Metrics and Visualisation for Crime Analysis and Genomics

Jeroen F. J. Laros

This research is part of the DALE (Data Assistance for Law Enforcement) project as financed in the ToKeN program from the Netherlands Organization for Scientific Research (NWO) under grant number 634.000.430.

The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming Research and Algorithmics).

Cover: Stereogram of Figure 4.2.

ISBN: 978-90-9024936-0

Metrics and Visualisation for Crime Analysis and Genomics

Proefschrift

ter verkrijging van
de graad van Doctor aan de Universiteit Leiden,
op gezag van de Rector Magnificus prof. mr. P.F. van der Heijden,
volgens besluit van het College voor Promoties
te verdedigen op maandag 21 december 2009
klokke 15.00 uur

door

Jeroen Franciscus Jacobus Laros
geboren te Den Helder
in 1977

Promotiecommissie

Promotor:	prof. dr. J.N. Kok	
Co-promotor:	dr. W.A. Kosters	
Overige leden:	prof. dr. Th. Bäck	
	prof. dr. J.T. den Dunnen	(Leids Universitair Medisch Centrum)
	dr. H.J. Hoogeboom	
	prof. dr. X. Liu	(Brunel University)
	dr. P.E.M. Taschner	(Leids Universitair Medisch Centrum)

Contents

Chapter 1

Introduction

This introduction is structured as follows. The first three sections describe the main topics of the thesis. In the fourth section we give an overview.

1.1 Data Mining

Informally speaking, *Data Mining* [67] is the process of extracting previously unknown and interesting patterns from data. In general this is accomplished using different techniques, each shedding light on different angles of the data. Due to the explosion of data and the development of processing power, Data Mining has become more and more important in data analysis. It can be viewed as a subdomain of Artificial Intelligence (AI [61]), with a large statistical component [4, 28].

Amongst the patterns that can be found by the usage of Data Mining techniques, we can identify *Associations*. Examples of this can be found in market basket analysis. One of the (trivial) examples would be that tobacco and cigarette paper are often sold together. A more intricate example is that certain types of tobacco (light, medium, heavy) are correlated with different types of cigarette paper. This so-called *Association Mining* is an important branch of Data Mining. Other patterns that are frequently sought are *Sequential patterns*. Sequential patterns are patterns in sets of (time)sequences. These patterns can be used to identify trends and to anticipate behaviour of individuals. Associations and Sequential patterns will play a major role in this thesis.

Once patterns have been identified, we often need a visualisation of them to make the discovered information insightful. This visualisation can be in the form of graphs, charts and pictures or even interactive simulations.

Data Mining is commonly used in application domains such as marketing and fraud detection, but recently the focus also shifts towards other (more delicate) application domains, like pharmaceutics and *law enforcement*.

In this thesis we focus on the application domains law enforcement and sequence analysis. In law enforcement, we have all the prerequisites needed for

Data Mining: a plethora of data, lots of categories, temporal aspects and more. There is, however, a reluctance when it comes to using the outcome of an analysis. When used with care, Data Mining can be a valuable tool in law enforcement. It is not unthinkable, for example, that results obtained by Data Mining techniques can be used when a criminal is arrested. Based on patterns, this particular criminal could have a higher risk of carrying a weapon, or an syringe, for example. In law enforcement, this kind of information is called *tactical data*.

After the Data Mining step, statistics is usually employed to see how significant the found patterns are. In most cases, this can be done with standard statistics. When dealing with temporal sequences though, and lots of missing or uncertain data, this becomes exceedingly harder.

1.2 DNA

Deoxyribonucleic acid, abbreviated as *DNA* [26, 65], is a macromolecule that contains the genetic information of living organisms. It consists of four letters $\{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{T}\}$ or *nucleotides*. These four letters form very large strings.

In the last few decades, new *DNA sequencing* techniques have been developed to read these strings efficiently. These techniques typically output one or more long strings in plain text format. By analysis of these strings, differences between species and even individuals can be detected. Even without knowing what the differences are, we can make phylogenetic trees based upon substrings of a genome.

Eventually, certain aspects of an individual (parts of the *phenotype*) can be extracted from the DNA. It is not unthinkable that in the near future, forensic experts can determine hair and eye colour based upon DNA fragments found at a crime scene (this is already possible to some extent). At present, it is already possible to determine from which population group a (potentially highly damaged) fragment of DNA comes from based upon *Single Nucleotide Polymorphisms* [55, 73] or SNPs. These are locations in a genome where one letter may vary from person to person. If the distribution of each of these positions is known for all population groups, and if enough fragments containing SNPs can be found in a DNA sample, determining the origin of such a sample has become a matter of statistics.

Small unique substrings within a genome are also used for numerous purposes. They can be used as markers [23] for genes for example; if the marker is present, then the gene is present. An other practical application is for the isolation of certain parts of DNA. We find two unique substrings on both sides of the part that has to be isolated and by using a technique called *Polymerase Chain Reaction* [17] or PCR we can duplicate the isolated part.

In the later chapters we mainly focus on unique substrings and their use, such as the construction of phylogenetic trees, and a way to select DNA makers.

1.3 Metrics

In both the Data Mining part as well as the DNA part of this thesis, we shall use a new metric, designed for *multisets.* This metric is a highly configurable one. It requires a function that can be chosen by a domain expert. This function should reflect the difference between the number of occurrences of the same object within two multisets. For example, the difference between a person who steals zero bikes and someone who steals one bike is arguably larger than the difference between a person who steals 100 bikes and someone who steals 101 bikes. This difference must be given by the domain expert.

Although this metric was originally designed for criminal activities, using a different function makes it applicable in many other domains. We show that in the later chapters, where we use all substrings in two genomes (of different species). Here the same argument applies as described above.

1.4 Overview

The thesis is structured in three parts. In the first part of this thesis we focus on the application of Data Mining in the area of law enforcement, in particular the application of *particle systems* in this area. In the second part we shall investigate the metrics as mentioned in the previous section. The third part deals with DNA. In particular we shall show how the metrics can be applied in both the law enforcement field as well as in genomic research. We pay special attention to the visualisation of the results. Next, we discuss the contents of each chapter.

In Chapter 2, we give an extended overview of the *Particle model* and its capabilities. It is explained how the internals work. Several output surfaces and their merits are discussed (one of them is described in detail in Chapter 3). The Particle model iterates over all pairs of points and pushes two points apart if they are too close and pulls them together if they are too far away. This model allows for several distance functions; both metric and non-metric functions are discussed.

We also give an in depth description of the *push and pull* forces that can be used and their expected influence on the output. Furthermore, the meaning of the axes in the output figure (such as the one in Figure 1.1) has always been poorly understood. We try to make this more insightful. Finally, we compare this technique with several other dimension reduction methods.

Chapter 3 introduces a visualisation algorithm that, given a set of points in high-dimensional space, will produce an image projected on a 2-dimensional torus. The algorithm is a push and pull oriented technique which uses a sigmoid-like function to correct the pairwise distances.

We describe how to make use of the torus property and show that using a torus is a generalization of employing the standard closed unit square. Experiments (of which a sample is shown in Figure 1.2) show the merits of the method.

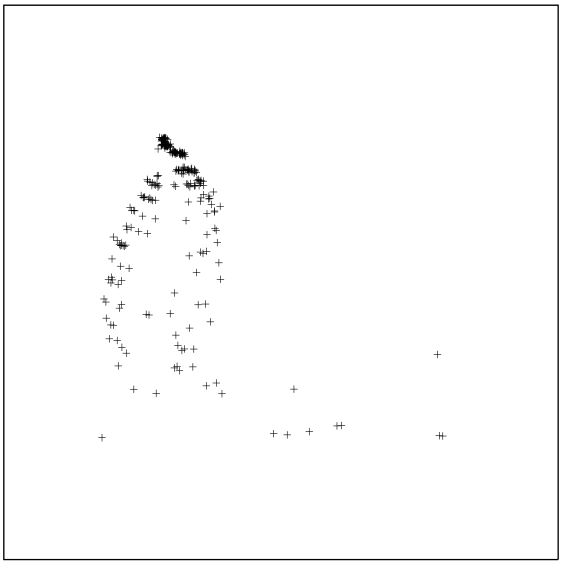

Figure 1.1: Projection of criminal careers using the Particle model; every point represents a single career

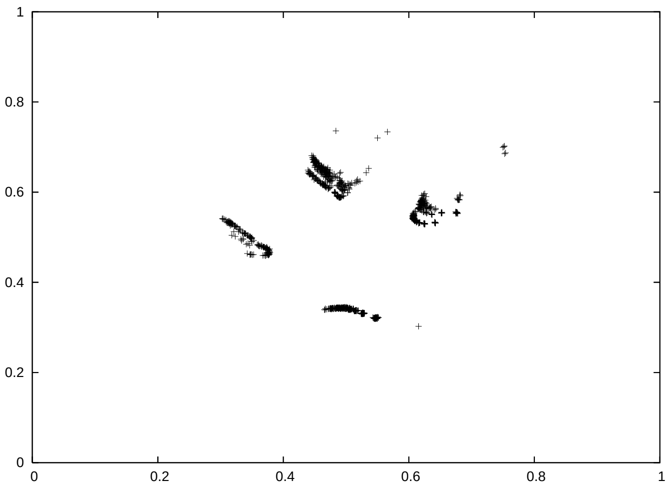

Figure 1.2: Visualising criminal careers on a torus; also a different metric is used

Chapter 4 focuses on a new method of the analysis of the errors introduced by multidimensional scaling techniques.

The error of an item in the output of these methods is associated with a charge, which is then interpolated to define a field, as seen in Figure 1.3. We give a general method on how to define this field, give several fine tuning

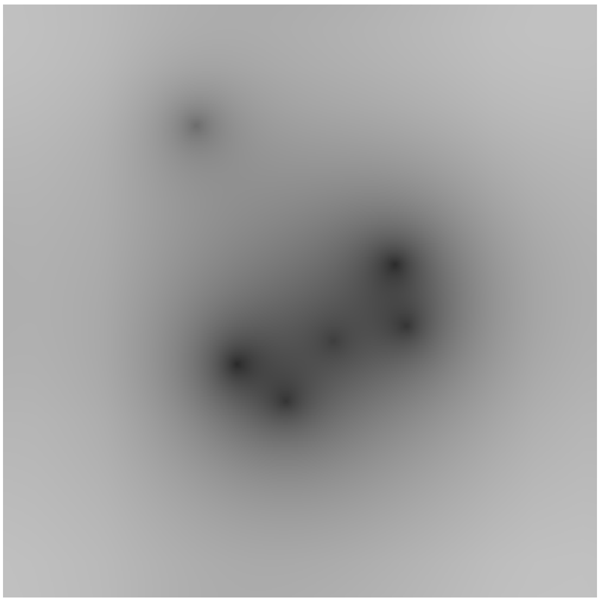

Figure 1.3: Errors visualised by interpreting them as a charge

techniques to highlight different aspects of the error and provide some examples to illustrate the usability of this technique.

In Chapter 5, we give an application of the usage of the *edit distance* between criminal careers to find criminals with a similar history. Also, an attempt is discussed to use these *neighbouring careers* to make a prediction about the future activities of criminals.

In Chapter 6, a new class of distance measures (metrics) designed for multisets is proposed. These distance measures are parametrised by a function f which, given a few simple restrictions, will always produce a valid metric. This flexibility allows these measures to be tailored for many domain-specific applications. We apply the metrics in bio-informatics (genomics), criminal behaviour clustering and text mining. The metric we propose also is a generalization of some known measures, e.g., the Jaccard distance and the Canberra distance. We discuss several options, and compare the behaviour of different instances.

The concept of multiset sequences is common in a number of different application domains. In Chapter 7 we introduce a new metric for the similarity between these sequences. Various types of alignment are used to find the shortest distance between two sequences. This distance is based on the well-defined distance measure for multisets from Chapter 6. Employing this, a pairwise distance can be defined for two sequences. Apart from the pairwise distances, the occurrence of holes (for timestamped sequences) can also be used in determining similarity; several options are explored. Applications of this metric to the analysis of criminal careers and access logs are reviewed.

Chapter 8 focuses on finding short (dis)similar substrings in a long string over a fixed finite alphabet, in this case a genome. This computationally intensive task has many biological applications. We first describe an algorithm to detect substrings that have edit distance to a fixed substring at most equal to a given e.

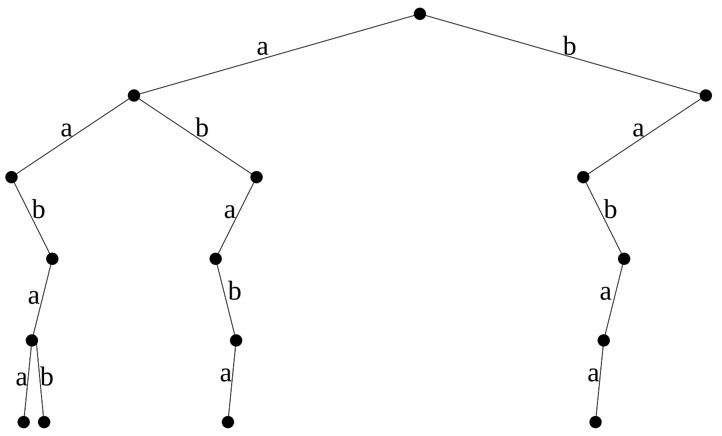

Figure 1.4: Trie of unique strings of length 5 originating from the string "abaababaabaaba"

We then propose an algorithm that finds the set of all substrings that have edit distance larger than e to all others by using a trie, as seen in Figure 1.4. Several applications are given, where attention is paid to practical biological issues such as hairpins and `GC` percentage. An experiment shows the potential of the methods.

In Chapter 9, we introduce a new way of determining the difference between full genomes, based upon the occurrence of small substrings in both genomes.

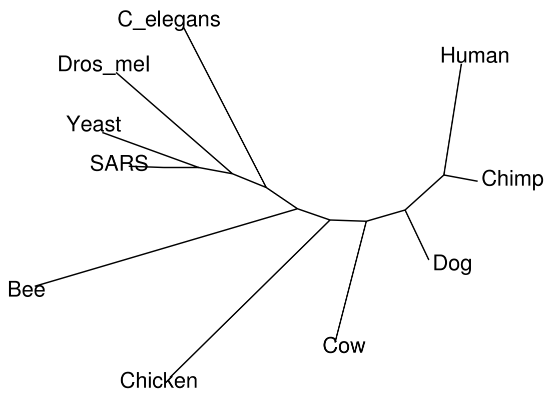

Figure 1.5: Phylogenetic tree based upon rare substrings

Basically we count the number of occurrences of all substrings of a certain length and use that to determine to what extent two genomes are alike. Based on these numbers several difference measures can be defined, e.g., a Euclidean distance in the vector space that has the same dimension as the number of possible substrings of a certain length, a multiset distance, or other measures. Each of these measures can be applied for phylogenetic tree generation, as shown in Figure 1.5. We also pay attention to some other visualisations and several statistics.

In Chapter 10 we propose a novel visualisation method for DNA and other long sequences over a small alphabet, which is based on the construction of the family of Rauzy fractals for infinite words. We use this technique to find repeating structures of widely varying length in the input string as well as the identification of coding segments. An example output of this visualisation technique is shown in Figure 1.6.

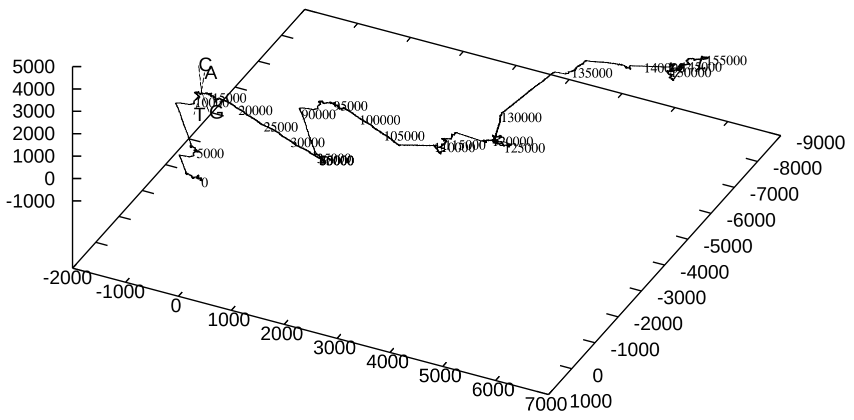

Figure 1.6: The first 160,000 nucleotides of the human Y-chromosome

Other properties of the input can also come to light using this technique.

1.5 List of publications

Next we give an overview of publications on which this thesis is based.

Chapter 3: Visualisation on a Closed Surface
This chapter is based on a paper published in the proceedings of the 19th Belgium-Netherlands Conference on Artificial Intelligence (BNAIC 2007) [42].

Chapter 6: Metrics for Mining Multisets
This chapter is based on a paper published in the proceedings of the Twenty-seventh SGAI International Conference on Innovative Techniques and Applications of Artificial Intelligence (AI-2007) [40]. A two page overview is also published in the proceedings of the 20th Belgium-Netherlands Conference on Artificial Intelligence (BNAIC 2008) [41].

Chapter 8: Selection of DNA Markers

This chapter is based on a paper published in the IEEE journal Transactions on Systems, Man, and Cybernetics, Part C [32].

Chapter 9: Substring Differences in Genomes
This chapter is based on a paper of which a one page overview is published in the proceedings of the Benelux Bioinformatics Conference (BBC 2008).

Chapter 10: Visualising Genomes in 3D using Rauzy Projections
This chapter is based on a paper which is presented in the 1st International ISoLA Workshop on Modeling, Analyzing, Discovering Complex Biological Structures, which was held on 4–5 June of 2009 in Potsdam, Germany.

The following publications on related subjects were co-authored during the PhD thesis:

Tri-allelic SNP markers enable analysis of mixed and degraded DNA samples
This paper is published in the Elsevier journal Forensic Science International: Genetics [73].

Onto Clustering of Criminal Careers
This paper is published in the proceedings of the Workshop on Practical Data Mining: Applications, Experiences and Challenges (ECML/PKDD-2006) [10].

Data Mining Approaches to Criminal Career Analysis
This paper is published in the proceedings of the Sixth IEEE International Conference on Data Mining (ICDM 2006) [9].

Temporal extrapolation within a static clustering
This paper is published in Foundations of Intelligent Systems, proceedings of ISMIS 2008 [14]. A two page overview is also published in the proceedings of the 20th Belgium-Netherlands Conference on Artificial Intelligence (BNAIC 2008) [15].

An Early Warning System for the Prediction of Criminal Careers
This paper is published in the proceedings of the 7th Mexican International Conference on Artificial Intelligence (MICAI 2008) [68].

Enhancing the Automated Analysis of Criminal Careers
This paper is published in the proceedings of SIAM Workshop on Link Analysis, Counterterrorism, and Security 2008 (LACTS2008) [13].

Part I

The Push and Pull Model with applications to criminal career analysis

Chapter 2

Randomised Non-Linear Dimension Reduction

The field of dimension reduction has provided a set of algorithms that can be used for the visualisation of high dimensional data. In this set, some well-known instances have been studied and used to a great extent, while others have not. In this chapter, we discuss the properties of the push and pull [43] model, also known as the particle or spring model. We first analyse the basic properties of the algorithm and discuss many variants and applications. A number of natural extensions and induced models are given as well.

2.1 Introduction

The general algorithm for 2-dimensional visualisation tries to solve the following problem: We have a pairwise distance matrix as input and as output we desire a 2-dimensional picture that represents the distances in the input matrix as good as possible. In general the data in the input matrix is derived from a high dimensional input space and can not be embedded perfectly in a 2-dimensional space.

The algorithm operates by placing points (or particles) randomly on a surface, the number of points corresponding with the number of rows (and columns) of the input matrix. Now the algorithm iterates in some way over (all) pairs of points, somewhat adjusting the position of the points in question according to the distances defined in the input matrix.

The algorithm can be terminated when the points no longer move, or when sufficiently many iterations have been done. The termination criteria are specified by the user.

2.2 The surface

Since this model is searching for an optimal arrangement of particles on a given surface, we can ask ourselves if we can use different surfaces to improve the dimension reduction. Normally we use a unit square to visualise the input data, but other choices are also available. We can look at closed or even semi-closed surfaces, for example, surfaces that have two or more (simply) identified boundaries. Also infinite surfaces are a possibility [8].

We could for example identify two of the sides of a unit square to obtain a (topological) cylinder. The push and pull algorithm will work exactly the same on an object like this, the only thing that needs to be adjusted is the distance function; on a cylinder there is a maximum distance in one direction. Notice that we only make a topological 2-dimensional cylinder. There is no curvature of the space at all.

Identifying more sides of the unit square will result in a fully closed surface like a globe, torus, Klein bottle or real projective plane. Again, the distance function must be adjusted for each of these surfaces (since our picture will still be a square) but the idea stays the same. Of these closed surfaces, a torus is perhaps the most natural choice to make. Again, notice that we use a topological 2-dimensional torus, again there is no curvature as would be the case with a 3-dimensional torus.

Using a closed surface has the advantage that an embedding of non-flat data is sometimes possible. We have more freedom to place our points since we can implicitly move in a third dimension. For example, we can perfectly embed points taken from the surface of a cylinder on a 2-dimensional torus, but not on the unit square. Conversely, we can embed the unit square on a 2-dimensional torus. Therefore a closed surface is a better choice than a normal unit square, although it must be noted that the end-user might find it confusing in the beginning.

2.3 Metric algorithms

Perhaps the best way to describe this model it to view the points as particles that have two types of forces working on them, an attracting and a repulsing one. These particles are bound to a surface with (normally) two dimensions. This loose description leaves a lot of freedom. We can use different types of forces. For example, the forces do not even have to be symmetrical. An other choice can be the surface, that does not have to be a unit square, but can also be a closed surface (see Chapter 3) like a torus.

Now we shall give the metric variant of the push and pull algorithm. It is split into two parts: a part that adjusts the positions of the particles and a part that iterates over a sequence of tuples of points and calls the adjustment function in each iteration.

