some restrictions, like requiring the result of a specific clustering process to be a *partition*, that process is equivalent to a classification. These restrictions apply to most clustering tasks, nevertheless clustering is usually considered a separate field within statistics, and methods and notation differ somewhat.

To cluster objects, a *distance* must be associated to each pair of objects. The simplest distance measure is the (weighted or unweighted) Euclidean distance between some numerical representation of the two objects, but other distance measures are frequently used. The numerical representation is quite often simply a list of properties of the objects. One important issue is whether the distance should be metric or not.

6.4.2. Clustering methods and a generic clustering process

Before clustering starts, the data set(s) should be analysed by statistical means. This is to determine whether there is any *cluster tendency* in the data set. Most clustering processes will return a set of clusters no matter if there really is a cluster tendency in the data set, and so avoiding or at least detecting malformed clusters are important.

The next step is to decide which method to use. The different methods can be taxonomically divided into groups in several ways; an obvious way is to separate between *top down* and *bottom up* methods. Bottom up methods start by considering each object a cluster and proceed by joining clusters, usually in an iterative process, until some predetermined stopping criteria is satisfied. The merging of two clusters can be lossy (non-nested), i.e., the new cluster is a set of objects, so information about sub-clusters are lost, or non-lossy (nested), i.e., the new cluster is a set of clusters. Top down methods, on the other hand, start with a cluster consisting of all objects in the data set and then (usually iteratively) split the cluster(s). Other methods start with a set of randomly chosen clusters and iteratively refine them; the vector quantization and the isodata method described below are examples of this.

Note that the result of a clustering process is either a single-level (flat) or a multi-level clustering. In the first case, the result is a set of sets of objects. In the last case, each cluster either contains objects or clusters[2] and the result is a tree in which the leaves are objects and non-leaf nodes are clusters. After clustering the data set, the resulting clustering must be evaluated – this will briefly be discussed in section 6.7.

---

[1] Or else a number between 0 and 1 determines to which degree an object is part of a cluster, this is called *fuzzy clustering*.

[2] Or, conceivably, a mixture of both.

6.5. *Hierarchical Clustering*

6.5.1. General

Hierarchical clustering is a bottom up non-lossy clustering technique that is widely used. Briefly, the method works by building a distance matrix and repeatedly joining the two clusters that have the minimum common distance. The result of merging two clusters is a cluster with two clusters as children, thus the result of a hierarchical clustering is a binary tree.

Hierarchical clustering being a standard method, several implementations exist and most major statistical packages provide at least one. The implementation can be simple and a sample algorithm is outlined in the next paragraph.

Different methods usually differ in how the distance between two clusters (of more than one point) is computed. Three simple and standard measurements are *single link*age, *complete linkage* and *average linkage*. With single and complete linkages, the distance between two clusters is the minimum or maximum distance between any two points in the two clusters, respectively. With average linkage, the distance is the average distance between pairs of points with one point from each cluster[1].

6.5.2. A simple hierarchical clustering algorithm

Input: A set of n objects and a way to measure the distance between two objects or between two sets of objects.

Output: A root *node* to a binary tree, where each node is either a leaf node containing one of the original objects or a non-leaf node with two nodes as children.

Steps:

1. Construct a set of n nodes, where node i is a leaf node containing object i.

2. Construct an n times n distance matrix, where element (x, y) is the distance between node i and node j.

3. Find the smallest number in the distance matrix, let the co-ordinates of this number be (i, j).

4. Create a node k with node i and j as children. Replace node i with node k and remove j from the distance matrix.

5. Update the distance matrix to reflect the changes.

6. Repeat steps *3* through *5* until only one node remain; this root node is then the output of the algorithm.

---

[1] Or a simplification of this, e.g. the average of the minimum and maximum distances between the two clusters

6.6. The Isodata algorithm

Another algorithm widely used is the *isodata algorithm*, also known as the *k-means* or the *c-means* algorithm. The idea is to start with a set of points that represent clusters, called *point* representatives, and iteratively set the point representatives to the mean of the vectors that are nearest them. While computationally efficient and simple to implement, the algorithm can be quite effective in identifying compact clusters. However, it has two weaknesses: As clusters are represented by *points*, the algorithm is not good at finding shell-shaped clusters. In addition, it tends not to deal accurately with clusters of significantly different sizes. Several variants of the algorithm exist, optimised for various conditions.

The basic isodata algorithm can be described as follows (adapted from [13]):

Input: A set $X = \{x_i\}$ of $N=||X||$ numerical vectors of size k, and m, the number of clusters to produce

Output: An 1-by-n array b so that x_i is in cluster/category $b(i)$

Steps:

- **Initialisation:** Let θ be an 1-by-m array of vectors of size k, the *point representatives*, and initialise it with random values
- **Repeat**

  For i = 1 to N

  - Determine the closest point representative θ_j for x_i
  - Set $b(i)=j$

  For i = 1 to m

  - Let θ_i be the mean of all vectors $x_j \in X$ that has $b(j)=i$

  Until no change in θ occurs between two successive iterations
- **Return** the cluster identification matrix b.

6.7. *Comparing classifications*

6.7.1. General

Consider a set S of objects that has been classified by two methods, returning the single-level classifications C and D. We want to compare the two classifications C and D to see how much they agree, and need some definition of agreement. One application is if one of the classifications is correct, by some *external criteria*, and the other classification approximates that one, e.g. if we have a manually constructed classification and want to develop automated methods to replace the manual ones. Other uses are to verify a classification *without* external criteria, by comparing it to randomly generated classifications.

If each group in the two classifications we are to compare has a class label associated with it, and these class labels are comparable, comparison is simple. Several measures exist, a simple one is to count the number of objects that in C is classified in the same category as in D, and express this as a percentage of the total number of objects. While simple and intuitive, this is not always the optimal measurement, as it can be misleading. If, in the correct classification, one group contains a huge part of the total set of entries, a high percentage score could be achieved by a (clearly not very useful) classifier that puts *all* the entries in that category.

In general, we can not assume that we have class labels for both classifications, or, to be precise, that we can compare class labels from different classifications. The only information available is which objects are grouped together. Next, a set of general methods for such cases is explained. Note that these methods apply to single-level classifications only. Methods for comparing nested clusters are more complicated, and are not discussed here.

6.7.2. The rand statistic and similar indices

Consider the set *S* and classifications *C* and *D* as previously. The goal is to find a statistical index that tells to which degree *C* matches *D* (or v.v.). The idea is to compare all pairs of objects from *S*, and, for both *C* and *D*, see if they are in the same category (i.e., have the same class label). In this manner, the amount of agreement is computed.

The basic algorithm is as follows ([13], p. 549):

1. Initialise four counters SS, SD, DS, DD to zero.
2. For each pair (s1, s2) of objects from *S*, do:
   – if s1 and s2 have the same class label in *C* and in *D*, increment SS.
   – if s1 and s2 have the same label in *C* but different labels in *D*, increment SD.
   – if s1 and s2 have different labels in *C* but the same label in *D*, increment DS.
   – if s1 and s2 have different labels in both *C* and *D*, increment DD

Let m1 = SS + SD, m2 = SS + DS and M = SS + SD + DS + DD[1]. We then use the three statistical indices defined as follows:

- *Rand statistic:* $$\mathrm{R} = \frac{\mathrm{SS} + \mathrm{DD}}{M}$$
- *Jaccard coefficient:* $$\mathrm{J} = \frac{\mathrm{SS}}{SS + SD + DS}$$
- *Fowlkes and Mallows index:* $$\mathrm{FM} = \frac{\mathrm{SS}}{\sqrt{\mathrm{m}_1 \times \mathrm{m}_2}}$$

It is clear that for all of these indices, the higher the value the better the agreement. For the rand statistic and the Jaccard coefficient the range is (0, 1), with 1 meaning a total match of all pairs. This requires the number of categories in the two classifications to be equal, which in general is not always the case.

Another important statistical index is the Hubert Γ statistic, which measures the correlation between two matrices. For two *symmetrical* matrixes X and Y the formula is

Hubert statistic: $$\Gamma(\mathrm{X}, \mathrm{Y}) = \frac{\sum_{i=1}^{N-1} \sum_{j=i+1}^{N} X_{ij} Y_{ij}}{M}$$

[1] Note that M is the total number of pairs from ***S***, that is M=N*(N-1) / 2 where N=|***S***| is the number of objects in ***S***.

6.7.3. Probability distributions and Monte Carlo estimations

It is important to understand that the statistical indices in themselves do not tell very much, except for extreme values. Not knowing the statistical distribution of the numbers in the data set, the *probability density function* (*pdf*) of an index is unknown. This means that we know neither the *expected values* of the statistical indices nor the *significances* of the values. To yield usable information, an index should therefore be compared to indices of *random data set*; i.e., a Monte Carlo approach should be used to estimate the pdf of the index. The average indices of random classifications will then be comparable to the index of the classification in question. This approach will not be described in detail here; readers are referred to [13] or [14] for more information.

An outline of one possible way to do a Monte Carlo estimation of the Hubert Γ statistic will be given here. This corresponds to a simple Monte Carlo approach under the *random label hypothesis* as presented in [13].

First, consider a set X of N input vectors, an N-by-N distance matrix P and a partition C of the objects in X. We want to measure the degree to which the partition C matches the data set X.

Note that the partition C can be viewed as a mapping function $c: X \rightarrow \{1..m\}$, where m is the number of clusters in C. Now construct an N-by-N matrix Y, so that $Y(i,j)=0$ if $c(\ X(i)\ )$ equals $c(\ X(j)\ )$ and 1 otherwise, and compute the Hubert Γ statistic of the matrices P and Y. Let γ be the result; γ tells us to which degree Y matches P.

Next, construct r mappings $c_i: X \rightarrow \{1..m\}$, corresponding to r random classifications of X. For each mapping i, compute the corresponding Hubert Γ statistic γ_i. Typical values for r is 100 or 200.

We can now use the γ_i values to estimate the expected value of γ, and find out if the clustering C is statistically superior to random clusterings.

7. DESCRIPTION OF THE CLASSIFICATION SYSTEM

In this chapter, implementation of the classification system is described in somewhat detail. First, an overview of the different components is given in section 7.1. The different components are then described more fully in the following sections.

7.1. Overview

The classification system consists of several components. The starting points are the TOPS and CATH data files, which are used for analysing, classification and classification comparisons of TOPS diagrams in several processes. The main functionality of the system is depicted in a block diagram in figure 7-1. There are four major parts or software packages:

Central in the system is a software package used for analysis of TOPS diagrams, the *TOPS package*, described in section 7.2. The TOPS package provides a set of methods for producing vector representations of diagrams and writing these to (flat-text) files. In addition, it has functionality for extracting selections of diagrams, based on their CATH entry, and for identifying each vector with a string version of their CATH number, suitable for classification purposes.

The *Cluster package* described in section 7.3 has methods for (unsupervised) hierarchical clustering of diagrams, and for comparing classifications using the statistical indices described in the previous chapter. This package is usually called directly from the TOPS package, without the need to go through intermediary files.

The *SOFM package* contains methods for applying the SOFM algorithm to vector representation of diagrams. This package is described in section 7.3. After training a network, it can produce topological maps showing relationships between diagrams. In addition, the network can be used to create revised vector representations of diagrams with fewer dimensions, in our case two[1]. The vectors can then be used as input to the cluster package or to the LVQ package described next.

The *LVQ package* uses the LVQ algorithm to perform supervised classification of the TOPS diagrams. After the LVQ has been trained on a set of diagrams, classification can be done on other (or the same) set, and the accuracy (compared to CATH) is found. See section 7.4 for a complete description.

The CATH files are used to find the CATH category of each TOPS entry. This information can be used to select on which diagrams the programs are to be working. The CATH number can also be included in the text file version of the vector representations.

---

[1] This is connected to the *dimension-reducing* property of the SOFM algorithm, as described in section 5.1.

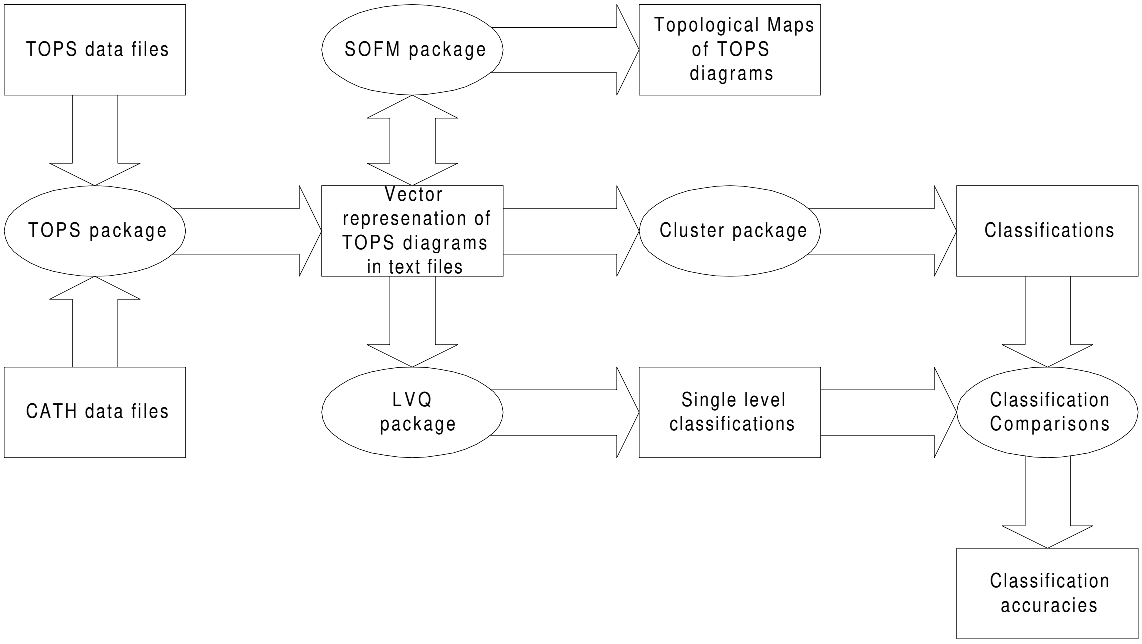

Figure 7-1: Overview of the classification system

This diagram shows the important functionality and connections in the classification system. Rectangular shapes denote data and round shapes denote processes, i.e. programs / software packages. Note that the SOFM package, whose main output is the topological maps, also produces vector representations of TOPS diagrams in text files with the same format as the files used by the LVQ package. The output from the SOFM package can therefore be used as the basis for both supervised and unsupervised classification. This is shown by the bi-directional arrow between the SOFM package and the vector representations. The classification comparison is actually performed by the cluster package, but for clarity it is here shown separated.

7.2. The TOPS package

7.2.1. Overview

The TOPS package can be further divided into several sub-units, see figure 7-2, and uses abstract representations of the TOPS diagrams internally. This enables fairly complex properties to be found easily, as well as making it easier to find new features. The clustering functionality is generalised and separated from the TOPS package. While in most cases it is called from the TOPS package, it can also operate on text files and is treated as a separate package in the next section.

Initially, the program reads the TOPS and CATH data files. The files are actually read only once, and later stored in an internal format for faster access at start-up. Several different vector representations are possible – new representations can be designed at run time – and which TOPS diagrams to include can be specified. Commands are specified at the command line, given in a script file or entered interactively. For instance, the *command "java ns.tops.Main –Ddebug.level=0 @cluster1.txt"* sets the debug level to zero and then runs the file script file *"cluster1.txt"*. Typical script files will set up parameters and then run a command like "cluster" to perform clustering, using the given parameters. The other important command is "generate", which generates data files for processing by the SOFM package or the LVQ package.

The following paragraphs describe the different aspects of the package. The appendix includes UML static structure diagrams.

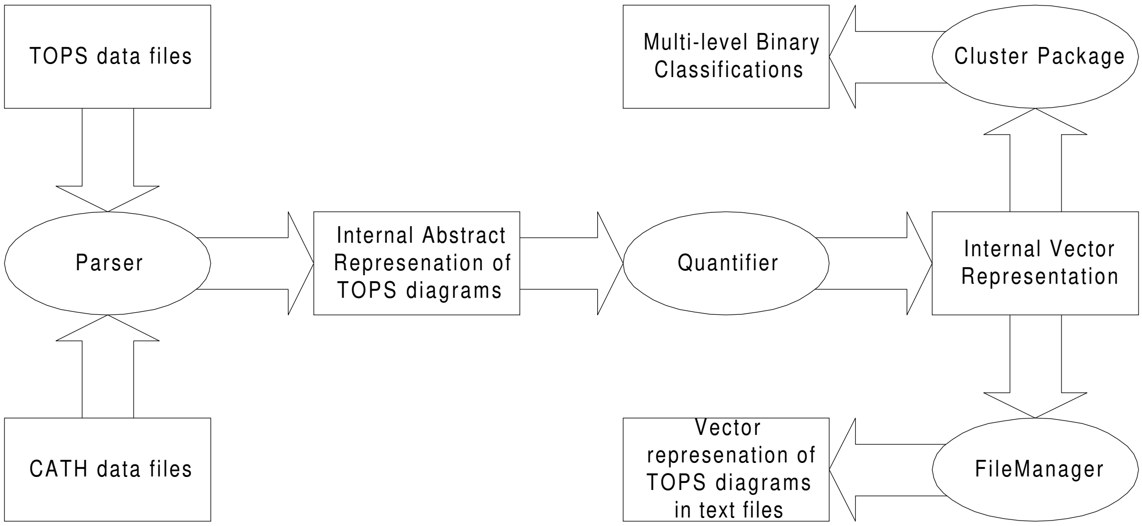

Figure 7-2: The TOPS package

A block diagram describing the main entities of the TOPS package. Note that the cluster package, while called from the TOPS package, is a separate entity. It can operate on text files of the same format as the LVQ and SOFM package. Normally, however, the cluster package will be called directly from the main program in the TOPS package, as shown here.

7.2.2. Internal abstract representations

Remember that, formally, a diagram is a set of SSEs and relations on those SSEs (paragraph 3.2.1). This is reflected in the class *Diagram*, representing TOPS diagrams: Each diagram has an array of *SSE*s, an array of *Chiralities* and an array of *HBonds* relations. In addition, each diagram has a domain name (see footnote on p. 19) and a CATH entry named *CATHEntry* (classes in italics). The CATH entry is assigned by looking up the domain name in the CATH file, and is used for selection of diagrams and for later identification.

The structural information found in TOPS diagrams is stored as an array of *Structures*. Each structure is one of the types mentioned in 3.2.2 and consists of a type label and a reference to which SSEs are part of that structure.

Each SSE can be one of four types: Alpha helix, Beta strand, N-terminal or C-terminal. Chiralities and handedness relations, being binary, has a direction and a left and a right SSE (or a from and a to SSE, in the case of HBonds).

Constructing the internal data representation is the job of the parser. Based on a modified version of a *Backus-Naur Form* (BNF) description of the format of the different data files, Java code that parses the data is automatically constructed. This is done using a parser generator known as JavaCC ("Java compiler compiler")[1]. The grammar and source code for the parser is found in the file "Tops.jj" in the appendix.

[1] See http://www.metamata.com for more information.

7.2.3. Extracting quantitative information – the Quantifier

The important aspect of the TOPS package is its ability to extract quantitative information from diagrams. In order to maintain generality, this functionality is separated into two classes, using one class to extract numerical properties from diagrams and another class to construct a complete numerical vector from a diagram. These two classes are *DomainProperty* and *Quantifier*, respectively, and they represent functions: *DomainProperty: Diagram → float* and *Quantifier: Diagram → float[]*. Both are abstract superclasses, and their subclasses will be described below. We will refer to instances of the Quantifier class as quantifiers and instances of the DomainProperty class as domain properties or simply properties.

The reason for this separation is that it allows us to use properties of diagrams to build complex aggregated rules, specifiable at run time. At the same time, not all quantifiers are aggregations of domain properties, this goes for quantifiers that has a global context, or quantifiers for which suitable DomainProperties do not exist. The *PairwiseQuantifier*, described below, is an example of the latter.

To all quantifiers is assosicated an array of corresponding *weights*; that is, each number of the quantifiers output vector can be weighted independantly of the other. This allows weighting according to importance, or adjusting for different magnitudes among equally important fields.

Five subclasses of the *DomainProperty* class have been implemented, these range from simple, e.g. the number of SSEs of each kind, to complex, like the ratio of parallel hydrogen bonds to anti-parallel.

The domain properties implemented so far are:

- SSE count; the number of SSEs of each of the four kinds (strand, helix, N-terminal, C-terminal). This class can thus have four distinct instances.
- Structure count; the number of super-secondary structures of each of the six kinds (see 3.2.2). This class can thus have six distinct instances.
- Chirality; the ratio of left-handed chiralities to right-handed.
- Crossings, i.e., the number of times the backbone crosses from one side of a sandwich to another[1].
- Parallel; the ratio of parallel h-bonds to anti-parallel h-bonds[2].

These DomainProperties represent the most essential properties of a diagram. Normally the first two properties will be the most important, as will be evidenced in table 8-1, see below. With these properties, most of the local structural properties in a diagram are captured. Other properties could focus on the long-range interactions between distinct structures, to capture the essentials of (classes of) proteins. As will become clear in the next chapter, in some cases such a global view seems to be needed to extract the characteristics that distinguish one class of proteins from another.

Two subclasses of the *Quantifier* class have been implemented, the *Rule* class and the *PairWiseQuantifier* class. The first represents aggregations of domain properties – it is simply an (ordered) set of domain properties. It can be represented as a text string, containing names of domain properties and optionally weights. Entries are separated with comma, e.g. “strands, sheets*2” represents an simple rule that returns the number of strands and the number of sheets multiplied with two.

---

[1] This property only applies to sandwiches. The number is actually the average over all sandwiches in a diagram, or zero if the diagram contains no sandwiches.

[2] This property only applies to sheets and is the average over all sheets in a diagram, or zero if there are none.

Following a common approach in quantification of sequences over a limited alphabet, the *PairwiseQuantifier* counts the occurrences of pairs of succeeding SSEs. It returns four values, the number of alpha-alpha pairs, the number of alpha-beta pairs etc, so that given a diagram with the SSE sequence *"hehehee"* the (unweighted) output would be *[0 3 2 1]*. Several classification projects have used this or similar approaches within the field of computational biology, and it is sometimes called a *neighbourhood matrix*.

In addition to the numerical values, the result of a quantification of a diagram includes a simple representative string. This string is important, as it is used to determine the class of that vector. Normally, the string will consists of the CATH identification of the diagram, down to some given accuracy, depending on at which level we want to classify diagrams. Alternatively, the string id could be the PDB domain id or a combination of PDB and CATH codes.

7.2.4. Selections

To extract selection from the total set of diagrams, an instance of the *CathMatcher* class is used. Basically, it is an abstract data type representation of a CATH identity string that allows wildcards; e.g. *"2.60.*"* matches all diagrams in any topology of the sandwich-architecture of the mainly-beta class. Which selection to use is decided at run time, by parsing a user-supplied expression.

7.2.5. Statistics

The result of applying a quantifier need not only be used for classifying. The class *Stat* has methods that, given a selection of diagrams and a quantifier, output several statistical indices (average, standard deviation, co-variance and kurtosis). By providing a rule that lists (unweighted) domain properties as quantifier, information of average number of strands and helixes, average number of sandwiches etc is listed for each CATH category of interest. This information can then be used for further statistical analysis.

7.2.6. Data files

Each time the CATH database changes a new version of the *CATH transition table* is released. This text file is a mapping from PDB-id's to the five-level CATH classification, and is read by the parser when constructing the CATH map used internally. Other input data files include the TOPS database described in section 3.2.4, and a text file containing a list of rules (Rule quantifiers). This last file is read at start-up and can be modified by the user.

An intermediary file format, not described in the block diagrams, is used for faster access at start-up. It is a binary *serialisation*, utilising Java's built-in object persistence mechanism. When the program is installed and whenever necessary, for instance when new CATH or TOPS data files are downloaded, the program parses these files, creates the internal representation and stores this. This decreases the start-up work and saves space.

The output files are constructed using the format of the SOFM and LVQ packages. The files start with a line containing a single number n, the length of the numerical vectors. Each proceeding line consists of n floats and ends with an identification string. The function of the identification string varies with the use of the file, but, in general, it appears as a *class label*.

All files can be stored or read in zipped format simply by providing a filename that ends with *.gz*; this is the default for serialisation, CATH and TOPS data files.