In Figure 2.1 we see how the adjustment of the points works. Assume that $\tilde{p}$ and $\tilde{q}$ are too far away from each other. The algorithm below describes how two

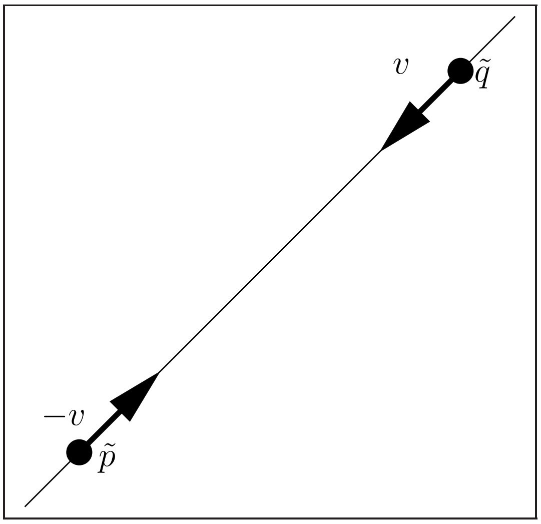

Figure 2.1: Metric correction

vectors v and $-v$ are calculated, over which the points are translated toward each other.

```
$PushPull\_Metric\ (p, q)$ ::
    **var** $correction$;
    **var** $v$;
    $correction = \alpha \cdot f(d_{\text{realised}}(\tilde{p}, \tilde{q}), inflation \cdot d_{\text{desired}}(p, q))$;
    $v = correction \cdot (\tilde{q} - \tilde{p})$;
    $\tilde{p} \leftarrow \tilde{p} - v$;
    $\tilde{q} \leftarrow \tilde{q} + v$;
```

Adjust $\tilde{p}$ and $\tilde{q}$.

In the algorithm above, p and q are input points. By $\tilde{p}$ we denote the coordinates of a point p in the target surface; we refer to $\tilde{p}$ as the *realisation* of p. The function $d_{\text{desired}}(p, q)$ is the distance between p and q as given in the input matrix, or perhaps by some other means. The function f returns a value between -1.0 and 1.0; if the realised distance is larger than the desired distance, the function will in general return a negative value, indicating that the points must be pulled together. The choices for this function are discussed in Section 2.3.1.

The value $d_{\text{realised}}(\tilde{p}, \tilde{q})$ is the distance between points $\tilde{p}$ and $\tilde{q}$ in the target surface. Usually, the Euclidean distance is employed for this. The global parameter α is the learning rate, which may decrease over time, or may be altered by the user. This parameter is discussed in Section 2.6. The global parameter *inflation* is used to utilize the entire output space. It is often set to 1.0 and is discussed in detail in Section 2.3. Note that if $\tilde{p} = \tilde{q}$, we can not determine a direction for the vector v, only the magnitude. To overcome this shortcoming, we can perhaps introduce a small distortion in the position of $\tilde{p}$ or $\tilde{q}$. Fortunately, in practice the algorithm has more than two input points, and other points will disturb the positions of $\tilde{p}$ and $\tilde{q}$, so in practice we can ignore this shortcoming.

```
MetricMainLoop ( ) ::
    while NotDone do
        choose some sequence $u = (u_1, u_2, \ldots, u_n)$
            of tuples of points;
        for $i \leftarrow 1$ to $n$ do
            PushPull_Metric($u_i$);
```

Iterate over tuples of points.

The main loop iterates over some sequence of tuples of n points. In general we iterate over all combinations of tuples exactly one time in each iteration. The order of these tuples is preferred to be random.

2.3.1 Forces

In this section we shall focus on functions that have input values between 0 and $\frac{1}{2}$ and have output values between -1 and 1 on this interval. The maximum value of $\frac{1}{2}$ is chosen because it is the maximum distance between two points on a torus. Functions for the normal bounded unit square will have input values between 0 and 1.

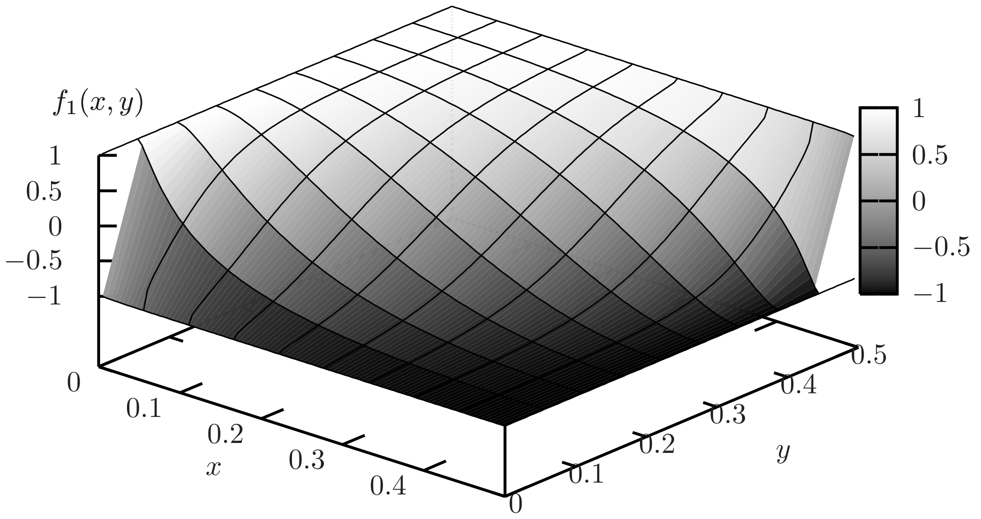

Figure 2.2: Correction function $f_1(x, y)$

The forces used for the alteration of the particles are usually symmetrical and can be described by a correction function that yields a positive value when two particles need to be placed closer together, and a negative value when they need to be pushed apart. There are many choices for such a function, a linear one being the most widely used, but other, mostly sigmoid like functions, can perform better since these functions are "aggressive" when a particle is not in the right position.

In Figure 2.2 we see such a function. The precise definition of this function is:

$$f_1(x,y) = \begin{cases} \cos(\pi \log_t(2x(t-1)+1)) & \text{if } \ y \neq \frac{1}{4} \\ \cos(\pi 2x) & \text{if } \ y = \frac{1}{4} \end{cases}$$

where $t = (1 - 1/(2y))^2$, $0 \leq x \leq \frac{1}{2}$, $0 < y < \frac{1}{2}$. So this function will be a deformed cosine for any fixed y, and will be 0 if x equals y. Furthermore, it is 1 whenever x equals 0 and -1 if x equals $\frac{1}{2}$.

Note that $f_1(\frac{1}{4}+x, \frac{1}{4}+y) = -f_1(\frac{1}{4}-x, \frac{1}{4}-y)$. This property is indeed quite desirable, since ... We will refer to it as the *symmetry property*.

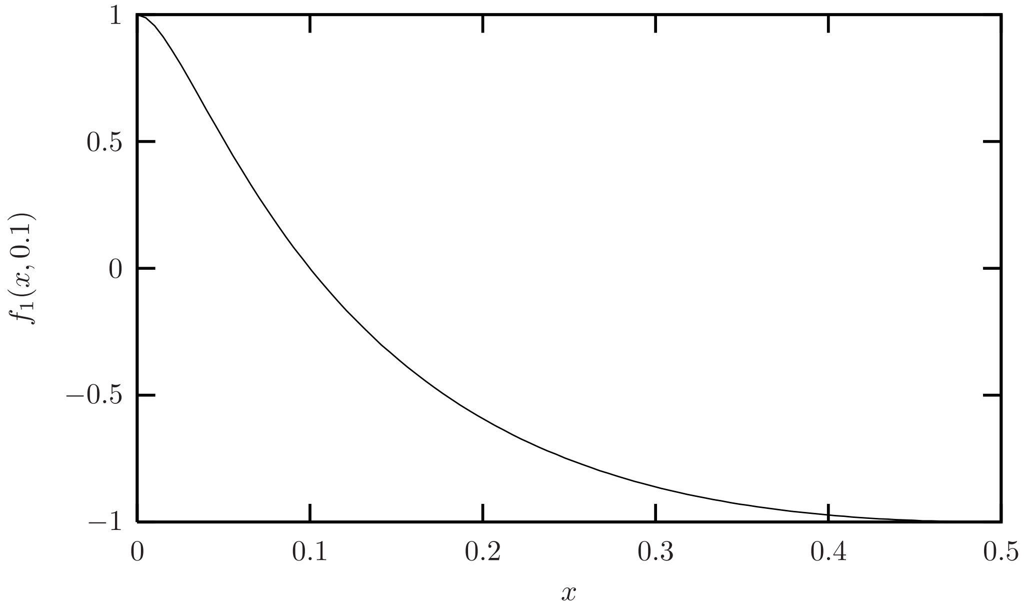

Figure 2.3: Correction function $f_1(x, 0.1)$

In the situation from Figure 2.3, the desired distance between two particles is 0.1. If the realised distance is larger than this, the function gets negative very quickly. On the other hand, the function will assume a positive value significantly larger than 0 when the realised distance is only slightly smaller than the desired distance.

For practical purposes, we can use any function that resembles the one given above, such as

$$f_2(x,y) = \cos(\pi(2x)^{\tan(\pi y)})$$

In order to get the symmetry property, one is lead to:

$$f_3(x,y) = \begin{cases} \cos(\pi(2x)^{4y}) & \text{if } 0 \le y \le \frac{1}{4} \\ -\cos(\pi(1-2x)^{4(\frac{1}{2}-y)}) & \text{if } \frac{1}{4} \le y \le \frac{1}{2} \end{cases}$$

In practice, this will be the kind of functions we will be using.

Another choice of function could be a function with a plateau, one that is zero on and near the desired distance.

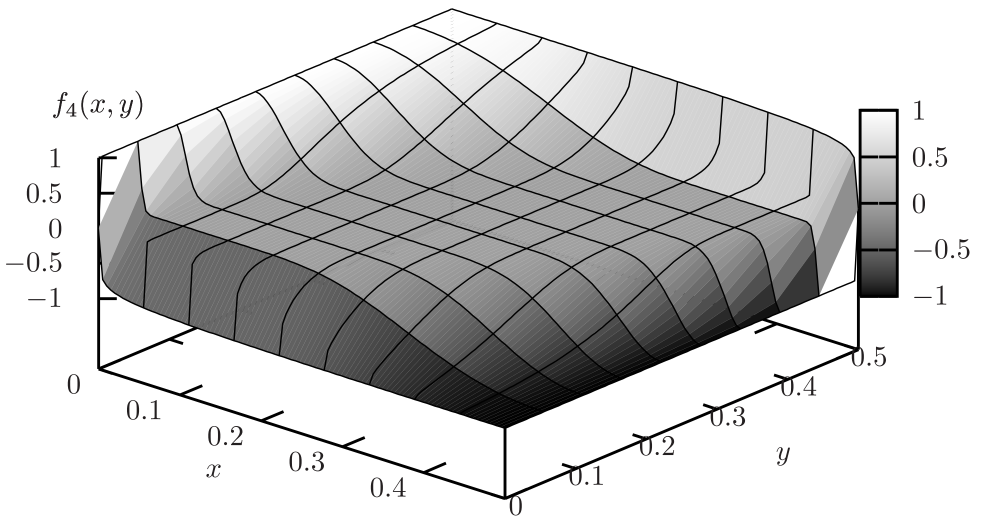

Figure 2.4: Correction function $f_4(x,y)$ with a plateau

In Figure 2.4 we see such a function, of which the exact definition is:

$$f_4(x,y) = \begin{cases} \cos^{25}(\pi \log_t(2x(t-1)+1)) & \text{if } y \neq \frac{1}{4} \\ \cos^{25}(\pi 2x) & \text{if } y = \frac{1}{4} \end{cases}$$

where $t = (1 - 1/(2y))^2$, $0 \le x \le \frac{1}{2}$, $0 < y < \frac{1}{2}$. Again, the symmetry property holds and in this particular case, if x roughly equals y, the function will not return a very large correction.

Also, reports have been made of even more complicated functions [12], each giving different results and emphasising different aspects of the data, or different aims, varying from creating a global picture to the desire to speed up the algorithm.

In Figure 2.5 we see an instance of a hybrid function, as used in a previous study [12]:

$$f_5(x,y) = \begin{cases} \cos^{3}(\pi \log_t(2x(t-1)+1)) & \text{if } y \neq \frac{1}{4} \\ \cos^{3}(\pi 2x) & \text{if } y = \frac{1}{4} \end{cases}$$

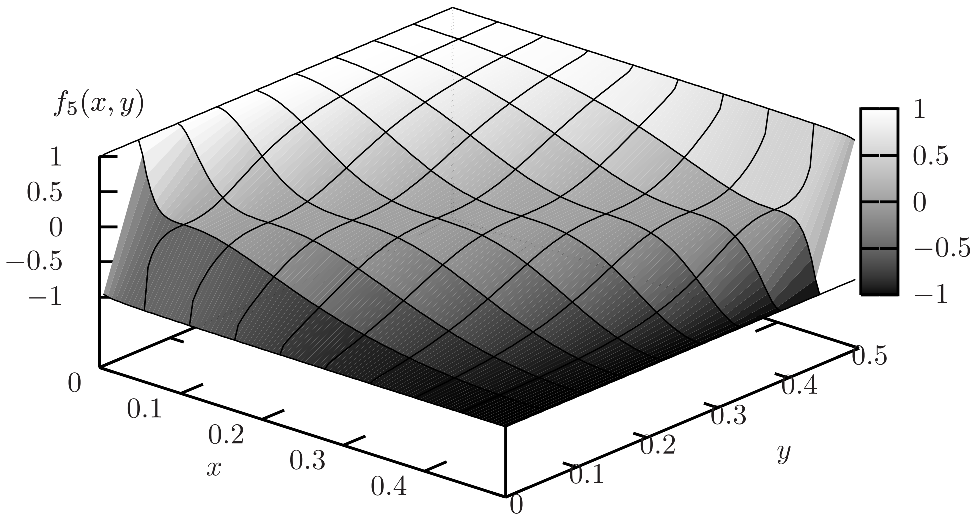

Figure 2.5: Hybrid correction function $f_5(x, y)$

where $t = (1 - 1/(2y))^2$, $0 \leq x \leq \frac{1}{2}$, $0 < y < \frac{1}{2}$. Although in the original paper a slightly different function is used, the shape is similar.

In general there is no clear way to determine which function to use. A function with a plateau will give a more global picture, since the exact distances are less important than the global placement and a sigmoid like function leads to fast convergence. It largely depends on the desired end result which function is preferred.

As mentioned before, we can also use a non-symmetrical function to describe the forces working on the particles, giving rise to a whole different set of images. Such a function would be a combination of a push force and a pull force, where the two are not each others inverse. We could for example use a logarithmic function for pushing and a linear function for pulling.

A useful example would be on a torus where we use a push function that is positive (but declining) everywhere and *no* pull function. At first glance this would seem not to work at all, but since we are working on a torus, we still get a valid dimension reduction, because the push force wraps around the torus (it is positive everywhere). So two points that are supposed to be far from each other will be pushed harder than points that are supposed to be near to each other. This results in a model where the points that are the farthest from each other have a dominance over other points and will force them into the correct position.

Using this model has the disadvantage that it is slow in comparison to the model with both forces. One big advantage, though, is that there is no need for an inflation parameter, the inflation is done automatically. For this reason, the push-only variant on a closed surface is an interesting model that should be investigated further.

The model allows for the online adding and removing of points [14], since the algorithm can be resumed at any point. In general adding or removing one point will not result in big global differences, only local ones, so on resuming the algorithm one will quickly find a new optimum.

A consequence of this observation is that the properties of the points can be altered online to make a simulation of flowing particles, or that a single item can be traced.

A drawback of this model is that the algorithm can get stuck in a local optimum. This is almost always the case with randomised algorithms that do local optimisations. In practical situations, however, this seems to happen rarely.

Another drawback is that the result of two runs of the algorithm will produce different pictures. In most cases, this is the result of a rotation or mirroring, which can be countered by adding three reference points to the data that the algorithm will not alter. If the difference is the result of the fact that the algorithm has found a different local optimum however, using reference points will not be of any use.

The main advantage of the algorithm is that it is fast and very flexible. If, for example, no (global) optimum can be found, the parameters can be changed online to better suit the data. We can also change these parameters to get out of a local optimum or to improve the embedding by multiplying the distances by a factor to make use of a closed surface.

An other point that is worth stressing is that we only need pairwise distances to generate the dimension reduction. The original coordinates need not to be known.

Furthermore, this form of dimension reduction is non-linear, as we shall see in the next section.

2.4 Axes

Many questions arise about the meaning of the axes when using this technique. The meaning is hard to understand: in general we can only say that it is a non-linear combination of the (maybe even unknown) input dimensions, which are already “warped” by the metric used to derive the distance matrix.

To illustrate the non-linearity of the algorithm we can take some points uniformly distributed on a sphere and try to embed them onto a 2-dimensional surface. A linear dimension reduction technique will produce a picture where two halves of the sphere will be projected upon each other. The push-pull technique however will generate a different picture.

In many cases there can be a correlation between one of the input dimensions and a direction in the resulting picture. For example, in the following picture we see the clustering of criminal “careers”, where time is an implicit dimension in our input data. The data stems from the Dutch national police and because of the sensitive nature can not be disclosed. The input of the push and pull algorithm is a distance matrix, obtained by calculating the edit distance between two criminal careers. The careers themselves are defined as a string of *multisets*,

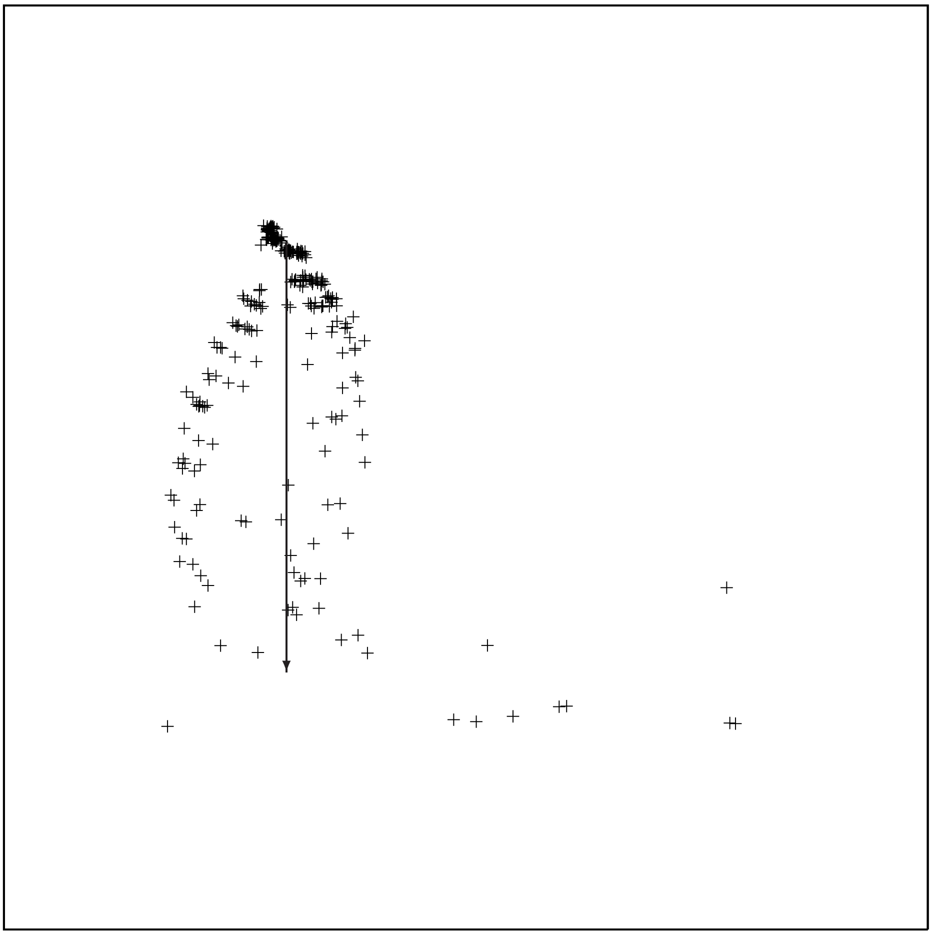

Figure 2.6: Criminal careers

where each multiset corresponds to the nature and frequency of crimes in one year.

With a metric for multisets (see Chapter 6) we can calculate a pairwise distance between two multisets and with the standard alignment algorithms [54, 66] we calculate the pairwise distance between the criminal careers.

In the resulting picture (seen in Figure 2.6) there is a correlation between time and the direction of the arrow. At the base of the arrow there is a cluster of short careers and as we proceed to the head of the arrow the length of the careers increase. Note that the same direction may correspond to a whole different combination of input dimensions at an other position in the picture.

So in general we can say that a direction has a meaning in the pictures, but that this meaning is not uniform in the whole picture. Only locally we can say something about the directions, globally this structure can be complex.

In *Principal Component Analysis* (PCA) [35,57], on the axes we always have a *linear* combination of the dimensions of the input space. For *Principal Curves and Surfaces* (PCS) [18, 29] and *Self Organising Maps* (SOM) [39] the data is projected onto a lower dimensional manifold, which does not need to be linear. It does, however, need to be continuous (smooth). In the push-pull algorithm this is not the case in general.

Both SOMs and PCA/PCS need the input points to operate. The push-pull algorithm only requires the pairwise distances, like most Multi Dimensional Scaling (MDS) [16] techniques. Unlike SOMs and PCA/PCS no parametrisation of the manifold is given as part of the output.

2.5 The non-metric variant

Since the metric used for the calculation of pairwise distances is a parameter in the dimension reduction, we can also use the topological ordering of distances to derive a projection onto a low-dimensional surface. The idea behind this is to make an embedding in such a way that the relative order of distances is preserved, but in general not the distances themselves. This means that the objective is a picture where the distance between two points is smaller than the distance between two other points if and only if this is also true in the input data.

Next we discuss the non-metric variant of the algorithms that do the dimension reduction. This variant is also split into two parts; one part is responsible for the adjustment of two tuples of particles, and the other part iterates over a sequence of tuples of tuples and calls the adjustment function in each iteration.

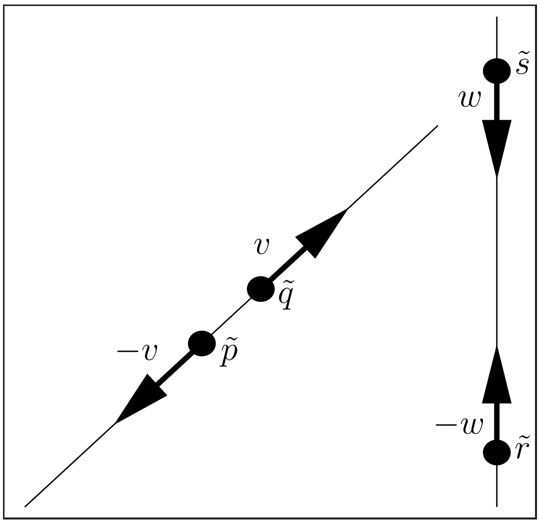

Figure 2.7: Non-metric correction

In Figure 2.7 we see how the adjustment of the points works. Assume that

$\tilde{p}$ and $\tilde{q}$ are too close to each other with respect to $\tilde{r}$ and $\tilde{s}$. The algorithm below describes how four vectors v, $-v$, w and $-w$ are calculated, over which the points are translated respectively.

```
PushPull_NonMetric $((p,q),(r,s))$ ::
    var $correction \leftarrow 0.0$;
    var $v, w$;
    if $d_{\text{realised}}(\tilde{p},\tilde{q}) < d_{\text{realised}}(\tilde{r},\tilde{s})$ and $d_{\text{desired}}(\tilde{p},\tilde{q}) > d_{\text{desired}}(\tilde{r},\tilde{s})$ then
        $correction \leftarrow \alpha \cdot \epsilon$;
    if $d_{\text{realised}}(\tilde{p},\tilde{q}) > d_{\text{realised}}(\tilde{r},\tilde{s})$ and $d_{\text{desired}}(\tilde{p},\tilde{q}) < d_{\text{desired}}(\tilde{r},\tilde{s})$ then
        $correction \leftarrow -\alpha \cdot \epsilon$;
    $v \leftarrow correction \cdot (\tilde{q} - \tilde{p})$;
    $w \leftarrow -correction \cdot (\tilde{s} - \tilde{r})$;
    $\tilde{p} \leftarrow \tilde{p} - v$;
    $\tilde{q} \leftarrow \tilde{q} + v$;
    $\tilde{r} \leftarrow \tilde{r} - w$;
    $\tilde{s} \leftarrow \tilde{s} + w$;
```

Adjust $(\tilde{p},\tilde{q})$ and $(\tilde{r},\tilde{s})$.

The non-metric variant of the algorithm adjusts four points each iteration by a small amount $\alpha\ \epsilon$, if the realised distance of one tuple is smaller than the other one, but the desired distance is larger, the points in the first tuple are pushed apart and the ones in the second tuple are pulled togeter. This variant is covered in Section 2.5.

```
NonMetricMainLoop ( ) ::
    while NotDone do
        choose some sequence $u = (u_1, u_2, \ldots, u_n)$
            of tuples of tuples of points;
        for $i \leftarrow 1$ to $n$ do
            PushPull_NonMetric$(u_i)$;
```

Iterate over tuples of tuples of points.

The main loop iterates over some sequence of tuples of tuples of points. In general we iterate over all combinations of two tuples exactly one time in each iteration. The order of these tuples is preferred to be random.

The correction function f used to alter the realisation of the points is highly configurable. If f is very small for all input values, the algorithm will reach a stable state if one exists with high probability. The only provision is that the total amount of distances is preserved. The reason that this will lead to a good embedding in general is because distances can be divided infinitely many times (in principle), so a valid ordering will nearly always be found if f is small enough for all input values.

This, however, is not very practical when the objective is a fast algorithm. To increase the effectiveness of f, several methods can be used like letting f depend on the number of input points, or on the amount of iterations (see Section 2.6). Another interesting method would be to let f depend upon the relative amount of points that already have a good position; this idea is discussed further in Section 2.6.

Since only relative distances have meaning in a non-metric dimension reduction, the size of a picture is irrelevant for correctness. However, for insight we want to have it as large as the space permits. If we use a closed or semi-closed surface, this is even more preferable, because then the properties of that particular space can be exploited. In the first case, we can do the dimension reduction and afterwards inflate the picture (zoom in) to make use of the entire space. In the second case however, we need to have a kind of entropy law to make the points want to float away from each other. We can not simply use a zoom function, since some points will be put closer together because of the nature of the torus; it has a maximum distance.

Apart from an inflation function, we want the diversity of distances to be as large as possible. Since the dimension reduction is non-metric there is some freedom in general. There are several ways to utilize this freedom in order to make the picture more insightful. First we can use the freedom to make the diversity of distances as large as possible without compromising the topological order. This will in general emphasize the difference in distances between two pairs of pairs of points. Another way to utilise this freedom is to make the distances resemble the real distances without compromising the topological order. This will result in a dimension reduction that is non-metric, but tries to be “as metric as possible”.

2.6 Simulated annealing

In both the metric and non-metric variant, simulated annealing [61] can be used to speed up the algorithm and to force convergence. The general idea of simulated annealing is to use large alterations at the beginning of the algorithm and small ones near the end. In this particular case, the strength of the correction function can be subject to such a technique.

2.7 Comparison with other methods

In this sections we compare with PCA, PCS, SOM and MDS.

Principal component analysis (PCA) is a linear technique that requires the input points. As output a (hyper) plane is given with the projected points on it. This is a deterministic algorithm, and thus always produces the same image when given identical input data. The fact that it is linear results in a linear combination of the input dimensions on the axes of the output picture. This might not give the best result though.

Principal Curves and Surfaces (PCS) is a non-linear technique that requires the input points. As part of the output, a parametrised manifold is given, onto which the points are projected. Although this dimension reduction technique is non-linear, the manifold is continuous (or smooth), which needs not be the case in the push-pull algorithm. PCS is, like PCA, also a deterministic technique.

In a **Self Organising Map** (SOM), a field of vectors is initialised randomly and then trained with input examples. The vector that looks most like the example, is altered in such a way that it looks even more like the example and further more, its neighbours are also altered, but to a lesser extent.

This results in a non-linear output manifold, similar to one used in PCS. The technique itself is non-deterministic though. Because of the non-determinism, the non-linear output manifold and the training component where mostly local changes are made, there is a strong relation with the push and pull model.

Push-pull has much resemblance with **Classical MDS**. First of all, they both are non-linear, only require the pairwise distances and minimise a stress function. A difference is that the stress function in Classical MDS is explicitly defined, where the stress function in Push-pull is not. The correction function can in a way be seen as part of the stress function and summation over the correction of all pairs of input points would result in a stress function. Like MDS, emphasis can be given to small distances (through the correction function). An other difference is that the correction is not a global, but a local one (hence the correction function as opposite to the stress function). MDS uses *gradient descent* to alter the position of the projected points, whereas push-pull makes local changes. The non-metric variant of MDS has a strong resemblance to non-metric push-pull for the same reasons.

Stochastic Neighbour Embedding [30] is a probabilistic dimension reduction technique where neighbourhood identities are preserved. The neighbours of each object in high-dimensional space are defined using probability techniques. A noticeable difference with other techniques is that not necessarily every high-dimensional object is associated with a single one in the low-dimensional space.

2.8 Conclusions and further research

In this chapter we have given an overview of the push and pull model. We have shown the flexibility of the model and we have given guidelines on how to interpret and use the parameters of this model.

Further studies is required for the push-only variant on a closed surface. The advantages are clear: the input values are automatically scaled in such a way that the output space is optimally used.

Chapter 3

Visualisation on a Closed Surface

In this chapter, we discuss a visualisation algorithm that, given a set of points in high-dimensional space, will produce an image projected on a 2-dimensional torus. The algorithm is a push and pull oriented technique which uses a sigmoid-like function to correct the pairwise distances. We describe how to make use of the torus property and show that using a torus is a generalization of employing the standard closed unit square. Experiments show the merits of the method.

3.1 Introduction

In many situations one wants to cluster and/or visualise data [67]. In this chapter we will describe a method to visualise a perhaps large set of data points on a 2-dimensional surface. This surface is basically the unit square U in $\mathbb{R}^2$, with sides identified in such a way that it topologically is a *torus*: left and right boundary are identified, and so are top and bottom boundary, see Figure 3.1 below. The resulting surface has no boundaries. As distance between two points a and b in U we just take the minimum of the ordinary Euclidean distance between a and any point from $\{b + (k, \ell) \mid k, \ell \in \{-1, 0, 1\}\}$. This surface will be referred to as "the" torus. Note that the distance is *not* the one that arises when a torus is embedded in $\mathbb{R}^3$ in the usual way (as a doughnut). In our case a visualisation as a unit square is more appropriate, remembering that left-hand and right-hand side are near to one another (and also top side and bottom side).

We start with a finite set of n data points $\{p_1, p_2, \ldots, p_n\}$. We use a given metric d to compute the distance $d_{ij} = d(p_i, p_j)$ between p_i and p_j $(i, j \in \{1, 2, \ldots, n\})$, which yields a symmetric matrix $D = (d_{ij})_{i,j=1}^n$. This matrix D will be the basis for our further actions. Its entries will be referred to as the *desired distances*. Our goal is to obtain points $\{p'_1, p'_2, \ldots, p'_n\}$ (the so-called *current points*) in U, in such a way that the distance between p'_i and p'_j in U (the *current distance*) resembles d_{ij}, the desired distance between p_i and p_j, as much

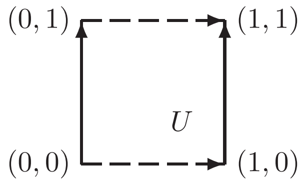

Figure 3.1: Unit square with sides identified: the torus.

as possible for $i, j \in \{1, 2, \ldots, n\}$. The difference between the current distances and the desired distance is therefore minimised. Together, the current points constitute the *current configuration*. Once this configuration is established, it can be used for all sorts of clustering purposes.

Our algorithm repeatedly takes two current points, and pushes them together or pulls them apart with a *correction factor*, depending on the relation between desired and current distance. We use an *inflation factor* and a *correction multiplier* to improve the current configuration. Note that the distances in U do not change when one rotates, mirrors or translates all points. Since our method makes use of random elements, visualisations might be the same under rotation, mirroring or translation, but it is also possible that they are actually different.

There are many methods that perform a dimension reduction. We mention Multi Dimensional Scaling (MDS, see [5,28]) and Principal Component Analysis (PCA, see [28]) as two well-known statistical methods. Other methods include several types of (competitive) neural networks, such as Kohonen's Self Organizing Maps (SOMs, see [28]) and vector quantization (again, see [28]). A comparison of all these methods is beyond the scope of this chapter (e.g., see [22]), we just mention two issues. First, our method is intuitive, very fast and requires no complicated mathematical operations, such as matrix inversion. Second, the use of the torus appears to be both natural and easy to describe; it also performs better than the previously used closed unit square (with boundary, cf. [43]), but still has all its merits. Notice that when using a 0.5×0.5 sub-square of U, one has this situation as a special case.

In Section 3.2 we sketch the background, and mention some alternative topologies. The method is described in Section 3.3. Section 3.4 has experiments, and we conclude in Section 3.5.

3.2 Background

In this section we mention some issues concerning our method. We will also point out a few difficulties that might arise, and some other possibilities.

As specified above, the surface we use is not the standard 2-dimensional unit square in the Euclidean space $\mathbb{R}^2$, but a 2-dimensional torus. The main advantage of using such a manifold is that there are more degrees of freedom in such a space.

A disadvantage of using a torus is that it is impossible to contract every circle to a point, and thus there are configurations possible where clusters are wrapped around the torus and thus might get stuck in a “local minimum”. A solution to this is to use a sphere (where each circle can be contracted to a point), but the projection of a globe onto a flat 2-dimensional space gives a distorted image (just like the map of the earth, the polar regions usually appear much larger than they actually are).

Another way to prevent the potential wrapping around the torus is to use a non-random initialization. If all points are initialized in one (small) area, the process will most likely not result in a configuration where wrapping is an issue. This can even be forced by placing a maximum distance (determined by the circumference of the torus) on the correction part of the algorithm.

There are more possibilities for such surfaces, like the non-orientable Klein bottle (obtained when identifying the dotted arrows from Figure 3.1 in opposite direction) or the real projective plane, but from all these, the metric on a torus (as specified above) is most like the standard Euclidean one, so it is natural to choose this object.

3.3 Algorithm

The algorithm we use is a *push and pull* oriented one, where the correction factor depends on the difference between the desired distance d_{desired} and the current distance d_{current}. This current distance, or rather its square, between two points $a = (x_1, y_1)$ and $b = (x_2, y_2)$ from U can be efficiently computed by:

```
$d_{\text{current}}((x_1, y_1), (x_2, y_2))$ ::
    **var** $x_3 \leftarrow x_2$;
    **var** $y_3 \leftarrow y_2$;
    **if** $x_1 - x_2 > 0.5$ **then** $x_3 \leftarrow x_3 + 1.0$;
    **if** $x_1 - x_2 < -0.5$ **then** $x_3 \leftarrow x_3 - 1.0$;
    **if** $y_1 - y_2 > 0.5$ **then** $y_3 \leftarrow y_3 + 1.0$;
    **if** $y_1 - y_2 < -0.5$ **then** $y_3 \leftarrow y_3 - 1.0$;
    **return** $(x_1 - x_3)^2 + (y_1 - y_3)^2$ ;
```

Quadratic distance between points in U

The point $b' = (x_3, y_3)$ is the (or more precisely, a) point from $\{b + (k, \ell) \mid k, \ell \in \{-1, 0, 1\}\}$ that realizes the shortest distance to a. This point will also be used later on. The maximal quadratic distance between any two points from U equals 0.5. (We will omit the word “quadratic” in the sequel.)

Instead of a linear or a constant function (of the current distance) to calculate the amount of correction, we can and will use a sigmoid-like function, or rather a family of functions. This function must adhere to some simple constraints, enumerated below. So we want a function $f = f_{d_{\text{desired}}}$ which is defined on $[0, 0.5]$, where 0.5 is the maximum distance between two points (on the torus). We must have, with $0 < d_{\text{desired}} < 0.5$ fixed:

- $f(0) = \rho$
- $f(0.5) = -\rho$
- $f(d_{\text{desired}}) = 0$

Here $\rho \in (0, 1]$ is the so-called *correction multiplier*. So when the current distance is as desired, f has value 0 — and so has the correction. The resulting correction factor *corrfac* equals $f(d_{\text{current}})$. If $d_{\text{desired}} = 0$, we make it slightly larger; similarly, if $d_{\text{desired}} = 0.5$, we make it slightly smaller.

We will use

$$f_{d_{\text{desired}}}(x) = \begin{cases} \rho\ \cos\left(\pi \log_t(2x(t-1)+1)\right) & \text{if } d_{\text{desired}} \neq 0.25 \\ \rho\ \cos\left(\pi 2x\right) & \text{if } d_{\text{desired}} = 0.25 \end{cases}$$

where $t = (1 - 1/(2d_{\text{desired}}))^2$; this function satisfies all the constraints. Figure 3.2 depicts $f_{0.1}$ and $f_{0.25}$, with $\rho = 1$.

The reason we choose a function like this, is because the correction of a point will be large when the error of that point is large. Only when the error is close to zero, the correction will be small. Other functions, like sigmoids will have the same behaviour.

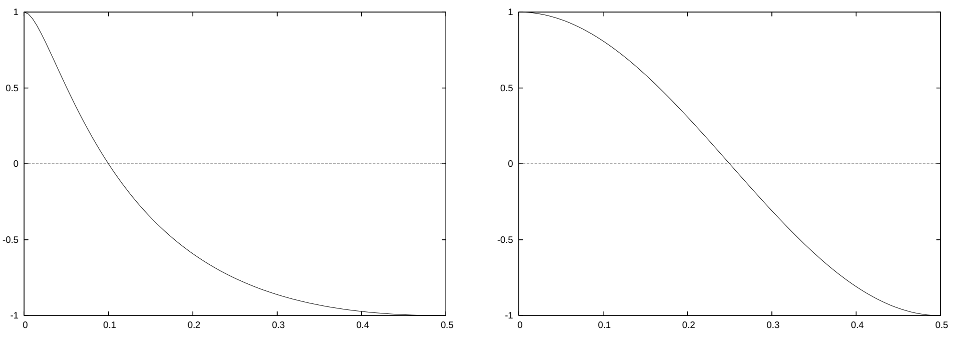

Figure 3.2: $f_{d_{\text{desired}}}$ with $d_{\text{desired}} = 0.1$ and $d_{\text{desired}} = 0.25$, $\rho = 1$.

Now suppose we want to "push and pull" two given points $a = (x_1, y_1)$ and $b = (x_2, y_2)$ in U; we first compute $b' = (x_3, y_3)$ as in the distance calculation of d_{current} above. Then the coordinates x_1 and y_1 of a are updated through

$$\begin{aligned} x_1 &\leftarrow x_1 + \textit{corrfac} \cdot |d_{\text{desired}} - d_{\text{current}}| \cdot (x_1 - x_3)\ /\ 2 \\ y_1 &\leftarrow y_1 + \textit{corrfac} \cdot |d_{\text{desired}} - d_{\text{current}}| \cdot (y_1 - y_3)\ /\ 2 \end{aligned} \tag{3.1}$$

A positive *corrfac* corresponds with pushing apart, a negative one with pulling together. In a similar way, the coordinates x_3 and y_3 of b are updated in parallel.

If a coordinate becomes smaller than 0, we add 1, and if it becomes larger than 1, we subtract 1. Together we will refer to this as Equation (3.1).

The basic structure of the algorithm is as follows:

```
initialize all current points in a small region of $U$
**while not** *Ready* **do**
    update all pairs (in arbitrary order) with Equation (3.1)
```

The push and pull algorithm

The algorithm terminates when the standard deviation and the mean error ($\sum_{|\text{pairs}|} |d_{\text{desired}} - d\text{current}|/|\text{pairs}|$) no longer change.

We now introduce the inflation factor σ, and secondly comment on the correction multiplier ρ.

The *inflation factor* $\sigma > 0$ can be used in the following way: Equation (3.1) is changed to

$$x_1 \leftarrow x_1 + \mathit{corrfac} \cdot |\sigma \cdot d_{\text{desired}} - d_{\text{current}}| \cdot (x_1 - x_3) \ / \ 2. \tag{3.2}$$

This can be useful in several ways. If, for example, all distances are between 0 and 0.2, one might argue that it is useful to multiply these distances by 2.5 to get a better spreading. This argument is especially valid if the resulting clustering cannot be realized in the plane, but can be embedded on a torus. Inflation with the right factor can make the overall error drop to zero in this case, while using the original distances will always result in a non-zero overall error.

Even if all distances are between 0 and 0.5, inflation or deflation can still be beneficial. For example, the input data can be such that inflation or deflation will result in the correct clustering of a large part of the input, while not using an inflation factor will result in a much higher overall error. An example of such input data would be a torus that is scaled between 0 and 0.2, with a few points outside this region. Normal clustering would result in a flat image where the points outside the torus region would have correct distances to the torus region, but with the correct inflation factor, the torus will be mapped on the entire space, and the few points outside the region will be misclustered. This results in a clustering where the overall error is small.

In practice we often take $\sigma = 1$.

The correction multiplier ρ is a parameter which controls the aggressiveness of the correction function. Initially this factor is set to 1, but for data that can not be embedded in the plane, lowering this factor can be beneficial.

If, for example, most of the distances are near the maximum, the correction function will push them so far apart, that they are pushed towards other points at he other side of the torus. This can result in the rapid fluctuation between two or more stable states. These states are probably not the global minimum for the clustering error, and therefore not the end result we desire. Increasing the correction multiplier will counter this effect.

3.4 Experiments

In this section we describe several experiments, both on synthetic and real data. The experiments are of an exploratory nature. We try to give a good impression of the merits of the algorithm.

We start with a synthetic dataset. In the left-hand picture of Figure 3.3 we see the original data points (on a "flat" 2D plane), from which a distance matrix is derived to serve as test data for the visualisation algorithm. In this picture we see seven spheres of which three are unique: the topmost two and the one in the center. The other four spheres are copies of one another. The total number of points is 700 and all distances are between 0.0 and 0.5.

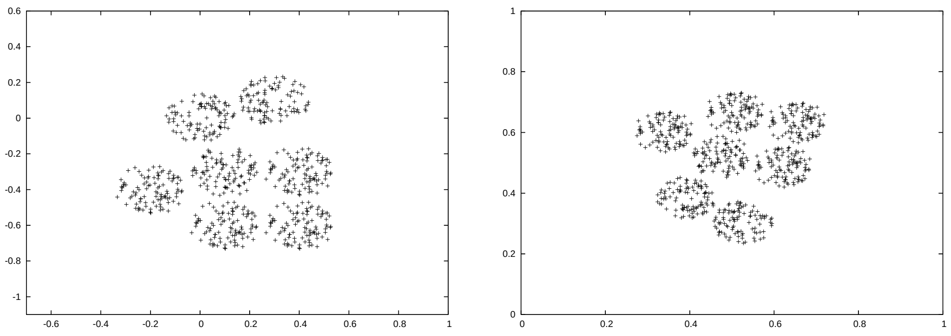

Figure 3.3: Original data (left) and visualisation (right).

After only a few iterations of our algorithm, the right-hand picture of Figure 3.3 appears. Notice how it resembles the input data, except for a mirroring and a rotation. All distances are preserved almost perfectly. Remember that only the pairwise distances were used by the algorithm. The mean error in this picture is 0.00004 and the standard deviation is 0.00003. As a final remark, "flat data" will always cluster within a sub-square of size 0.5×0.5.

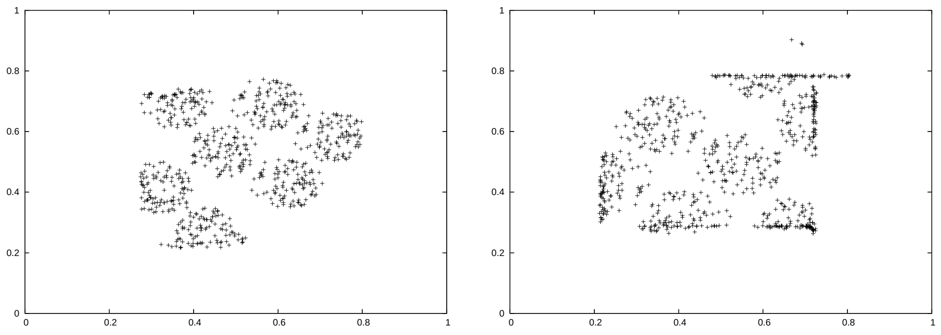

Figure 3.4: Visualisation of flat data with an invalid inflation factor.