7.3. The SOFM package

The *Self-Organizing Map Program Package*, or *SOM_PAK* for short, is a software bundle designed by the SOFM programming team of the Helsinki University of Technology. Its authors include Teuvo Kohonen, the inventor of the SOFM algorithm. The bundle consists of programs for initialising, training and visualisations of feature maps, and is available from the Helsinki University[1] free of charge for researchers and students.

Programmed in ANSI-C, SOM_PAK is available for a broad range of platforms, including MS-DOS and most UNIX flavours. It operates on flat ASCII text files, given as parameters to the different programs.

A detailed description of the different programs in the package can be found in the accompanying documentation [19]; the treatment in this section will be brief.

Initialising a feature map consists of selecting a set of reference vectors, the codebook. Two programs perform this: *randinit*, which initialises the codebook vectors to random values, and *lininit*, which uses the eigenvectors of the input data vectors for initialisation. Note that both initialisation schemes normally lead to the same result, although *lininit* may converge faster. For this thesis *randinit* was selected, with a map size of 20 by 20 (ie, the output layer has 20 times 20 neurons).

Training the network is done by the main program, *vsom*; several runs of this program are normally required. In addition to names of data files, the number of training iterations and the *learning parameter* must be specified. Following Kohonens recommendations, the learning parameter will decrease linearly to zero from the specified value. For our purposes training periods of length 2000 and 20000 were deemed sufficient, using learning parameters of 0.1 and 0.02, respectively. These values are based on Kohonens recommendations, and while further experimenting probably could find better values, they are likely to be adequate.

After using the program *vcal* to label the map units (neurons in the output layer) according to the samples in the input data file, the program *umat* is run. *Umat* produces standard Post-script maps, where the map units are labelled with the label of the best-matching input sample.

7.4. The LVQ package

LVQ_PAK, or *The Learning Vector Quantization Package*, is another software bundle provided by Helsinki University of Technology. Developed by (almost) the same people, LVQ_PAK shares enough similarities with SOM_PAK to be treated in the same manner and the data files can be used interchangeably. Like SOM_PAK, LVQ_PAK is programmed in ANSI-C, available for most major platforms, free of charge for use in research and available from the same address.

The LVQ_PAK operates in a similar manner to SOM_PAK. First, a set of reference vectors is chosen. LVQ_PAK provides two initialising programs, *eveninit* and *propinit*. *Eveninit* allocates the same number of reference vectors to each class, while *propinit* uses proportional initialisation (see §6.3.2).

After initialisation, the program *balance* may be called. *Balance* performs one step of the OLVQ1 algorithm and then adjusts the codebook vectors so that the medians of the shortest distances between the codebook vectors in all classes are equalised.

---

[1] URL: ftp://cochlea.hut.fi/pub/

The next step is training the LVQ. Four programs are provided: *olvq1*, *lvq*1, *lvq2* and *lvq3*, corresponding to the four algorithms described in section 6.3.3. In this work, only *olvq1* is used; it uses default optimised parameters and only the number of iterations needs to be provided. For our purposes, 10000 iterations are considered sufficient, following Kohonens recommendations.

To actually perform a classification, the program *classify* is used; given a set of input vectors, it assigns to each sample a class label. The accuracy of this classification can be measured by the program *accuracy*. Note that when evaluating the quality of a classification, the mapping between the *correct class label* and the *class label returned by the* classify *program* are known. This makes comparisons easy: Simply measure the ratio of input samples classified correctly to the total number of input samples. *Accuracy* yields this ratio for each class and the total.

In order to verify the correctness of classifications, all data files used in this thesis are split into a *training set* and a *test set*. The test set is not used in any *codebook changing* operations, only when measuring the quality of classifications. The samples in the test set are either chosen randomly, with a given probability distribution, or chosen so that each relevant class is represented exactly once.

7.5. The Cluster Package

7.5.1. Overview

The Cluster Package is another software bundle. Implemented in Java, for reasons stated earlier, the package clusters general Java objects using either the isodata algorithm or standard hierarchical clustering. It can also operate on text files, using a simple data format compatible with that of LVQ_PAK and SOM_PAK.

Its main components are the *Node* class, used for representation of binary trees, the *ClassificationComparator* class, used for comparisons of matrices and classifications, and two classes that perform clustering.

7.5.2. Hierarchical clustering

The class *HierarchicalClustering*, as the name implies, performs hierarchical clustering using the algorithm described in 6.5.2. The default linkage method is *complete link*, ie, the distances between two clusters is the maximum of the distances between any two nodes in separate clusters. Other linkage methods implemented are *single link* (minimum of distances) and *simple average* (maximum distance, but adjusted for number of nodes in a cluster). Taking as input a (one-dimensional) array of objects to be clustered and a distance function, the method *cluster* returns the *root node* of a *binary tree*. In the case of the TOPS package, the typical parameters would be an array of diagrams and a distance function based on the Quantifier class[1]. The nodes are instances of the class *Node*, which has operations for accessing the left or right sub-tree or, in the case of leaf nodes, the leaf item itself.

The class has a *main* method and so can be called from the command line. The only parameter is the name of a file, and the program assumes the file to be of the same format as the data files used by

[1] The simplest case being, of course, the Euclidean distance between the vectors returned by the Quantifier.

SOM_PAK and LVQ_PAK (7.2.6). Processing the vectors and assigning the string ids as names, Euclidean distance between the vectors is used as a distance function. A binary tree is then constructed. The output of the program consists of a parenthesised version of the tree, as in *{ { x1, x2 }, { { y1, y2 }, z1 }*. This standard notation can be understood by standard software packages for visualisation or processing.

The implementation is kept simple and runs in $O(n^4)$ worst-case time, where n is the number of objects to cluster. Faster solutions are possible, but, for our usage, the efficiency of the package is sufficient: Clustering 1000 instances takes less than a minute on a standard desktop computer.

In addition to the clustering functionality, a method that merges (sub)-trees according to specific criteria is implemented. This method takes as input a maximum value for the distance between two nodes in a cluster, and returns a classification (*Collection* of *Collections*) in which all subnodes of trees with an internal distance less than the given value are merged. In this manner, a single level classification is returned, and this classification can then be compared to other classifications with the methods described later. Note that this classification method is primitive, and if a single-level classification is desired, other algorithms perform better.

7.5.3. Distance measures

To measure the distance between diagrams, an abstract super-class *Distance* was defined. The only useful implementation is the *QuantifierDistance*, which, as the name implies, is based on a quantifier. Given two diagrams, the distance function computes the Euclidean distance between the vectors returned by the quantifier. The value returned is the result of a sigmoid function applied to this distance, to have a fixed range for the values. The sigmoid function in question is the logistic function $\phi(x)$, defined as

$$\varphi(x) = \frac{1}{1 + e^{-ax}}$$

7.5.4. The *Isodata* class

The *Isodata* class contains methods to perform clustering using the isodata algorithm described in 6.6. In addition, methods for constructing proximity matrices and calculating the Hubert Γ statistic are implemented. For efficiency, the representation of classifications is different: Normally, a partition P of a set of objects $S=\{s_i\}$ is represented by an array of integers, where the integers denote cluster labels; this means that the object s_i is in cluster P[j]. Note that the integer labels are only internal labels, and not comparable to other classifications. Thus, if we have two partitions P_1 and P_2, comparing them is done by comparing class labels within the partitions, and not between. Given two objects s_i and s_j, they are in the same cluster in P_1 if and only if $P_1[i]=P_1[j]$, similarly for P_2. Methods for changing from one representation to another are also implemented.

The main method is *cluster*, which takes as input an array of real-valued vectors and the desired number of clusters. Note that with the isodata algorithm, the number of clusters must be specified and the algorithm is only guaranteed to converge with the squared Euclidean distance function, thus no other parameters are needed.

The implementation is simple yet efficient; clustering 10000 vectors of size 2 into 8 clusters takes approximately 10 seconds. The complexity is roughly $O(N^x m^x r^x s)$, where N is the number of entries, s is the size of those entries, m is the number of clusters and r is the number of iterations necessary. Typical values for r are 10-50.

7.5.5. Comparing classifications

The four statistical indices described in section 6.7 are implemented as a part of the clustering package and are found in the class *ClassificationComparator*. The method *compare* takes as input the two collections to be compared and an integer identifying the method to use (*RAND*, *JACCARD* or *FOWLKE*), and returns a float value with the desired index. The method *hubert* takes as input two proximity matrices and returns a float value with the standard (not normalised) Hubert Γ statistic.

The method *montecarlohubert* takes as input a proximity matrix, a source of randomness and two integers r and m, and returns a sorted array of size r of real values. These values are the result of producing a random partition, performing an isodata clustering of the random partition and comparing the resulting classification to the proximity matrix.

All algorithm runs in $O(n^2)$, and in effect, the computations are more than fast enough (usually less than a second on a standard desktop computer). However, these comparison only works for single-level classifications. Methods that compare trees can be implemented, using similar techniques.

8. RESULTS

In this chapter the results obtained with the classification system so far is presented. Section 8.1 gives an overview of which results are included in the thesis and why. Sample topological maps are shown in section 8.2 along with explanations and interpretations. Section 8.3, the main section, presents a representative set of classifications obtained using various quantifiers, as well as the statistics obtained when comparing these classifications to the CATH classification. Finally, in section 8.4, the results of the unsupervised classifications are given.

8.1. Overview

8.1.1. Selections and general procedures

The TOPS classification system has been tested using the two kinds of quantifiers *Pairwise* and *Rule* described in the previous chapter. The quantifiers have been used for the creation of topological maps as well as for supervised and unsupervised classification. Another variation is represented by the selection of diagrams to work on; three selections were chosen:

- All diagrams – a complete set of all TOPS diagrams, which corresponds to the latest release of the BDP database and so with the most determined protein structures. With this selection, the diagrams were classified in one out of four categories and compared to the Class-assignment in the CATH system.
- All diagrams in the second Class in the CATH system, the *mainly beta* domains. This time the diagrams were classified in one out of 18 categories, the different Topologies in the mainly-beta class.
- All diagrams in the *sandwich* topology in the CATH system (2.60.*). As before, the classification is done corresponding to the categories in the sub-level in CATH – in this case, the 26 different Homologous super-families.

The first selection was chosen mostly as a test of the system: In CATH, classification at this level is based only on SSE counts. This means that classification using a simple rule that only counts SSEs should achieve high accuracy. The second selection was chosen on the grounds that TOPS cartoons are better suited to describe proteins with high beta strand content, due to their weak description of alpha helix interactions. The third selection was decided to be on the next level in the CATH hierarchy, and the data set chosen is simply the Topology with the highest number of proteins.

For each of these selections, two sets were generated, a training set and a test set. The training sets were used when using supervised classification and for the initial training of the SOFM when performing self-organising. The test sets were used for the final (visible) topological maps and for measuring the accuracy of the classifications performed using LVQ. The diagrams in the test set were chosen randomly, by picking 20 percent of the diagrams to be in the test set, and ensuring that all categories found in the training set had at least one representative in the test set.

For each of the selections and for each of the five quantifiers – see below – the following procedure was applied:

1. The numerical output of the quantifiers on the given set was written to text files.
2. A self-organising map was trained on the training set, and topological maps were created using the test set and manually inspected.
3. A learning vector quantization was trained on the training set, classification performed on the test set and the corresponding accuracy found.
4. Clustering was applied to the complete set, and the classification compared to CATH.

For the supervised classifications, all classes are labelled. This means that the classification accuracy is simply a count of matches: For each category produced by the LVQ algorithm, the entries were compared to the corresponding category in CATH and the percentage of correctly classified entries computed. In the unsupervised classifications, there are no comparable class labels, so the results of comparisons are the statistical indices mentioned in section 7.5.4.

8.1.2. Quantifiers

Five quantifiers were tested, one Pairwise instance (*P*) and four Rule instances (*R0-R3)*, see section 7.2.3. The emphasis has been on the Rule quantifier, as this is the most general kind.

The four Rule quantifiers used were

- *R0*: “strands/6.4623, helices/4.9025, sheets/0.9908, barrels/0.063, csheets/0.0701, vcsheets/0.0252, sandwiches/0.2257, unknowns, parallel/0.0745, crossings/0.6405, chiralities/1.056”, where all properties are weighted with the inverse of the average value over all diagrams.
- *R1*: “ strands/9.1706, helices/2.1506, sheets*2, barrels*5, csheets, vcsheets*5, sandwiches*5, unknowns, parallel, crossings/1.6352”, based on the properties considered important, and partly weighted based on the total statistics for all diagrams.
- *R2*: “strands, helices, sheets, barrels, csheets, vcsheets, sandwiches, unknowns, parallel, crossings, chiralities”, that is, a unweighted listing of all implemented properties.
- *R3*: “Strands, helices“, that is, only counts number of strands and helixes.

The weights given here are either one (whenever not specified), the inverse of the average value, or some “suitable” number, based on the importance of the corresponding property. This prevents fields with large values (eg the number of SSEs of a specific kind) dominating the vectors and thus biasing the classifications. Being a naive approach, this weighting could be improved by weighting statistically significant fields higher than fields found to be irrelevant.

8.2. Topological maps

Fifteen maps were created, of which three are showed in figures 8-1 to 8-3. The maps were created using a SOFM with 25 neurons organised in a 5-by-5 grid. The SOFM was trained on the selections described in the previous section, using LVQ1 as learning algorithm for 22000 iterations.

After the training phase, a test set was chosen. These entries were then run through the network, and a winner neurone chosen. The class label (in this case, the CATH id) was written on the winner neurone's place in the topological map. Note that only one class label is shown; this means that if two entries with different class labels produces the same winner neurone, the class label of the second entry will hide the first. This is a limitation in SOM_PAK, which could have been avoided by using the raw data to create customised graphical presentations. That, however, was considered beyond the scope of this thesis.

While showing some interesting properties, the maps are not unambiguous: Diagrams from the same classes are often placed far apart, and diagrams from different classes are sometimes packed close together. Yet some coherence is obvious: The diagrams tend to be grouped together with diagrams of the same CATH category. The effect is somewhat easier to see when using maps with a higher number of neurons (maps with 50 neurons were tested), but because of their high resolution they are not suitable for printing and so are not included here.

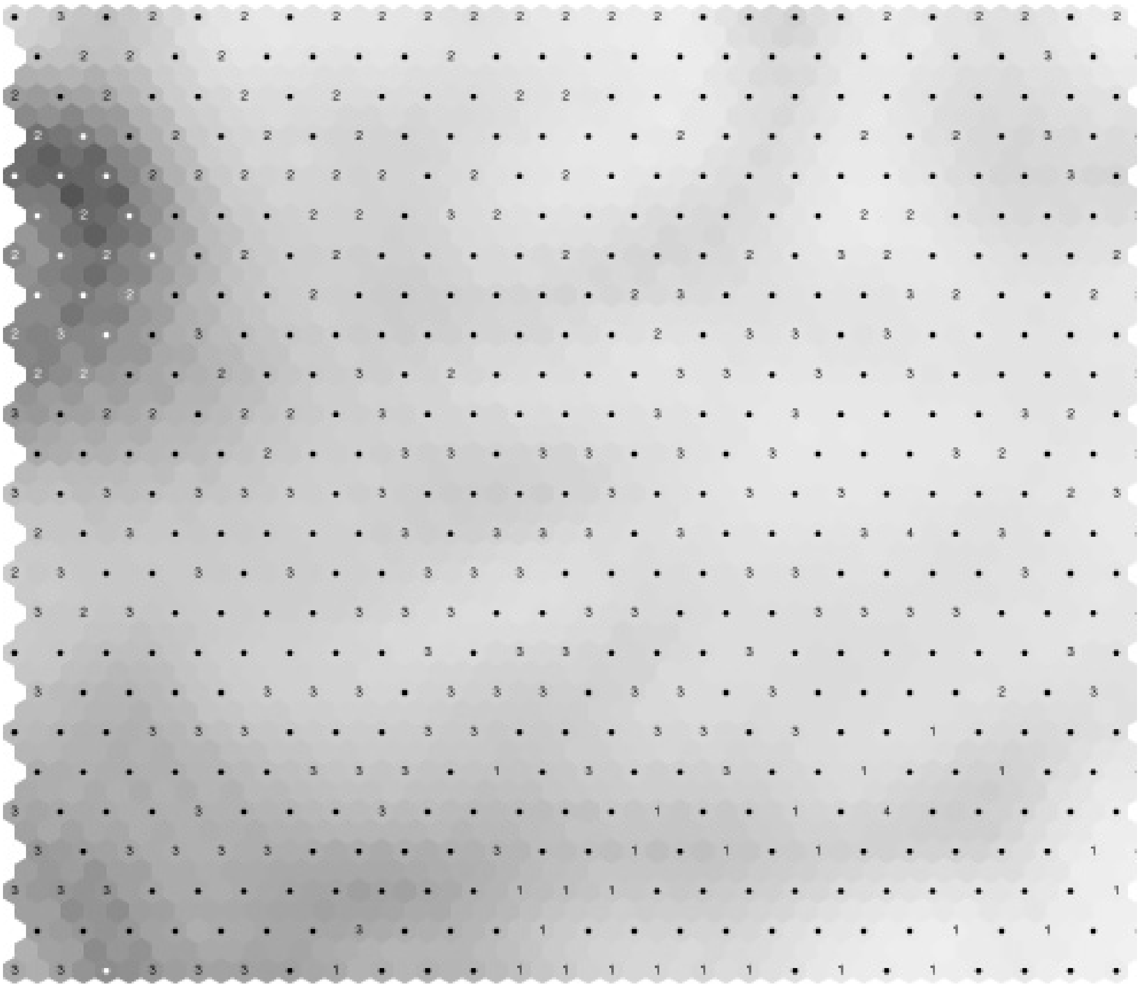

Figure 8-1: Topological map of all domains

This topological map shows relationships between diagrams from separate classes in the top level of the CATH hierarchy (Class). The 25x25 points represent neurons. If an entry with class label x *is "won" by neuron* y*, neurone* y *has the class label* x *associated with it. It can be seen that diagrams tend to be found next to other diagrams from the same CATH category, for instance the mainly-alpha(1) diagrams are found in the lower (right). The greyscale on the map mark areas with large differences, while areas with mostly white background are similar (less different). This suggest, for instance, that the group of mainly-beta domains found in the upper left corner are distinctly different from their neighbouring mainly-beta domains. The fact that the alpha+beta diagrams (class 3) are spread around the map, can be explained by looking at the quantifier used. As the quantifier takes beta-interactions into account, and the alpha-helix description in TOPS is weak, variations within the alpha+beta class will be larger than variations within the other classes.*

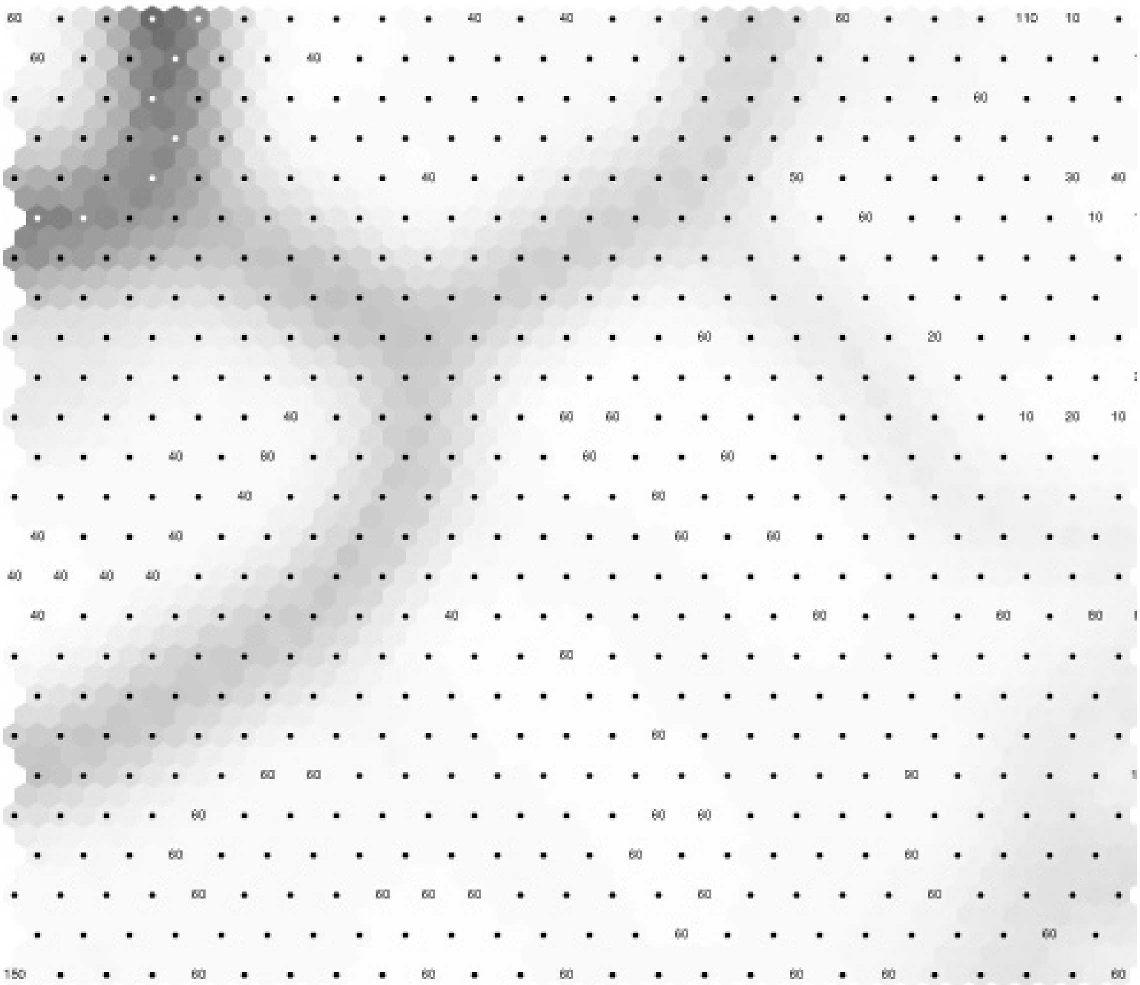

Figure 8-2: Topological maps of mainly-beta domains

In this map, only protein domains belonging to the mainly-beta class were used. For clarity, the labels only show the architecture – the class, for domains in this map, is mainly-beta (2). Note the boundaries between, e.g., the 2.40 and the 2.60 groups in the upper left corner. Another interesting group is found in the upper right corner, where several proteins from different architecture are in the same group. This might suggest that the differences between those vectors are smaller than their difference from some of the other groups, e.g. 2.40 or 2.60. The map also reflects the fact that the majority of the proteins belong to the 2.40 or the 2.60 architecture.