In Figure 3.4 we see the same test data, except that the distances have been multiplied by a factor 1.5 in the left-hand picture, and by 2.6 in the right-hand one. This results in a non-correct embedding, since the maximum distance in this space is 0.5. The effects can be seen in Figure 3.4, in particular in the right-hand picture. In both pictures a translation has been applied in order to center most points. Though the full 1.0×1.0 square has been used, most current points reside in the smaller 0.5×0.5 square, as is clearly visible in the right-hand picture.

The top-left sphere is forced closer to the bottom-left one than is possible. This results in the flattening of the spheres at the outermost edges. This effect can be explained by considering the overall error (which is minimized). By a local deformation, the overall error is kept small. The effect can also be seen (to a lesser extent) in the middle-left sphere. Notice that the effect is absent in the top-central sphere because of the void at the bottom-center of the picture (there is nothing to collide with at the other side of the torus).

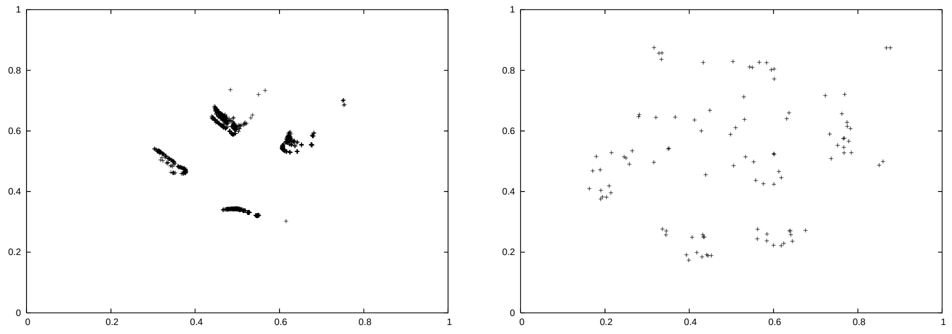

Figure 3.5: Visualisation of criminals, non flat case. Left: with categories; right: without categories.

In Figure 3.5, left, we see a visualisation of real data. We have taken a database of 1,000 criminal records supplied to us by the Dutch national police, and divided the crimes into three categories (light, middle, heavy): each record has three integers, describing the number of crimes in the respective categories. The distance measure we use is one defined on multisets and is described in Chapter 6. It basically averages the absolute differences between the numbers of crimes.

The resulting matrix cannot be embedded in the plane, but it almost could, since the mean error is relatively small 0.00494 and so is the standard deviation 0.00529. We refer to this type of situation as a "non flat case". An indication that the data is almost flat, is that the clustering stays within the 0.5×0.5 sub-square, and inflation increases the error. There are four main clusters in this picture, where:

- The leftmost one consists of criminals that have committed relatively light crimes. They all fall into the categories light and middle.
- The top one consists of all-rounders, they have all committed crimes in all

categories.

- The rightmost one consists of criminals that have only committed light and heavy crimes, nothing in between.
- The bottom one consists of criminals that have only committed light crimes, all of them fall into the category light.

Then there is a very small cluster in the top-right corner of the picture, this is a cluster of people who have only committed heavy crimes. This is apparently non-standard behaviour for a criminal. There are a few other isolated points in this picture, they all are people with a strange criminal record.

In Figure 3.5, right, we see the clustering of 100 criminals based upon the same distance measure as in Figure 3.5, but now we do *not* categorize the crimes; here the records have 80 attributes. The result is a scattered image (largely due to the lack of similarity), occupying a large part of the unit square, and only a few local clusters. We make use of inflation factor $\sigma = 2$ and correction multiplier $\rho = 1/16$ here, to produce the picture with a mean error of 0.02636 and a standard deviation of 0.01786. All visualisations are obtained within a few seconds.

Finally, we show an example from chemistry. The dataset we use, the so-called `4069.no_aro` dataset, contains 4,069 molecules (graphs); from this we extracted a lattice containing the 2,149 most frequent subgraphs. These are grouped into 298 structural related patterns occurring in the same molecules using methods presented in [25], resulting in a 298 by 298 distance matrix; the distance between graphs is based on the number of co-occurrences.

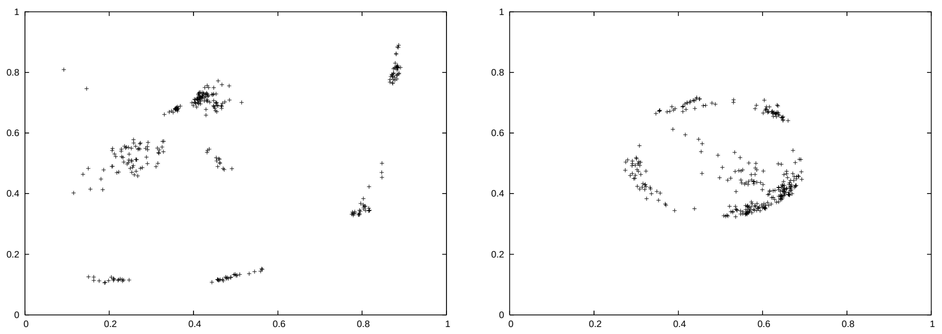

Figure 3.6: Two visualisation of a dataset with molecules.

Figure 3.6 shows two visualisations. The left-hand picture has mean 0.03488 and standard deviation 0.03117, with parameters $\rho = 0.048$ and $\sigma = 1.1$; the right-hand picture has mean 0.05029 and standard deviation 0.03200, with parameters $\rho = 0.031$ and $\sigma = 0.5$. The latter picture is what we would have gotten when we had used a bounded unit square. The first picture gives a better embedding, with a lower error. The groups that pop up can be used by a biologist to investigate biological activity.

3.5 Conclusions and further research

We conclude that our algorithm is able of giving adequate visualisations on the torus. Starting from a set of data points and their pairwise distances, it quickly provides an embedding on this surface. The algorithm is fast, flexible and easy to use, for instance for clustering purposes.

The method was originally developed for the analysis of criminal records (see Section 3.4), and performs quite well in this case, but it also appears to be applicable in other fields.

For further research, we would like to examine other topologies, such as a sphere. Yet another possibility is to somehow fix current points, once they have reached a good position with respect to *many* other points. And finally, the online addition of new points.

Chapter 4

Error Visualisation in the Particle Model

In this chapter, we introduce a new method of analysing the errors introduced by multidimensional scaling techniques. We associate an error of an item in the output of these methods with a charge, and then interpolate this charge to define a field.

We give a general method on how to do this interpolation and we provide several methods and fine tuning techniques to calculate the charge of the items.

4.1 Introduction

The *push and pull* model [43], also known as the particle or spring model [19] is a way of accomplishing a dimension reduction. However, as is the case in each dimension reduction algorithm, an error is introduced in the output in almost all cases. This error can be uniformly distributed throughout the whole picture (this would perhaps be preferable), or it can be accumulated in one or more parts of the picture. In the latter case, parts of the constructed dimension reduction can be useless and thus visualisation of the error would gain insight in the quality of the picture.

If an error map could be constructed for a dimension reduction, a potential user could use this map to assess the quality of the original picture and decide which parts are usable or not. By making a continuous map of the error, various standard techniques, like looking at the gradient, can be used for further analysis.

Error visualisation is important for dimension reduction techniques, because (apart from some trivial cases) we know that the produced image is only a projection of the original data in some way, an approximation so to speak. Because of this, the resulting image has errors and these errors might or might not be uniformly distributed throughout the image. In the first case, there is no further analysis to be done, in the latter case however, it can very well be that a part of the outcome of a dimension reduction is completely unusable, while an other part is still of value. To find out which parts of such an image are valuable we need to gain insight into the error distribution throughout the image.

Populair methods to visualise the error usually give each item in the output picture a colour, or a grey value, denoting its error, but with many points this can give a rather chaotic picture which is hardly usable.

A general way of constructing an error map is given in Section 4.2. Furthermore, two natural implementations are given, along with a general threshold function to filter out small deviations in the error maps. An artificial example dataset and various experiments on it are given in Section 4.3. We conclude in Section 4.4, we speculate upon the application of our method in other dimension reduction techniques and give suggestions for further research.

4.2 Constructing the error map

In the push and pull model, the distances between any two particles p and q from a given set is given by an input matrix. We call these distances the *desired distances*, denoted by $d_{\text{desired}}(p, q)$. The actual distances in the picture we call the *realised distances*, denoted by $d_{\text{realised}}(p, q)$. Clearly, the goal of the push and pull model is to minimise the difference between d_{desired} and d_{realised} for all pairs of points.

The output of the push and pull technique is a statical model of particles, where in general there is a difference between the desired and realised distances. The difference between these distances contributes to the error of a particle, in some way. There are several choices of functions to calculate the total error connected with a particle. This will be discussed in the next two subsections.

Once the total error connected with a particle is calculated, we can make a plot of the error propagating through space. We let the error decay as the distance to the particle grows. The error of a particle can be seen as analogous to a charge, an *error charge* so to speak. We let this error propagate through the space, analogous to the way an electrical charge propagates. We use a variant of the well-known *law of Coulomb* to calculate the field strength at any distance from the particle. When there are multiple particles, we use the sum of the field strength values to calculate the strength of the *error field* at any point in the space.

The scalar form of Coulomb's law, which describes the magnitude of the electrostatic force F on two charged particles q_1 and q_2, is defined as:

$$F = k_e \frac{q_1 q_2}{r^2}$$

where r is the distance between the particles and k_e is Coulomb's constant.

Of course, we are not bound to the law of Coulomb to indicate the decline of the field strength. The space that is the result of the dimension reduction technique needs not to be flat, so using a different function to calculate the error field is quite acceptable. Furthermore, we have no particular reason to use Coulomb's law, except for the analogue with nature and therefore mans familiarity with it.

In a model with n particles $\{p_1, p_2, \ldots, p_n\}$, we use the following formula to calculate the field strength at position (x, y) in the error map:

$$\mathit{field}(x, y) = \frac{1}{n} \sum_{i=1}^{n} \frac{\mathit{error}(p_i)}{(1 + d((x, y), \mathit{pos}(p_i)))^{\gamma}}$$

Where γ is the speed at which the field drops off. Choosing $\gamma = 1$ will result in a linear drop off, $\gamma = 2$ will result in a drop off similar to the one in the law of Coulomb. In this paper, we shall use $\gamma = 0.5$. The function d calculates the Euclidean distance between the point (x, y) and the location of the particle p_i, which is given by the $\mathit{pos}(p_i)$. The function *error* is a generalised function returning an error value. In the following subsections we discuss a number of possibilities for this function and their consequences for the error map.

A first and natural choice for the error function follows from the assumption that each particle has an equal amount of influence over each other particle. Thus we use the average of all pairwise errors to calculate the total error of a particle. We refer to this average als the *global error*.

So in a model consisting of n particles, the global error of a particle $q \in \{p_1, p_2, \ldots, p_n\}$ is calculated with the following formula:

$$\mathit{error}_{\text{global}}(q) = \frac{1}{n} \sum_{i=1}^{n} |d_{\text{desired}}(p_i, q) - d_{\text{realised}}(p_i, q)|$$

After applying this function to each particle in the model, we can make the error map with the *field* function, as described above.

If one wants to gain more insight in the (possibly dense) clusters that can arise, a natural approach is to use a decline function for the error itself. This cancels the errors of remote particles (and possible overshadowing effects of those remote particles) and focuses on the particles in the area. To calculate this error, we use a weighed average of all pairwise errors, where the weight is a (declining) function of the distance between the particles. We refer to this weighted average as the *local error*.

Again, we have a model consisting of n particles. To calculate the local error of a particle $q \in \{p_1, p_2, \ldots, p_n\}$, we use the following formula:

$$\mathit{error}_{\text{local}}(q) = \frac{1}{n} \sum_{i=1}^{n} \frac{|d_{\text{desired}}(p_i, q) - d_{\text{realised}}(p_i, q)|}{(1 + d_{\text{realised}}(p_i, q))^{\delta}}$$

Here δ is the speed at which the error of a particle loses its influence over other particles (comparable to the function of γ in the *field* function). In this chapter we use $\delta = 0.5$.

The local error field for each point in the error map can be calculated with the *field* function again.

4.2.1 Minimum correction

A technique that can be applied before we calculate either the local or the global error field is to define a threshold for the error and subtracting this threshold from the particles that have a higher error. Using such a threshold will filter out small errors that might not be of interest.

A natural choice for the threshold is the minimum error that occurs. We shall call this the *minimum correction* for the remainder of this chapter. To do the minimum correction, we first calculate the minimum of the total errors of all particles

$$err_{\min} = \min(error(p_1), error(p_2), \ldots, error(p_n))$$

and then subtract $err_{\min}$ from the error of each particle:

$$error'(p_i) = error(p_i) - err_{\min} \quad (1 \leq i \leq n)$$

For pictures with a high number of particles, of which the errors have a broad distribution, one can choose a higher threshold to see which points are responsible for most of the error.

4.3 Experiments

To give a good impression of the effect of the different choices that can be made, we have constructed a small, artificial dataset.

$$\begin{pmatrix} 0 & \frac{1}{2} & \frac{1}{2} & \frac{1}{2}\sqrt{2} & \frac{3}{20} & \frac{9}{10} \\ \frac{1}{2} & 0 & \frac{1}{2}\sqrt{2} & \frac{1}{2} & \frac{3}{20} & \frac{9}{10} \\ \frac{1}{2} & \frac{1}{2}\sqrt{2} & 0 & \frac{1}{2} & \frac{3}{20} & \frac{9}{10} \\ \frac{1}{2}\sqrt{2} & \frac{1}{2} & \frac{1}{2} & 0 & \frac{3}{20} & \frac{9}{10} \\ \frac{3}{20} & \frac{3}{20} & \frac{3}{20} & \frac{3}{20} & 0 & \frac{9}{10} \\ \frac{9}{10} & \frac{9}{10} & \frac{9}{10} & \frac{9}{10} & \frac{9}{10} & 0 \end{pmatrix}$$

Table 4.1: Sample input

In Table 4.1, we see the pairwise distances of six defining points from an object that resembles the one shown in Figure 4.1 (leaving out one point). The points a, b, c and d form a square, point f has a large distance to the corner points, making it the top of a pyramid. Point e is only defined as being very close to the corners of this square. So close in fact, that the triangle inequality does not hold because of this point e. If we take out e, the remaining points would have been embeddable in $\mathbb{R}^3$ though, as can be seen in Figure 4.1.

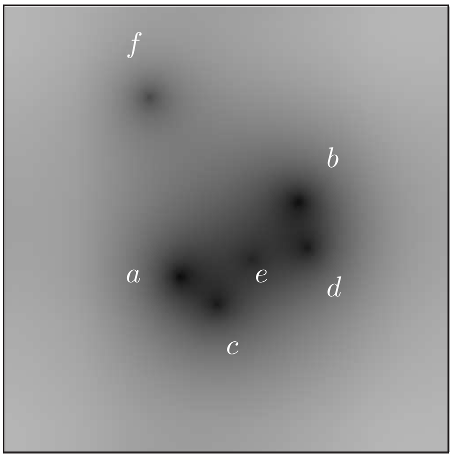

Figure 4.2: Global error map

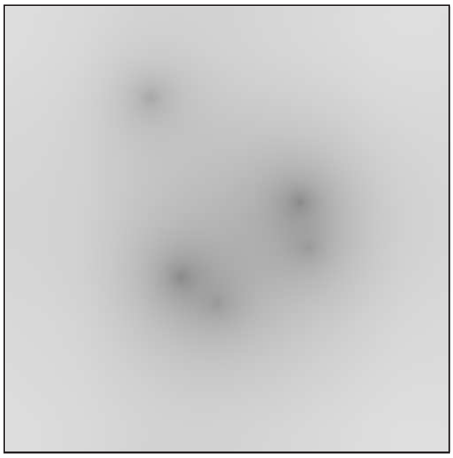

Figure 4.3: Global error map with minimum correction

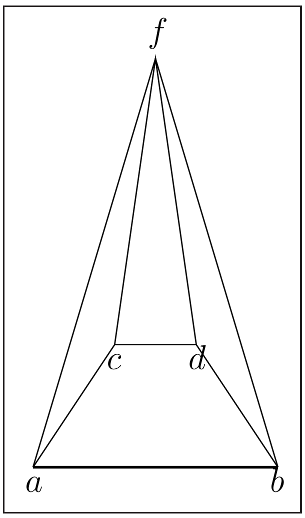

Figure 4.1: Approximate sample input (apart from one point)

As mentioned before, apart from point e, the object is a pyramid; the submatrix consisting of the first four rows and columns give the distances of the square ground plane and the last row and column give the distances to the top. The total figure can certainly not be embedded in a 2-dimensional plane (even a pyramid would be a problem), so we must introduce a number of approximations to make an embedding possible.

In Figure 4.2, we see the global error map of a dimension reduction done on the data in Table 4.1, where, as mentioned before, we use $\gamma = 0.5$. The shape is what we would expect. The ground plane of the pyramid, consisting of points a, b, c and d are the four points in the centre of the image, now in the form of a trapezoid. Point e is in the centre of this trapezoid and point f can be found in the upper left of the picture.

From this map, it is clear that the two points a and b, located on the broad side of the trapezoid have the largest error.

If we apply the minimum correction on the global error map, we get a picture as seen in Figure 4.3. Point e has vanished from this map and it is even more clear which points have the largest error.

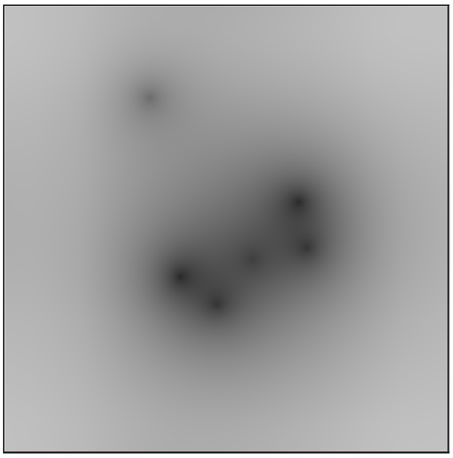
Figure 4.4: Local error map

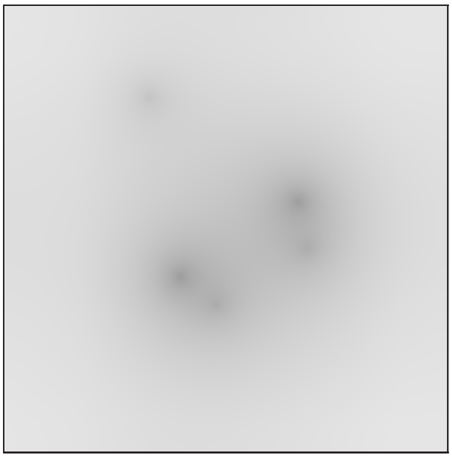
Figure 4.5: Local error map with minimum correction

In Figure 4.4 we see the local error map of the dimension reduction done on the data given in Table 4.1, where, as mentioned before, we use $\gamma = 0.5$ and $\delta = 0.5$. When compared to Figure 4.2, we see that the top of the pyramid, point f, is relatively less prominent in the local error map. The difference between the points in the ground plane, points a, b, c and d and the other points, e and f, however, has gotten larger. This can be explained by the disturbing influence of point e, that apparently is responsible for most of the errors of the points in the ground plane.

If we apply the minimum correction on the local error map, we get a picture as seen in Figure 4.5. Again, point e has vanished and this time point f is almost invisible as well. The only points that are still clearly visible are the ones that have a significant error because of a local disturbance.

Because the error map is a "continuous" map, we can apply some standard computer graphics techniques on it for further analysis. We can for example compute the derivative of the error map,

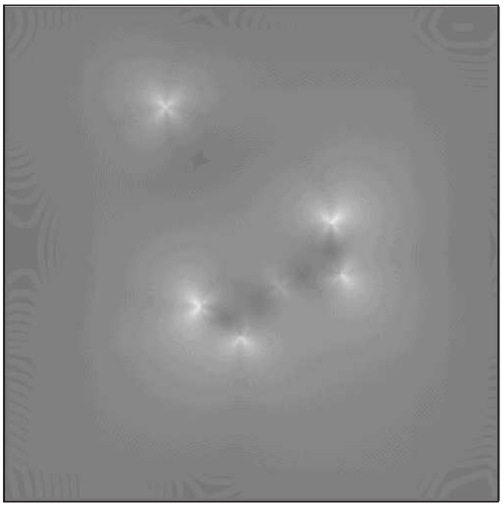

Figure 4.6: Gradient of the global error map

In Figure 4.6 we see such a derivative. We made this image by taking the maximum gradient in a 5×5 square, iterated over all points in the original image.

What we can see from this picture is the “stability” of the error field. In the dark areas, the error does not change very much, so they can be called stable. The white areas are the ones that have large fluctuations in the error. For more complicated input data, this can be of use to assess the quality of the error map.

4.4 Conclusions and further research

In this chapter, we have shown that the construction of an error map can be useful for the analysis of the quality of (parts of) the output of a dimension reduction technique. We have given several natural approaches to construct different kinds of error maps, each emphasising different aspects of the error under consideration.

The method described in this chapter can be applied to other dimension reduction techniques as well. For example, it can be applied for Principle Component Analysis [35,57], Principle Curves [29] and Multidimensional Scaling [5,16] and their (metric) variants without alteration. The non-metric variants of Multidimensional Scaling require a radically different error function, because these techniques don’t use a metric distance in their visualisation.

On the other hand, Self Organising Maps [39] have their own way of constructing an error map: by calculating a gradient from the original map.

Apart from the examples we have given, other kinds of visualisations are possible. We can, for example, use different values for γ to alter the speed at which the error field drops off. This can be useful if we want to analyse the visualisation of a large number of input points. In that case we could increase the drop off speed to analyse the errors of particles that are very close together.

Automatically determining the value of γ would be an interesting follow up for this research.

We can also use different decline functions. This might be preferable if we have some expert knowledge of the input data for example.

Chapter 5

Temporal Extrapolation Using the Particle Model

In the field of criminal investigations people on the task force usually have a gut feeling about trends in criminal careers. For example, it is reasonable to assume that a drug addict that has been doing petty theft and some other minor criminal offences, combined with drug abuse, will keep doing to do this in the future. On the other hand, someone of whom we see an incline in the severity of the crimes, has a high probability of committing even worse crimes in the future.

In this chapter we will focus on a way to find similar careers and perhaps to automatically make a prediction of a future path of a criminal career by looking at the trends in the neighbouring criminal careers.

5.1 Introduction

Under the assumption that criminals that have a similar history, will have a similar future (at least in the near future), we propose an analysis based upon a large database (obtained from the Dutch national police, which for privacy reasons can not be disclosed) which consists of information on approximately one million criminals and their criminal history. We use a dimension reduction technique for noise reduction and to simplify the calculations. We use metrics designed especially for criminal careers as a basis for this dimension reduction.

In earlier work [14] temporal extrapolation was successfully attempted within a static clustering. In this work, criminals were classified by progressively adding their career to the static clustering. In this way, the first year resulted in a position in the clustering, the first two years were an other point and so on. The next point is determined by a 2-dimensional parametric extrapolation.

A disadvantage of this technique is that it relies on the linearity of the dimension reduction technique. If the produced image is done with a non-linear technique, the result may be less usable, depending on the input data.

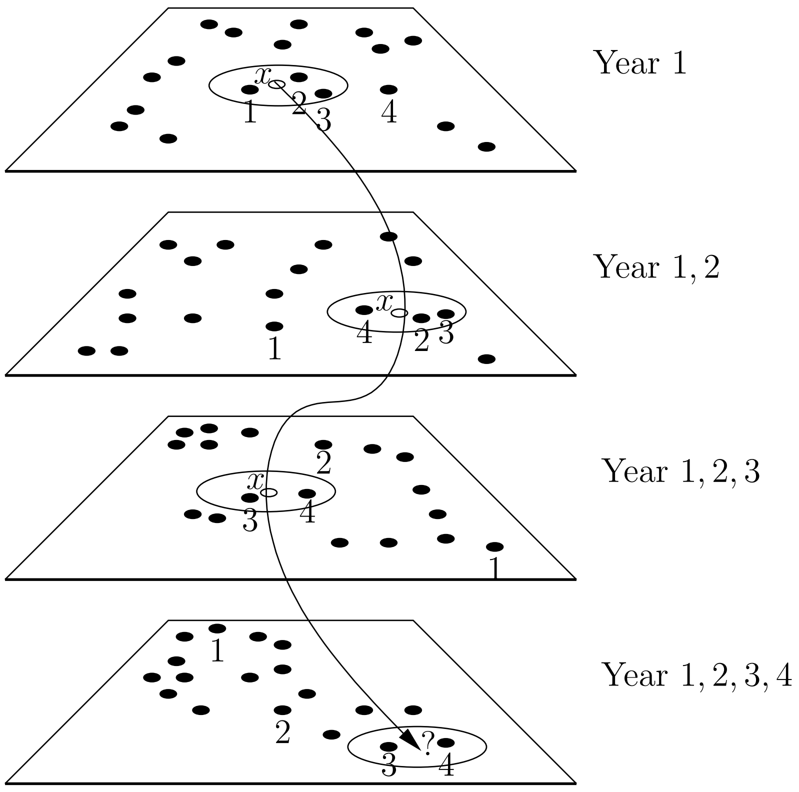

Figure 5.1: Flowing model

To accommodate for the shortcomings of the previous extrapolation algorithm, we use the idea of progressively adding a career to all points, instead of one. By doing this, we create a dynamical model that changes through time as the careers of the data points progress. By keeping track of the neighbourhood of the target point, we can extrapolate the end position of the target without making any assumptions about the space at all.

In Figure 5.1 we see an example. In the upper plane, only the first year of each career is considered, in the second plane the first two and so on. In each plane we see the tracked point (an open circle) denoted by x, and a couple of points that are or have been in the neighbourhood of x, they are denoted by 1, 2, 3 and 4.

We decide upon a neighbourhood in some way and keep track of the careers in this neighbourhood throughout the years. In the last step, we either use the neighbouring careers as such and return them for further analysis, or we can look at the subsequent crimes in all of these careers and by employing a weighing scheme, do a prediction for future crimes.

The extrapolation can be done in two ways: We can either do a new dimension reduction for each added year, or we can alter the input matrix to add a new year. If we would only do one extrapolation per target point, and if a non-deterministic dimension reduction technique is used, the latter choice might be the best one, because if the dimension reduction technique gets stuck in a local optimum, the on-line adding of new data would use this optimum as a starting point. Running multiple extrapolations though, is better in both cases and obsoletes the on-line option.

We use a *particle*- or *push and pull* model (see Chapter 2) for the dimension reduction step, because it is fast and has flexibilities that other techniques lack, such as the choice of an output space (see Chapter 6. We use a *torus* as output space in this chapter because we are only interested in the (local) neighbourhoods of each point, which the torus preserves. Furthermore, the fact that the torus is a closed surface gives more freedom for a valid visualisation.

5.2 Parameters

There are a number parameters in this approach. We shall discuss the parameters of each step in detail.

Generating the distance matrices First of all, we make a number of distance matrices (one for each year as the career progresses). As a distance measure we use the *edit distance* between two multiset sequences (see Chapter 7, each of these multisets represents the activities of one person in one year. This step itself already has a large amount of parameters. We can either use *local* or *global alignment*, we can use *expanded* or *non-expanded* careers and we have to decide on a *gap penalty* and a *edit penalty*. These parameters are discussed in more detail in Chapter 7. All of these parameters should be decided upon by a domain expert.

Dimension reduction When the distance matrices have been generated, we do a dimension reduction step. We mainly do this for *noise reduction*, but it also has the nice side effect that it makes the rest of the calculations less demanding. After this dimension reduction step, we need to decide upon the size of the neighbourhood and how to determine it. One could for example use a fixed radius with the tracked point as a centre, or we could use k nearest neighbours.

Since we use a model that can produce non-linear mappings onto the output space, we can not make the neighbourhood too large, we can not make it too small either, because it might be empty then. These two observations make the choice for the radius a difficult one, and even dependent on the input data, or sometimes even on the tracked point.

The k nearest neighbours approach solves most of these problems, but the non-linearity issue is still bothersome. Also, we obviously need to assign a sensible value for k.

We propose a hybrid solution for the non-linearity issue. We choose k nearest neighbours, but we also record their distance to the tracked point. This way we can use this distance as a weighing function in the extrapolation by using $1/\mathit{distance}$ as a weight for example.

Weighing of the neighbouring careers Finally, we need to come up with

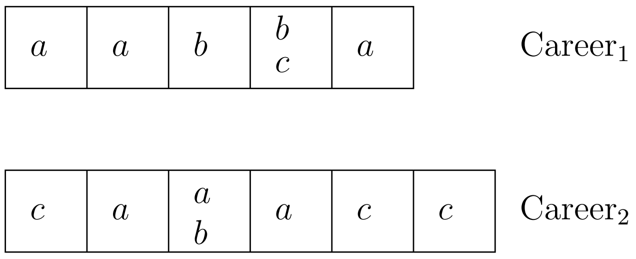

Figure 5.2: Two careers

weights that accentuate the difference between neighbouring points in the first year, neighbouring points in the first two years and so on. We do this because we assume that similarity in the last year is more significant than similarity in the first year.

Prediction As an extra step, we can use the neighbouring careers as a predictor for the future actions of the person corresponding to the tracked point. Again, there are multiple choices to consider. We can for example only use the data from the next year as a predictor, we could use multiple years, or we can take the remaining career of each neighbouring career to make a prediction. In the latter case, we propose a weighing scheme that assigns high weights for crimes in the near future and low weights for those in the distant future.

5.3 Extrapolation method

To do the extrapolation, we first need to calculate the distance matrices for the progressing careers. Suppose we have n objects and we want to do the extrapolation over m years, We first have to make the matrix for the first year, then the one for the first two, and so on. This pre-calculation step results in m symmetric matrices of $n \times n$ distances. We denote the matrix which consists of the distances for the first year only by M_1, the one for the first two years be M_2 and so on.

In Figure 5.2, we see two careers. To generate M_1, we take a look at the first field of each career, to generate M_2, we take a look at the first two fields. The distance measure we use for this comparison is the edit distance of multiset sequences (see Chapter 7).

We start the extrapolation by making a dimension reduction of the points corresponding to the distances in M_1 and pinpointing the career c we want to follow. We choose a radius, which defines the neighbourhood of the career and then we list all the points in this neighbourhood. We call this set $N_1 = N_1(c)$.

Then we either alter the distance matrix M_1 by replacing the values with the ones in M_2 and not reinitialising the visualisation for the reasons mentioned above (to avoid getting into a different local or global optimum), or we make a new visualisation. Either way, the points in the visualisation will adjust to

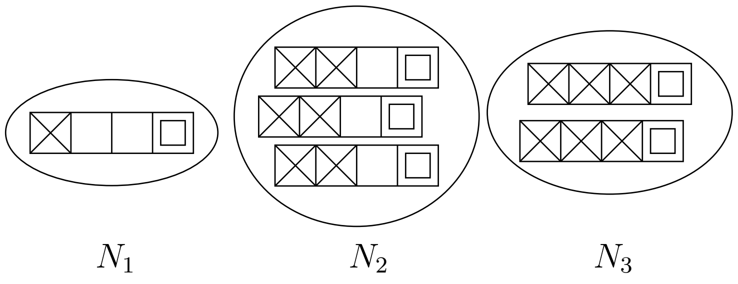

Figure 5.3: Neighbourhood sets for $m = 4$

the new data (M_2) and if the visualisation reaches a stable state again, we can extract the new neighbourhood set $N_2 = N_2(c)$ in the same way as we did above. We repeat this process until we have all $m - 1$ neighbourhood sets.

In Figure 5.3 we see a number of neighbourhood sets. The elements of the careers that are considered are marked with a cross. So in N_1 only the first element of each career is considered. In N_2 the first two elements are considered and so on. Each m-th element is marked with a square to denote the element that is used for the extrapolation.

After this step, we can have a look at element m in each career in the neighbourhood sets. It is reasonable to use a weighing scheme (where N_i has weight w_i), which assigns a large weight to careers in N_{m-1} and a small one to the ones in N_1, to emphasise the temporal aspect of the extracted data.

The weight for each neighbouring career x is calculated as follows:

$$w(x) = \sum_{i=1}^{m-1} \sum_{i:x \in N_i} w_i$$

We can now apply this calculated weight to the m-th element of career x and by doing this for all neighbouring careers, we get a multiset of weighed activities. The weight can be used as an indicator of chance. The outline of the algorithm is as follows:

```
Predict (c) ::
    for $i = 1$ to $m - 1$ do
        compute $M_i$
        do a dimension reduction for $M_i$
        determine $N_i = N_i(c)$
    for $x \in \cup_{i=1}^{m-1} N_i$ do
        compute $w(x)$
    compute prediction using $w(x)$
```

Prediction for a career c based upon neighbouring careers

5.4 Experiments

In our experiments, we took a random selection of 112,326 criminals from our input dataset (consisting of the criminal activities of Dutch offenders) and applied a filter to select those people that had a career of at least four years. This resulted in a new database consisting of 1,617 criminal careers.

The crimes (elements of the multisets) were categorised in 8 categories, which the Dutch National Police themselves use and therefore seem reasonable. These categories have not been weighed in our experiments, although some categories are representative for minor offences, while others are representative for severe crimes. We have chosen not to weigh crimes, because we want to find a prediction in more detailed behaviour, while predicting the nature of future crimes would require this weighing scheme.

For the construction of the distance matrices, we used local alignment as a scheme to calculate the edit distance. The gap penalty was set to the maximum edit penalty, which was 1.0. Note that because we search for the edit distance between strings of the same length (in every step, we only consider the first n years of a career that is longer than n), there is no difference between local and global alignment.

In the dimension reduction step, we used a fixed number of neighbours $k = 10$ and used the hybrid weighing function to compensate for "neighbours" that are far away. We made a dimension reduction for three years and recorded all the neighbouring careers for each step. Each dimension reduction was repeated four times to compensate for the possibility of local optima.

Year	1	1–2	1–3
Value	1	4	16

Table 5.1: Weights used for extrapolation

The weights for careers found in each step is given in Table 5.1. These weights were in turn divided by the distance from the tracked point, as explained in Section 5.2. In this set up, we predict the future crime with 63% accuracy, vs. the 11.1% we would get if a random result would have been returned.

5.5 Conclusions and further research

Because the dimension reduction technique was not used for the normal 2-D visualisation, but as a noise filter only, we can use a dimension reduction technique that reduces the input space to a higher dimensional space than the standard 2-D. In this way, the embedding of the calculated distances will probably be a lot better, but in general we shall need a larger radius to find a sufficient amount of neighbours in order to do the extrapolation. In other words, the accuracy will improve (up to a certain point, we still need the noise reduction of course), but at a cost. The computational complexity of the problem will increase severely.

Part II

Metrics

Chapter 6

Metrics for Mining Multisets

In this chapter, we propose a new class of distance measures (metrics) designed for multisets, both of which are a recurrent theme in many data mining applications. One particular instance of this class originated from the necessity for a clustering of criminal behaviours.

These distance measures are parametrised by a function f which, given a few simple restrictions, will always produce a valid metric. This flexibility allows these measures to be tailored for many domain-specific applications.

In this chapter, the metrics are applied in bio-informatics (genomics), criminal behaviour clustering and text mining. The metric we propose also is a generalization of some known measures, e.g., the Jaccard distance and the Canberra distance. We discuss several options, and compare the behaviour of different instances.

6.1 Introduction

In many fields data mining is applied to find information in large amounts of data. A few example areas are bio-informatics, crime analysis and of course computer science itself. In data mining, *multisets* (also referred to as bags) are a recurring theme. Finding *distance measures* or *metrics* (for multisets) is one aspect of data mining [67]. When a suitable measure is found, many types of analysis, such as clustering, can be performed on specific documents, DNA and other instances of multisets.

The reasons for finding distance measures are very diverse. In crime analysis [9] for example, it is possible to determine the distance between two criminals based on their behaviour (their crime record). In bio-informatics comparing two species with only the information of their DNA (or short fragments of it) can be done. This is especially useful in forensic applications where DNA strands are frequently damaged, so the fragments that are extracted from samples cannot be given a place on the genome. Even without the information of the placement of the DNA fragments found, it is possible to differentiate between species and even individuals by using techniques described in this chapter. We finally mention market basket analysis, where distances between multisets are basic for further analysis. As a motivating example, the distance between two customers can or cannot take into account the numbers of purchases of individual products (thus providing either multisets or sets), and it is also possible to stress the difference between 1 and 2 sales on the one hand and, e.g., 41 and 42 sales on the other hand.

In Section 6.3 we give a new class of distance measures that are suitable for comparing multisets. The class has a parameter f (a function) that has a couple of simple properties which, if met, will always produce a valid metric. To the best of our knowledge, the class is new, and generalizes several of the more well-known distance measures mentioned in Section 6.2 and Section 6.3.

For different domains, different problems arise and different distance measures will be needed. For many of them, a tailor made function f can be provided and if the given restrictions apply to f, no further effort has to be made with respect to the validity of the metric. We mention several examples in the applications in Section 6.4. Different choices of f may lead to different visualisations. Furthermore, choosing such a function f is rather straightforward and intuitively easier to do than constructing a metric directly.

6.2 Background

Finite multisets from a universe with n elements can be viewed as points in n-dimensional space. For example, the multiset $\{a, b, a, a, b, a\}$ can be abbreviated to $\{a^4, b^2\}$ (since the order of elements is irrelevant) and by leaving out the element names, we get the vector $(4, 2)$ in 2-dimensional space. Several known distance measures can be applied. We mention the most important ones. In Section 6.3 we will show the relation with our metric. In all cases, we consider multisets X, Y over $\{1, 2, \ldots, n\}$, and let $x_i \in \mathbb{R}_{\geq 0}$ (resp. y_i) be the number of times that i $(i = 1, 2, \ldots, n)$ occurs in X (resp. Y).

- Minkowski distance of order p [67]

$$d(X, Y) = \left(\sum_{i=1}^{n} |x_i - y_i|^p \right)^{1/p}$$

  For $p = 1$, we get the Manhattan distance; for $p = 2$, we get the well-known Euclidean distance; if we let $p = \infty$, we get the Chebyshev distance or L_∞ metric.

- Canberra distance

$$d(X, Y) = \sum_{i=1}^{n} \frac{|x_i - y_i|}{x_i + y_i}$$

When both x_i and y_i are zero, the fraction is defined as zero. Often the distance is divided by the number of indices i for which at least one of x_i or y_i is non zero.

- Jaccard distance for sets [34]

$$d(X,Y) = \sum_{i=1}^{n} |x_i - y_i| \;/\; \Big(n - \sum_{i=1}^{n}(1-x_i)(1-y_i)\Big)$$

- Bray-Curtis (Sorensen) distance (often used in botany, ecology and environmental science) [6]

$$d(X,Y) = \sum_{i=1}^{n} |x_i - y_i| \;/\; \sum_{i=1}^{n}(x_i + y_i)$$

- Mahalanobis distance (generalized form of the Euclidean distance) [48] This is an example of a metric that requires a more complicated scheme: the covariance matrix of the data must be computed, which is quite time-consuming. We will not further discuss this type of metric here.

6.3 The metric

In this section we will define our new *class of metrics*. As a parameter we have a function f that must meet several properties.

Let f be a function $f : \mathbb{R}_{\geq 0} \times \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$ with finite supremum M and the following properties:

$$\begin{aligned} f(x,y) &= f(y,x) && \text{for all } x,y \in \mathbb{R}_{\geq 0} && (6.1)\\ f(x,x) &= 0 && \text{for all } x \in \mathbb{R}_{\geq 0} && (6.2)\\ f(x,0) &\geq M/2 && \text{for all } x \in \mathbb{R}_{>0} && (6.3)\\ f(x,y) &\leq f(x,z)+f(z,y) && \text{for all } x,y,z \in \mathbb{R}_{\geq 0} && (6.4) \end{aligned}$$

For a multiset X, let $S(X)$ denote its underlying set. For multisets X, Y with $S(X), S(Y) \subseteq \{1,2,\ldots,n\}$ we define $d_f(\emptyset,\emptyset) = 0$ and

$$d_f(X,Y) = \frac{\sum_{i=1}^{n} f(x_i,y_i)}{|S(X) \cup S(Y)|}$$

if both X and Y are non-empty. Again, $x_i \in \mathbb{R}_{\geq 0}$ (resp. y_i) is the number of times that i ($i = 1,2,\ldots,n$) occurs in X (resp. Y; usually x_i and y_i are integers); $|S(X) \cup S(Y)|$ is the number of elements in $X \cup Y$, seen as set. Note that $0 \leq d_f(X,Y) \leq M$, $d_f(X,Y) = d_f(Y,X)$ and $d_f(X,Y) = 0 \Rightarrow S(X) = S(Y)$. If f also satisfies

$$f(x,y) = 0 \Rightarrow x = y \quad \text{for all } x,y \in \mathbb{R}_{\geq 0} \qquad (6.5)$$

we have $d_f(X,Y) = 0 \Rightarrow X = Y$. It is clear that properties (1), (2) and (4) must hold in order to ensure that we have a metric; indeed, just consider the case where $n = 1$.

The function f specifies the difference between the number of occurrences of a particular element in two multisets. Constructing such a function is natural and can easily be done by domain experts. Also note that the function f is defined for all positive real numbers; this property is only used when weights are involved (see Section 6.3), and it also makes the proof below more general.

We now show that d_f satisfies the *triangle inequality*, and therefore is a metric.

Theorem 1. *For all X, Y, Z with $S(X), S(Y), S(Z) \subseteq \{1, 2, \ldots, n\}$ we have:*

$$d_f(X,Y) \leq d_f(X,Z) + d_f(Z,Y)$$

Proof. We may assume that not both X and Y are $\emptyset$. If $d_f(X,Z)+d_f(Z,Y) \geq M$ we are done, since $d_f(X,Y) \leq M$. So we may assume that $d_f(X,Z)+d_f(Z,Y) < M$. Now

$$\begin{aligned} d_f(X,Y) &= \frac{\sum_{i=1}^{n} f(x_i,y_i)}{|S(X) \cup S(Y)|} = \frac{\sum_{i \in S(X) \cup S(Y)} f(x_i,y_i)}{|S(X) \cup S(Y)|} \\ &\leq \frac{\sum_{i \in S(X) \cup S(Y)} f(x_i,z_i) + \sum_{i \in S(X) \cup S(Y)} f(z_i,y_i)}{|S(X) \cup S(Y)|} \\ &= \frac{\sum_{i \in S(X) \cup T} f(x_i,z_i) + \sum_{i \in S(Y) \cup T} f(z_i,y_i)}{|S(X) \cup S(Y)|} \end{aligned}$$

where the set T is defined by $T = S(Z) \cap (S(X) \cup S(Y))$. We have

$$\begin{aligned} \sum_{i \in S(X) \cup T} f(x_i,z_i) &= \sum_{i \in S(X) \cup S(Z)} f(x_i,z_i) - \sum_{i \in S(Z) \setminus T} f(0,z_i) \\ &\leq \sum_{i \in S(X) \cup S(Z)} f(x_i,z_i) - \frac{tM}{2} \end{aligned}$$

with $t = |S(Z) \setminus T|$. We conclude

$$\begin{aligned} d_f(X,Y) &\leq \frac{\sum_{i \in S(X) \cup S(Z)} f(x_i,z_i) + \sum_{i \in S(Y) \cup S(Z)} f(z_i,y_i) - tM}{|S(X) \cup S(Y)|} \\ &= \frac{d_f(X,Z)|S(X) \cup S(Z)| + d_f(Z,Y)|S(Y) \cup S(Z)| - tM}{|S(X) \cup S(Y)|} \end{aligned}$$

Now $-tM \leq -t(d_f(X,Z)+d_f(Z,Y))$ (because of the assumption that $d_f(X,Z)+d_f(Z,Y) < M$). So, noting that $|S(X) \cup S(Z)| = t + |S(X) \cup T|$ (and similarly for $|S(Y) \cup S(Z)|$) we get

$$\begin{aligned} d_f(X,Y) &\leq \frac{d_f(X,Z)|S(X) \cup T| + d_f(Z,Y)|S(Y) \cup T|}{|S(X) \cup S(Y)|} \\ &\leq d_f(X,Z) + d_f(Z,Y) \end{aligned}$$

since $|S(X) \cup T| \leq |S(X) \cup S(Y)|$ (and similarly for $|S(Y) \cup T|$). $\square$ $\square$

Before studying several properties of the metric, we first notice that its behaviour deviates from that of standard distance measures. As an example, if we have two given points, and we move one of these in a "new" dimension, the distance changes considerably whereas in the Euclidean case it does not.