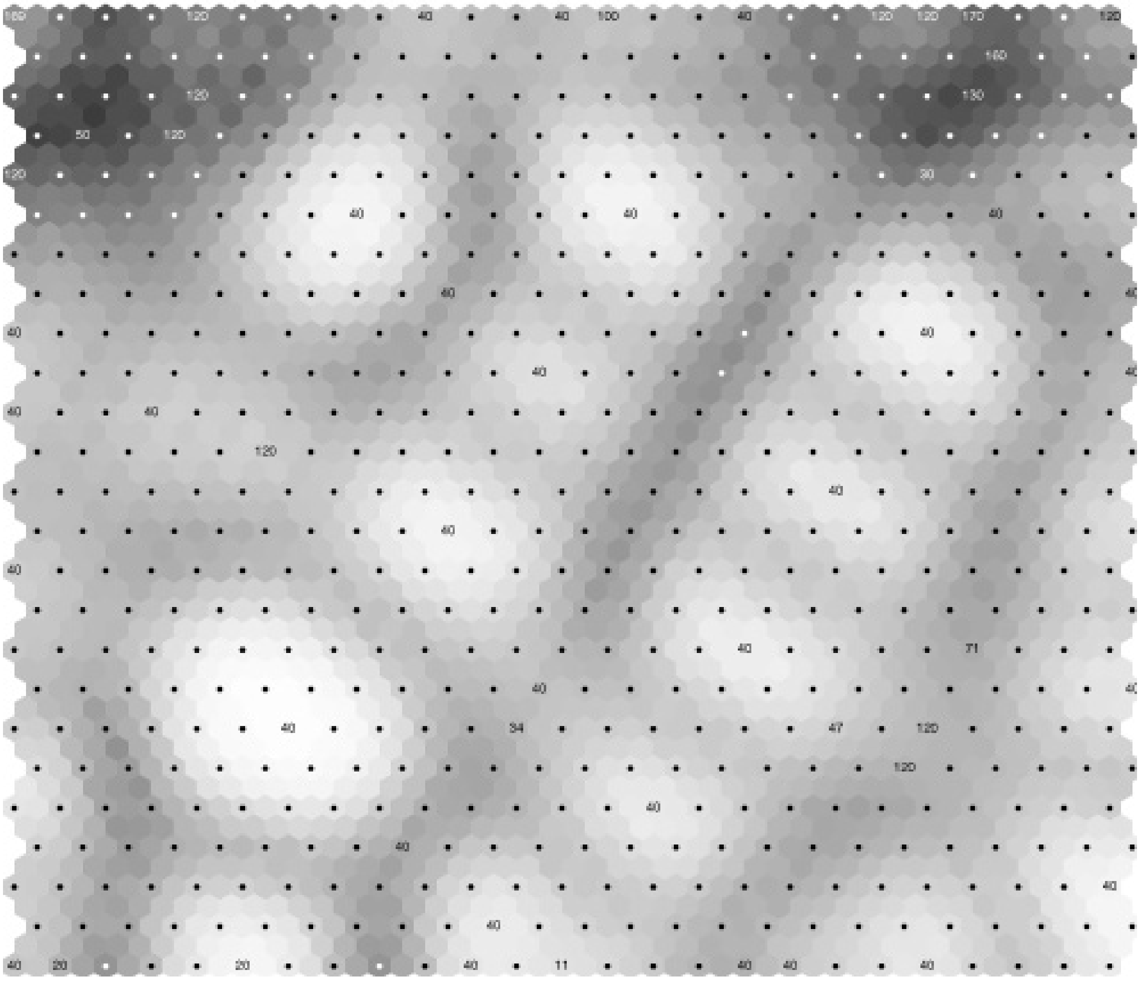

Figure 8-3: Topological maps of sandwich topologies

In this map, only protein domains belonging to the sandwich architecture of the mainly-beta class (2.60) were used. The labels only show the topology id. Again, the map reflects the fact that the majority of the proteins belong to two topologies, the 2.60.40 and 2.60.120. The 2.60.40 topologies are spread all over the map, with clear boundaries. With few exceptions, all other topologies are found close together in the upper right and left corner, or at the bottom borderline. This might indicate that with this quantifier, the immunoglobulin-like topologies (2.60.40) have greater variation of possible values than the other topologies. The jelly rolls (2.60.120), on the other hand, seems to consist of two easily distinguishable classes.

8.3. LVQ classifications

For each of the three selections, similar tables are produced, listing the output from the LVQPAKs *accuracy* program (7.4). The tables show classification accuracy for each of the five quantifiers, and the weighted total in percent.

The accuracy of classification on the top level is rather good, but this is to be expected, due to the simple classification procedure used at this level in CATH. All five quantifiers yield approximately the same results on the different classes except for the fourth class where the results vary from 52% to 63%.

Class label	Size	R0	R1	R2	R3	P
1	85	94.12	85.88	92.94	90.59	84.71
2	151	88.08	86.75	90.07	90.07	89.40
3	172	80.23	80.81	83.14	76.16	77.91
4	19	57.89	52.63	63.16	63.16	52.63
Total	**427**	**84.78**	**82.67**	**86.65**	**83.37**	**82.20**

Table 8-1: Supervised classification of all classes

This is the accuracy obtained when classifying all diagrams in the database, which corresponds (more or less) to all determined protein structures. Note that class 4 is "Other", domains not grouped in the first three categories for some reason, usually for having low secondary structure content. Bad accuracy for this category should be expected, especially considering the low number of entries. This holds true for all quantifiers, so at least they fail consistently, which is interesting by itself.

Table 8-2 and 8-3, showing the results when classifying mainly-beta diagrams and diagrams from the *sandwich* architecture in the *mainly-beta* class, respectively, are more interesting. The values this time cover a wider spectrum, which is natural considering that R3 and P only takes into account the amount (and order) of alpha and beta SSEs, without considering structural properties any further. R0, R1 and R2 count the number of super-secondary structures, in addition to the SSE count, and predictably have the highest total score. This is explained by looking at the different architectures in the mainly-beta class. Remember that when classifying into different architectures the focus is on the overall shape, and super-secondary structures must be taken into account for classifications to be feasible.

Class label	Size	R0	R1	R2	R3	P
2.10	11	81.82	63.64	63.64	27.27	27.27
2.20	7	85.71	85.71	85.71	85.71	71.43
2.30	5	20.00	40.00	20.00	40.00	40.00
2.40	42	73.81	76.19	85.71	45.24	42.86
2.50	1	0.00	0.00	0.00	0.00	0.00
2.60	81	92.59	98.77	92.59	75.31	87.65
2.70	1	0.00	0.00	0.00	0.00	0.00
2.80	3	66.67	0.00	33.33	0.00	0.00
2.90	1	0.00	0.00	0.00	0.00	0.00
2.100	1	0.00	0.00	0.00	0.00	0.00
2.102	1	0.00	0.00	0.00	0.00	0.00
2.110	1	0.00	0.00	0.00	100.00	100.00
2.120	2	100.00	100.00	100.00	100.00	100.00
2.130	1	100.00	100.00	100.00	100.00	100.00
2.140	1	0.00	0.00	0.00	0.00	0.00
2.150	1	0.00	0.00	100.00	0.00	0.00
2.160	1	0.00	0.00	0.00	0.00	0.00
2.170	2	0.00	0.00	0.00	0.00	0.00
Total	**163**	**77.91**	**79.75**	**79.75**	**58.28**	**63.19**

Table 8-2: Supervised classification of mainly-beta diagrams

Accuracy when classifying all diagrams in the mainly-beta class. Note that the quantifiers that only consider SSE contents and organisation, R3 *and* P, *score noticeably lower than the quantifiers that take super-secondary structures into account. The numbers also suggest that 2.120 and 2.130 are distinctly different from the other vectors, as they are both classified correctly by all quantifiers despite having only one and two representatives, respectively.*

The total accuracy, varying from below 60% to 80%, is rather low but still useful. However, the problem is the difference in magnitudes: A majority of the categories has only a single entry, and as the data-set generating algorithm chooses at least one from each category to be in the test set, the categories with only one entry (singletons henceforward) are not necessarily represented in the training set at all. Out of 18 categories in the mainly beta class only 9 have more than one topology. Unfortunately, all categories in the CATH classification system share this property: At the second and third level of classification many (>50%) groups have only a single element. A solution to this problem is to include the singletons in both sets, but then the test would not be representative: Testing a supervised algorithm on entries that are part of the training set is not appropriate. Another solution is to exclude the singletons from the test set. While this would undoubtedly be interesting, and surely raise the total scores, it was felt that the test set should be as representative as possible. Thus using LVQ on these datasets is error-prone, or rather inaccuracy-prone: Unless the single entries are highly distinguishable, chances are high that entries belonging to those categories will be classified incorrectly.

At both levels, a few categories contain most of the entries. This leads to a high total score, although, in fact, most categories are not used at all. Being a property of the nature of the classification of CATH, this is a problem at most levels. A way to compensate would be to weight the singular entries higher, e.g. by duplicating those vectors. However, unless the entries are significantly different from the other entries, this will not work well.

On the third level, all quantifiers yield approximately the same accuracy, between 60 and 70 percent, which by it self could be considered satisfactory. However, again all quantifiers fail at classifying singletons; even the best quantifier only classified 2 out of 18 singletons correctly.

Another interesting fact is that the R0, R1 and R2 quantifiers, which differs mostly in weights and not in which properties are included, have only minor differences in scores on all levels. This is to be expected: The SOFM algorithm is robust enough not to be (too much) influenced by differences in magnitude between different elements of the vectors.

Class label	**Size**	**R0**	**R1**	**R2**	**R3**	**P**
2.60.9	1 entries	0.00	0.00	0.00	0.00	0.00
2.60.11	1 entries	0.00	0.00	0.00	0.00	0.00
2.60.20	4 entries	0.00	0.00	0.00	0.00	0.00
2.60.30	2 entries	0.00	0.00	0.00	0.00	0.00
2.60.34	1 entries	0.00	0.00	0.00	0.00	0.00
2.60.35	1 entries	0.00	0.00	0.00	0.00	0.00
2.60.40	64 entries	95.31	95.31	96.88	96.88	95.31
2.60.43	1 entries	0.00	0.00	0.00	0.00	0.00
2.60.47	1 entries	0.00	0.00	0.00	0.00	0.00
2.60.50	1 entries	100.00	0.00	0.00	0.00	0.00
2.60.60	1 entries	100.00	100.00	100.00	0.00	0.00
2.60.71	1 entries	0.00	0.00	0.00	0.00	0.00
2.60.80	2 entries	0.00	0.00	0.00	0.00	0.00
2.60.90	1 entries	0.00	0.00	0.00	0.00	0.00
2.60.97	2 entries	0.00	0.00	0.00	0.00	0.00
2.60.98	1 entries	0.00	0.00	0.00	0.00	0.00
2.60.100	1 entries	0.00	0.00	0.00	0.00	0.00
2.60.110	1 entries	0.00	0.00	0.00	0.00	0.00
2.60.120	21 entries	61.90	61.90	71.43	61.90	71.43
2.60.130	1 entries	100.00	100.00	100.00	0.00	0.00
2.60.160	1 entries	0.00	0.00	0.00	0.00	0.00
2.60.169	1 entries	100.00	100.00	100.00	100.00	100.00
2.60.170	1 entries	0.00	0.00	0.00	0.00	0.00
2.60.175	1 entries	0.00	0.00	0.00	0.00	0.00
Total	**113 entries**	**69.03**	**68.14**	**70.80**	**67.26**	**68.14**

Table 8-3: Supervised classification of sandwich diagrams

The accuracy obtained when classifying diagrams in the sandwich architecture. Again all quantifiers score well on the largest group, drawing their total scores up, but fail to classify singletons correctly.

8.4. Clustering

The cluster package was used to perform clustering on the selections and quantifiers given in section 8.1. Both the hierarchical and the isodata algorithm were tried. The results of the hierarchical clusterings were binary trees, but for the purpose of comparison, the binary trees were collapsed to flat, single-level classifications. These classifications were then compared to CATH using the *ClassificationComparator* class. The results of the isodata clusterings were single-level classifications, which also were compared to the CATH classification.

The isodata algorithm performed considerable better than the hierarchical algorithm, as expected. This might be caused by inadequate comparisons: Collapsing a hierarchical structure by the simple means provided in the cluster package can loose too much information. See [13] for better methods to identify the significant structure in a hierarchical clustering, or for methods to compare hierarchical clusterings to single-level clusterings. The statistics obtained by the isodata algorithm on the three selections are given in table 8-4 to 8-6.

Note that this does not constitute a complete clustering process, including cluster validation. It is to be considered a test of the system and a basis for further analysis.

	R0	**R1**	**R2**	**R3**	**P**
Rand statistic	0.44	0.67	0.68	0.67	0.69
Jaccard coefficient	0.26	0.31	0.32	0.31	0.34
Fowlkes and Mallows	75.86	NaN	18.37	NaN	34.43

Table 8-4: Unsupervised classification of all diagrams

Sample statistical indices when comparing isodata classifications at the top-level to the classification in CATH. NaN *(Not-a-Number) represents overflow, which (usually) indicates a high level of agreement. Note that different statistical indices yield different relative results, e.g., according to the Fowlkes and Mallows index R0 is better than R2 while according to the rand statistic R2 is better than R0.*

	R0	R1	R2	R3	P
Rand statistic	0.74	0.70	0.71	0.64	0.64
Jaccard coefficient	0.36	0.25	0.28	0.21	0.18
Fowlkes and Mallows	1.78	1.66	NaN	NaN	1.10

Table 8-5: Unsupervised classification of mainly-beta architectures

Sample statistical indices when comparing isodata classifications of mainly-beta architectures to the classification in CATH. Note the comparatively lower rand statistics of the "simple" quantifiers P and R3, the same effect is found when using supervised classification (see discussion in 8.3).

	R0	R1	R2	R3	P
Rand statistic	0.62	0.66	0.71	0.71	0.75
Jaccard coefficient	0.18	0.16	0.36	0.37	0.35
Fowlkes and Mallows	0.31	0.28	0.54	0.55	0.51

Table 8-6: Unsupervised classification of barrel topologies

Sample statistical indices when comparing isodata classifications of barrel topologies to the classification in CATH. Note the comparatively higher rand statistics of the "simple" quantifiers P and R3, the same effect is found when using supervised classification (see discussion in 8.3).

As stated before, the statistical indices by them selves do not tell much. For all classifications, a Monte Carlo approach was used to find the estimated value of the Hubert Γ statistic, following the simplified hypothesis test outlined in 6.7.3. The result of this approach is given in table 8-7.

Selection (CATH entries)	R0	R1	R2	R3	P
*	429.59 360.71	62.21 51.96	167.57 138.52	146.66 119.22	114.88 96.33
2.*	445.19 422.90	77.16 73.37	116.32 112.51	91.78 88.68	76.38 73.44
2.40.*	596.63 569.50	48.38 47.62	70.28 72.76	63.72 65.18	52.44 52.10

Table 8-7: Monte-Carlo approximation of Hubert Γ statistic

This table shows the Hubert Γ statistic for the given quantifiers and selections. For each cell, the first (top) number is the Γ statistic for the isodata classification, and the second number is the expected value, estimated by Monte-Carlo approximation (100 iterations) under the random label hypothesis. In all cases where the calculated values were higher than the expected value, they were higher than all 100 Monte-Carlo values.

We observe that at the architecture and class levels the Γ statistics of the isodata classifications are higher than of all 100 random classifications. The obvious interpretation is that the isodata classifications are better than random, and that it has captured some clusters inherent in the data. Generally, if the statistic is higher (for right-tailed pdf distributions like that of Γ) than *100-p* percent of the random classifications we can assume, with a significance level of p, that the classification is significantly better than random. Note, however, that this is not a complete statistical analysis and further work is required to verify that, e.g., the random label hypothesis holds. The analysis, unfortunately, is beyond the scope of this thesis.

At the third level, the results are more disappointing and a bit confusing: With two of the quantifiers, R2 and R3, the expected values (according to the Monte-Carlo approximation) are higher than the actual values. This deserves further investigation.

9. Summary and Discussion

9.1. Overview

In this thesis, a framework for classification of higher-level structural descriptions of proteins is presented. The classification systems reads TOPS and CATH data files, and allows selection of diagrams, statistical analysis, unsupervised (hierarchical and isodata) clustering and, via external packages, supervised classification and the formation of topological maps.

The system is highly generalised and extendable. Being divided into logical modules, the system can easily adapt to other uses, for instance comparisons with other classification systems (than CATH) or other classification methods. The protein domains are represented using abstract data types, based on the structural information contained in TOPS diagrams. Numerical vectors are created using functions, of which several types exist. New functions can be defined, and this functionality deserves more testing. The clustering functionality is separated in a package by itself, and can be used for other clustering tasks.

Note that methods based on neural network or similar techniques are unlikely to be used as reference classification methods: For a fixed, standardised classification, deterministic, explicit and, preferably, intensional methods are needed. However, methods along the lines described here can be used for preliminary or tentative classifications and for analysis.

9.2. Discussion

The system has been tested on the current TOPS and CATH databases, with supervised and unsupervised classifications and the formation of topological maps. The classifications have been compared to the CATH system. A very high level of agreement being considered unlikely, the results of comparisons are promising but not satisfactory. In the case of proteins belonging to the mainly-beta class (for which the TOPS diagrams are more adequate, due to their weak alpha helix structure description), the highest total supervised classification accuracy was 80 percent. Classification accuracy depends heavily on which properties are included, and it is obvious that further statistical analysis of the properties is needed. This could also reveal interesting statistical properties of the different categories in existing (manual and semi-automatic) classification systems, e.g. SCOP and CATH.

The low accuracy problem is mainly caused by the singletons, i.e. categories with only one single entry; these are in general not classified correctly. Highest performance is found when using the LVQ classification system, as is natural: In most cases, supervised methods will perform better than unsupervised ones.

TOPS diagrams, being a formal and very compact structural description, have several useful features. Of these, pattern matching and searching is perhaps the most interesting: Due to their compactness, a database of all known (determined) structures can be searched in a small amount of time. It is therefore interesting to note that despite their simplicity, TOPS diagrams clearly retain structural information enough for classification to be possible. This is shown by the accuracy of these first feeble attempts of classifications.

As stated above, for the classifications to be considered satisfactory, a higher level of accuracy is required. Whether this can be achieved using the approach chosen in this thesis remains to be seen.

9.3. Further work

Clearly further work is required in this area of research. At first sight, a complete statistical analysis of the numerical properties of the TOPS diagrams could have been done on an earlier stage, to single out the important ones. Principal components analysis, or other statistical methods, could reveal which variables were significant at the different levels. In addition, more work should be done on quantifiers, to reveal more subtle characteristics of diagrams.

Another issue left unfinished is to combine the methods and framework described in this thesis with the work of Gilbert, Westhead and co-workers, described earlier, to include treatment of patterns in the TOPS package. This would enable easy retrieval of patterns found descriptive of CATH categories. Presumably, intensional approaches are better suited for this classification task.

Further work could also include:

- Integration with domain assigning and TOPS cartoons generating programs, in order to provide an initial / tentative / suggestive classification for CATH and SCOP.
- Statistical analysis of the results of the clustering algorithms.
- Compare with other classification systems, eg SCOP.
- Further experimenting with new rules and new domain properties.
- Making an applet and a web interface.
- Testing classification of patterns.

10. REFERENCES

[1] Thomas E. Creighton, 1993: "Proteins: Structures and molecular properties", 2nd edition. W. H. Freeman and Company, New York, USA

[2] João Setubal and João Meidanis, 1997: "Introduction to computational molecular biology". PWS Publishing Company, Boston, MA, USA

[3] Carl Branden and John Tooze, 1991: "Introduction to protein structure". Graland Publishing, London, UK

[4] Burkhard Rost and Sean O'Donoghue, 1997: "Sisyphus and prediction of protein structure", CABIOS **13**: 345-356

[5] A. D. Michie, C.A. Orengo, J.M. Thornton, 1996: "Analysis of Domain Structural Class using an Automated Class Assignment Protocol". J.Mol.Biol Vol.262, p.168-185

[6] W. R. Taylor and C. A. Orengo, 1989: "Protein structure alignment" Journal of Molecular Biology **209**, pages 1-22

[7] M. J. E. Sternberg and J. M. Thornton, 1977, *J. Mol. Biol..*, **110**, 269-283.

[8] T.P. Flores, D.M. Moss, and J.M. Thornton, 1994: "An algorithm for automatically generating protein topology cartoons", *Protein Engineering* **7**(1):31-37

[9] D.R. Gilbert, D.R. Westhead and J.M. Thornton, 1998: "A constraint-based system for protein motif-searching, pattern discovery and structure comparison", ERCIM/COMPULOG Workshop on Constraints, CWI, Amsterdam, the Netherlands

[10] Simon Haykin, 1994: "Neural Networks: A Comprehensive Foundation". Prentice Hall, London, UK

[11] I. Aleksander, H. Morton, 1992: "An Introduction to Neural Computing". Chapman & Hall, London, UK

[12] Teuvo Kohonen, 1997: "Self-Organizing Maps", 2nd edition. Springer-Verlag Berlin Heidelberg, Germany

[13] Sergios Theodoridis and Konstantinos Koutroumbas, 1999: "Pattern Recognition". Academic Press, San Diego, California, USA

[14] Edward R. Dougherty, 1990: "Probability and statistics for the engineering, computing and physical sciences". Prentice Hall, London, UK.

[15] D. E. Rumelhart and D. Zipser, 1985: "Feature discovery by competitive learning", Cognitive Science **9**, 75-112

[16] C. E. Shannon, 1948: "A mathematical theory of communication", Bell System Technical Journal **27**, p. 379-423, 623-656

[17] B. Widrow and M. E. Hoff, 1960: "Adaptive switching circuits" IRE WESCON Convention Record, pages 96-104

[18] P. J. Werbos, 1974: "Beyond regression: New tools for prediction and analysis in the behavioural sciences." Ph.D. Thesis, Harvard University, MA, USA.

[19] Teuvo Kohonen, Jari Kangas and Jorma Laaksonen, 1992: "SOMPAK: The self-organizing map package", Ver. 1.2. Helsinki University of Technology, Helsinki, Finland

[20] Teuvo Kohonen, Jari Kangas, Jorma Laaksonen, Kari Torkkola, 1992: "The Learning Vector Quantization Program Package", ver 2.1. Helsinki University of Technology, Helsinki, Finland

[21] Teuvo Kohonen, 1982: "Self-organised formation of topologically correct feature maps." *Biological Cybernetics* **43**, pages 59-69.

[22] E. Säckinger, B. E. Boser, J. Bromley, Y. LeCun and L. D. Jackel, 1992: "Application of the ANNA neural network chip to high-speed character recognition", *IEEE Transactions on Neural Networks* **3**, pages 498-505

[23] M. Cohen, H. Franco, N. Morgan, D. Rumelhart and V. Abrash, 1993: "Context-dependent multiple distribution phonetic modelling with MLPs.", In *Advances in Neural Information Professing Systems* (S. J. Hanson, J. D. Cowan and C. L. Giles, eds), pages 649-657. Morgan Kaufmann, San Mateo, CA, USA

[24] T. J. Sejnowkski and C. R. Rosenberg, 1987: "Parallel networks that learn to pronounce English text.", *Complex Systems* **1**, pages 145-168

[25] T. Kohonen, 1988: "The Neural Phonetic Typewriter", *Computer* **21**, p. 11-22.

[26] T. Kohonen, 1987: "State of the art in neural computing" *IEEE International Conference on Neural Networks* **1**, p. 77-89

[27] E. A. Ferran and P. Ferrara, 1991: "Topological maps of protein sequences.", *Biological Cybernetics* **65**, 451-458

[28] J. Orlando, R. Mann. and S. Haykin, 1990: "Classification of sea-ice using a dual-polarized radar" *IEEE Journal of Oceanic Engineering* **15**, pages 228-237

[29] H. J. Ritter, T. M. Martinetz and K. J. Schulten, 1992: "Neural Computation and Self-Organizing Maps: An Introduction". Addison-Wesley, Reading, MA, USA.

[30] J. S. Baras and A. LaVigna, 1990: "Convergence of Kohonen's learning vector quantization." International Joint Conference on Neural Networks **3**, pages 17-20. San Diego, CA, USA

Appendix A: PROGRAMS

Logical View

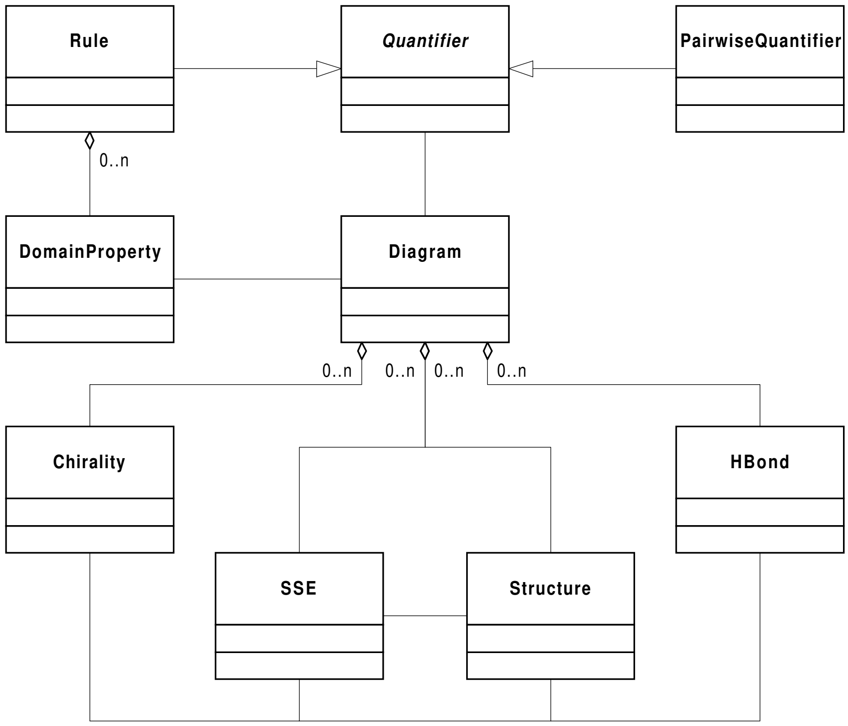

Figure A-1: Central data structure

This static structure diagram shows the main classes in the data structure. The arrows indicate aggregation (diamond shape at owner class), specialisation/generalisation (arrowhead at super-class) or general associations (use-relationships) between classes.