Interesting properties of this measure are:

- If X and Y are "normal" sets, i.e., $x_i, y_i \in \{0,1\}$ $(i = 1, 2, \ldots, n)$, we note that

$$d_f(X,Y) = f(1,0)\frac{|X \setminus Y| + |Y \setminus X|}{|X \cup Y|} = f(1,0)\left(1 - \frac{|X \cap Y|}{|X \cup Y|}\right)$$

- $d_f(\emptyset, \underbrace{(1, \ldots, 1)}_{n}) = nf(1,0)/n = f(1,0)$.

Here we use the notation $(x_1, x_2, \ldots, x_n)$ for the multiset X, where again x_i denotes the number of times the element i occurs in X (cf. the example in Section 6.2).

A variant of this measure can be defined as follows:

$$\widetilde{d}_f(X,Y) = \frac{\sum_{i=1}^{n} f(x_i, y_i)}{|S(X) \cup S(Y)| + 1}$$

By using this measure, we can drop the separate definition of $\widetilde{d}_f(\emptyset, \emptyset)$. Another advantage of this measure is that $d(\emptyset, \{x\}) = d(\emptyset, \{x, y\}) = \ldots = \frac{1}{2}f(1,0)$, while $\widetilde{d}(\emptyset, \{x\}) = \frac{1}{2}f(1,0)$, $\widetilde{d}(\emptyset, \{x, y\}) = \frac{2}{3}f(1,0)$ and so on. All conditions for a distance measure hold, since this function is still symmetric, the distance between identical multisets is zero, and the triangle inequality holds. To show the latter property, we can use a proof that is analogous to the one above, except for the last step, in which we replace $|S(X) \cup T|/|S(X) \cup S(Y)| \leq 1$ by $|S(X) \cup T|/(|S(X) \cup S(Y)| + 1) \leq 1$. Another way of proving it is by adding a new element $*$ that is present once in each multiset. This reduces the problem to the property shown above: $\widetilde{d}_f(X,Y) = d_f(X \cup \{*\}, Y \cup \{*\})$.

The application of weights for certain elements can be done by multiplying the number of elements to which the weight must by applied by the weight. These weights need not be integers, which is the reason why f is defined on real numbers in Section 6.3. As an example, suppose we have the multiset $X = (1, 2, 1)$ and we want to apply the weight 10 to the first element. The resulting multiset X' is defined by $X' = (10, 2, 1)$. We shall return to this issue in Section 6.4.

In order to obtain more reasonable and intuitive measures, the following restriction can be posed upon f:

$$f(x,y) \leq f(x', y') \quad \text{if } x' \leq x \leq y \leq y' \tag{6.6}$$

It then follows that $\lim_{k \to \infty} f(k, 0) = M$. In this case it is easy to show that the condition that $f(x,0) \geq M/2$ is mandatory for the triangle inequality to hold. Indeed, let $X = (k, 0, \ldots, 0)$, $Y = (0, \ldots, 0)$ and $Z = (0, \ell, \ldots, \ell)$. With $|S(X)| = 1$, $|S(Y)| = 0$ and $|S(Z)| = n - 1$, we have

$$\begin{aligned} d_f(X,Y) &= f(k,0) \to M \text{ when } k \to \infty \\ d_f(X,Z) &= \frac{f(k,0) + (n-1)f(\ell,0)}{n} \to \frac{M + (n-1)f(\ell,0)}{n} \text{ when } k \to \infty \\ d_f(Z,Y) &= \frac{(n-1)f(\ell,0)}{n-1} = f(\ell,0) \end{aligned}$$

Now $d_f(X,Y) \le d_f(X,Z) + d_f(Z,Y)$ implies $f(k,0) \le 2f(\ell,0)$ (let $n \to \infty$). With $f(\ell,0) < M/2$ for some $\ell > 0$ this is not true, so the triangle inequality does not hold.

A natural way to generate a suitable f is the following. Start with a function $g : \mathbb{R}_{\ge 0} \to \mathbb{R}_{\ge 0}$, and put $f(x,y) = |g(x) - g(y)|$. Clearly, properties (1), (2) and (4) hold for f. We may take $g(0) = 0$. If in addition g is an increasing function with $\lim_{x\to\infty} g(x) = M$ and $g(x) \ge M/2$ for $x \in \mathbb{R}_{>0}$, f also satisfies properties (3) and (6.6). If g is injective, e.g., if g is strictly increasing, (5) holds too.

Typical examples include:

- $g(x) = 1$ for x with $0 < x \le 1$ and $g(x) = M = 2$ for x with $x \ge 1$
- $g(x) = 1/2$ for x with $0 < x < L$ and $g(x) = M = 1$ for x with $x \ge L$; here L is a (large) constant
- $g(x) = 1/2$ for x with $0 < x \le 1$ and $g(x) = x/(x+1)$ for x with $x > 1$ ($M = 1$), see Section 6.4; note that if we only use integer arguments, we just need the "$x/(x+1)$ part"
- $g(x) = 1/2$ for x with $0 < x \le 1$ and $g(x) = (2^x - 1)/2^x$ for x with $x > 1$ ($M = 1$)

We conclude with a more intuitive explanation of the metric. Consider two vases filled with marbles of different colours. We first take a look at the marbles of the first colour. If both vases contain many marbles of this colour, the difference should be small, but the difference between one marble and no marbles should be large. The exact difference can be tuned by altering the function f, which specifies the distance between groups with a different number of marbles of the same colour.

When looking at all colours, we repeat the procedure above and divide by the amount of colours we have encountered. This differs from division by the total number of marbles, or by (some variation of) the total number of possible colours. This latter option, for instance chosen in case of the Euclidean distance, does not keep track of the "sizes" of the multisets under consideration. Choosing the total number of marbles as denominator — as the Bray-Curtis distance does — has the disadvantage that adding one marble of a fresh colour is hardly noticed, while our metric is much more sensible to this. Our metric emphasizes the number of different colours.

Relation with other distance measures

Many well-known distance measures are special cases of the one we describe here. For example the Jaccard distance can be constructed by any f with $f(1,0) = 1$, where the multisets must be "normal" sets, so $A = S(A)$ and $B = S(B)$. As noted before, this results in the following formula for sets:

$$d(X,Y) = \frac{|X\backslash Y| + |Y\backslash X|}{|X \cup Y|}$$

To produce the Canberra distance (with the extended denominator) we use the following f:

$$f(x,y) = \frac{|x-y|}{x+y} \text{ for } (x,y) \neq (0,0)$$

and $f(0,0) = 0$. Note that this f cannot be constructed by a function g in the way explained above.

6.4 Applications

In this section we use the following function for f:

$$f_0(x,y) = \frac{|x-y|}{(x+1)(y+1)}$$

This function satisfies properties (1)–(6) mentioned in the previous section; it is a result of using $g(x) = x/(x+1)$. This function has the interesting property that if both x and y are large, the resulting value is small. For example, the (pairwise) distance between 0 and 1 is larger than the distance between 8 and 9, which is intuitive in many applications concerning multisets.

The visualisation algorithm we use in this section is a randomized push-and-pull oriented algorithm [43], comparable to a competitive neural network. It gives a projection of the original points in a 2-dimensional space. The reason we use this algorithm is because it is fast and able to give a clustering for many data points, where normal dimension reduction algorithms perhaps would fail. For the purpose of this chapter, there is no need to go into detail concerning this algorithm. We only state here that the Euclidean distances between points in the 2-dimensional space approximate the original distances as good as possible.

Plagiarism: When comparing two documents while ignoring the context and the semantics, we can make a multiset of words in the documents. To accommodate for the difference in lengths of two documents, we can increase the weight for each word in the smallest document by the relative size of the documents. In this way, identical copies of the same text will be detected.

In this chapter we will not further elaborate on this; we only mention the flexibility of the measures, which allows for many user-defined alterations.

Genomics: We will give an example of an instance of our distance measure in the genomics domain. We do not claim this particular instance is the best for clustering species, but from the illustrative example it can be seen that this instance does work.

A *genome* of some biological species can be considered as a long string over a small finite alphabet, usually $\{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$; it can be converted into a multiset by using a sliding window of length n to count the occurrence of each substring (or factor) of length n. Of course the number of occurrences will depend on n: the larger n is, the lower the number of occurrences will be on average.

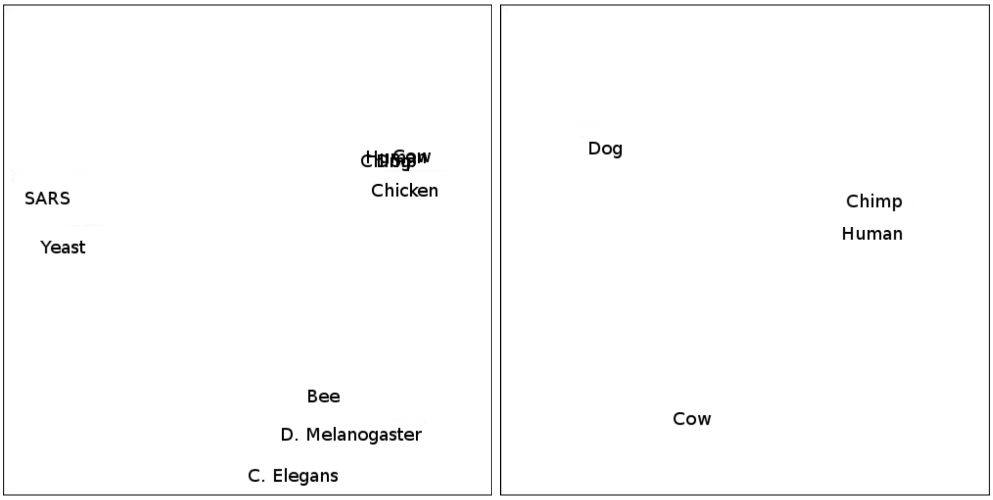

Figure 6.1: Visualisation for ten species; left: all ten; right: the four mammals

By determining the number of occurrences of each factor in two genomes, we obtain two multisets that can be compared to each other. If we use our distance measure with the function mentioned above, the occurrences of unique or almost unique substrings will account for most of the difference between the genomes. Factors that occur many times in both genomes are accounted for accordingly.

In this way we compare two species mostly on the number of differences between rare substrings in their DNA. In Figure 6.1 a clustering based on this distance is shown; DNA [69] of ten species (SARS, Yeast, Bee, C. Elegans, Drosophila Melanogaster, Chicken, Cow, Dog, Chimp and Human) is used; the right part of the figure zooms in on the four mammals, which are very close together in the left part of the figure (the labels are practically on top of each other). The sizes of the genomes vary from $3.69 \cdot 10^4$ for SARS to $3.60 \cdot 10^9$ for Cow. As in the case of Plagiarism, we here also compensate for the difference in sizes.

Apart from this type of clustering, other visualisations are possible too: the metric can also be used to generate a phylogenetic tree, for example.

Criminal records

For comparing criminal records [9], the above function is very well suited. When we make a multiset from criminal records, we get for example a multiset where the first element represents bicycle theft, the second one represents violent crimes, and so on. The difference between no crime and one or more crimes in each category accounts for a large difference, while having two large numbers in each category accounts for almost no difference at all. This is rather useful, since two people who steal bikes on a regular basis, can be seen as much alike.

Of course there are some differences between the categories which one might want to accentuate. For example, a murder is considered a much more severe offence than a bicycle theft. One way to accommodate for this difference is to use a vector of weights $W = (w_1, w_2, \ldots, w_n)$ with $w_i \in \mathbb{R}_{\geq 0}$ and to make the following adjustments to the distance measure:

$$d_f^W(X, Y) = \frac{\sum_{i=1}^n f(w_i x_i, w_i y_i)}{|S(X) \cup S(Y)|}$$

It is easy to prove that this adjustment does not change the fact that the distance formula is still a good metric.

Now, by choosing the vector of weights carefully (this must be done by an expert in criminology) we can assign relative weights for crimes. In our example, we can set the weight for bicycle theft to 1 and the weight for murder to a large integer to accentuate the severity of the crime.

As a test case, we made the following synthetic dataset with fictional crimes $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$ and $\mathcal{D}$ of increasing severity, and criminals ranging from 1 to 10. For each criminal the number of crimes in each category is given. For instance, 1 is innocent, 2 is an incidental small criminal, 6 is a one-time offender of a serious crime, and 10 is a severe all-round criminal.

	1	2	3	4	5	6	7	8	9	10
$\mathcal{A}$	0	2	10	0	0	0	0	2	0	2
$\mathcal{B}$	0	0	0	2	0	0	2	4	0	2
$\mathcal{C}$	0	0	0	0	1	0	2	0	3	2
$\mathcal{D}$	0	0	0	0	0	1	1	0	5	2

Table 6.1: Ten criminals, four crimes

In the top-left picture of Figure 6.2 we see a clustering of these ten criminals with the standard f_0. In the picture right next to it, we applied weights $1, 10, 100, 1000$, respectively, to the crimes, to specify the weight of the crime. We now see that criminals 7 and 10 are very close together, but at the same time, criminals 2 and 3 also stay close. “Criminal” 1 is surprisingly rather close to the two criminals who have committed relatively light crimes. The reason that criminals 5 and 6 are close together is because they are one-time offenders, and have a large distance to the rest of the group.

In the bottom-left picture, we see a clustering with f chosen in such a way that we get the Jaccard distance, so we treat the criminals as sets. Notice that criminals 2 and 3 now have distance zero to each other (the labels are on top of each other in this picture). The bottom-right clustering uses a totally different f: $f_1(x, y) = \frac{3}{2} - f_0(x, y)$ for $(x, y) \neq (0, 0)$ and $f_1(0, 0) = 0$. Note that, e.g., $f_1(0, 1) = 1 > \frac{3}{4} = f_1(0, 3)$, so property (6.6) does not hold for f_1. Now criminals with disjoint behaviour are grouped, leading to a “dissimilarity” clustering.

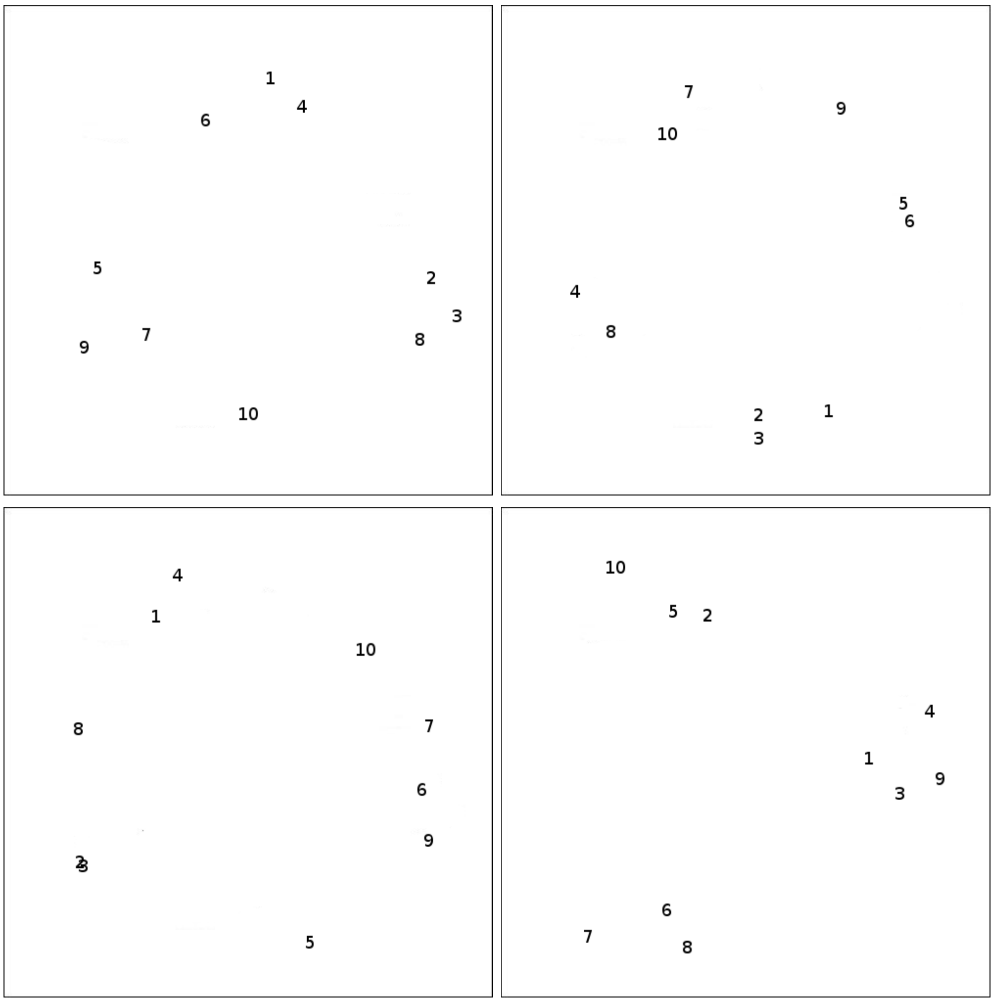

Figure 6.2: Four different clusterings for ten criminals

6.5 Conclusions and further research

In this chapter we have proposed a new flexible distance measure, that is suitable in many fields of interest. It can be fine tuned to a large extent.

We can use this measure as a basis for further analysis, like the analysis of criminal careers. In that case, we suggest that the distance measure is used as a basis for alignment to make the best match between two careers. By doing this, and by comparing sub-careers, we might be able to extrapolate criminal behaviour based upon the criminal record through time. We also want to apply the measure to a real, large database. Finally, we would like to examine the relation with more statistically oriented measures.

Chapter 7

Alignment of Multiset Sequences

The concept of multiset sequences is common in a number of different application domains. This chapter introduces a new metric for the similarity between these sequences. Various types of alignments are used to find the shortest distance between two sequences. This distance is based on a well-defined distance measure for multisets.

Employing this, a pairwise distance can be defined for two sequences. Apart from the pairwise distances, the occurrence of holes (for timestamped sequences) can also be used in determining similarity; several options are explored. Applications of this metric to the analysis of criminal careers and access logs are reviewed.

7.1 Introduction

Data mining techniques are often employed to extract information from large quantities of data [67]. One of the recurring concepts in data mining is that of *multisets* (also known as bags). A multiset is a set over a finite alphabet where elements may be present more than one time. For example, a vase filled with coloured marbles is a multiset, there can be more than one marble of a particular colour in the vase. Another common theme is data with a strict ordering in some sense, i.e., a temporal aspect. Such information is usually in the form of timestamps.

In *bio-informatics* [26, 65], and especially the field of genomics, the concept of *alignment* is well-known. Various types of alignments are used to match specific strings of DNA to each other. They are usually employed to measure how different the target string is from the other strings in the experiment. The more different it is, the more interesting it might be. If two strings of DNA are close to each other in this sense, there is a reasonable chance that these strings will stick to each other (or hybridise as the biologists call it), thus ruining the experiment.

Strings that have a very high edit distance to all other strings (within a specific genome) are therefore a research area with many applications, the construction of *primers* [17, 37] and *micro arrays* [58] being two of them.

A good example dealing with ordered multiset sequences is the analysis of *criminal careers*, a concept used in the law enforcement area [50, 51].

We can represent a criminal career (the criminal activities a person exhibits throughout his or her life) as a sequence of multisets. Each element (of a multiset) representing a crime. Since someone can commit multiple crimes and also repeat a crime within a certain timeframe, the adoption of the multiset paradigm is natural. In the analysis of criminal careers, the temporal aspect of the elements in a career (the crime is committed on a certain date) is also an important characteristic. Combining the properties of this distribution with the frequency of the crimes can provide valuable insight into the behaviour of criminals. By calculating distances between careers, one can make predictions about the threat someone poses at the time of arrest, or an analyst could make predictions about the future development of a starting delinquent.

Another research area concerned with the analysis of time sequences is knowledge discovery from (web) access logs, e.g., the activity at a certain time is related to the parts of the world where it is day or night. Also, crawlers and spiders tend to cause a more or less cyclic pattern.

Our method aims to calculate the distance between two sequences of multisets in order to provide an insight in similarities between, e.g., careers or web browsing behaviour. We explore different kinds of alignment and methods of representation of the multiset sequences in order to obtain different visualisations and clusterings, each one accentuating a different aspect of the sequences under consideration. We use the visualisation just as a means to show the fertility of the alignment method.

Apart from the direct applicability in the above mentioned analysis, this new metric can be used as a stepping stone for the enhancement of existing techniques, a good example being temporal extrapolation [14], that depends upon valid metrics.

The overview of the rest of this chapter is as follows. In Section 7.2 we provide information about the concept of alignment and explain the different types of alignment used within our approach. In Section 7.3 we explain how we adapt current alignment methodologies to work with multisets and how they deal with missing data. In Section 7.4 we describe the datasets on which we can apply our techniques and we mention the details of the application of the techniques on the datasets. We conclude in Section 7.5.

7.2 Background

Research of the alignment of strings has been prominent for a number of years, resulting in the development of a wide variety of algorithms. Most of them are calculating the difference between two sequences, which can be done by calculating the so-called *edit distance* [45]. This is defined by the minimum number of edits needed to transform one string into an other. Most often, an *edit* is defined as either an insertion, a deletion or the rewriting of a symbol. Each of these three operations leads to a certain *penalty* that can be different and may even depend on the symbols in the sequence itself. The sum of these penalties is then an upper bound for the edit distance. If the penalty is minimal (usually this is also a minimum number of edits), we speak of *the* edit distance.

The goal of alignment is to minimize this distance for two sequences, in practice arranging the sequences in such a way that the similarities between the two sequences are "placed" below each other in a visualisation. Such a placement is called a *layout*. In a layout *gaps* (usually denoted by `-`'s) represent insertions or deletions. By definition, (and with slight abuse of language) an *alignment* is one of the optimal layouts possible. The problem of constructing an alignment starts with the calculation of the edit distance, yielding a number of different solutions. One of them, selected arbitrarily, is then traced back to generate an alignment. Depending on the purpose of the algorithm, there are several types of alignment available.