Logical View

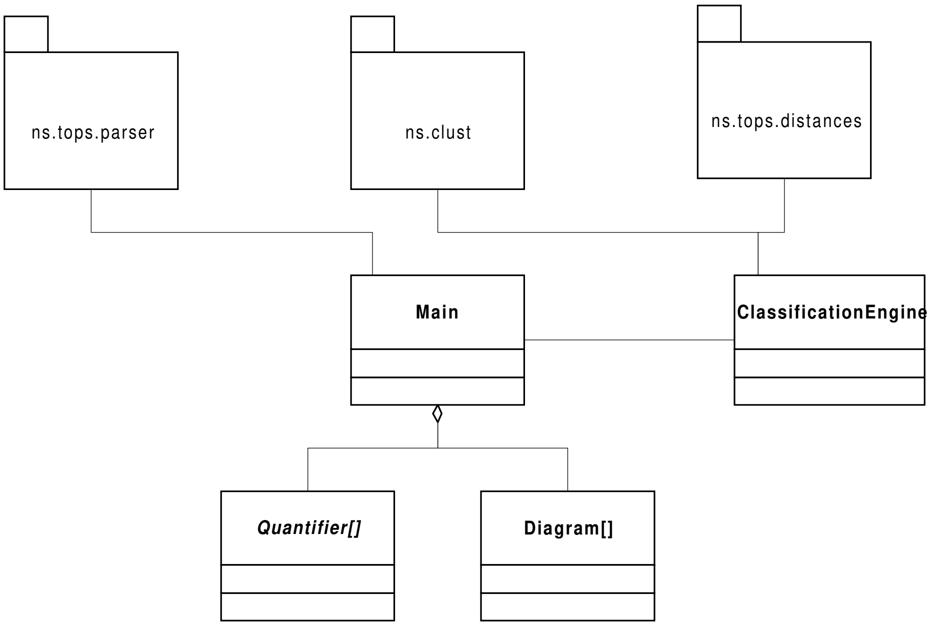

Figure A-2: Modules in the TOPS package

The TOPS package is divided into several modules, bound together by the Main *class. Of the classes shown, those representing diagrams are found in the* ns.tops.diagrams *package, the rest in* ns.tops*.*

CLASSES / FILES

ns.tops.cath.CathEntry 75
ns.tops.cath.CathMatcher 76
ns.tops.diagram.Diagram 77
ns.tops.diagram.SSE 78
ns.tops.diagram.Structure 79
ns.tops.diagram.Chirality 79
ns.tops.diagram.HBond 80
ns.tops.diagram.Point 80
ns.tops.diagram.CrossingProperty 81
ns.tops.diagram.ParallelProperty 81
ns.tops.diagram.DomainProperty 82
ns.tops.diagram.ChiralityProperty 82
ns.tops.diagram.NoOfStructuresProperty 83
ns.tops.diagram.TopsConstants 83
ns.tops.distances.AbstractDistance 84
ns.tops.distances.QuantifierDistance 84
ns.tops.distances.CathDistance 85
ns.tops.Quantifier 85
ns.tops.Rule 86
ns.tops.PairwiseQuantifier 87
ns.tops.Holder 88
ns.tops.PropLanguage 88
ns.tops.ClassificationEngine 89
ns.tops.CathClassificationEngine 89
ns.tops.Main 90
ns.tops.Stat 94
ns.tops.FileManager 95
ns.tops.DataSetGenerator 97
ns.tops.parser.Tops.jj 98
ns.clust.Isodata 101
ns.clust.HierarchicalClustering 103
ns.clust.Node 104
ns.clust. ClassificationComparator 105

ns.tops.cath.CathEntry

```
package ns.tops.cath;

import ns.tops.*;
import java.util.*;
import java.io.*;

/** Simple class representing CATH-entris. Parses or
unparses
  * them according to the normal dot-notation.
  * Also contain methods for reading the CATH-file and
  * creating a HashTable of PDB-entries and CATH-
classifications.
  */

public class CathEntry
   implements Serializable
{
   // Class variables
   static final int CATH_LEVELS = 5;

   // Protected object variables
   protected final int [] myFields = new int[ CATH_LEVELS
+ 1 ];

   // Main accessor-method:

   /** Get the number of the class at a given level.
     * @param i Level. Legal values are 1 through 5.
     * @return -1 if illegal index, category index
otherwise
     */
   public int get( int i )
   {
      return ( ( i <= CATH_LEVELS && i >= 1 ) ? myFields[
i ] : -1 );
   }

   /** Full string representation, eg "1.10.40.40.580".
*/
   public String toString() { return toString( 5 ); }

   /** String representation with a given number of
levels.*/
   public String toString( int levels )
   {
      if( levels < 1 || levels > CATH_LEVELS )
         throw new IllegalArgumentException(
            this.getClass().toString() + ": " + levels );

      StringBuffer sb = new StringBuffer( "" +
myFields[1] );
      for( int i = 2; i <= levels; i++ )
         sb.append( "." ).append( myFields[ i ] );
      return sb.toString();
   }

   /** Returns true iff the objects identify the same
family.
     * @param o A CathEntry to compare too.
     * @return true iff o has the same class,
architecture,
     *   topology, superfamily and family.
     * @throw ClassCastException iff o not is a
CathEntry.
     */
   public boolean equals( Object o )
   {
      if( o.getClass() != CathEntry.class )
         throw( new ClassCastException(
            getClass() + "->" + o.getClass() )
         );
      CathEntry ce = (CathEntry)o;
      boolean eq = true;
      for( int i = 1; i < CATH_LEVELS; i++ )
         if( this.get( i ) != ce.get( i ) )
            return false;
      return true;
   }


   // Constructors: Hide no-options-constructor

   protected CathEntry() {}


   /** Given a string, eg "1.10.40.40.580", returns a
     * CATHEntry object representing that class.
     * @return null if invalid string, a valid CATHEntry
otherwise.
     */
   public static CathEntry parse( String rep ) {
      CathEntry c;
      try{
         c = new CathEntry();
         // Get cath-numbers:
         StringTokenizer st = new StringTokenizer( rep,
"." );
         for( int i = 1; i <= CATH_LEVELS; i++ )
            c.myFields[ i ] = Integer.parseInt(
st.nextToken() );
      }
      catch( Exception e ) {
         e.printStackTrace();
         return null;
      }
      return c;
   }


   /** Reads cath-entries from a named file. */
   public static Map readCATH( String fname )
   {
      try{
         return readCATH( new BufferedReader(
            new FileReader( fname )
         ) );
      }
      finally{ return null; }
   }

   /** Reads cath-entries from a text stream. Assumes the
     * file is in the same format as cath_all.pl.
     */
   public static Map readCATH( Reader r )
   {
      try
      {
         HashMap cath = new HashMap();
         StringBuffer sb = new StringBuffer();
         BufferedReader fr = new LineNumberReader( r );
         fr.readLine();

         while( fr.ready() )
         {
            String s = fr.readLine();

            // Check if it is an entry, if not just skip:
            if( s.indexOf( '(' ) != 0 ) continue;

            String p = s.substring( 2, 8 );
            String t = s.substring( 11);
            String c = t.substring( 0, t.indexOf( '\'' )
);

            cath.put( p, parse( c ) );
         }
         r.close();
         return cath;
      }
      catch( Exception e ){ e.printStackTrace(); return
null; }
   }
}
```

ns.tops.cath.CathMatcher

```
package ns.tops.cath;

import java.util.*;
import java.io.*;
import ns.Util;

/** Simple class to select diagrams
  * according to their CATH-classification.
  */

public class CathMatcher
   implements Serializable
{

   // Object variables

   protected int [][] values = { null, null, null, null,
null };

   /** matches( null ) always returns
MATCH_UNCLASSIFIEDS.
     * Default is false.
     */
   public boolean MATCH_UNCLASSIFIEDS = false;

   /** If true, a null value in the values array
     * matches everything. Default is true.
     */
   public boolean NULL_MATCHES_ALL = true;

   /** Set values. */
   public void setValues( int [][] v )
   {
      if( v != null && v.length == values.length )
         values = v;
      else
         throw( new RuntimeException(
            "CathMatcher.setValues: Invalid v. "
         ) );
      return;
   }

   /** Get values. */
   public int [][] getValues() { return values; }


   /** Returns a readable version of this CathMatcher. */
   public String toString()
   {
      StringBuffer sb = new StringBuffer();
      for( int i = 0; i < values.length; i++ ) {
         if( values[ i ] == null || values[ i ].length ==
0  )
            sb.append( "*" );
         else
         {
            sb.append( "" + values[ i ][ 0 ] );
            for( int j = 1; j < values[ i ].length; j++ )
               sb.append( "/" + values[ i ][ j ] );
         }
         if( i < values.length-1 )
            sb.append( "." );
      }
      return sb.toString();
   }


   /** True iff this matcher matches ce. */
   public boolean matches( final CathEntry ce )
   {
      if( ce == null )
         return MATCH_UNCLASSIFIEDS;

      Util.debug( "Cathmatcher.matches: " + ce, 15 );

      int [] target = ce.myFields;

      // default for boolean is false, I hope:
      boolean [] b = new boolean[ 5 ];

      for( int i = 0; i < values.length; i++ )
      {
         int [] tmp = values[ i ];

         if( tmp == null )
            b[ i ] = NULL_MATCHES_ALL;
         else
         {
            innerfor: for( int j = 0; j < tmp.length; j++
)
               if( tmp[ j ] == target[ i + 1 ] )
               {
                  b[ i ] = true;
                  break innerfor;
               }
         }
      }

      return ( b[0] && b[1] && b[2] && b[3] && b[4] );
   }

   /** This matcher will match all cathentries.*/
   public CathMatcher() {}

   /** This matcher will match all cathentries in class
i. */
   public CathMatcher( int i )
   {
      int [] tmp = { i };
      values[ 0 ] = tmp;
   }

   /** This matcher will match all cathentries in
     * class i and architecture j.
     */
   public CathMatcher( int i, int j )
   {
      int [][] tmp = { { i }, { j }, null, null, null };
      setValues( tmp );
   }


   /** Parses strings to CathMatchers.
     * Syntax: "class{/class}*.arch{/arch}* etc".
     * @return A valid CathMatcher or null.
     */
   public static CathMatcher parse( String s )
   {
      try{
         CathMatcher cm = new CathMatcher();
         StringTokenizer st = new StringTokenizer( s, "."
);

         for( int j = 0; j <= 4; j++ ) {
            String s2 = st.nextToken();
            if( s2.equals( "*" ) )
               cm.values[ j ] = null;
            else {
               StringTokenizer tmp = new StringTokenizer(
s2, "/" );

               int [] ss = new int[ tmp.countTokens() ];
               for( int i = 0; i < ss.length; i++ )
                  ss[ i ] = Integer.parseInt(
tmp.nextToken() );
               cm.values[ j ] = ( ss.length == 0 ? null :
ss );
            }
         }
         return cm;
      }
      catch( Exception e ) { e.printStackTrace(); return
null; }
   }

}
```

ns.tops.diagram.Diagram

```
package ns.tops.diagram;

import ns.Util;
import ns.tops.*;
import ns.tops.cath.*;

import ns.tops.parser.TopsParser;

import java.util.ArrayList;
import java.util.Map;
import java.util.Collection;
import java.util.StringTokenizer;

import java.io.*;

/** A representation of a TOPS diagram.
  * Contains the structures and number of strands and
helixes.
  * A tops diagram is a triple ( SSEs, H-Bonds,
Chiralities ) where
  * <ul>
  * <li>SSE [] getSSEs </li>
  * <li>HBond [] getHBonds</li>
  * <li>Chirality [] getChiralities</li>
  * </ul>
  *
  * Those define a set of structures, Structure []
getStructures()
  */

public class Diagram
    implements java.io.Serializable, TopsConstants,
Comparable
{

   // Class constants

   public static final int ID_ALL = -1;
   public static final int ID_DOMAIN = 0;


   // Class variables
   public static int stringID = ID_DOMAIN;

   // Object variables - main data structure

   protected CathEntry myCE = null;
   protected CathEntry cathEntry;
   protected String myDomainName = "xxxxxx";

   protected SSE [] mySSEs;
   protected Structure [] myStructures;
   protected HBond [] myHBonds;
   protected Chirality [] myChiralities;

   protected int [] noOfStructs;


   // Main accessor methods:

   public SSE [] getSSEs() { return mySSEs; }
   public void   setSSEs( SSE [] sses ) { mySSEs = sses;
return; }

   public HBond [] getHBonds() { return myHBonds; }
   public void     setHBonds( HBond [] HBonds ) {
      myHBonds = HBonds; return;
   }

   public Chirality [] getChiralities() { return
myChiralities; }
   public void         setChiralities( Chirality []
Chiralities ) {
      myChiralities = Chiralities; return;
   }

   public Structure [] getStructures() { return
myStructures; }
   public void         setStructures( Structure [] s ) {
      myStructures = s; return;
   }

   public String getDomainName() { return myDomainName; }
   public void   setDomainName( String n ) {
      myDomainName = n; return;
   }

   public String toString()
   {
      switch( stringID ) {
         case ID_ALL: return getDomainName() + "/" +
            getCathEntry().toString( 5 );
         case ID_DOMAIN: return getDomainName();
         default: return getCathEntry().toString(
stringID );
      }
   }

   // Some additional accessor methods:

   public CathEntry getCathEntry() { return myCE; }
   public void setCathEntry( CathEntry ce ) { myCE = ce;
}
   public int [] getNoOfStructs() { return noOfStructs; }
   public int getNoOfStrands() { return noOfStructs[
STRAND ]; }
   public int getNoOfHelixes() { return noOfStructs[
HELIX ]; }

   /** Complete string representation. */
   public String unparse()
   {
      StringBuffer sb = new StringBuffer();
      sb.append( "Diagram: " ).append( EOL );
      sb.append( "   name=" + myDomainName )
            .append( EOL ).append( EOL );

      sb.append( "   SSEs: [ " );
      for( int i = 0; i < mySSEs.length; i++ )
         sb.append( mySSEs[ i ] + " " );
      sb.append( "]" ).append( EOL ).append( EOL );

      sb.append( "   Structs: [ " );
      for( int i = 0; i < myStructures.length; i++ )
         sb.append( myStructures[ i ] + " " );
      sb.append( "]" ).append( EOL ).append( EOL );

      sb.append( "   HBonds: [ " );
      for( int i = 0; i < myHBonds.length; i++ )
         sb.append( myHBonds[ i ] + " " );
      sb.append( "]" ).append( EOL ).append( EOL );

      sb.append( "   Chirs: [ " );
      for( int i = 0; i < myChiralities.length; i++ )
         sb.append( myChiralities[ i ] + " " );
      sb.append( "]" ).append( EOL ).append( EOL );

      return sb.toString();
   }

   // Constructors

   protected Diagram() { return; }

   public Diagram( String dname, SSE [] sselist,
         Structure [] structs, HBond [] hblist, Chirality
[] chirs )
   {
      setDomainName( dname );
      mySSEs = sselist; myStructures = structs;
      myHBonds = hblist; myChiralities = chirs;

      // assign noOfStructs:
      noOfStructs = new int[ TopsConstants.UNKNOWN+1 ];
      for( int i = 0; i < myStructures.length; i++ )
         noOfStructs[ myStructures[ i ].getType() ]++;

      for( int i = 0; i < mySSEs.length; i++ )
         noOfStructs[ mySSEs[ i ].getType() ]++;
      return;
   }

   /** Given a string from the TOPS database,
     * returns a Diagram-object representing that
diagram.
     * @param cath A HashMap of CATH-entries.
     * @return null if invalid string, a valid Diagram
otherwise.
     */
   public static Diagram parse( String s, Map cath ) {
      return TopsParser.parseDiagram( s, cath );
   }


   /** Reads diagrams from a datafile.
     * Same as readDiagrams( f, null, cath );
     */
   public static Diagram [] readDiagrams( Reader r, Map
cath ) {
      return readDiagrams( r, null, cath );
   }


   /** Reads diagrams from a datafile.
     * Normal usage: Diagrams [] d =
     * readDiagrams( new FileReader( "cartoons.pl" ) );
     * This method uses buffering, ie do not provide a
     * buffered reader as that only will decrease
performance.
     *
     * @param r A character stream containing the
diagrams.
     * @param cath A map from pdp-names to cath-entries
     * @param cm A cathmatcher that the diagram must
match
     * @return An array of diagrams.
     */
   public static Diagram [] readDiagrams( Reader r,
         CathMatcher cm, Map cath )
   {
      Util.tock();
```

```java
      Util.verbose0( "Reading diagrams" );
      Collection c;
      try{ c = TopsParser.parseDiagrams( r, cath ); }
      catch( Exception e ) { e.printStackTrace(); return
null; }
      Util.verbose( " done (" + Util.tick()/1000 + "
secs)" );
      return selectDiagrams(
         (Diagram[]) c.toArray( new Diagram[ 0 ] ), cm
      );
   }

   /** Selects diagrams according to CATH-category.
     * @param A matcher that the cath of the diagrams
     *   must match, eg new CathMatcher( 3, 40 ).
     */

   public static Diagram [] selectDiagrams(
         Diagram [] diags, CathMatcher cm ) {
      Diagram [] res = null;
      try
      {

         Diagram [] tmp = new Diagram[ diags.length ];
         int index = 0;

         for( int i = 0; i < diags.length; i++ )
         {
            boolean matches = ( cm == null || cm.matches(
                  diags[ i ].getCathEntry() ) );
            if( matches )
               tmp[ index++ ] = diags[ i ];
         }

         res = new Diagram[ index ];

         for( int i = 0; i < index; i++ )
            res[ i ] = tmp[ i ];

      }
      catch( Exception e ) { e.printStackTrace(); }
      return res;
   }


   /** Compares domain id's. */
   public int compareTo( Object o ) {
     Diagram d = (Diagram)o;
     if( this.equals( o ) )
       return 0;
     else
       return myDomainName.compareTo( d.myDomainName );
   }

}
```

ns.tops.diagram.SSE

```java
package ns.tops.diagram;

import java.util.*;
import java.io.*;
import ns.Util;

/** Secondary structure element.*/

public class SSE implements java.io.Serializable
{
   protected int no = -1;
   protected int dir = 0;
   protected String name ="";
   public int myType = TopsConstants.UNKNOWN;

   public final int getDir() { return dir; }
   public void setDir( int d ) {
      if( d >= -1 || d <= 1 ) dir = d; return;
   }

   public final int getNo() { return no; }
   public void setNo( int newNo ) { no = newNo; return; }

   public final String getName() { return name; }
   public void setName( String s ) { name = s; return; }

   public int  getType() { return myType; }
   public void setType( final int type )
   {
      if( type >= TopsConstants.STRAND &&
            type <= TopsConstants.CTERM )
         myType = type;
      else
   throw( new RuntimeException(
this.getClass().toString() +
           ".setType(" + type + "): invalid type." ) );
      return;
   }


   public String toString() {
      return getName() + ":" + getType() + "(" + getDir()
+ ")";
   }

   protected SSE() { return; }

   public SSE( String n ) { setName( n ); return; }
   public SSE( int type, int no, int dir ) {
      setType( type );
      setNo( no );
      setDir( dir );
      return;
   }
}
```

ns.tops.diagram.Structure

```
package ns.tops.diagram;

/** Class for the different third-level structures in
diagrams.
  * Each instance has a integer type, as defined in
  * TOPSConstants, and an array
  * of SSE elements.
  */

public class Structure
    implements java.io.Serializable
{
  protected SSE [] mySSEs = null;
  protected int myType = TopsConstants.UNKNOWN;

  public int getType() { return myType; }
  public void setType( int s ) {
    if( s >= TopsConstants.STRUCT_START &&
         s <= TopsConstants.UNKNOWN )
      myType = s;
    return;
  }

  public SSE [] getSSEs() { return mySSEs; }
  public void setSSEs( SSE [] sses ) { mySSEs = sses; }

  // Constructors:

  protected Structure() { return; }

  public Structure( int struct_type, SSE [] sses ) {
    mySSEs = sses;
    setType( struct_type );
  }

   public String toString()
   {
      StringBuffer sb = new StringBuffer();
      sb.append( TopsConstants.struct_ids[ myType ] + "([
" );
      for( int i = 0; i < mySSEs.length; i++ )
         sb.append( mySSEs[ i ].getName() + " " );
      sb.append( "])" );
      return sb.toString();
   }

   /** If result[ i ] == 1/2 then strand number
     * i belongs to the 1st/2nd sheet.
     */
   int [] getSheetArray( Diagram owner )
   {
      if( myType != TopsConstants.SANDWICH )
         return null;

      int [] t = new int[ owner.getSSEs().length ];
      HBond [] hb = owner.getHBonds();

      // Initialize first sheet:
      t[ mySSEs[ 0 ].getNo() ] = 1;


      // First get members of the first sheet:
      boolean changed = true;
      while( changed )
      {
         changed = false;
         for( int i = 0; i < hb.length; i++ )
         {

            int from = hb[ i ].from.getNo();
            int to   = hb[ i ].to.getNo();

            if( t[ to ] == 1 && t[ from ] != 1 )
            {
               t[ from ] = 1;
               changed = true;
            }
            else if( t[ from ] == 1 && t[ to ] != 1 )
            {
               t[ to ] = 1;
               changed = true;
            }
         }
      }

      // Then get the rest:
      for( int i = 0; i < mySSEs.length; i++ )
         t[ mySSEs[ i ].getNo() ] =
            ( t[ mySSEs[ i ].getNo() ] == 1 ? 1 : 2 );

      return t;
   }

}
```

ns.tops.diagram.Chirality

```
package ns.tops.diagram;

import ns.tops.*;
import ns.tops.cath.*;

import java.util.*;
import java.io.*;
import ns.Util;

/** Class representing chiralities.
  */

public class Chirality implements java.io.Serializable
{
   SSE from, to;
   public int handedness;
   public static final int LEFT = -1, RIGHT = 1;

   public Chirality( SSE f, int d, SSE t )
   {
      from = f; to = t; handedness = d;
   }

   public String toString() {
      return "(" + from.getName() + "," +
         handedness + "," + to.getName() + ")";
   }
}
```

ns.tops.diagram.HBond

```
package ns.tops.diagram;

import java.util.*;
import java.io.*;
import ns.Util;

/** Secondary structure binary directed relation. */

public class HBond implements java.io.Serializable
{
   public SSE from, to;
   public int direction;
   public static final int DOWN = -1, UP = 1;

   public HBond( SSE f, int d, SSE t )
   {
      from = f; to = t; direction = d;
   }

   public String toString() {
      return "(" + from.getName() + "," + direction + ","
+
            to.getName() + ")";
   }

}
```

ns.tops.diagram.Point

```
package ns.tops.diagram;

import java.util.*;
import java.io.*;
import ns.Util;

/** TOPS diagram element - parsed and created,
  * but currently ignored.
  */

public class Point implements java.io.Serializable
{
   int x, y;
   public Point( int x, int y ) {
      this.x = x; this.y = y; return;
   }
}
```

ns.tops.diagram.CrossingProperty

```
package ns.tops.diagram;

import ns.tops.diagram.*;
import ns.Util;
import java.util.Collection;
import java.util.Vector;
import java.util.Arrays;
import java.util.HashSet;

/** Count crossings in sandwiches. */

public class CrossingProperty extends DomainProperty
{

   public CrossingProperty( String id, String desc )
   {
      setID( id );
      setDescription( desc );
      return;
   }

   /** Returns the average times the backbone crosses
from
     * the first sheet to the second sheet in the
sandwiches.
     */
   public float valueOf( final Diagram d ) {

      int sum = 0;
      int no = 0;

      for( int strindex = 0;
            strindex < d.getStructures().length;
            strindex++ ) {

         if( d.getStructures()[ strindex ].getType() !=
               TopsConstants.SANDWICH )
            continue;

         no++;

         Structure sw = d.getStructures()[ strindex ];
         int [] sheets = sw.getSheetArray( d );

         // move along the backbone and
         // count the times it crosses from one sheet to
another

         if( false )
         {
            StringBuffer deb = new StringBuffer(
                  "sheets: { " + sheets[ 0 ] );
            for( int i = 1; i < sheets.length; i++ )
               deb.append( ", " + sheets[ i ] );
            deb.append( " }" );
            Util.debug( deb, 12 );
         }

         SSE [] sa = d.getSSEs();

         int cur = 0;
         for( int i = 1; i < sa.length-1; i++ )
         {
            int tmp = sheets[ sa[ i ].getNo() ];
            if(
               ( cur == 1 && tmp == 2 ) || ( cur == 2 &&
tmp == 1 )
            )
               sum++;
            cur = tmp;
         }
      }

      if( no == 0 )
         return 0;
      else
         return ( (float)sum ) / (float)no;
   }
}
```

ns.tops.diagram.ParallelProperty

```
package ns.tops.diagram;

import ns.tops.diagram.*;
import java.util.Collection;
import java.util.Vector;
import java.util.Arrays;

/** Measures degree of parallelity. */

public class ParallelProperty extends DomainProperty
{

   public ParallelProperty( String id, String desc )
   {
      setID( id );
      setDescription( desc );
      return;
   }

   /** Returns the ratio of parallel hbonds in the
sheets...
     * Does not include the sheets found in sandwiches.
     * @return float in the range [0..1], 0 meaning
     *   only parallel hbonds.
     */
   public float valueOf( Diagram d )
   {
      Structure [] structs = d.getStructures();

      int sum = 0;
      int no = 0;

      for( int strindex = 0; strindex < structs.length;
strindex++ )
      {
         if( structs[ strindex ].getType() !=
TopsConstants.SHEET )
            continue;
         no++;

         int up = 0;
         int tot = 0;

         Structure s = structs[ strindex ];
         Collection sses = Arrays.asList( s.getSSEs() );

         HBond [] hbtmp = d.getHBonds();

         for( int i = 0; i < hbtmp.length-1; i++ )
         {
            if( sses.contains( hbtmp[ i ].from ) &&
                  sses.contains( hbtmp[ i ].to ) )
            {
               if( hbtmp[ i ].direction == HBond.UP )
                  up++;
               tot++;
            }
         }
      if( tot != 0 )
        sum += (int) ( ( ( float )up ) / (float) tot );

      }

      if( no == 0 )
     return 0;
      else
     return ( (float)sum ) / (float)no;
   }
}
```

ns.tops.diagram.DomainProperty

```
package ns.tops.diagram;

import ns.tops.diagram.*;
import java.util.*;

/** A DomainProperty acts on a Domain and assigns it a
value.
  * This is an abstract base class. All properties are
  * identified with unique string id's.
  */

public abstract class DomainProperty
    implements java.io.Serializable
{

   protected DomainProperty() { return; }

   /** Unique ID.*/
   protected String myID = "undefined_property";

   /** Set the id. */
   public void setID( String id ) { myID = id; }

   /** Get the id.
    * @deprecated
    */
   public String getID() { return myID; }

   /** Each property has a description. */
   protected String myDesc = null;

   /** Get the user description of this property. */
   public String getDescription() { return myDesc; }

   /** Set the user description of this property. */
   public void setDescription( String s ) { myDesc = s;
return; }

   /** This implementation returns the DomainProperty's
     * unique ID. Subclasses may or may not override
this.
     */
   public String toString() { return getID(); }

   /** Reads the known properties and stores them in an
HashMap. */
   public static Map readProps()
   {
      Map m = new HashMap();
      String [] sa = { "sheets", "barrels", "csheets",
"vcsheets",
            "sandwiches", "unknowns", "strands",
"helixes" };

      m.put( "sheets", new NoOfStructuresProperty(
"sheets",
            TopsConstants.SHEET    , "beta sheets" ) );
      m.put( "barrels",new NoOfStructuresProperty(
"barrels",
            TopsConstants.BARREL   , "barrels" ) );
      m.put( "csheets",    new NoOfStructuresProperty(
"csheets",
            TopsConstants.CSHEET   , "curved sheets" ) );
      m.put( "vcsheets",   new NoOfStructuresProperty(
"vcsheets",
            TopsConstants.VCSHEET  , "vcurved sheets" )
);
      m.put( "sandwiches", new NoOfStructuresProperty(
"sandwiches",
            TopsConstants.SANDWICH , "sandwiches" ) );
      m.put( "unknowns",   new NoOfStructuresProperty(
"unknowns",
            TopsConstants.UNKNOWN  , "unknown structures"
) );
      m.put( "strands",    new NoOfStructuresProperty(
"strands",
            TopsConstants.STRAND   , "beta strands" ) );
      m.put( "helixes",    new NoOfStructuresProperty(
"helixes",
            TopsConstants.HELIX    , "alpha helixes" ) );
      m.put( "crossings",  new CrossingProperty(
            "crossings", "Average no of crossings in
sandwiches" ) );
      m.put( "parallel",   new ParallelProperty(
            "parallel", "Average ratio of p. hbonds in
sheets" ) );
      m.put( "chiralities",new ChiralityProperty(
            "chiralities", "Number of r-chirs minus l-
chirs" ) );

      return m;
   }

   /** The method!! */
   public abstract float valueOf( Diagram d );
}
```

ns.tops.diagram.ChiralityProperty

```
package ns.tops.diagram;

import ns.tops.diagram.*;
/** Counts chiralities. */
public class ChiralityProperty extends DomainProperty
{
   public ChiralityProperty( String id, String desc )
   {
      setID( id );
      setDescription( desc );
      return;
   }

   public float valueOf( Diagram d )
   {
      Chirality [] chirs = d.getChiralities();
      int sum = 0;
      for( int i = 0; i < chirs.length; i++ )
         if( chirs[ i ].handedness == Chirality.RIGHT )
            sum++;
         else if( chirs[ i ].handedness == Chirality.LEFT
)
            sum--;
      return sum;
   }
}
```

ns.tops.diagram.NoOfStructuresProperty

```
package ns.tops.diagram;

import ns.tops.*;

/** Simple domain-property used to count the 5 fixed
structures.
  * Each instance has an <it>index</it>, as specified in
  * TopsConstants.
  */

public class NoOfStructuresProperty extends
DomainProperty
{
   protected int myIndex = TopsConstants.STRAND;

   public void setIndex( int ind ) { myIndex = ind;
return; }
   public int getIndex() { return myIndex; }

   public float valueOf( Diagram d )
   {
      return d.getNoOfStructs()[ myIndex ];
   }


   public NoOfStructuresProperty( String id, int
structIndex )
   {
      myID = id;
      myIndex = structIndex;
   }

   public NoOfStructuresProperty( String id, int si,
String desc )
   {
      myID = id;
      myIndex = si;
      myDesc = desc;
   }

}
```

ns.tops.diagram.TopsConstants

```
package ns.tops.diagram;

/** Defines some useful constants. */


public interface TopsConstants{
   public static final String EOL   =
      System.getProperty( "line.separator" );

   public static final int STRAND   = 0;
   public static final int HELIX    = 1;
   public static final int NTERM    = 2;
   public static final int CTERM    = 3;

   public static final int SHEET    = 4;
   public static final int BARREL   = 5;
   public static final int CSHEET   = 6;
   public static final int VCSHEET  = 7;
   public static final int SANDWICH = 8;
   public static final int UNKNOWN  = 9;

   public static final int STRUCT_START = 4;
   public static String [] struct_ids = {
        "e", "h", "n", "c", "s", "b", "c", "v", "sw",
"u"
   };
}
```

ns.tops.distances.AbstractDistance

```
package ns.tops.distances;

import ns.Util;
import ns.tops.diagram.*;

import ns.tops.*;
import ns.tops.cath.*;


/** An abstract superclass for classes representing
methods
  * for finding a measure of the distance between two
diagrams.
  */

public abstract class AbstractDistance
    implements java.io.Serializable, ns.clust.Distance
{

   /** The method!! */
   public abstract float getDistance( Diagram d1, Diagram
d2 );

   /** String representation. */
   public String toString() { return
this.getClass().toString(); }

   /** Convenience method.
     * Subclasses can override this for effiency, if
needed.
     */
   public float [] getDistances(
         final Diagram d, final Diagram [] diagrams ) {
      float [] da = new float[ diagrams.length ];
      for( int i = 0; i < diagrams.length; i++ )
      {
         da[ i ] = getDistance( d, diagrams[ i ] );
      }
      return da;
   }

   /** Convenience method.
     * Subclasses can override this for effiency, if
needed.
     */
   public float [][] getDistances( final Diagram []
diagrams1,
         final Diagram [] diagrams2 ) {
      float [][] da = new float[
            diagrams1.length ][ diagrams2.length ];
      for( int i = 0; i < diagrams1.length; i++ )
         da[ i ] = getDistances( diagrams1[ i ],
diagrams2 );
      return da;
   }

   /** Convenience method.
     * Subclasses can override this for effiency, if
needed.
     * @return A distance matrix.
     */
   public float [][] getDistanceMatrix(
         final Diagram [] diagrams ) {
      return getDistances( diagrams, diagrams );
   }

   /** Calls getDistance( (Diagram)d1, (Diagram)d2 ). */
   public float distance( Object o1, Object o2 ) {
      return getDistance( (Diagram)o1, (Diagram)o2 );
   }


   /** Static helper method. */
   public static float [] transform( double [] m ) {
      float [] t = new float[ m.length ];
      for( int i = 0; i < m.length; i++ )
        t[ i ] = (float) m[ i ];
      return t;
   }

   /** Static helper method. */
   public static double [] transform( float [] m ) {
      double [] t = new double[ m.length ];
      for( int i = 0; i < m.length; i++ )
     t[ i ] = (double) ( m[ i ] );
      return t;
   }

   /** Static helper method. */
   public static void abs( double [] da ) {
      for( int i = 0; i < da.length; i++ )
         da[ i ] = Math.abs( da[ i ] );
      return;
   }

}
```

ns.tops.distances.QuantifierDistance

```
package ns.tops.distances;


import ns.Util;
import ns.tops.Quantifier;
import ns.tops.diagram.Diagram;

import java.io.Serializable;

/** Uses Euclidean distance between quantifier-vectors.
  */

public class QuantifierDistance extends AbstractDistance
{

   protected Quantifier q;
   protected static float a = 0.5f;

   /** Default constructor. */
   public QuantifierDistance( Quantifier r ) {
      if( r == null )
         throw( new RuntimeException(
            "QuantifierDistance: q==null" )
         );
      q = r;
   }

   /** Returns the result of a sigmoid function applied
to
     * the euclidean distance between
     * quantifier-vectors for the diagrams.
     * @return Euclidean distance.
     */
   public float getDistance( Diagram d1, Diagram d2 ) {
      float [] v1 = q.valueOf( d1 );
      float [] v2 = q.valueOf( d2 );

      float sum = 0;
      for( int i = 0; i < v1.length; i++ )
           sum += Math.pow( v1[ i ] - v2[ i ], 2 );

      return sigmoid( Math.sqrt( sum ) );
   }

   public static float sigmoid( double d ) {
      return (float) ( 1.0f / ( 1 + Math.exp( -a* d ) )
);
   }

   /** String representation. */
   public String toString() {
return this.getClass() + ": q=" +
q; }

}
```

ns.tops.distances.CathDistance

```
package ns.tops.distances;

import ns.Util;
import ns.tops.diagram.*;

import ns.tops.*;
import ns.tops.cath.*;


/** An abstract superclass for classes representing
methods
  * for finding a measure of the distance between two
diagrams.
  */

public class CathDistance
   extends AbstractDistance
{

   protected int myLevel = 1;

   /** Default constructor with level as parameter. */
   public CathDistance( int level ) {
      myLevel = level;
   }

   /** The method!! */
   public float getDistance( Diagram d1, Diagram d2 ) {
      if( d1.getCathEntry().get( myLevel ) ==
            d2.getCathEntry().get( myLevel ) )
         return 0;
      else
         return 1;
   }

   /** String representation. */
   public String toString() { return
this.getClass().toString(); }


}
```

ns.tops.Quantifier

```
package ns.tops;

import ns.tops.diagram.Diagram;

import java.util.Locale;
import java.text.DecimalFormat;
import java.text.NumberFormat;
import java.text.FieldPosition;
import ns.Util;

/** A representation of a rule that transforms a diagram
into a
  * vector. Abstract base class with implementation of
most methods.
  */


public abstract class Quantifier implements
java.io.Serializable
{

   protected static byte numberSize = 2;
   protected static byte decimals = 3;

   protected static NumberFormat dnf =
      NumberFormat.getNumberInstance( Locale.US );
   static
   {
      dnf.setMaximumFractionDigits( decimals );
      dnf.setMinimumFractionDigits( decimals );
      dnf.setMaximumIntegerDigits( numberSize );
      dnf.setGroupingUsed( false );
   }

   protected static String formatFloat( float d )
   {

      StringBuffer sb = new StringBuffer();

      FieldPosition fp = new FieldPosition(
dnf.INTEGER_FIELD );
      dnf.format( d, sb, fp );

      int missing = numberSize - fp.getEndIndex();

      for( int i = 0; i < missing; i++ )
         sb.insert( 0, ' ' );

      return sb.toString();
   }


   protected float [] myWeights;

   /** Returns the weights vector. */
   public float [] getWeights() { return myWeights; }

   /** Sets the weights for this quantifier. */
   public void setWeights( float [] newWeights )
   {
      if( newWeights != null &&
         myWeights.length == newWeights.length )
         myWeights = newWeights;
   }

   /** Returns the length of vectors created by this
quantifier. */
   public int getSize() {
      return ( myWeights == null ? -1 : myWeights.length
);
   }

  /** Same as toString() */
  public String unparse()  { return toString(); }

   /** Calls stringOf( diagram, true ). */
   public String stringOf( Diagram diagram ) {
      return stringOf( diagram, true );
   }


   /** Returns a string version of the vector values of
the diagram,
     * optionally including the name and class of the
structure.
     * @return A string, eg "0.0 2.5 1.666
pdpbladi/cathbladi
     */
   public String stringOf( Diagram diagram, boolean
header )
   {
      StringBuffer res = new StringBuffer();
      float [] values = valueOf( diagram );
      for( int i = 0; i < values.length; i++ )
         res.append( formatFloat( values[ i ] ) + " " );

      if( header )
         res.append( diagram.toString() );
      return res.toString();
```

```
   }

  // Plural methods - just for convenience


   /** Calls valueOf( d[ i ] for all i's. */
   public float [][] valueOf( Diagram [] d )
   {
      float [][] res =
         new float[ d.length ][ valueOf( d[ 0 ] ).length
];
      for( int i = 0; i < d.length; i++ )
         res[ i ] = valueOf( d[ i ] );
      return res;
   }

   /** Calls stringOf( d[ i ] ) for all i's. */
   public String [] stringOf( Diagram [] d )
   {
      String [] res = new String[ d.length ];
      for( int i = 0; i < d.length; i++ )
         res[ i ] = stringOf( d[ i ] );
      return res;
   }

  // Object ([]) methods - just for convenience

   /** Calls valueOf( (Diagram) o ). */
   public float [] valueOf( Object o ) {
      return valueOf( ( Diagram ) o );
   }

   /** Calls valueOf( (Diagram[]) o ). */
   public float [][] valueOf( Object [] o ) {
      return valueOf( ( Diagram[] ) o );
   }

   /** Calls stringOf( (Diagram) o ). */
   public String stringOf( Object o ) {
      return stringOf( ( Diagram ) o );
   }

   /** Calls stringOf( (Diagram[]) o ). */
   public String [] stringOf( Object [] o ) {
      return stringOf( ( Diagram[] ) o );
   }

   /** Gives a string describing fields - used in eg
statistics. */
   public String [] getHeaders() { return new String[
getSize() ]; }

  // The abstract methods!!

    /** The method: Calculates the actual values, given a
diagram. */
   public abstract float [] valueOf( Diagram d );



}
```

ns.tops.Rule

```
package ns.tops;

import ns.Util;
import ns.tops.diagram.*;
import ns.tops.parser.TopsParser;

import java.io.*;
import java.util.*;
import java.text.*;

/** A representation of a rule that transforms a
  * diagram into a vector. A Rule is an aggregation
  * of DomainProperties.
  */

public class Rule extends Quantifier implements
java.io.Serializable
{

   // Object variables

   protected DomainProperty [] props = null;
   protected String name = new String();


   // Main accessor

   public DomainProperty [] getProps() { return props; }


   // Constructor

   public Rule( DomainProperty [] p, Float [] ws, String
name )
   {
      props = p;
      myWeights = new float[ ws.length ];
      for( int i = 0; i < ws.length; i++ )
         myWeights[ i ] = ws[ i ].floatValue();
      this.name = name;

      return;
   }

   /** Calculates the actual values, given a diagram. */
   public float [] valueOf( Diagram d )
   {
      float [] valueOf = new float[ myWeights.length ];
      for( int i = 0; i < props.length; i++ )
         valueOf[i] = myWeights[i] * props[i].valueOf( d
);
      return valueOf;
   }

   /** Given a string representing a rule, returns a
Rule-object
     * representing that rule.
     * @return null if invalid string, a valid Rule
otherwise.
     */
   public static Rule parse( final String s, final Map
props )
   {
      return TopsParser.parseRule( s, props );
   }

   /** Reads a set of rules from a text (ASCII) file.
     * @param fname Name of the file to read the rules
from
     * @return An array of the rules in the file
     */
   public static Rule [] readRules(
      final String fname, final Map props )
   {
      try
      {
         Collection c = TopsParser.parseRules(
            new FileReader( fname ), props );
         return ( Rule[])c.toArray( new Rule[ 0 ] );
      }
      finally { return null; }
   }

   /** Returns an array of the names of the
domainproperties.*/
   public String [] getHeaders()
   {
      String [] getHeaders = new String[ getSize() ];
      for( int i = 0; i < getHeaders.length; i++ )
         getHeaders[ i ] = props[ i ] +
            ( myWeights[ i ] != 1.0 ? "*" +
            formatFloat( myWeights[ i ] ) : "" );
      return getHeaders;
   }

   /** String representation. */
   public String toString()
   {
```

```
      return "Rule " + name + " (size: " + getSize() +
")";
   }

   /** Full string representation, including weights. */
   public String unparse() {

      String [] headers = getHeaders();
      StringBuffer sb = new StringBuffer( "[ " ).
         append( headers[ 0 ] );
      for( int i = 1; i < headers.length; i++ )
         sb.append( ", " + headers[ i ] );
      sb.append( "; " + this.toString() + "]" );
      return sb.toString();
   }

}
```

ns.tops.PairwiseQuantifier

```
package ns.tops;

import ns.Util;
import ns.tops.diagram.*;
import java.text.*;

/** Creates a float vector with the frequencies of pairs
of SSEs. */

public class PairwiseQuantifier extends Quantifier
   implements TopsConstants
{
   public static boolean qualityCheck = false;

   protected static final int ALPHA_ALPHA = 0;
   protected static final int BETA_ALPHA  = 1;
   protected static final int ALPHA_BETA = 2;
   protected static final int BETA_BETA  = 3;

   protected PairwiseQuantifier() {
      float [] w = { 1.0f, 1.0f, 1.0f, 1.0f };
      myWeights = w;
      return;
   };

   public PairwiseQuantifier( float [] w ) {
      this();
      setWeights( w );
      return;
   }

   public float [] valueOf( Diagram d ) {

      SSE [] sses = d.getSSEs();
      float [] result = new float[ 4 ];
      int cur = sses[ 1 ].getType();
      for( int i = 2; i < sses.length-1; i++ )
      {

         if( cur == HELIX )
         {
            if( sses[ i ].getType() == HELIX )
               result[ ALPHA_ALPHA ]++;
            else if( sses[ i ].getType() == STRAND )
               result[ ALPHA_BETA ]++;
            else if( qualityCheck )
               Util.bug( this.getClass().toString() +
                  " .valueOf: " + d.getDomainName() + ",
1" + i );
         }
         else if( cur == STRAND )
         {
            if( sses[ i ].getType() == HELIX )
               result[ BETA_ALPHA ]++;
            else if( sses[ i ].getType() == STRAND )
               result[ BETA_BETA ]++;
            else if( qualityCheck )
               Util.bug( this.getClass().toString() +
                  " .valueOf: " + d.getDomainName() + ",
2, " + i );
         }
         else if( qualityCheck )
            Util.bug( this.getClass().toString() + "
.valueOf: " +
               d.getDomainName() + ", 3, " + i );
         cur = sses[ i ].getType();
      }
      // Quality check:
      if( qualityCheck )
      {
         int sum = 0;
         for( int i = 0; i < 4; i++ )
            sum += result[ i ];
         if( sum != sses.length - 3 )
            Util.bug( this.getClass().toString() + "
.valueOf: " +
               d.getDomainName() + ", 4, sum = " + sum +
                  ", should be " + ( sses.length-3 ) );
      }
      for( int i = 0; i < 4; i++ )
         result[ i ] *= myWeights[ i ];
      return result;
   }
   public String [] getHeaders()
   {
      String [] getHeaders = { "ALPHA_ALPHA",
"BETA_ALPHA",
         "ALPHA_BETA", "BETA_BETA" };
      return getHeaders;
   }

}
```

ns.tops.Holder

```
package ns.tops;

import ns.tops.diagram.*;
import ns.tops.cath.*;
import java.util.Map;

/** Just a basic holder... */

public class Holder implements TopsConstants
{
   public Map cath;
   public Map props;
   public Rule [] rules;
   public Diagram [] diagrams;

   public Map getCath() { return cath; }
   public Map getProps() { return props; }
   public Rule [] getRules() { return rules; }
   public Diagram [] getDiagrams() { return diagrams; }

   public String toString()
   {
      return
         "Holder:" + EOL +
         "   " + cath.size() + " cath-elements" + EOL +
         "   " + props.size() + " properties" + EOL +
         "   " + rules.length + " rules" + EOL +
         "   " + diagrams.length + " diagrams" + EOL;
   }

}
```

ns.tops.PropLanguage

```
package ns.tops;

/** Constants used for reading properties files. */

public interface PropLanguage
{

   static final String DEBUG_LEVEL = "debug.level";
   static final String VERBOSE_LEVEL = "verbose.level";
   static final String DIAGRAM_STRINGID =
"diagram.stringid";

   /** Folder of data files. */
   static final String DATA_FOLDER = "tops.data.folder";

   // Statistics stuff
   static final String STAT_MIN_ARCHES =
"stat.minimumArches";
   static final String STAT_CLASS= "stat.class";

   static final String DIAG_RE = "tops.diagrams.matcher";
   static final String INCLUDE_NULL_ENTRIES =
      "tops.diagrams.includeNullEntries";

   // Classification stuff

   static final String CLASSIFICATION_ENGINE =
      "classification.engine";
   static final String CLASSIFICATION_COLLAPSE_LEVEL =
      "classification.collapse.level";

   // Main stuff

   /** Name of main data file. */
   static final String DATA_FILE = "tops.data.filename";
   /** Name of cartoons file. */
   static final String CARTOONS_FILE =
"tops.data.cartoons";
   /** Name of rules file. */
   static final String RULES_FILE = "tops.data.rules";
   /** Name of cath file. */
   static final String CATH_FILE =
"tops.data.cathentries";
   /** Zip method of cartoons file. */
   static final String CARTOONS_ZIP =
"cartoons.zip.method";

   static final String generatorOutputType =
      "generator.output.type";
   static final String generatorOutputLevels =
      "generator.output.levels";

   /** DataSetGenerator training set output file. */
   static final String TRAINING_OUTPUT_FILE_NAME =
      "generator.training.filename";
   /** DataSetGenerator test set output file. */
   static final String TEST_OUTPUT_FILE_NAME =
      "generator.test.filename";
   /** ID of splitter. */
   static final String SPLITTER =
"generator.splitter.id";
   /** Argument to splitter, if applicable. */
   static final String SPLITTER_ARG =
      "generator.splitter.argument";
   /** Number of rule to use in DataSetGenerator. */
   static final String RULE_NUMBER = "rules.number";

   static final String QUANTIFIER = "quantifier.id";
   static final String QUANTIFIER_ARG =
"quantifier.argument";
   static final String TOPOLOGY = "topology";

}
```