A *global alignment* [54] is a form of alignment that has a high preference for matching entire sequences to each other. This makes a global alignment most useful when aligning two sequences that have roughly the same size. It can also be used when there is a good reason within the application domain to assume that short sequences are different from long sequences. By using global alignment, the distance between such sequences will be large by default.

A *local alignment* [66] puts less penalty on gaps at the beginning or end. Therefore it is most suitable to align small strings to large ones. The small string will be matched to the most compatible substring of the large one.

We will not discuss the inner workings of these (well-known) algorithms here. Both global and local alignments are efficiently calculated by employing *dynamic programming* techniques in quadratic time ($\mathcal{O}(mn)$) and usually in linear space ($\mathcal{O}(\min(m, n))$) [31], where m and n are the lengths of the sequences.

Related work can be found in the alignment of event sequences [49] and musical sequence comparison [36], where raw events, rather than bags of events, are considered as elementary units. An other difference with these techniques is that we operate within a discrete domain, whereas other techniques operate on continuous spatio-temporal data, where raw events can be identified, and the need for the use of multisets is absent.

Example 1. In this example we have two sequences $U =$ `AABABBBAA` and $V =$ `AAAABBAA` of consecutive symbols (or atomic actions) from the alphabet {`A`, `B`}. All penalties equal 1. First an alignment is constructed for the strings U and V; one of the possible alignments is depicted below:

```
    012345678
U = AABABBBAA
    || ||| ||
V = AAAABB-AA
```

This alignment (an optimal layout) has two errors, one at position 2 and one at position 6. Other alignments with two errors are also possible. □

In the case of alignment, the edit distance is the distance measure for two sequences of symbols. The first requirement of such a distance measure is stating the difference between the symbols (a distance for the elements, from which the edit distance follows). A distance matrix like the one below is typical:

	A	B	-
A	0	x	y
B	x	0	z
-	y	z	$-$

Here $x, y, z \in \mathbb{R}_{\geq 0}$ are domain specific constants. So in an alignment where A has to be matched to B, we get a penalty x, and when B is matched to a gap, then the penalty is z. Since the matching of two gaps never occurs, this value is omitted from the distance matrix.

Normally a distance matrix like the one from the example will suffice for an alignment. In genomics there are various distance matrices to give a distance between two nucleotides, based upon different experimental data.

When dealing with crimes, where for example A is a bicycle theft and B is robbery, a domain expert should specify y by giving the absolute severity of A and z by giving the absolute severity of B. The maximum or average sentence for each crime might be a reasonable value (in days of captivity for example). The value of x can then be set to the difference in severity between crimes A and B.

In the case of access logs, we can assign different distances to elements according to their source. For example, if a certain network is known for its many bots, we can increase or decrease the distance to other elements depending on what we want to investigate. We can also do this for countries if we want to accentuate (or under-emphasize) hits from a certain area. This can potentially be realised by using a GIS database.

The main bottleneck in this method for multiset strings is that using a predefined distance matrix is not feasible because the number of multisets is too large or even infinite. Since each symbol may be used arbitrarily often, there are infinitely many multisets. For example, a criminal can steal several (potentially very many) bikes in one year.

In this chapter we present a solution to this problem and suggest some (domain specific) distance measures.

7.3 Alignment adaptation

For the alignment of general structures, we need at least two things: a valid distance measure for the elements (in our case multisets) of the structure and a way to deal with so-called “holes”.We first discuss the pairwise comparison and then the conversion into sequences of multisets.

Pairwise comparison

In order to do alignment on structures that are not bare sequences a distance matrix with pre-calculated values might not be sufficient any more. In our case, we want to know the distance between multisets. To do this, we use the following approach (see Chapter 6) which generalises well-known distance measures like the Jaccard [34] and the Canberra distance.

Let f be a function $f : \mathbb{R}_{\geq 0} \times \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$ with finite supremum M and the following properties:

$$\begin{array}{rcll}
f(x,y) & = & f(y,x) & \text{for all } x,y \in \mathbb{R}_{\geq 0} \\
f(x,x) & = & 0 & \text{for all } x \in \mathbb{R}_{\geq 0} \\
f(x,0) & \geq & M/2 & \text{for all } x \in \mathbb{R}_{>0} \\
f(x,y) & \leq & f(x,z) + f(z,y) & \text{for all } x,y,z \in \mathbb{R}_{\geq 0}
\end{array}$$

For a multiset X, let $S(X)$ denote its underlying set. For multisets X, Y over the set $\{1, 2, \ldots, n\}$ we define (if $X \neq \varnothing$ or $Y \neq \varnothing$)

$$d_f(X,Y) = \frac{\sum_{i=1}^{n} f(x_i, y_i)}{|S(X) \cup S(Y)|}$$

and $d_f(\varnothing, \varnothing) = 0$. Here x_i, resp. y_i, denote the integer number of elements in category i for multiset X, resp. Y.

As function f we will use the following function ($x, y \in \mathbb{R}_{\geq 0}$, both ≥ 1, or $x = 0$ or $y = 0$):

$$f(x,y) = \frac{|x-y|}{(x+1)(y+1)}$$

(Other values with $x < 1$ or $y < 1$ are not needed.) In Chapter 6 it was shown that this d_f is a decent distance measure; note that the supremum M equals 1. The reason that we use this particular function is because the difference between two different small numbers (like 0 and 1) is large in comparison to the difference between two distinct large numbers. For our purposes, this is natural, because the fact that someone has committed a crime is more important than the number of crimes that were committed in that category. Indeed, the fact that someone has stolen a bike, is more important than the number of bikes that were stolen.

In the case of access logs, the same reasoning can be used. It is important to see the difference between 0 and 1 hits from a certain area, but the difference between 100 and 101 hits is not very important.

We could also have chosen a function like

$$f(x,y) = \frac{|x-y|}{x+y+1}$$

or any other function that has the required properties. Such a choice must in general be made by a domain expert. We want to emphasise that for the alignment of other multiset sequences, a totally different function might be needed. It all depends on the underlying data represented by the multisets.

Sequences: From actions to multisets

In this subsection we describe how an action sequence is converted into a sequence of multisets, paying special attention to the introduction of so-called *holes* (empty multisets).

Example 2. In this example we again have two sequences U = BAAA and V = BAA of consecutive symbols (actions) from the alphabet {A, B}. Again, all penalties equal 1. This time, however, each action has a distinct timestamp associated with it: $t(U_0) = 0$, $t(U_1) = 1$, $t(U_2) = 2$, $t(U_3) = 3$, $t(V_0) = 0$, $t(V_1) = 1$ and $t(V_2) = 2$.

A possible alignment is:

```
     0123
U =  BAAA
     | ||
V =  B-AA
```

The edit distance of this alignment is 1, but the timestamps of the symbols in positions 3 and 4 do not match. When we look at an other alignment like

```
     0123
U =  BAAA
     |||
V =  BAA-
```

we see that the edit distance itself does not change, but since the timestamps do match in this case, we can say this alignment is better than the first one. □

In general, let $U = U_0U_1 \ldots U_n$ be an ordered sequence of $n+1$ timestamped atomic actions. The timestamps are denoted by $t(U_i)$, where $0 \leq i < n$; so $t(U_0) \leq t(U_1) \leq \ldots \leq t(U_n)$. Let $\Delta t_i = t(U_{i+1}) - t(U_i)$ be the time difference between two consecutive actions ($0 \leq i < n - 1$). The first step is to combine actions with (exactly) the same timestamp into multisets. The timestamp of such a multiset is naturally defined as that of one of its elements. So we get an ordered sequence of timestamped multisets.

Now we can still further combine consecutive multisets. This can for instance be done if their timestamps are sufficiently close, or if there is some natural urge to combine them, e.g., into years. In the latter case the new timestamp would be the year, in the former the average could be chosen. However, in any case we assume that the timestamps of all sequences under consideration are from a finite set, e.g., years ranging from 1988 to 2008.

We slightly abuse the notation from the first paragraph, and still use Δt_i as the distance between consecutive multisets from a given sequence. Clearly, in many situations these Δt_i's are not evenly distributed. Sometimes they are, and constitute the same linear range for all sequences under consideration. If this is not the case, there are two natural options. The first is to introduce *holes* (empty multisets) to fill in the "missing" multisets. These holes arise in particular when data is missing or absent, either on purpose or by accident.

If holes at the start or the end of a sequence are created, they can or cannot be omitted — with care. The adding of empty multisets is called *expansion*, and the resulting sequence is called *expanded.* The second option is just to omit these holes, leading to shorter sequences of different lengths. These sequence are referred to as *non-expanded.*

As in Example 1, we have to provide a distance value if a multiset (e.g., a hole in the case of expanded sequences) is aligned to a gap (-). Consider, for example, aligning the multiset sequences $\{1, 2\}, \varnothing, \{3, 3\}$ and $\{1, 2\}, \{3\}$. In order to find an optimal layout, we compute the distance between $\varnothing$ and -, and that between $\{3, 3\}$ and -. We have chosen for a fixed large value, i.e., 1. Note, however, that more intricate possibilities do exist.

7.4 Experiments

We shall now describe two datasets on which we shall apply the techniques described in Section 7.3. The datasets are (unfortunately) not publicly available. However, our focus is on showing the possibilities of our methods and not on efficiency issues. These datasets should be viewed as examples of general databases which contain time stamped bags of atoms.

First we provide a brief explanation of the terms used in this section. There are two types of alignment we use, being *global* and *local* alignment as explained in Section 7.2.

Furthermore there are several options to scale the distances between 0 and 1. We implemented two, namely an *absolute* one, which uses a constant (usually the length of the largest sequence in the database) for scaling. Another option is to use *relative* scaling, where we use the length of the largest sequence in the pairwise alignment to scale the distances.

Finally we implemented two ways to deal with the absence of data. As explained in Section 7.3, we can have *expanded* and *non-expanded* sequences.

7.4.1 Criminal careers

The first database consists of approximately one million (anonymised) criminals and their crimes grouped per year. In this particular case, we only know the number of crimes in certain categories per criminal per year. The number of categories is nine.

Example 3. In Table 7.1 we see a table of two criminals and the crimes they committed over the years.

	1999	2000	2001	2002	2003
A	$\{1,2\}$	$\{3\}$	$\{1,1,3\}$	$\{2,3\}$	$\{3\}$
B	$\{3,3\}$	$\varnothing$	$\{3,4\}$	$\{3,3\}$	$\{3,4\}$

Table 7.1: Two criminal careers.

Per criminal, each year can be viewed as a multiset of crimes (1, 2, 3 and 4 are crimes in this example) and the entire criminal career is a sequence of multisets. The second criminal has a hole for the year 2000. □

In order to do analysis on the behaviour of these criminals, we use the previously defined distance measure for sequences of multisets. With this, we can identify trends in criminal behaviour and try to extrapolate future behaviour.

As mentioned before, we try different combinations of alignment, scaling and treatment of holes. To illustrate the difference between expanded and non-expanded sequences, we first give an example.

Example 4. Suppose we have a criminal that has committed the following crimes:

year	1998	1999	2002	2003	2004
crimes	$\{1,2\}$	$\{3,3\}$	$\{2\}$	$\{1,2\}$	$\{2,3\}$

Table 7.2: Example criminal.

We can now use the non-expanded sequence $(\{1,2\},\{3,3\},\{2\},\{1,2\},\{2,3\})$. The years 2000 and 2001 in which the perpetrator was inactive, are simply ignored. The only thing that is preserved is the order in which the multisets of crimes are committed.

We can also use the expanded sequence $(\{1,2\},\{3,3\},\varnothing,\varnothing,\{2\},\{1,2\},\{2,3\})$ for our alignment. The years where there was no activity are explicitly denoted as empty multisets. This way both the order and the time aspect are preserved. □

Depending on the type of analysis, one of these choices is to be preferred. In the case of criminal careers, a hole in the activities may be the result of a sentence (jail). Depending on various assumptions we may or may not want to include the holes in our analysis. In this section we shall see the difference between both choices. Furthermore, no weighing has been applied (meaning that each type of crime is considered equal in severity).

The pictures below are obtained by a dimension reduction algorithm comparable to Multi Dimensional Scaling [5]. This technique iterates over all pairs of data points and adjusts the position of these points on a 2-dimensional plane according to the desired distance between them. It is a kind of competitive neural network which is halted when the position of the points no longer change. The output of this algorithm is an embedding of the points in a 2-dimensional *torus* (see Chapter 3). The usage of a torus improves the embedding for data that is non-flat (i.e., can not be embedded in a normal 2-dimensional plane). So one should be aware that the boundaries of the pictures are identified and that the maximum distance in each of these pictures is actually half of the diagonal. We remark that we just use this technique to give an impression of the results obtained by the usage of our metric. The distances are of importance,

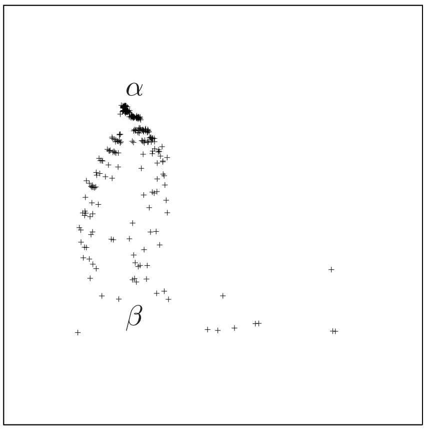

Figure 7.1: Database of criminal careers. Global alignment, absolute scaling, non-expanded careers.

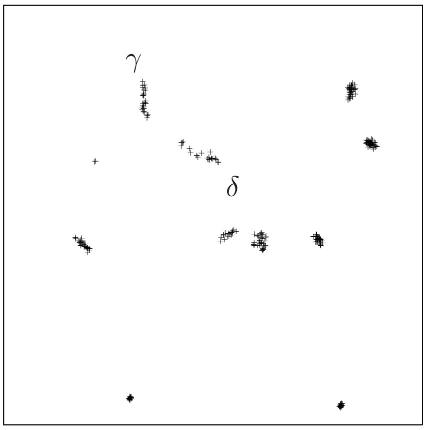

Figure 7.2: Database of criminal careers. Global alignment, relative scaling, non-expanded careers.

not the dimension reduction technique itself. Using other dimension reduction techniques will provide similar results.

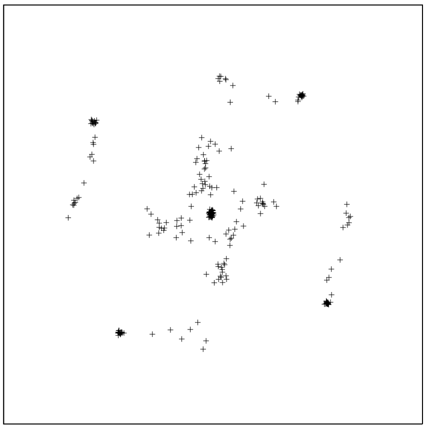

Figure 7.3: Database of criminal careers. Local alignment, absolute scaling, non-expanded careers.

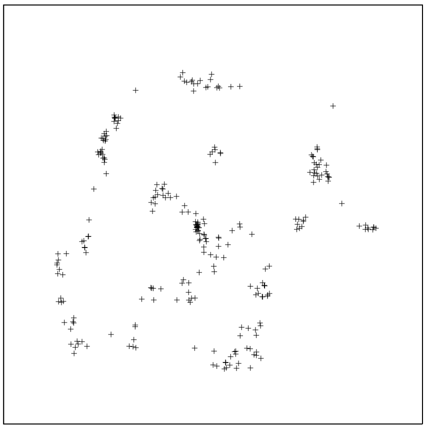

Figure 7.4: Database of criminal careers. Local alignment, relative scaling, non-expanded careers.

In Figures 7.1–7.4 we see a visualisation of 5,000 criminals; all pictures were made with non-expanded careers. In Figure 7.1 we used a global alignment and an absolute scaling factor. In Figure 7.2 we used a global alignment with a relative scaling. Figure 7.3 is made with local alignment and an absolute scaling factor and Figure 7.4 is made with local alignment and a relative scaling.

In the first two of these pictures, there is an emphasis on the length of the careers, which in this case means the amount of years in which a criminal has been active (since all non-active years are ignored). In Figure 7.1 for example, we see a large cluster indicated with the letter α, these are all short careers. As we move from α to β the length of the careers increases.

In Figure 7.2 the clusters indicated with γ and δ contain only long careers. The other clusters have careers of roughly the same size and their elements have been clustered according to the similarity in career.

In the other two pictures, we see something different emerging: the long careers are no longer put into the same cluster, but more emphasis is given to similarity in criminal activity. For example, the central cluster in Figure 7.3 consists mainly of criminals who have committed non-violent crimes involving large sums of money.

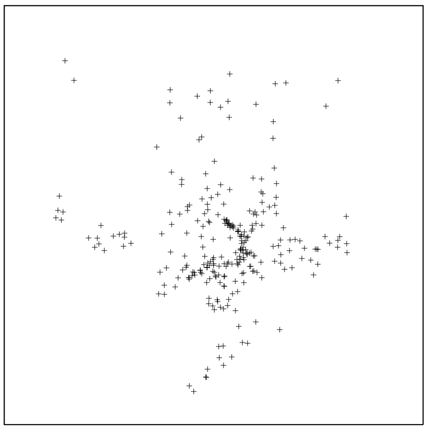

Figure 7.5: Database of criminal careers. Local alignment, relative scaling, expanded careers.

When we use expanded careers as in Figure 7.5, we see a far more scattered image than in the previous ones. The reason for this is that the number of dimensions increases, combined with the fact that the gap penalty is constant. We see that a series of holes in a career will result in a rather big distance, whereas this is not the case with non-expanded careers. On an individual basis, this technique can be useful because it is able to detect an overlap in imprisonment, the clusters, however, will be very small (perhaps indicating criminals who work closely together).

7.4.2 Access logs

As a second dataset we used the access log of a web server. For confidentiality reasons we can not disclose the actual dataset. We looked at the class-A network (i.e., the first number of the IP address) from which a hit in this log originated.

Within a timeframe of sufficient length, these hits will result in a multiset since more than one hit can originate from a specific network. Note that events with different timestamps will be combined into multisets, and that usually holes do not occur — except for periods when the web server was inaccessible, for example.

A sequence of these multisets can be defined in multiple ways. First of all, it is natural to choose a timeframe of constant length. For the choice of the consecutive timeframe however, there are many possibilities. One can choose to have an overlap in the timeframes; this will result in sequences having an overlap by design. Especially for the analysis of access logs, this is a good choice, since when visualising the data, the consecutive sequences will be placed next to each other, giving a very natural view.

Of course one can vary the overlap; this will result in visualising different aspects of the data. A large overlap will result in highly rigid strains of sequences, which only a great similarity with other sequences can break. Decreasing the overlap will result in more loose chains, and more emphasis is given to similarity between sequences based upon their content instead of the chronological order.

In our experiment, we were interested in the class-A network from which a hit in the log originated. We grouped all hits within 10 minutes together into a multiset and made a sequence of multisets of length 10. The subsequent sequence starts 10 minutes after the first one, therefore having an overlap of at least 80% (only the first and last multiset can differ from its predecessor).

For this experiment we took the first twelve hours of the log and another part of twelve hours, exactly one week later. Since all sequences in this test set are of the same length, there is no difference between local and global alignment, there is also no difference between absolute and relative scaling and finally, there is no difference between expanded and non-expanded careers, since in the log there are hardly any time windows of 10 minutes where no activity occurs.

In Figure 7.6 we see that a lot of consecutive sequences (denoted by numbers in this picture) are very close to each other and are neatly arranged in ordered threads. The sequences denoted by the numbers 26 to 31 and 101 to 108 are prominent examples. This behaviour occurs throughout the picture, but is perhaps not clearly visible because of the overlapping numbers.

In Figure 7.7 we see exactly the same visualisation, but we have replaced the numbers with +s for a clearer view. Again, we see the ordered threads, but now we see that some of these threads align next to each other and sometimes even intersect. This indicates a similar behaviour in these consecutive sequences.

We also expected a correlation between sequences one week apart, but except for the pair $(8, 86)$ (in the top cluster of Figure 7.7), no evidence of this could be found.

7.5 Conclusions and further research

We have introduced a new metric that is a natural way of defining a distance between sequences of multisets. We have used the example of criminal careers

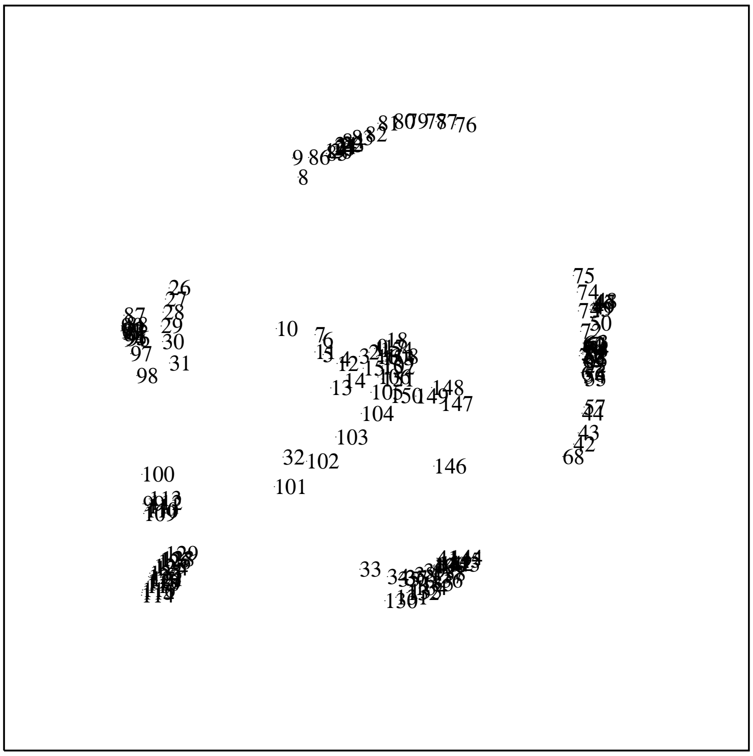

Figure 7.6: Access log, two periods of 12 hours, 1 week apart.

to illustrate this point and to emphasise the flexibility of this new metric. As a second example, we used access logs to show that our metric can be applied in a different domain. We used a visualisation technique to show the merits and possibilities of the method, highlighting several user-controlled features.