ns.tops.ClassificationEngine

```
package ns.tops;

import ns.tops.diagram.*;

/** Interface for classes that classify diagrams
according
  * to given rules. Classes that implement this interface
  * contain one method to classify a set of Diagrams, and
  * may contain additional methods. No constructors are
specified;
  * an engine may require a set of diagrams on which to
train,
  * and additional parameters.
  */

public interface ClassificationEngine
{

   /** Returns a classification of the given diagrams.
     * No restrictions on the number of levels or
structure of
     * classification are given.
     * @param diagrams An array of diagrams to be
classified.
     * @return A Collection containging Collections of
Diagrams
     */

   public java.util.Collection classify( Diagram []
diagrams );
}
```

ns.tops.CathClassificationEngine

```
package ns.tops;

import banda.stat.clust.*;
import java.util.*;
import ns.Util;
import ns.tops.cath.*;
import ns.tops.diagram.*;

/** Classifies known structures according to their CATH-
entry.
  * Uses the Diagrams pdp-id to obtain a classification,
given
  * a map between pdp-id's and Cath-entries.
  */


public class CathClassificationEngine
   implements ClassificationEngine
{

   // object variables:
   private  int level = 1;

   /** Hide empty constructor. */
   private CathClassificationEngine () {};

   /** Constructor.
     * @param level The level on which to classify.
     *          Legal values are 1 to 5
     */
   public CathClassificationEngine( int level )
   {
      if( level >= 1 || level >= 5 )
         this.level = level;
      else throw new IllegalArgumentException(
         this.getClass().toString() + ": illegal level("
+ level + ")" );
   };

   /** Returns a flat classification as an array. */
   public static int[] fastClassify( final Diagram[]
diagrams, final int level ) {
      final int[] s = new int[ diagrams.length ];
      for( int i = 0; i < diagrams.length; i++ )
         s[ i ] = diagrams[ i ].getCathEntry().get( level
);
      return s;
   }

   /** Creates a 2-level classification of the diagrams
according
     * to their CathEntry at the given level.
     */
   public Collection classify( final Diagram [] diagrams
)
   {
      final int[] c = fastClassify( diagrams, level );
      Map main = new HashMap();
      for( int i = 0; i < c.length; i++ ) {
         Integer key = new Integer( i );
         Collection ctemp = (Collection) main.get( key );
         if( ctemp == null ) {
            ctemp = new HashSet();
            main.put( key, ctemp );
         }
         ctemp.add( diagrams[ i ] );
      }
      return main.values();
   }

   /** Creates a 2-level classification of the diagrams
according
     * to their CathEntry at the given level.
     */
   public static Collection classify( Diagram []
diagrams, int lev ){
     return new CathClassificationEngine( lev ).classify(
diagrams );
   }


}
```

ns.tops.Main

```
package ns.tops;

import ns.Util;
import ns.tops.diagram.*;
import ns.tops.cath.*;
import ns.tops.parser.TopsParser;
import ns.tops.distances.*;
import ns.clust.*;

import java.io.*;
import java.util.*;
import java.util.zip.*;
import java.text.*;

/** Main class: Used to update database, test classes,
  * run DataSetGenerator etc.
  *
  * TODO: Fix / delete methods, rename stuff etc... lots!
  */

public class Main implements PropLanguage, TopsConstants
{

   // Some really hardcoded sensible defaults:

   protected static String optionsSuffix = "-D";
   protected static String scriptSuffix = "@";
   protected static String propertiesFile =
"tops.properties";
   protected static String propertiesHeading =
      "# User-defaults for tops-program. ";
   protected static String defaultPropertiesFile =
      "tops.default.properties";

   // read ini-file at startup:
   static Properties properties = getProperties();

   // Main holder
   protected static Holder myHolder = null;

   public static Holder getHolder()
   {
      if( myHolder == null )
         myHolder = FileManager.loadHolder();
      return myHolder;
   }


   protected static void usage()
   {
      String help =
            "Usage:     java tops.Main [ option | command
) * "+EOL+
            "           where command is: update | test |
" +
                        "defaults | generate | gzip |..."
+ EOL +
            "           and option is:     " +
optionsSuffix +
                        "prop=value" + EOL +
                        "See readme.html for more help.";
      System.out.println( help );
   }

   public static void main( String [] args) throws
Exception
   {
      Runtime r = Runtime.getRuntime();
      if( Util.verboseLevel >= 15 )
         properties.list( System.out );

      parseOption( null );

      System.out.println();

      if( args.length == 0 )
         doInteractive();
      else
         doCommands( args );
      return;
   }

   private static void doInteractive()
   {
      System.out.println( "Interactive mode. Enter
commands, " +
          "separated by enter or space. " + EOL +
          "Enter quit to finish. " );
      try
      {
         BufferedReader r = new BufferedReader(
            new InputStreamReader( System.in) );
         while( true ) //r.ready()
         {
            System.out.print( ">" );
            String command = r.readLine();
            if( command.equals( "quit" ) )
               break;
            else if( command.trim().equals( "" ) )
               continue;
            String [] args = new String[ 1 ];
            args[ 0 ] = command;
              System.out.println( "Doing command: " +
command );
            doCommands( args );
         }
      } catch( Exception e ) { e.printStackTrace(); }
      return;
   }

   private static void doCommands( String [] args )
throws Exception
   {
      for( int i = 0; i < args.length; i++ ) {
           args[ i ] = args[ i ].trim();
         if( args[ i ] == null || args[ i
].trim().equals( "" ) ) continue;
         else if( args[ i ].indexOf( scriptSuffix ) != -1
)       doScript( args[ i ] );
         else if( args[ i ].indexOf( optionsSuffix ) != -
1 )      parseOption( args[ i ] );
         else if( args[ i ].equals( "interactive" ) )
doInteractive();
         else if( args[ i ].equals( "wait" ) )
Util.waitForInput( "waiting..." );
         else if( args[ i ].equals( "update" ) )
myHolder = FileManager.updateAll();
         else if( args[ i ].equals( "test" ) )
test();
         else if( args[ i ].equals( "defaults" ) )
setProperties();
         else if( args[ i ].equals( "print" ) )
printDiagrams( args, i+1 );
         else if( args[ i ].equals( "stat" ) )
Stat.doit();
         else if( args[ i ].equals( "generate" ) )
DataSetGenerator.doit();
           else if( args[ i ].equals( "list" ) )
properties.list( System.out );
           else if( args[ i ].equals( "read" ) )
getHolder();
         else if( args[ i ].equals( "gzip" ) )
FileManager.gzipIt( args[ i + 1 ] );
         else if( args[ i ].equals( "gunzip" ) )
FileManager.gunzipIt( args[ i + 1 ] );
         else if( args[ i ].equals( "cluster" ) )
cluster();
           else if( args[ i ].equals( "r" ) )
properties = getProperties();
         else
            usage();
      }

      // that's it
      return;
   }



   private static void doScript( String fname )
   {
      try
      {
         fname = fname.substring( scriptSuffix.length()
);
         BufferedReader r = new BufferedReader( new
FileReader( fname ) );
         Collection c = new Vector();
         while( r.ready() )
            c.add( r.readLine() );
         doCommands( ( String[] )c.toArray( new String[ 0
] ) );
      }
      catch( FileNotFoundException e ) { Util.out( "File
" + fname + " not found. " ); }
      catch( Exception e ) { Util.out( "Scripterror: " +
e ); e.printStackTrace(); }
   }

  private static void parseOption( String arg ) {
    try
      {
     if( arg != null ) {
       String prop = arg.substring(
optionsSuffix.length(), arg.indexOf( '=' ) );
       String val = arg.substring( arg.indexOf( '=' ) + 1
);
       Util.verbose( "Setting \"" + prop + "\" to \"" +
val + "\"", 5 );
       properties.setProperty( prop, val );
     Util.verboseLevel = Integer.parseInt(
properties.getProperty( VERBOSE_LEVEL ) );
     Util.debugLevel   = Integer.parseInt(
properties.getProperty( DEBUG_LEVEL ) );
     Diagram.stringID  = Integer.parseInt(
properties.getProperty( DIAGRAM_STRINGID ) );
   }
  }
    catch( Exception pe ) { pe.printStackTrace();
Util.verbose( "Illegal option.", 3 ); }
    return;
  }
```

```
   public static void setProperties()
   {
      try
      {
         properties.store( new FileOutputStream(
propertiesFile ), propertiesHeading );
      }
      catch( Exception e ) { e.printStackTrace(); }
   }


   private static Properties getProperties()
   {
      Properties properties = new Properties(
getDefaults() );
      try
      {
         properties.load( new FileInputStream(
propertiesFile ) );
      }
      catch( FileNotFoundException fnfe )
      {
         System.err.println( "DomainProperty file " +
propertiesFile + " not found. " + EOL +
               "Use \"java ns.tops.main d\" to create a
default properties file. " );
      }
      catch( Exception e ) { e.printStackTrace(); }
      parseOption( null );
      return properties;
   }


   private static void test()
   {

      Holder h = getHolder();

      Map pdpMap = new HashMap();

      for( int i = 0; i < h.diagrams.length; i++ )
         pdpMap.put( h.diagrams[ i ].getDomainName(),
h.diagrams[ i ] );

      Map props = h.props;
      Map cath = h.cath;
      Rule [] rules = h.rules;
      Diagram [] diagrams = h.diagrams;

      Util.verbose( EOL );

      // Check serialization
      Util.verbose( "Serialization" );
      Util.verbose( "   Cath-entries: " + cath.size() );
      Util.verbose( "   Properties: " + props.size() );
      Util.verbose( "   Diagrams:" + diagrams.length );
      Util.verbose( "   Rules: " + rules.length );
      Util.verbose( "" );

      // CathMatcher:
      Util.verbose( "CathMatcher:" );
      String diagRE = properties.getProperty( DIAG_RE );
      boolean b = Boolean.valueOf(
properties.getProperty( INCLUDE_NULL_ENTRIES )
).booleanValue();

      Util.verbose(  "   To parse: " + diagRE + "(" + b +
")" );
      CathMatcher cm = CathMatcher.parse( diagRE );

      cm.MATCH_UNCLASSIFIEDS = b;
      Util.verbose0( "   toString: " + cm.toString() );
      Util.verbose( ( cm.MATCH_UNCLASSIFIEDS ?
"(inclusive)" : "(exclusive)" ) );

      // Check selection
      Util.verbose( "Selection:" );
      diagrams = Diagram.selectDiagrams( diagrams, cm );
      Util.verbose( "   Got " + diagrams.length + "
diagrams matching " + cm );

      StringBuffer sb = new StringBuffer();
//      for( int i = 0; i < Math.min( diagrams.length,
100*4 ); i+=100 )
      for( int i = 0; i < Math.min( diagrams.length,
100*2 ); i+=100 )
         sb.append( diagrams[ i ].toString() );

      Util.verbose( "   2 sample diagrams: " +
sb.toString() );
      Util.verbose( "" );

      // Other stuff:

      Util.verbose( "DomainProperty:" );
      String fullName = null; Diagram d=null;
      //      String [] pdpName = { "2stv00", "2bopA0",
"2sttA0", "2tbvA0", "13pkA1" };
      String [] pdpName = { "2stv00", "2bopA0" };
      for( int i = 0; i < pdpName.length; i++ )
      {
         d = (Diagram)pdpMap.get( pdpName[ i ] );
         if( d == null )
            continue;
         String property = "parallel";
         fullName = d.getDomainName() + "/" + cath.get(
d.getDomainName() );
         DomainProperty p = (DomainProperty) props.get(
property );

         if( p == null )
            Util.debug( "Ups: p == " + p, 5 );

         if( d == null )
            Util.debug( "Ups: p == " + p, 5 );

         Util.verbose( "   Sample diagram with fullname "
+ fullName );
          Util.verbose( "      has tostring() " +
d.toString() );
         TopsParser.printDiagram( d );
         Util.verbose( "      has " + p.valueOf( d ) + "
" + p );
         Util.verbose( "" );
      }

      // Rules

      Util.verbose( "Rule" );

      Rule r = rules[ 0 ];
      r.numberSize = 3;

      Util.verbose( "   Rule:" + r );
      Util.verbose( "   applied to  " + fullName + "
yields: " );
      Util.verbose( "   " + r.stringOf( d ) + EOL );
      Util.verbose( "   rule.unparse() = " + r.unparse()
);

      // Properties:

      Util.verbose( "Properties: " );
      Util.verbose( props );
      Util.verbose( "---" );


   }



   private static void printDiagrams( String [] args, int
index )
   {
      Holder h = getHolder();

      Map pdpMap = new HashMap();
      for( int i = 0; i < h.diagrams.length; i++ )
         if( h.diagrams[ i ].getDomainName() == null )
            Util.debug( "ups altså. " );
         else
            pdpMap.put( h.diagrams[ i ].getDomainName(),
h.diagrams[ i ] );

      Quantifier q = getQuantifier();

      Util.verbose( "Quantifer: " + q );

      for( int i = index; i < args.length; i++ )
      {
         Diagram d = (Diagram) ( pdpMap.get( args[ i ] )
);

         if( d == null )
            Util.debug( "ups: d == null" );

         Util.verbose( d.unparse() );

         Util.verbose( q.stringOf( d ) + "" );
      }
      return;
   }


   public static Collection merge( Collection c )
   {
      Collection main = new Vector();
      for( Iterator i = c.iterator(); i.hasNext(); )
      {
         Object o = i.next();
         if( o instanceof Collection )
            main.addAll( merge( (Collection)o ) );
         else
            main.add( o );
      }
      return main;
   }

   static void printCollection( Collection c, PrintStream
out )
   {
      printCollection( c, "", 2, 0, out );
      System.out.println( "done printing collection c" );
   }
```

```
   public static Properties getDefaults()
   {
      Properties p = new Properties();
      try{ p.load( new FileInputStream(
defaultPropertiesFile ) ); }
      catch( Exception e ) { e.printStackTrace(); }
      return p;
   }



   public static Quantifier getQuantifier()
   {
      String id = properties.getProperty( QUANTIFIER );
      Util.debug( "getQuantifier: id = " + id + ".", 16
);
      if( id.indexOf( "Rule" ) != -1  )
      {
          int i = 0;
          try{
            i = Integer.parseInt( properties.getProperty(
QUANTIFIER_ARG ).trim() );
           } catch( Exception e ) { e.printStackTrace();
Util.out( "Generator: Warning: Setting i = 0. " ); }
         Util.debug( "getQuantifier: returning " +
getHolder().rules[ i ], 10 );
         return getHolder().rules[ i ];
      }
      else if( id.indexOf( "Pairwise" ) != -1 )
      {
         Quantifier q = new PairwiseQuantifier();
         float [] d = { 1/0.3222f, 1/1.6663f, 1/1.672f ,
1/6.5729f };
         q.setWeights( d );
         return new PairwiseQuantifier();
      }
      else {
         Util.out( "Ups: got null quantifier" );
         return null;
      }
   }

   public static CathMatcher getCathMatcher()
   {
      return CathMatcher.parse( properties.getProperty(
DIAG_RE ) );
   }

   public static Diagram [] getDiagrams()
   {
      CathMatcher cm = getCathMatcher();
      return Diagram.selectDiagrams(
getHolder().diagrams, cm );
   }

   public static DataSetGenerator.Splitter getSplitter()
   {
      if( true )
         return DataSetGenerator.getSplitter(
Integer.parseInt( properties.getProperty( "level" ) ), 15
);
      String id = properties.getProperty( SPLITTER );
      if( id.indexOf( "Training" ) != -1 )
         return
DataSetGenerator.getTrainingOnlySplitter();
      else if( id.indexOf( "Random" ) != -1 )
      {
         int p = Integer.parseInt(
properties.getProperty( SPLITTER_ARG ) );
         return DataSetGenerator.getRandomSplitter( p );
      }
      else if( id.indexOf( "Representative" ) != -1 )
      {
         int l = Integer.parseInt(
properties.getProperty( SPLITTER_ARG ) );
         return
DataSetGenerator.getRepresentativeSplitter( l );
      }
      else
         return null;
   }

   public static PrintWriter getTestOut()
   {
      try{ return new PrintWriter( new FileOutputStream(
properties.getProperty( TRAINING_OUTPUT_FILE_NAME ) ) );
      } catch( Exception e ) { e.printStackTrace();
return null; }
   }

   public static PrintWriter getTrainingOut()
   {
      try{ return new PrintWriter( new FileOutputStream(
properties.getProperty( TEST_OUTPUT_FILE_NAME ) ) );
      } catch( Exception e ) { e.printStackTrace();
return null; }
   }

   private static void printCollection( Collection c,
String prefix, int level, int lines, PrintStream out )
   {
      if( lines >= 20 && out.equals( System.out ) ) {
         Util.waitForInput();
         lines = 0;
      }
      else
         lines++;
      if( level == 0 )
         c = merge( c );


      final String fix = "   ";
      out.println( prefix + c.getClass().toString() + ":
" + c.size() + " elements. " );
      out.flush();
      lines++;
      for( Iterator i = c.iterator(); i.hasNext(); )
      {
         Object o = i.next();
         if (o == null)
            continue;
         if( o.getClass() == Diagram.class ) {
            out.println( prefix + fix + ( (Diagram)o
).getCathEntry() );
            out.flush();
            lines++;
         }
         else
            printCollection( (Collection)o, prefix + fix,
level-1, lines++, out );
      }
      return;
   }


   private static void cluster() throws Exception {

      String f1 = properties.getProperty( "cluster.out"
);
      PrintStream out = f1.equals( "System" ) ?
System.out : new PrintStream( new FileOutputStream( f1 )
);

      f1 = properties.getProperty( "cluster.resout" );
      PrintStream resout = new PrintStream( new
FileOutputStream( f1 ) );

      Util.out( "Sending results to " + f1 );
      resout.println( "Output started at " + new Date()
);

      int level = Integer.parseInt(
properties.getProperty( "level" ) );
      CathMatcher cm = getCathMatcher();

      // Quantifiers:
      Quantifier [] q = new Quantifier[ 5 ];
      for( int i = 0; i < 4; i++ ) {
         properties.setProperty( "quantifier.argument",
Integer.toString( i ) );
         q[ i ] = getQuantifier();
      }
      q[ 4 ] = new PairwiseQuantifier();

      out.println( "Cathmatcher: " + cm );

      Diagram [] diagrams = Diagram.selectDiagrams(
getHolder().diagrams, cm );
      Diagram.stringID = level;

      out.println( "\n\n\nLevel " + level + ":" );

      out.print( "Getting cath classification" );
      Util.tock();
      final int[] cathClassification =
CathClassificationEngine.fastClassify( diagrams, level );
      final int m;
      {
         HashSet set = new HashSet();
         for( int i = 0; i < cathClassification.length;
i++ )
            set.add( new Integer( cathClassification[ i ]
) );
         m = set.size();
      }
      out.println( " done, " + m + " categories. (" +
Util.tick() + " millis)\n" );


      final int monty = Integer.parseInt(
properties.getProperty( "monty" ) );

      // For each quantifiers:
      for( int func = 0; func < q.length; func++ )
      {
         Util.tock();
           out.print( "Calling quantifer..." );
```

```
           final float[][] x = q[ func ].valueOf(
diagrams );
           out.println( "done (" + Util.tick() + "
millis)" );

           out.print( "Calling isodata algorithm, level="
+ level + ", diags=" + diagrams.length
              + ", q=" + q[ func ] );
           Util.tock();
           final int [] classification = Isodata.cluster(
x, m );
         out.println( " done (" + Util.tick() + "
millisecs)" );

           out.print( "Getting original statistics: " );
           Util.tock();
           float [] firstStats =
ClassificationComparator.fastStats( cathClassification,
classification );
         out.println( "   ... done (" + Util.tick() + "
millisecs)" );


         out.print( "Getting Hubert for original values"
);
         Util.tock();

         final int N = x.length;

         // Generate a proximity matrix to use with
hubert:
         final float[][] P = new float[ N ][ N ];
         for( int i = 0; i < N; i++ )
            for( int j = 0; j < N; j++ )
               P[ i ][ j ] = Isodata.sqeu( x[ i ], x[ j ]
);


         // Generate another proximity matrix to use with
hubert:
         final float[][] Y = new float[ N ][ N ];
         for( int i = 0; i < N; i++ )
            for( int j = 0; j < N; j++ )
               Y[ i ][ j ] = classification[ i ] ==
classification[ j ] ? 0 : 1;

         final float origHubert =
ClassificationComparator.hubert( P, Y );
         out.println( " done (" + Util.tock() + ",
origHubert=" + origHubert );

         out.print( "Getting " + monty + " Monte-Carlo
classifications and comparing them" );
         Util.tock();
         final float[] mchuberts =
ClassificationComparator.montecarlohubert( P, monty, m,
new Random() );
         out.println( " done (" + Util.tick() + "
millis), origHubert=" + origHubert );

            resout.println();  resout.println();
resout.println();
            resout.println( "Got isodata classification:
" );
            resout.println( "   level=" + level );
            resout.println( "   cath.size=" +
cathClassification.length );
            resout.println( "   clust.size=" +
classification.length );
            resout.println( "   q=" + q[ func ] );
            resout.println();
            resout.println( "Standard indices: " );

            String [] names =
ClassificationComparator.NAMES;

            for( int i = 0; i < names.length-1; i++ )
               resout.println( "   " + names[ i ] + ": "
+ formatFloat( firstStats[ i ] ) );
            resout.println( "   " + names[ names.length-1
] + ": " + ( firstStats[ names.length-1 ] ) );


            float hub = 0;
            for( int i = 0; i < mchuberts.length; i++ )
               hub += mchuberts[ i ];
            hub /= mchuberts.length;

            resout.println( "Origiginal hubert: " +
formatFloat( origHubert ) + "(" + formatFloat( hub ) +
")" );
            resout.println( "Average mc-hubert: " + hub
);
            resout.println( "Largest mc-hubert: " +
mchuberts[ mchuberts.length-1 ] );
            resout.println(); out.println();
            resout.flush();

     }

     resout.println( "Output ended at " + new Date() );
     resout.close();
```

```
   }

   protected static NumberFormat dnf =
      NumberFormat.getNumberInstance( Locale.US );
   static
   {
      dnf.setMaximumFractionDigits( 2 );
      dnf.setMinimumFractionDigits( 2 );
      dnf.setMaximumIntegerDigits( 3 );
      dnf.setGroupingUsed( false );
   }

   protected static String formatFloat( float d )
   {

      StringBuffer sb = new StringBuffer();

      FieldPosition fp = new FieldPosition(
dnf.INTEGER_FIELD );
      dnf.format( d, sb, fp );

      int missing = 3 - fp.getEndIndex();

      for( int i = 0; i < missing; i++ )
         sb.insert( 0, ' ' );

      return sb.toString();
   }


}
```

ns.tops.Stat

```
package ns.tops;

import ns.Util;
import ns.tops.diagram.*;
import ns.tops.cath.*;
import ns.tops.distances.*;

import VisualNumerics.math.Statistics;
import java.util.*;
import java.io.*;

/** A class for obtaining and displaying statistical
  * information about quantifiers and sets of diagrams.
  */

public class Stat
   implements java.io.Serializable, PropLanguage,
TopsConstants
{

   static Properties properties = Main.properties;

   public static void main( String [] args )
   {
      doit();
   }

   public static void doit()
   {
     Diagram [] diags = Diagram.selectDiagrams(
Main.getHolder().diagrams, Main.getCathMatcher() );
      System.out.println( "Statistics for " +
Main.getCathMatcher() + "(" + diags.length + " diags):
\n" );
      stat( diags, Main.getQuantifier(), new PrintWriter(
System.out, true ) );
   }


  public static void stat( Diagram[] diags, Quantifier q,
PrintWriter out ) {

    float [][] data = getData( diags, q );

    final int [] tabs = { 20, 10, 10, 10, 10, 8 };
    String [] fields = { "Field", "Average", "VarCoEff",
"Kurtosis", "Std dev", "Total" };
    String [] headers = q.getHeaders();

    for( int i = 0; i < fields.length; i++ )
      out.print( Util.rightPad( fields[ i ], tabs[ i ] )
);
      out.println();

      for( int i = 0; i < q.getSize(); i++ )
     {
       out.print( Util.rightPad( headers[ i ], tabs[ 0 ]
) );
       for( int j = 0; j < data[i].length; j++ )
            out.print( Util.rightPad( formatFloat( data[
i ][ j ], 4 ), tabs[ j+1 ] ) );
       out.println();
     }
      out.println();
  }


   public static float [][] getData( Diagram [] diags,
Quantifier r )
   {
      // First, avoid exceptions:
      if( diags == null || r == null ) return null;

      // Util.verbose0( "Calculating values, " +
diags.length + " diagrams:" );


      // Run through all diagrams, get values, compute
average, stddev and total

      float [][] ar = new float[ diags.length ][
r.getSize() ];
      for( int i = 0; i < diags.length; i++ )
      {
         ar[ i ] = r.valueOf( diags[ i ] );
      }

      // Compute std and average for each collumn of ar

      float [][] getData = new float[ r.getSize() ][ 5 ];

      for( int i = 0; i < r.getSize(); i++ )
      {
         float [] da = new float[ ar.length ];
         for( int j = 0; j < ar.length; j++ )
         {
            da[ j ] = ar[j][i];
         }
         float std = (float)
Statistics.standardDeviation(
QuantifierDistance.transform( da ) );
         float ave = (float) Statistics.average(
QuantifierDistance.transform( da ) );
         float coVar = (float) (std / ave);
         float kurt = (float) Statistics.kurtosis(
QuantifierDistance.transform( da ) );

         float sum = ave*da.length;
         getData[ i ][ 0 ] = ave;
         getData[ i ][ 1 ] = coVar;
         getData[ i ][ 2 ] = kurt;
         getData[ i ][ 3 ] = std;
         getData[ i ][ 4 ] = sum;
      }

      return getData;
   }



   public static void printStats( Diagram [] diags,
Quantifier q, PrintWriter out )
   {
      // Assumes q.getSize() and q.getHeaders().length
matches!

      Util.debug( "printStats called, " + diags.length +
" diagrams, q=" + q, 5 );

      final int [] tabs = { 20, 10, 10, 10, 10, 8 };
      String [] fields = { "Field", "Average",
"VarCoEff", "Kurtosis", "Std dev", "Total" };

      String [] headers = q.getHeaders();

      // First get the data:
      float [][] data = getData( diags, q );

      for( int i = 0; i < fields.length; i++ )
         out.print( Util.rightPad( fields[ i ], tabs[ i ]
) );
      out.println();

      for( int i = 0; i < q.getSize(); i++ )
      {
         out.print( Util.rightPad( headers[ i ], tabs[ 0
] ) );
         for( int j = 0; j < data[i].length; j++ )
            out.print( Util.rightPad( formatFloat( data[
i ][ j ], 4 ), tabs[ j+1 ] ) );
         out.println();
      }
      out.println();
   }



   public static String formatFloat( float d, int digits
)
   {
      int exp = (int) Math.pow( 10, digits );
      int newInt = (int) ( d*exp );

      String formatFloat = "" + (float) newInt/exp;
      Util.debug( "exp: " + exp + ", newInt: " + newInt +
", s: " + formatFloat + ", d: " + d, 15 );

      return formatFloat;
   }


}
```

ns.tops.FileManager

```
package ns.tops;

import ns.Util;
import ns.tops.diagram.*;
import ns.tops.cath.*;
import ns.tops.parser.TopsParser;
import ns.tops.distances.*;

import banda.stat.clust.*;

import java.lang.reflect.*;
import java.io.*;
import java.util.*;
import java.util.zip.*;
import java.text.*;

/** FileManager class: Used to update database, load
files etc.
  */

public class FileManager implements PropLanguage,
TopsConstants
{

   // Some defaults
   protected static final String [] zipExts =
      { "", ".deflated", ".zip", ".gz" };
   protected static final Properties properties =
Main.properties;

   public static void main( String [] args) throws
Exception
   {
      return;
   }


   public static Map readProps()
   {
      return DomainProperty.readProps();
   }


   public static Map readCATH()
   {
      Map readCATH = null;
      try
      {
         readCATH = CathEntry.readCATH( getReader(
            properties.getProperty( CATH_FILE ) ) );
      }catch( Exception e ) { e.printStackTrace(); }

      if( readCATH == null )
         Util.bug( "pokker!!" );
      else
         Util.debug( "CathMap OK!!!!!!", 5 );
      return readCATH;
   }


   /** Reads the original source files, creates the
     * data file used internally and returns the data.
     */
   public static Holder updateAll()
   {
      Holder myHolder = new Holder();
      try
      {
         long t1 = System.currentTimeMillis();

         // Get folders and stuff:

         String dataFolder = properties.getProperty(
            DATA_FOLDER );
         String cartoonsFile =   properties.getProperty(
            CARTOONS_FILE );
         String rulesFile =      properties.getProperty(
            RULES_FILE );

         // Properties:
         myHolder.props = DomainProperty.readProps();

         // Cath
         myHolder.cath = readCATH();

         // diagrams:
         Reader r = new InputStreamReader(
            getInputStream( cartoonsFile ) );
         myHolder.diagrams = Diagram.readDiagrams(
            r, null, myHolder.cath );


         // rules:
         myHolder.rules = (Rule[]) TopsParser.parseRules(
            new FileInputStream( rulesFile ),
myHolder.props ).
               toArray( new Rule[ 0 ] );

         // Done, now serialize and quit.
         saveHolder( myHolder );
         long elapsed = ( System.currentTimeMillis() - t1
) / 1000;
         Util.verbose( "Total time: " + elapsed + "
secs)" + EOL );
      }
      catch( Exception e ) { e.printStackTrace(); }

      return myHolder;
   }


  /** Writes the data to the location specified in
    * the properties file.
    */

   public static void saveHolder( Holder h )
   {
      Util.verbose0( "Writing data : ", 2 );
      long t1 = System.currentTimeMillis();
      try
      {
         ObjectOutputStream s = new ObjectOutputStream(
            getOutputStream( properties.getProperty(
DATA_FILE ) ) );
         s.writeObject( h.cath );
         Util.verbose0( "cath ", 2 );

         s.writeObject( h.props );
         Util.verbose0( "props ", 2 );

         s.writeObject( h.rules );
         Util.verbose0( "rules ", 2 );

         s.writeObject( h.diagrams );
         Util.verbose0( "diagrams ", 2 );

         s.close();
      }
      catch( Exception e )
      {
         e.printStackTrace();
      }
      long elapsed = ( System.currentTimeMillis() - t1 )
/ 1000;
      Util.verbose( "done (" + elapsed + " secs)", 2 );
   }


  /** Loads the data from the location
    * specified in the properties file.
    */
   public static Holder loadHolder()
   {
      Util.verbose0( "FileManager: Reading data from
disk: ", 2 );
      System.out.flush();
      long t1 = System.currentTimeMillis();

      Holder h = new Holder();
      try
      {
         ObjectInputStream s = new ObjectInputStream(
            getInputStream( properties.getProperty(
DATA_FILE ) ) );

         h.cath = (HashMap) s.readObject();
         Util.verbose0( "cath ", 2 );
         System.out.flush();

         h.props = (HashMap) s.readObject();
         Util.verbose0( "props ", 2 );
         System.out.flush();

         h.rules = (Rule[]) s.readObject();
         Util.verbose0( "rules ", 2 );
         System.out.flush();

         h.diagrams = (Diagram[]) s.readObject();
         Util.verbose0( "diagrams ", 2 );
         s.close();
      }
      catch( ClassCastException e )
      {
         updateAll();
      }
      catch( Exception e )
      {
         e.printStackTrace();
      }
      long elapsed = ( System.currentTimeMillis() - t1 )
/ 1000;
      Util.verbose( "done (" + elapsed + " secs)", 2 );

      return h;
   }

  /** Returns a BufferedReader for the given filename.
    * Unzip zipped files if fname ends with gz(ip).
    */
  public static BufferedReader getReader( String fname )
    {
```

```
      return new BufferedReader( new InputStreamReader(
         getInputStream( fname ) ) );
    }

  /** Returns an OutputStream for the given filename.
    * Unzip zipped files if fname ends with gz(ip).
    * @return BufferedOutputStream or GZipOutputStream
    */
   public static OutputStream getOutputStream( String
fname )
   {
      try
      {
         final int bufsize = 5*1024*1024;
//overkillisneverenough
         String zipTarget = ".gz";
         if( fname.indexOf( zipTarget ) != 0 )
            return new GZIPOutputStream(
               new FileOutputStream( fname ), bufsize );
         else
            return new BufferedOutputStream(
               new FileOutputStream( fname ), bufsize );
      } catch( Exception e ) { e.printStackTrace();
return null; }
   }

  /** Returns an buffered InputStream for the given
filename.
   * Zips if fname ends with gz(ip).
   * @return BufferedInputStream or GZipInputStream
   */
   public static InputStream getInputStream( String fname
)
   {
      try
      {
         final int bufsize = 5*1024*1024;
//overkillisneverenough
         String zipTarget = ".gz";
         if( fname.indexOf( zipTarget ) != 0 )
            return new BufferedInputStream(
               new GZIPInputStream( new
BufferedInputStream(
                  new FileInputStream(
                     fname ) ), bufsize ), bufsize );
         else
            return new BufferedInputStream(
               new FileInputStream( fname ), bufsize );
      } catch( Exception e ) { e.printStackTrace();
return null; }
   }




   /** Creates a gzip-file.
     * The file is untouched, a new file named fname.gz
in the same
     * dir is created.
     * @param fname The name of the file to be gzipped.
     */
   public static void gzipIt( String fname )
   {
      int cacheSize = 1024*1024;
      try
      {
         long t1 = System.currentTimeMillis();

         File inFile = new File( fname );
         Util.verbose( "Opening input file " + inFile);
         InputStream is = new BufferedInputStream(
            new FileInputStream( inFile ), cacheSize );

         File outFile = new File( fname + ".gz" );
         Util.verbose( "Opening output file " + outFile
);
         OutputStream os = new GZIPOutputStream(
            new FileOutputStream( outFile ), cacheSize );


         Util.verbose( "Zipping file: " );


         byte [] buf = new byte[ cacheSize ];
         boolean finished = false;
         while( ! finished )
         {
            if( is.available() > cacheSize )
               is.read( buf );
            else
            {
               buf = new byte[ is.available() ];
               is.read( buf );
               finished = true;
            }
            os.write( buf );
         }

         is.close();
         os.close();
         buf = null;

         long elapsed = (
            System.currentTimeMillis() - t1 ) / 1000;
         Util.verbose( "Done zipping (" + elapsed + "
secs.)" );
      }
      catch( Exception e )
      {
         Util.err( "Error: " + e );
         e.printStackTrace();
         System.exit( 1 );
      }

   }

   /** Unzips a gzip-file.
     * The file is untouched, a new file named fname
     * minus gz in the same dir is created.
     * @param fname The name of the file to be gzipped.
     */

   public static void gunzipIt( String fname )
   {
      int cacheSize = 1024*1024;
      try
      {
         long t1 = System.currentTimeMillis();

         File inFile = new File( fname );
         Util.verbose( "Opening input file " + inFile );
         InputStream is = new GZIPInputStream(
            new FileInputStream( inFile ), cacheSize );

         File outFile = new File(
            fname.substring( fname.length() - 3 ) );
         Util.verbose( "Opening output file " + outFile
);
         OutputStream os =
            new BufferedOutputStream(
               new FileOutputStream( outFile ), cacheSize
);


         Util.verbose( "Unzipping file: " );


         byte [] buf = new byte[ cacheSize ];
         boolean finished = false;
         while( ! finished )
         {
            if( is.available() > cacheSize )
               is.read( buf );
            else
            {
               buf = new byte[ is.available() ];
               is.read( buf );
               finished = true;
            }
            os.write( buf );
         }

         is.close();
         os.close();
         buf = null;

         long elapsed = ( System.currentTimeMillis() - t1
) / 1000;
         Util.verbose( "Done unzipping (" + elapsed + "
secs.)" );
      }
      catch( Exception e )
      {
         Util.err( "Error: " + e );
         e.printStackTrace();
         System.exit( 1 );
      }

   }


}
```

ns.tops.DataSetGenerator

```
package ns.tops;

import ns.Util;
import ns.tops.cath.*;
import ns.tops.parser.*;
import ns.tops.diagram.*;

import java.io.*;
import java.util.*;

/** Class that generates test- and training sets. */

public class DataSetGenerator
   implements PropLanguage, TopsConstants
{

   /** 0 */
   public static final int SOM = 0;
   /** 1 */
   public static final int LVQ = 1;

   public static int state = SOM;
   public static int level = 1;

   /** A splitter! */
   protected interface Splitter
   {
      public void add( Diagram d, Collection trainingSet,
         Collection testSet );
   }

   public static void main( String [] args ) { doit(); }

   public static void doit()
   {
      PrintWriter testOut = Main.getTestOut();
      PrintWriter trainingOut = Main.getTrainingOut();
      Quantifier q = Main.getQuantifier();

      state = Integer.parseInt(
Main.properties.getProperty( "state" ) );
      String top = Main.properties.getProperty( TOPOLOGY
);

      level = Integer.parseInt(
Main.properties.getProperty( "level" ) );

      String header = "" + q.getSize();
      if( state == SOM )
         header += " " + top;

      testOut.println( header );
      trainingOut.println( header );

      generateDataSet( Main.getDiagrams(), q,
Main.getSplitter(), testOut, trainingOut );
      trainingOut.flush(); trainingOut.close();
      testOut.flush(); testOut.close();
   }

   public static Splitter getSplitter( final int levels,
final int percent )
   {
      return new Splitter()
      {
         final int l = levels;
         final int p = percent;
         final Map map = new HashMap();
         protected final Random randomizer = new Random(
1 );

         public String t( Diagram d )
         {
            CathEntry ce = d.getCathEntry();
            StringBuffer sb = new StringBuffer( "" +
ce.get( 1 ) );
            for( int i = 2; i <= l; i++ )
               sb.append( "." + ce.get( i ) );
            return sb.toString();
         }
         public void add( Diagram d, Collection
trainingSet, Collection testSet )
         {
            if( map.put( t( d ), d ) == null ||
randomizer.nextInt( 100 ) < p-1 )
               testSet.add( d );
            else
               trainingSet.add( d );
         }
         public String toString() { return
"supersplitter(" + l + ", " + p + ")"; }
      };
   }

   /** All diagrams goes to trainingset. */
   public static Splitter getTrainingOnlySplitter()
   {
      return new Splitter()
      {
         public void add( Diagram d, Collection
trainingSet, Collection testSet )
         {
            trainingSet.add( d );
         }
         public String toString() { return
"TrainingOnly()"; }
      };
   }

   /** Percent % goes randomly to testset, the rest to
trainingset. */
   public static Splitter getRandomSplitter( final int
percent )
   {
      return new Splitter()
      {
         protected final long seed = 1;
         protected final Random randomizer = seed == -1 ?
new Random() : new Random( seed );

         final int p = percent;
         public void add( Diagram d, Collection
trainingSet, Collection testSet )
         {
            if( randomizer.nextInt( 100 ) < p )
               testSet.add( d );
            else
               trainingSet.add( d );
         }
         public String toString() { return "Random(" + p
+ ")"; }
      };
   }

   /** One of each cath category goes to test, the rest
to trainingset. */
   public static Splitter getRepresentativeSplitter(
final int levels )
   {
      return new Splitter()
      {
         final int l = levels;
         final Map map = new HashMap();
         public String t( Diagram d )
         {
            CathEntry ce = d.getCathEntry();
            StringBuffer sb = new StringBuffer( "" +
ce.get( 1 ) );
            for( int i = 0; i <= l; i++ )
               sb.append( "." + ce.get( i ) );
            return sb.toString();
         }
         public void add( Diagram d, Collection
trainingSet, Collection testSet )
         {
            if( map.put( t( d ), d ) == null )
               testSet.add( d );
            else
               trainingSet.add( d );
         }
         public String toString() { return
"Representative(" + l + ")"; }
      };
   }

   protected static void writeSet( Quantifier q,
Collection c, PrintWriter out )
   {
      if( q == null || c == null | out == null )
         return;
      for( Iterator i = c.iterator(); i.hasNext();
out.print( stringOf( q, i.next() ) + EOL ) )
         ;
      out.close();
   }

  protected static String stringOf( Quantifier q, Object
o ) {
    if( state == LVQ )
      return q.stringOf( o );

    Diagram d = (Diagram) o;
    StringBuffer res = new StringBuffer( q.stringOf( d,
false ) );

    res.append( d.getCathEntry().get( level ) );
    return res.toString();
  }

   /** The method. */
   public static void generateDataSet( Diagram [] diags,
Quantifier q, Splitter sp, PrintWriter testOut,
PrintWriter trainingOut )
   {
      Util.verbose( "generateDataSet: ", 3 );
      Util.verbose( "   " + diags.length + " diagrams", 3
);
      Util.verbose( "   Quantifer: " + q, 3 );
      Util.verbose( "   Splitter: " + sp, 3 );
      Util.verbose( "   testOut: " + testOut, 3 );
      Util.verbose( "   training: " + trainingOut, 3 );
```

```
      Util.verbose( "   stringID: " + Diagram.stringID, 3
);

      int stringerType = Integer.parseInt(
Main.properties.getProperty( generatorOutputType ) );
      int stringerLevels = Integer.parseInt(
Main.properties.getProperty( generatorOutputLevels ) );

      Util.verbose0( "   Splitting...", 5 );
      Collection testSet = new Vector(), trainingSet =
new Vector();
      for( int i = 0; i < diags.length; i++ )
         sp.add( diags[ i ], testSet, trainingSet );

      Util.verbose0( " done,  writing test... ", 5 );

      writeSet( q, testSet, testOut );
      Util.verbose0( " done, writing training...", 5 );
      writeSet( q, trainingSet, trainingOut );
      Util.verbose( " done.", 5 );
      return;
   }
}
```

ns.tops.parser.Tops.jj

```
/**
  * Main TOPS parser.
  */

options
{
   LOOKAHEAD = 1;
   OPTIMIZE_TOKEN_MANAGER=true;
   STATIC=true;
   FORCE_LA_CHECK=true;
   ERROR_REPORTING=false;
}

PARSER_BEGIN(TopsParser)

package ns.tops.parser;

import ns.Util;
import ns.tops.*;
import ns.tops.diagram.*;
import ns.tops.cath.*;

import java.util.*;
import java.io.*;


/** Generated class to parse data files. */

public class TopsParser implements TopsConstants
{

   // Used when parsing rules:
   protected static Map myProps =
DomainProperty.readProps();

   // Used when parsing Diagrams:
   protected static SSE [] curSSEs;
   protected static Map myCathMap =
FileManager.readCATH();

   /** The main method tests the integrity of the given
datafiles.*/
   public static void main( String [] args )
      throws ParseException, FileNotFoundException
   {
      Util.tock();
      System.out.print( "Starting to parse: " );

      Collection elements;
      Iterator i;

      elements = parseRules( new FileInputStream( args[1]
), myProps );
      System.out.println( "done in " + Util.tick() + "
millisecs. " );
      System.out.println( "Total: " + elements.size() );
      i = elements.iterator();
      i.next();
      System.out.println( "Sample: " + i.next() );

      elements = parseDiagrams(
         new FileInputStream( args[0] ), myCathMap
      );
      System.out.println( "done in " + Util.tick() + "
millisecs. " );
      System.out.println( "Total: " + elements.size() );
      i = elements.iterator();
      i.next();
      System.out.println( "Sample: " + i.next() );
   }

   public static Rule parseRule( String rule, Map props )
   {
      try{
         myProps = props;
         if( jj_initialized_once )
            ReInit( new StringReader( rule ) );
         else
            new TopsParser( new StringReader( rule ) );
         return Rule();
      }
      catch( Exception e ) { e.printStackTrace(); return
null; }
   }

   public static Collection parseRules( InputStream is,
Map props )
   {
      try{
         myProps = props;
         if( jj_initialized_once )
            ReInit( is );
         else
            new TopsParser( is );
         return RuleFile();
      }
      catch( Exception e ) { e.printStackTrace(); return
null; }
   }
```

```
   public static Collection parseRules( Reader r, Map
props )
   {
      try{
         myProps = props;
         if( jj_initialized_once ) ReInit( r );
         else new TopsParser( r );
         return RuleFile();
      }
      catch( Exception e ) { e.printStackTrace(); return
null; }
   }


   public static Collection parseDiagrams(
      InputStream is, Map cathmap )
   {
      try{
         myCathMap = cathmap;
         if( jj_initialized_once ) ReInit( is );
         else new TopsParser( is );
         return DiagramFile();
      }
      catch( Exception e ) { e.printStackTrace(); return
null; }
   }

   public static Collection parseDiagrams( Reader r, Map
cathmap )
   {
      try{
         myCathMap = cathmap;
         if( jj_initialized_once ) ReInit( r );
         else new TopsParser( r );
         return DiagramFile();
      }
      catch( Exception e ) { e.printStackTrace(); return
null; }
   }

   public static Diagram parseDiagram( String diag, Map
cathmap )
   {
      try{
         myCathMap = cathmap;
         if( jj_initialized_once )
            ReInit( new StringReader( diag ) );
         else
            new TopsParser( new StringReader( diag ) );
         return Diagram();
      }
      catch( Exception e ) { e.printStackTrace(); return
null; }
   }

   public static void printDiagram( Diagram d ) {
      printDiagram( d, System.out ); }
   public static void printDiagram( Diagram d,
PrintStream out) {
      out.println( d.unparse() );
   }

}

PARSER_END( TopsParser )



// Initial definitions:


SKIP : /* WHITE SPACE */
{
  " "
| "\t"
| "\n"
| "\r"
| "\f"
}


SPECIAL_TOKEN : /* COMMENTS */
{
     < PERCENT_COMMENT: "%" (~["\n","\r"])*
("\n"|"\r"|"\r\n") >
  |  < HASH_COMMENT: "#" (~["\n","\r"])*
("\n"|"\r"|"\r\n") >
  |  < BAD_INPUT_FILE: "bad_inputfile.\n" >
}


TOKEN: /* LITERALS */
{
      < INT: ( <DASH> )? <DECIMAL_LITERAL> >
   |  < #DECIMAL_LITERAL: ( ( ["1"-"9"] (["0"-"9"])* ) |
"0" ) >
   |  < FLOAT:
         (["0"-"9"])+ "." (["0"-"9"])* (<EXPONENT>)?
         (["f","F","d","D"])?
       | "." (["0"-"9"])+ (<EXPONENT>)?
(["f","F","d","D"])?
       | (["0"-"9"])+ <EXPONENT> (["f","F","d","D"])?
       | (["0"-"9"])+ (<EXPONENT>)? ["f","F","d","D"]
      >
   |  < #EXPONENT: ["e","E"] (["+","-"])? (["0"-"9"])+ >
   |  < STRING_LITERAL: "'" ( ~[ "\'" ] )* "'" >
}


TOKEN : /* IDENTIFIERS */
{
      < Id: <LETTER> (<LETTER>|<DIGIT>)* >
   |  < #LETTER: ["_", "A"-"Z", "a"-"z" ] >
   |  < #DIGIT:  ["0"-"9" ] >
}


TOKEN : /* SEPARATORS */
{
      < SEPARATOR: ":-" >
   |  < LPAREN: "(" >
   |  < RPAREN: ")" >
   |  < LBRACE: "{" >
   |  < RBRACE: "}" >
   |  < LBRACKET: "[" >
   |  < RBRACKET: "]" >
   |  < SEMICOLON: ";" >
   |  < COMMA: "," >
   |  < DOT: "." >
   |  < DASH: "-" >
   |  < STAR: "*" >
   |  < DIV: "/" >
   |  < EQ: "=" >
}


   /* Diagram:==
      c(DomainName,
       ListOfNodeNumbers,
       ListOfHbonds,
       ListOfFixedStructures,
       ListOfChiralities ):-
       SetsOfNodeAssignments.
   */




Collection DiagramFile() :
{
   // ParseUnit: a file containing a set of diagrams
   Diagram x; Collection c = new ArrayList();
}
{
  ( x = Diagram() { c.add( x ); } )*  <EOF>
  { return c; }
}



Diagram Diagram():
{ String name; SSE [] sses; HBond [] hbonds;
   Structure [] structs; Chirality [] chirs; }
{
   <Id>
   <LPAREN>
         name = quotedString()               <COMMA>
         sses = InitialListOfNodeNumbers()   <COMMA>
         hbonds = ListOfHbonds()             <COMMA>
         structs = ListOfFixedStructures()   <COMMA>
         chirs = ListOfChiralities()
   <RPAREN>
   <SEPARATOR> SetsOfNodeAssignments( sses ) <DOT>
   {
      Diagram d = new Diagram( name, sses, structs,
hbonds, chirs );
      d.setCathEntry( ( CathEntry ) myCathMap.get( name )
);
      return d;
   }
}



SSE [] InitialListOfNodeNumbers():
{ Token x; Collection c = new ArrayList(); }
{
   <LBRACKET>
      ( x = <Id> { c.add( new SSE( x.toString() ) ); }
      ( <COMMA> x = <Id> { c.add( new SSE( x.toString() )
); } )* )?
   <RBRACKET>
   {
      curSSEs = (SSE[]) c.toArray( new SSE[0] );
      return curSSEs;
   }
}


SSE [] ListOfNodeNumbers():
{ SSE s; Collection c = new ArrayList(); }
{
   <LBRACKET> ( s = nodeId() { c.add( s ); }
      ( <COMMA> s = nodeId() { c.add( s ); } )* )?
   <RBRACKET>
```

```
   { return (SSE[]) c.toArray( new SSE[0] ); }
}


HBond [] ListOfHbonds():
{ Collection c = new ArrayList(); HBond x = null; }
{
   <LBRACKET>
      ( x = HbondsEntry() { c.add( x ); }
      ( <COMMA> x = HbondsEntry() { c.add( x ); } )* )?
   <RBRACKET>
   { return (HBond[]) c.toArray( new HBond[0] ); }
}

HBond HbondsEntry():
{ SSE f, t; int d; }
{
   f = nodeId() <DASH> d = hdir() <DASH> t = nodeId()
   { return new HBond( f, d, t ); }
}


Structure [] ListOfFixedStructures():
{ Collection c = new ArrayList(); Structure s;}
{
   <LBRACKET>
   (
      s = Fixed() { c.add( s ); }
      ( <COMMA> s = Fixed() { c.add( s ); } )*
   )?
   <RBRACKET>
   {
      return (Structure[])c.toArray( new Structure[ 0 ]
);
   }
}

Structure Fixed():
{ Token tT; SSE [] structs; }
{
      tT = <Id> <LPAREN> structs = ListOfNodeNumbers()
<RPAREN>
      {
     int type = UNKNOWN;
     for( int i = STRUCT_START; i < struct_ids.length;
i++ )
        if( struct_ids[ i ].equals( tT.toString() ) )
          type = i;
         return new Structure( type, structs );
      }
}


Chirality [] ListOfChiralities():
{ Collection c = new ArrayList(); Chirality chir; }
{
   <LBRACKET>
   (
      chir = ChiralityEntry() { c.add( chir ); }
      ( <COMMA> chir = ChiralityEntry() { c.add( chir );
} )*
   )?
   <RBRACKET>
   {
      return (Chirality[]) c.toArray( new Chirality[ 0 ]
);
   }
}

Chirality ChiralityEntry():
{ SSE f, t; int h; }
{
      <LPAREN>
         f = nodeId() <COMMA> h = hand() <COMMA> t =
nodeId()
      <RPAREN>
      { return new Chirality( f, h, t ); }
}


void SetsOfNodeAssignments( SSE [] sses ):
{}
{
   (
      NodeAssignment( sses )
      (
         <COMMA>
         NodeAssignment( sses )
      )*
   )?
   { return; }
}



void NodeAssignment( SSE [] sses ):
{ String id; SSE sse; }
{
   sse = nodeId()
   <EQ>
   NodeDesc( sse )
   { return; }
}


void NodeDesc( SSE sse):
{ Token noT; Token typeT; int d; }
{
   <LPAREN>
      noT =  <INT>      <COMMA>
      typeT = <Id>      <COMMA>
      d = dir()         <COMMA>
      point()           <COMMA>
      point()           <COMMA>
      ( <STRING_LITERAL> | <INT> )
   <RPAREN>
   {
      String type = typeT.toString();
      if( type.equals( "n" ) )
        sse.setType( NTERM );
      else if( type.equals( "c" ) )
        sse.setType( CTERM );
      else if( type.equals( "h" ) )
     sse.setType( HELIX );
      else if( type.equals( "e" ) )
     sse.setType( STRAND );
      else
         throw new ParseException(
            "NodeDesc(): unknown SSE type. "
         );

      sse.setNo( Integer.parseInt( noT.toString() ) );
      sse.setDir( d );
      return;
   }
}


SSE nodeId():
{ Token x; }
{
   x = <Id>
   {
      int i;
      for( i = 0; i < curSSEs.length; i++ )
         if( curSSEs[ i ].getName().equals( x.toString()
) )
            break;
      return curSSEs[ i ];
   }
}


/** Handedness: -1 (left), 0 (unknown), or 1 (right) */
int hand():
{ Token x; }
{
   x = <INT>
   { return Integer.parseInt( x.toString() ); }
}


/** Directon: -1 (down), 0 (unknown), or 1 (up) */
int dir():
{ Token x = null; }
{
   x = <INT>
   { return Integer.parseInt( x.toString() ); }
}

/** Directon: -1 (down), 0 (unknown), or 1 (up) */
int hdir():
{}
{
      "h(-1)"
      { return -1; }
   |  "h(1)"
      { return +1; }
}

/** Point: ( x, y ) */
Point point():
{ Token x, y; }
{
   <LPAREN>
      x = <INT> <COMMA> y = <INT>
   <RPAREN>
   { return new Point(
        Integer.parseInt( x.toString() ),
        Integer.parseInt( y.toString() )
     );
   }
}


Collection RuleFile() :
{ Rule x; Collection c = new ArrayList(); }
{
  ( x = Rule() { c.add( x ); } ) *
  <EOF>
  { return c; }
}


Rule Rule():
{
```

```
   Token com = null; Collection dpc = new Vector();
   Collection ws = new Vector();
}
{
   entries( dpc, ws ) <SEMICOLON>
   {
      DomainProperty [] dps = (DomainProperty[])
         dpc.toArray( new DomainProperty[ 0 ] );

      Float [] ds = (Float[]) ws.toArray( new Float[ 0 ]
);

      if( dps == null ||
          ds == null ||
          dps.length == 0 ||
          ds.length == 0 )
         Util.err( "ups: Error in tops.jj, " + dps + "
and " + ds );
      return new Rule( dps, ds,
         ( com == null ? "" : com.toString() )
      );
   }
}


void entries( Collection dpc, Collection ws ):
{
   DomainProperty curP; float curW;
   Token op = null;
   float w;
}
{
      curP = property() { dpc.add( curP ); }
      (
         ( op = <STAR> | op = <DIV> ) w = numeric()
         {
            if( op.toString().equals( "*")  )
               ws.add( new Float( w ) );
            else
               ws.add( w == 0.0 ? new Float( 1.0 ) :
                  new Float( 1.0 / w ) );
         }
      )?
      { if( ws.size() != dpc.size() )
         ws.add( new Float( 1.0 ) ); }
      ( <COMMA> entries( dpc, ws ) )?
   {
      return;
   }
}


DomainProperty property():
{ Token x; }
{
   x = <Id>
   { return (DomainProperty)myProps.get( x.toString() );
}
}


// General methods

float numeric():
{ Token x; }
{
   ( x = <FLOAT> | x = <INT> )
   { return Float.parseFloat( x.toString() ); }
}


String quotedString():
{ Token x; }
{
   x = <STRING_LITERAL>
   {
      String s = x.toString();
      return s.substring( 1, s.length() - 1 );
   }
}
```

ns.clust.Isodata

```
package ns.clust;

import java.util.Random;
import ns.Util;

/** An implementation of the isodata algorithm.
  */

public final class Isodata
{
   /** Writes dots to stdout, to show progress. */
   public static final boolean VERBOSE = true;

   /** Writes some debug info to stdout, to show what's
happening.*/
   public static final boolean DEBUG = false;

   /** Squared Euclidean distance. */
   public static float sqeu( final float [] v1, final
float [] v2 )
   {
      float sqeu = 0;
      for( int i = 0; i < v1.length; i++ )
         sqeu += Math.pow( v1[ i ] - v2[ i ], 2 );
      return sqeu;
   }


   /** Returns an array version of a clustering.
     * Complexity: O( x.length * x[ 0 ].length * m * iter
),
     *   where iter is the number of iterations
neccessary...
     *   typical values are < 50.
     * @param m number of clusters
     * @param x An array of float-vectors
     * @return An array clust so that the vector
     *   x[i] is in cluster number clust[i]
     */
   public static int[] cluster( final float[][] x, final
int m ) {

      // Create the result-matrix:
      final int[] b = new int[ x.length ];

      // Do the clustering
      // final float[][] theta = theta( x, m, b );
      theta( x, m, b, new Random() );

      return b;
   }


   static float[][] theta( final float [][] x, final int
m,
      final int[] b, final Random random )
   {

      if( b.length != x.length )
         throw new IllegalArgumentException(
            "Isodata: b.length must equal x.length"
         );

      final int N = x.length;
      final int size = x[ 0 ].length;

      final float [][] theta = new float[ m ][ size ];

      // Init the theta's:
      for( int i = 0; i < theta.length; i++ )
         for( int j = 0; j < size; j++ )
            theta[ i ][ j ] = random.nextFloat();

      for( boolean changed = true; changed; )
      {
         changed = false;
         if( VERBOSE )
            System.out.print( "." );
         else if( DEBUG )
            printFA( theta );


         for( int i = 0; i < N; i++ ) {

            // determine the nearest theta to x[i]  ->
curIndex
            float curDist = Float.MAX_VALUE;
            int curIndex  = Integer.MIN_VALUE;
            nearest_theta: for( int j = 0; j < m; j++ )
            {

               final float dist = sqeu( theta[ j ], x[ i
] );
               if( dist == 0 ) {
                  if( DEBUG ) {
                     System.out.println(
                        "Zero: theta[" + j + "], x[" + i
+ "]"
                     );
```

```
                    curDist = 0;
                 }
                 curIndex = j;
                 break nearest_theta;
              }
              else if( dist < curDist ) {
                 curIndex = j;
                 curDist = dist;
              }
           }

           // set b( i ) = curIndex
           b[ i ] = curIndex;

        }


        //determine theta[ ? ] as the mean of the
        //vectors x[ ? ] with b[ ? ] == ?:


        // Iterate over all theta[i]
        for( int i = 0; i < m; i++ ) {

           final float [] tempTheta = new float[ size ];
           int tempSize = 0;

           // Iterate over alle x[j]:
           for( int j = 0; j < N; j++ ) {

              //if b[j]==i then tempTheta += x[j] and
tempSize++;
              if( b[ j ] == i ) {
                 for( int index = 0; index < size;
index++ )
                    tempTheta[ index ] += x[ j ][ index
];
                 tempSize++;
              }
           }

           for( int index = 0; index < size; index++ ) {
              final float t = (
                 tempSize == 0 ? 0 : tempTheta[ index ]
/ tempSize
              );
              if( theta[ i ][ index ] != t ) {
                 theta[ i ][ index ] = t;
                 changed = true;
              }
           }
        }
     }

     return theta;
  }



  public static void main( String [] args ) {

     final Random random;
     if( args.length != 0 ) {
        int seed = Integer.parseInt( args[ 0 ] );
        System.out.println( "Using random seed " + seed
);
        random = new Random( seed );
     }
     else
        random = new Random();

     final int N = 10000;

     // generate test set:
     final float [][] centers = {
        { -15.0f, -3.0f },
        { 1.0f, -12.0f },
        { -1.0f, -3.0f },
        { 1.0f, 2.0f },
        { 3.5f, 3.5f },
        { 6.0f, 1.0f },
        { 0.5f, 3.5f },
        { 6.0f, 100.0f }
     };
     final int m = centers.length;
     final int size = centers[ 0 ].length;
     final float [][] x = new float[ N ][ size ];
     final int[] b = new int[ x.length ];

     for( int i = 0; i < N; i++ )
        for( int j = 0; j < size; j++ )
           x[ i ][ j ] = (float)random.nextGaussian() +
              centers[ i % centers.length ][ j ];

     // Cluster and write output
     System.out.print( "Clustering " + N + " vectors of
length " +
        size + " into " + m + " clusters" );
     Util.tock();
     final float [][] theta = theta( x, m, b, random );
     System.out.println( " done in " + Util.tick() + "
millis." );
     printFA( theta );
  }

  public static void printFA( float [][] fa ) {
     System.out.println( "-" );
     for( int i = 0; i < fa.length; i++ ) {
        System.out.print( "|" );
        for( int j = 0; j < fa[ 0 ].length; j++ )
           System.out.print( fa[ i ][ j ] + "; " );
        System.out.println();
     }
     System.out.println( "-" );
  }

}
```

ns.clust.HierarchicalClustering

```
package ns.clust;

import ns.Util;
import java.util.*;

/** Main class of the clust package; creates a binary
three with
  * a hierachical clustering, using the given linkage
method.
  */

public class HierarchicalClustering
{

   // Constants

  public static final int SINGLE_LINK = 0;
  public static final int COMPLETE_LINK = 1;
  public static final int SIMPLE_AVERAGE = 2;
  public static final int TEST_LINK = 99;

  protected static final boolean DEBUG = false;
  protected static boolean COUNT = true;

  // Object variables:

  private final Node [] nodes;
  private final int method;
  private final float[][] distMatrix;
  private final Distance distance;

  /** Whether to keep distances in the nodes or not.
    * Setting this to false may improve speed.
    */
  public boolean keepDist = true;

  /** Return the root node of the hiearchical clustering
    * with the given parameters.
    */
  public static Node cluster( final Object [] data, final
Distance d, final int method )
    {
      return new HierarchicalClustering( data, d, method
).cluster();
    }

  /** Constructs a cluster object with the given
parameters. */
  public HierarchicalClustering( final Object [] data,
final Distance distance, final int m )
    {
      method = m;
      this.distance = distance;

      nodes = new Node[ data.length ];
      for( int i = 0; i < nodes.length; i++ )
     nodes[ i ] = new Node( data[ i ] );

      distMatrix = new float[ data.length ][];
      for( int i = 0; i < data.length; i++ )
     {
       distMatrix[ i ] = new float[ i ];
       for( int j = 0; j < i; j++ )
         distMatrix[ i ][ j ] = distance.distance( data[
i ], data[ j ] );
     }
    }

  /** The method!! */

  public Node cluster()
    {
      for( int index = nodes.length; index > 1; index-- )
     {
       if( COUNT )
         System.out.print( "        \r" + index );

       // First: Find minimum value, assume distmatrix
OK:

       float min = Float.MAX_VALUE;
       int minX=-1, minY=-1;

       for( int i = 0; i < index; i++ )
         for( int j = 0; j < i; j++ )
           if( distMatrix[ i ][ j ] < min )
          {
            min = distMatrix[ i ][ j ];
            minX = i; // Note that i > j so that minX >
minY
            minY = j;
          }


       // Remove n[ minX ] and n[ minY ] and replace
them:
       {
         // Create a replacement for nodes minX and minY
         final Node tmp = new Node(
           nodes[ minX ],
           nodes[ minY ],
           distMatrix[ minX ][ minY ]
         );

         // Insert it into minY:
         nodes[ minY ] = tmp;

         // Swap X with last (index-1)
         nodes[ minX ] = nodes[ index-1 ];
       }


       // Now recalculate the distMatrix:
       {
         // First, recalculate
         for( int j = 0; j < minY; j++ )
           distMatrix[ minY ][ j ] =
          value(
                distMatrix[ minY ][ j ],
                distMatrix[ minX ][ j ]
                );

         for( int i = minX+1; i < index; i++ )
           distMatrix[ i ][ minY ] =
          value(
                distMatrix[ i ][ minX ],
                distMatrix[ i ][ minY ]
                );

         // Next, swap

         for( int i = 0; i < minX; i++ )
           distMatrix[ minX ][ i ] =
          distMatrix[ index - 1 ][ i ];

         for( int j = minX + 1 ; j < index - 1; j++ )
           distMatrix[ j ][ minX ] =
          distMatrix[ index - 1 ][ j ];

       }

       // and that's it, me think...

     }

      // Clear the counts?
      if( COUNT )
     System.out.print( "        \r" );

      return nodes[ 0 ];
    }


  /** Simple output method. */

  public void dumpMatrix( java.io.PrintStream out, int x,
int y )
    {
      out.println( "DistMatrix:" );
      for( int i = 1; i < distMatrix.length; i++ )
     {
       for( int j = 0; j < distMatrix[ i ].length; j++ )
         out.print( " " + ( ( i == x && j == y ) ? "-----
" : Util.formatFloat( distMatrix[ i ][ j ], 5 ) ) );
       out.println();
     }
    }


   // Hidden methods


  // Value method: uses the field method
  private final float value( float f1, float f2 )
    {
      switch( method )
     {
     case SINGLE_LINK: return Math.min( f1, f2 );
     case COMPLETE_LINK: return Math.max( f1, f2 );
     case SIMPLE_AVERAGE: return ( f1+f2 ) / 2;
     case TEST_LINK: return 42;
     default: throw( new RuntimeException( "cluster:
Unknown method." ) );
     }
    }



}
```

ns.clust.Node

```
package ns.clust;

import java.io.Serializable;
import java.util.*;


/** A node in a binary three. Each node is either a leaf
containing an item, or
  * a non-leaf containing two sub-threes, termed right
and left.
  */

public final class Node implements Serializable
{

  /** Left part of paranthesised String presentation.
    * See toString(). Default is '{'.
    */
  public static String LEFT_BRACE = "{";

  /** Right part of paranthesised String presentation.
    * See toString(). Default is '}'.
    */
  public static String RIGHT_BRACE = "}";

  // Hidden variables

  protected final Node left, right;
  protected final Object item;
  protected final boolean leaf;
  protected final int mySize;
  protected final int myDepth;
  protected final float myDist;


  // Constructors:

   Node( final Object i ) {
     item = i;
     leaf = true;
     mySize = 1;
     myDepth = 1;
     left = null;
     right = null;
     myDist = Float.MIN_VALUE;
   }

   Node( final Node l, final Node r ) {
      this( l, r, Float.MIN_VALUE );
   }


   Node( final Node l, final Node r, final float dist ) {
     left = l;
     right = r;
     leaf = false;
     mySize = left.mySize + right.mySize;
     myDepth = Math.max( left.myDepth, right.myDepth ) +
1;
     item = null;
     myDist = dist;
   }

  // Accessors:

  /** Returns true iff this node is a leaf */
  public boolean isLeaf() { return leaf; }

  /** Returns the number of leaf ancestors, or 1 if leaf.
*/
  public int size() { return mySize; }

  /** Returns the height of this node's subtree, 1 if
leaf. */
  public int height() { return myDepth; }

  /** Returns the distance between the sub-nodes, or
Float.MIN_VALUE if not assigned. */
  public float getDist() { return myDist; }

  /** Returns this node's item, or null if not leaf. */
  public Object getItem() { return item; }

  /** Returns the left children of this node, or null if
leaf. */
  public Node getLeft() { return left; }

  /** Returns the right children of this node, or null if
leaf. */
  public Node getRight() { return right; }

  /** Returns a paranthesied presentation of this node.
    * Uses LEFT_BRACE and RIGHT_BRACE.
    */
  public String toString() {
    if( leaf )
        return getItem().toString();
    else
        return LEFT_BRACE + left.toString() + ", " +
right.toString() + RIGHT_BRACE;
  }

  /** Adds all sub-elements to the collection c. */
  public void addElements( Collection c ) {
    if( leaf )
      c.add( item );
    else {
      left.addElements( c );
      right.addElements( c );
    }
  }


  /** Create a collection of collections given a minimum
distance.
    * If the internal distance is less than the given
parameter
    * the sub-elements are returned as a collection.
    * @return a Collection of collections of items.
    */
  public Collection collapse( float minDist ) {
     Collection c = new Vector();
     if( leaf ) {
        Collection d = new ArrayList();
        d.add( item );
        c.add( d) ;
     }
     else
        collapse( c, minDist );
     return c;
  }

   protected void collapse( Collection c, float minDist )
{
      if( leaf || myDist < minDist ) {
         Collection e = new ArrayList();
         addElements( e );
         c.add( e );
      }
      else {
         left.collapse( c, minDist );
         right.collapse( c, minDist );
      }
   }


}
```

ns.clust. ClassificationComparator

```
package ns.clust;


import java.util.*;
import ns.Util;

/** Compares classifications.
  */

public final class ClassificationComparator {

   // Constants

   public static final int RAND = 0;
   public static final int JACCARD = 1;
   public static final int FM = 2;

   public static final String [] NAMES = { "Rand   ",
"Jaccard", "FM     " };

   protected static final int SS = 0;
   protected static final int SD = 1;
   protected static final int DS = 2;
   protected static final int DD = 3;
   protected static final int SUM = 4;

   protected static final boolean DEBUG = false;
   public static final boolean VERBOSE  = true;

   protected static final String EOL =
         System.getProperty( "line.separator" );



   /* Returns the three implemented indices. */
   protected static float[] getStats( final int [] t ) {
      float [] res = new float[ 3 ];
      int m1 = t[ SS ] + t[ SD ];
      int m2 = t[ SS ] + t[ DS ];
      int M = t[ SUM ];
      res[ JACCARD ] = ( t[ SS ] / (float)( t[ SS ] + t[
SD ] + t[ DS ] ) );
      res[ FM ] = (float) ( t[ SS ] / Math.sqrt( m1 * m2
) );
      res[ RAND ] = ( t[ SS ] + t[ DD ] ) / (float)t[ SUM
];
      return res;
   }


   /* Returns the three implemented indices, given to
classifications. */
   public static float[] fastStats( final int[] c1, final
int[] c2 )
   {
      return getStats( getNumbers( c1, c2 ) );
   }


   /* Returns a binary proximity matrix. */
   public static boolean[][] proxMatrix( final int[] c )
   {
      boolean[][] Y = new boolean[ c.length ][];
      for( int i = 0; i < c.length; i++ )
         Y[ i ] = new boolean[ i ];

      for( int i = 0; i < Y.length; i++ )
         for( int j = i+1; j < Y.length; j++ )
            // Compare the pair (i, j)
            Y[ i ][ j ] = c[ i ] == c[ j ];
      return Y;
   }


   /* Returns the Hubert index. */
   public static float hubert( final float[][] X, final
float[][] Y )
   {
      if( X.length != Y.length || X[ 0 ].length != Y[ 0
].length )
         throw new IllegalArgumentException( "hubert:
wrong sizes." );

      float res = 0;
      final int N = X.length;
      final int M = (int) ( N * (N-1) / 2 );

      for( int i = 0; i < N-1; i++ )
         for( int j = 0; j < N; j++ )
            res += X[ i ][ j ] * Y[ i ][ j ];

      res /= M;
      return res;
   }

   /* Returns r generated Hubert indexes. */
   public static float[] montecarlohubert( final
float[][] distMatrix,
      final int r, final int m, final Random random )
   {
      final int N = distMatrix.length;
      final float[] hubs = new float[ r ];

      for( int iter = 0; iter < r; iter++ ) {
         if( VERBOSE )
            System.out.print( "." );
         // Generate a random partition:
         final int[] partition = new int[ N ];
         for( int i = 0; i < N; i++ )
            partition[ i ] = random.nextInt( m );

         final float[][] randMatrix = new float[ N ][ N
];
         for( int i = 0; i < N; i++ )
            for( int j = 0; j < N; j++ )
               randMatrix[ i ][ j ] = ( partition[ i ] ==
partition[ j ] ? 0 : 1 );

         hubs[ iter ] = hubert( distMatrix, randMatrix );
      }

      Arrays.sort( hubs );
      return hubs;

   }


   protected static int[] getNumbers( final int[] c1,
final int[] c2 )
   {
      if( c1.length != c2.length )
         throw new IllegalArgumentException(
            "getNumbers: c1.length!=c2.length"
         );

      final int m = c1.length;
      final int[] t = new int[ 5 ];

      for( int i = 0; i < m; i++ )
         for( int j = i+1; j < m; j++ ) {
            // Compare the pair (i, j)

            final boolean s1 = c1[ i ] == c1[ j ];
            final boolean s2 = c2[ i ] == c2[ j ];

            if( s1 && s2 )
               t[ SS ]++;
            else if( s1 && !s2 )
               t[ SD ]++;
            else if( !s1 && s2 )
               t[ DS ]++;
            else // if( !s1 && !s2 )
               t[ DD ]++;
         }

      t[ SUM ] = t[0]+t[1]+t[2]+t[3];
      return t;
   }

}
```