In practice, we often do not have real valued timestamps. Transactions are usually grouped together in days, months or even years. This can simplify alignment, since absence of data can be viewed as an empty transaction (one with no elements in it). Alignment of this type of sequences is straightforward.

A useful extension on alignment is the use of variable gap length (also see [47]). In many practical cases, the fact that there is a gap in the alignment is more important than the length of the gap. As long as the gap penalty increases with the length of the gap, and as long as the function governing the gap penalty is concave, the triangle inequality of the edit distance holds. Unfortunately, the efficiency of the alignment algorithm deteriorates. Because there is more data dependency, the time complexity will be $\mathcal{O}(n^3)$ and the space complexity will increase to $\mathcal{O}(n^2)$. Even when working with discrete timestamps, one could consider less penalty for consecutive empty transactions.

We also would like to investigate the usage of weights for the crimes. At

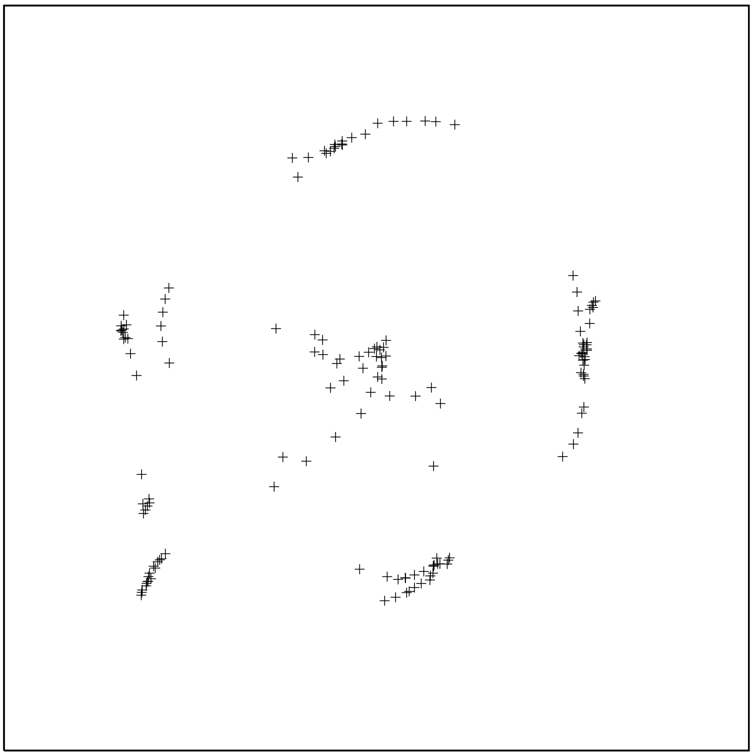

Figure 7.7: Same as Figure 7.6, with numbers replaced with +'s.

present, we have not done so because there is no clear weighing scheme for the categories available yet.

Part III

DNA

Chapter 8

Selection of DNA Markers

Given a genome, i.e., a long string over a fixed finite alphabet, the problem is to find short (dis)similar substrings. This computationally intensive task has many biological applications. We first describe an algorithm to detect substrings that have edit distance to a fixed substring at most equal to a given e. We then propose an algorithm that finds the set of all substrings that have edit distance larger than e to all others. Several applications are given, where attention is paid to practical biological issues such as hairpins and `GC` percentage. An experiment shows the potential of the methods.

8.1 Introduction

The genomes of several species have been available on the internet for a couple of years now. Because of the availability in an electronic format, computers can be used to do experiments that would normally take months or even years. Data mining research incorporates the exploration of raw data, identifying genes, designing primers, searching for differences between individuals or differences between species. Common in many of these techniques is the use of unique strings. Finding these strings is a challenging task and the main topic of this chapter.

There are a number of techniques that benefit from the existence of unique strings, segments of DNA that occur only once in the genome under consideration. Finding primers, i.e., markers for specific positions of the genome, is one of them. Primers can be used to identify genes that are associated with diseases, but also the identification or classification of blood samples can be done with primers. Furthermore unique strings can be used in phylogenetic research to make an ancestral tree of species or populations. Microarrays are also populated with a large number of unique or nearly unique strings. All these problems benefit from the possibility of generating large amounts of *dissimilar* substrings of the genome.

Some work on marker selection has been done in the past, e.g., a pragmatic approach for a relative small number of markers can be found in [70]. In [20] we see an approach that uses an alternative of the distance-based measure: the similarity-measure in neural networks for the analysis of DNA and amino-acids sequences; [56] is an overview of the application of Evolutionary Algorithms in bioinformatics tasks like gene sequence analysis, gene mapping, DNA fragment assembly and so on; [46] presents new techniques for recognizing promoters (E. Coli) given an unlabelled DNA sequence based on feature extraction and a neural network for classification; [58] is a practical application that computes primer pairs for genome scale amplification, and also does extraction of coding sequences and primer pair optimization; [27] is a primer design program that uses fuzzy logic to calculate primer qualities, and also uses suffix trees to enhance the spacial complexity; [37] looks at the multi criteria decision process of primer design and reports that trade-off between deviations from ideal values of all of the criteria can result in a considerable speedup. A practical application for retrieving and assembling gene sequences can be found in [71], where afterwards primer pairs are designed for amplification. For more information about the related subject of multiple alignment see [72], where dot-matrices are viewed as projections of unknown n-dimensional points and alignment for n sequences is done by image reconstruction in an n-dimensional space with noise.

However, our major concern is not the design of individual primers, but rather the creation of large sets of primers with special properties. In particular, these primers should be as dissimilar as possible. For example, in PCR experiments, a lot of primers can be put together to do several experiments at once. Because the amount of primers is very large in comparison with the target DNA, chances are that the primers react with each other instead of with the DNA. To prevent this, it is necessary to generate a set of primers that will not react with each other. This is done by selecting only those primers that are highly dissimilar from all other ones. This is a computationally challenging task, because in principle all strings must be compared to each other.

In Section 8.2 we address the combinatorial background. The major algorithms are described in Section 8.3. The *Proximity Search* algorithm finds all occurrences of a string s in a given set of strings, allowing at most e errors. The *Distance Selection* algorithm produces the strings that are at distance larger than e to all other strings in the set. For both algorithms a *trie* is used for efficiently storing the given set. As distance the so-called *edit distance* is employed, with some possible variations. In Section 8.4 we mention practical applications and their implications, e.g., the hairpin problem that arises from the occurrence of self-similar strings. An extensive experiment is reported in Section 8.5. To make our task more biologically relevant we have put some additional constraints on the markers. Apart from being unique, the markers have to adhere to minimum and maximum thresholds of `GC` content and bonding energy. Another constraint is on the combinatorial nature of the string itself: it should not contain a repeating sequence. Our basic approach does not provide us with the tools to deal with all these additional restrictions. In this section the method is augmented to handle these biological requirements. We conclude in Section 8.6.

8.2 Combinatorial background

In this section we sketch the combinatorial background to illustrate the memory requirements. We have about $3 \cdot 10^9$ nucleotides in the human genome, so we also have about $3 \cdot 10^9$ factors of a given length ℓ (for small ℓ). These huge numbers pose a severe restriction on the necessary data structures.

We determine unique substrings by counting the occurrence of all strings of a given length ℓ. The reason we chose for counting is because using a trie would result in enormous memory usage. If for example we make a trie to count the occurrences of length ℓ, we need 4^ℓ nodes in the lowest level. The number of internal nodes will be: $\sum_{i=0}^{\ell-1} 4^i = \frac{1}{3}(4^\ell - 1)$, which makes the total amount of nodes $\frac{1}{3}(4^{\ell+1} - 1)$. The number of branches will be four times as large, i.e., $\frac{4}{3}(4^{\ell+1} - 1)$. In the case of $\ell = 18$, we need a huge amount of memory only to hold the trie. Assume that we represent each node by means of four pointers. In this case each node consists of four 32-bits (= 4 bytes) pointers, which makes the total amount of used memory $4 \cdot \frac{1}{3}(4^{18+1} - 1) \approx 3 \cdot 10^{11}$ bytes (341 Gigabytes). However, since we can only address 4 Gigabytes of memory with a 32-bits integer, we have to use 64-bits pointers, which makes the amount of memory 683 Gigabytes. Furthermore, we need some sort of counter at the leaves of the trie (the nodes at the lowest level). This will increase the amount of memory by at least another 4^{18} bytes (another 64 Gigabytes).

Let us see how much memory it would take if we store all substrings of length 18 in the complete human genome in a trie. The worst case is when all strings are unique (which is possible, since the number of combinations (4^{18}) exceeds the number of substrings ($3 \cdot 10^9$). Let us assume that up to depth 15 the trie is complete and below this it has $3 \cdot 10^9$ nodes per level. Because the number of nodes in the last couple of levels exceed 2^{32} we have to use 64-bits pointers. Up to level 15 there are $\sum_{i=0}^{\ell+1} 4^i = \frac{1}{3}(4^{l+1} - 1) \approx 3 \cdot 10^9$ nodes, all having four 64-bits pointers. This results in a memory usage of approximately $106 \cdot 10^9$ bytes. The levels 16 and 17 both have $3 \cdot 10^9$ nodes with one pointer, resulting in $48 \cdot 10^9$ bytes. The leaves only need to have a counter. This sums up to a total memory usage of approximately $157 \cdot 10^9$ bytes, which is about 146 Gigabytes. This is why we chose to use direct indexing into an array (which in this case only uses 64 Gigabytes of memory). Details are given in Section 8.5.1. In the next section, however, we can still successfully apply tries, because in that case we already have selected a relatively small subset of the set of all substrings.

8.3 Proximity search and distance selection

In this section we describe two algorithms that find (dis)similar substrings of a given string. We consider the genome, for example that of a human, as a string over a finite alphabet Σ. Usually $\Sigma = \{\mathtt{a}, \mathtt{c}, \mathtt{g}, \mathtt{t}\}$, but the alphabet may also include other symbols when parts of the genome are unknown or marked. For this genome we want to extract a set of substrings of length ℓ that are highly dissimilar to each other. To make this precise we require that the selected strings have edit distance (or Levenshtein distance [45]) larger than e to all others. The values of ℓ and e ($1 \leq e \leq \ell$) are specified beforehand.

The *edit distance* between two strings is defined as the minimum number of simple operations that is needed to transform one string into the other; a simple operation is either a deletion, an insertion or a substitution of a single character. If only substitutions are allowed we obtain the Hamming distance. Determining the edit distance between two strings is an intensive operation (with quadratic complexity), and if we want to calculate the edit distance between all strings, we need quite a lot of calculation, of order $O(\ell^2 m^2)$, where m is the amount of strings.

By inserting the candidate strings in a trie T, we can calculate the edit distance to all the other strings in a more efficient way. So we now assume that the trie T contains a set of substrings of a given string; this set should not be too large. Our basic algorithm, referred to as *Proximity Search*, tries to find a string s in a trie with root T, allowing at most e errors, and proceeds as follows:

```
Proximity $(T, e, s)$ ::
   if MarkedAsEndpoint $(T)$ and $s = \lambda$ then
      Visit $(T)$;
   else
      for $\sigma$ in $\Sigma$ do
         if Exists $(T.\sigma)$ then
            if Head $(s) = \sigma$ then
               Proximity $(T.\sigma, e,$ Tail $(s))$;
            else if $e > 0$ then
               Proximity $(T.\sigma, e - 1,$ Tail $(s))$;
            if $e > 0$ then
               Proximity $(T.\sigma, e - 1, s)$;   % insertion
      Proximity $(T, e - 1,$ Tail $(s))$;          % deletion
```

Proximity Search.

The function *Exists* (T) returns true if and only if the node (pointer) T exists. The function *MarkedAsEndpoint* (T) returns true if and only if the root of T is special, i.e., marked as the end of a word. The function *Visit* (T) visits the root node, and does the necessary bookkeeping, e.g., outputs the path and/or marks the last node. The functions *Head* (s) and *Tail* (s) return the head and the tail of the string s, respectively. Finally, $T.\sigma$ is the pointer that leads to the child that corresponds with σ; and λ denotes the empty string.

Essentially we traverse the trie in a normal depth first way, except for the fact that we allow a number of errors. If we have read a character from the trie, we compare it with the first character of the string to be matched, if it matches, we do nothing, if it doesn't match, we decrease the amount of allowed errors. If the amount of allowed errors is less than zero, we exit the function. If we encounter the end of a word in the trie (referred to as an "endpoint"), and there are no characters left in the string, we have found a match. The end of a word is marked with a flag in the node. This node does not need to be a leaf, so we are able to compare words of different length. A similar approach can be found in [2].

The kind of trie is of no importance; we can either use a binary trie or a trie that has a maximum branching factor of that of the size of the alphabet (although it is of course advisable to use a branching factor that is a divisor of the size of the alphabet in the case that the branching factor is smaller). In our illustration, we use the latter.

We describe the algorithm through two examples, where the second one also allows for insertions and deletions. In both examples we assume that all nodes are marked as endpoints. In the first example we do not use the last three lines of our algorithm (responsible for insertions and deletions); effectively, we "visit" all strings at at most a given Hamming distance from the original one. In the examples we take $\Sigma = \{\texttt{a}, \texttt{b}\}$, and we always fill the trie with *all* combinations in Σ^* of the appropriate length.

Example 1 (Proximity Search, parallel alignment): In Fig. 8.1 we see a trie with all combinations of 3 letters over the alphabet $\{\texttt{a}, \texttt{b}\}$; all nodes in the trie are marked as endpoints. The word `abb` is highlighted as a path in the trie. If we want to find all strings of length 3 at Hamming distance 1 from the word `abb` we use our recursive algorithm. The black dots denote the strings that are found, that is, all strings with at most one mismatch. Notice that in this example no insertions or deletions are allowed.

So the outcome of our algorithm consists of `aab`, `aba`, `abb` and `bbb`. Because of the nature of our algorithm a lot of branches are not traversed, for example the subtrie under the path `ba` is skipped.

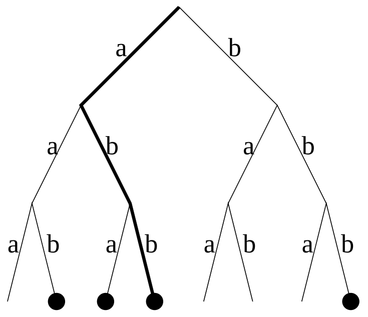

Figure 8.1: All words in $\{\texttt{a}, \texttt{b}\}^*$ at at most Hamming distance 1 from `abb`

The complexity of this algorithm, where we do not allow for insertions and deletions, is $(\ell k)^{e+1}$ where ℓ is the length of the string we are searching for, k is the size of the alphabet and e is the number of errors allowed. We assume that the trie is full. If this is not the case, the complexity is lower than this, and k must be replaced by some appropriate average.

Example 2 (Proximity Search, parallel alignment with insertions and dele-

tions): In Fig. 8.2 we see all strings that are at most at edit distance 1 from the word `abb`, where in this case we do allow insertions or deletions. Again, all nodes in the trie are marked as endpoints.

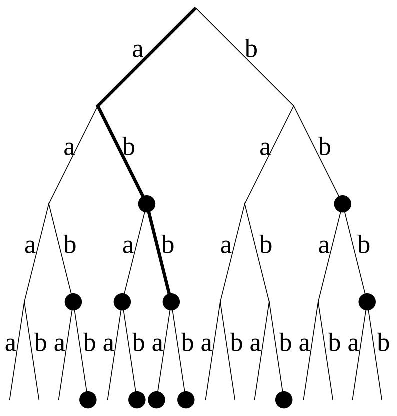

Figure 8.2: All words in $\{\texttt{a}, \texttt{b}\}^*$ at at most edit distance 1 from `abb`

In this case the complexity of the algorithm is $(3\ell k)^{e+1}$ The extra 3 is the result of the three times we enter the recursion per node. Henceforth, we shall refer to this algorithm as *Proximity Search*.

However, we want to know which strings are highly dissimilar. To do that we extend our algorithm in the following way. Again, let T be a trie containing a set of substrings of a given string, where the endpoints are marked as such. Initially, all these nodes are marked as "good". Let $L = L(T)$ be the set of all endpoints in the trie. First we do a Proximity Search for each string $s \in L$. If it finds more than one string, mark all these strings (including s itself) as "bad". If we are about to start a search for a string that is marked as "bad", skip it. The reason we can skip this search is because the distance between two strings is symmetric, and therefore we shall encounter all strings that would be marked as "bad" at a later stage in this part of the algorithm. This is an optimization which makes the algorithm run faster as it progresses. When this part of the algorithm is finished, we traverse the trie one more time and extract all strings that are not marked as "bad". The result is a set of strings that have a distance to each other string that exceeds a preset distance e. We shall refer to this algorithm as *Distance Selection*: it marks all strings in T as "bad" that have distance $\leq e$ to some other string in T (except for itself):

```
$DistanceSelection\ (T, L, e)$ ::
    for $s$ in $L$ do
        if not $IsBad\ (s, T)$ then
            $R$ := empty trie;
            $Proximity\ (T, e, s)$;
            if $ManyEndPoints\ (R)$ then
                $MarkAsBad\ (T, R)$;
```

Proximity Search.

The function *IsBad* (s, T) returns true if and only if the node at the end of the path s is marked as "bad". The function *ManyEndPoints* (T) returns true if and only if the trie T contains more than one node that is marked as an endpoint. The function *MarkAsBad* (T, R) marks all strings that occur in R as well as in T as "bad" in trie T. In this case, we let *Visit* (T) (from the function *Proximity*) add a string (with last node marked as endpoint) to an initially empty trie R of solutions it has encountered.

Example 3 (Distance Selection): Suppose the genome of a certain species is built up out of the letters from the alphabet $\{\texttt{a}, \texttt{b}\}$, for example `abaababaabaaba`. We consider all substrings of length $\ell = 5$. There are only four unique substrings in this genome, the other two (`abaab` and `baaba`) both occur three times. The four unique strings are put into a trie as shown in Fig. 8.3, where the leaves are the only endpoints. The first child is always an `a` and the second one is always a `b`. After the application of the sequential Distance Selection algorithm (with

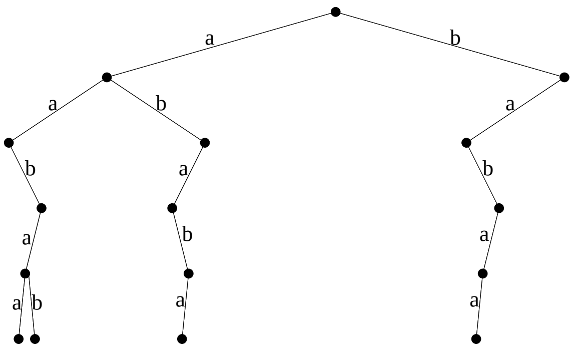

Figure 8.3: Trie of unique strings

distance $e = 1$) only one string is left: the string `ababa`.

The Distance Selection algorithm can be improved by propagating the "bad" label to the parent if all children are marked as "bad". If we find a string in a subtrie that is marked as "bad", we do not have to search the rest of the subtrie because all strings are already marked. This optimization can improve the speed of the Distance Selection algorithm a lot, especially if e is large.

To make this algorithm practical for DNA marker selection, we have to perform some adjustments. Most of them are necessary to make the initial trie as small as possible. If we don't do that, the trie will most likely become too large to handle, cf. Section 8.2. This has also been done in Example 3. However, this means that the selected strings are far away from each other, but not necessarily far away from all other substrings of the original genome. We shall give the different steps in the next section.

For the purposes sketched in Example 1 and Example 2 we can also use a directed acyclic graph (DAG) for memory efficiency, but in our examples we use a trie. This is in the first place because the examples are more clear and secondly because for the Distance Selection algorithm a DAG can not be used. This is because in a DAG more than one path can lead to a node that is marked as the end of a word. Marking such a node as "bad" could disqualify more than one word in the trie and could lead to undesirable results.

8.4 Applications

We indicate some biological applications for the algorithms proposed in the previous section.

8.4.1 Primer pair selection

Primers are used in techniques like the *Polymerase Chain Reaction* (PCR) [17] and *Multiplex Ligation-Dependent Probe Amplification* (MLPA) [64].

For primer selection, we use the following scheme (not necessarily in this order):

1. Determine all unique strings of length ℓ.
2. Filter for simple repeats.
3. Filter on `GC`-count.
4. Filter on bonding energy.
5. Put the remaining strings in a trie and use the Distance Selection algorithm to select the best strings.

The resulting strings are guaranteed to be highly dissimilar to all other strings in the resulting set. In general it is not true that they are highly dissimilar to all strings in the original set. Because these strings are used for primers, the main concern is that the primers do not stick together in real life experiments. The fact that they may bind to other parts of the genome is of less concern. The reason for that is because primers are used in pairs, and the chance of a primer pair having the same problems, is neglectable.

8.4.2 DNA marker selection

For DNA markers in general the story is somewhat different. In contrast with primers, general DNA markers are not used in pairs. Therefore the emphasis must be more on bonding to the DNA in the genome. Thus the edit distance between the markers becomes as important as the edit distance between the markers and all other substrings in the genome. Therefore we use a slightly adapted technique to select them; essentially it is the same as the one we used to select primers, except in the last step, where we use the Distance Selection algorithm in a different way.

We again extract all strings of length ℓ from the genome and test them to the trie with the Distance Selection algorithm (instead of the strings that are in the subset (and in the trie)).

8.4.3 Other applications

With a couple of alterations, we can use the same techniques to check primers for the formation of *hairpins* or *primer-dimers.* A hairpin in this context means that a primer is its own reverse complement, or at least up to a certain degree. The reverse complement is obtained by reversing the string and replacing every nucleotide with its complement (`A`$\leftrightarrow$`T`, `C`$\leftrightarrow$`G`). If a primer's head is the reverse complement of its tail, chances are that the primer folds and binds to itself. This is what we call a hairpin. A primer-dimer is a similar structure, where two primers stick together. There are two sorts of primer-dimers: the ones that stick with the heads to each other, and the ones that stick with the tails. Both hairpins and primer-dimers are undesirable structures that should be removed from a set of primers.

Although many programs can check things like this, we use the structure of the trie again to check every combination at once. Another advantage is that we can allow for one or more errors in the formation of hairpins or primer-dimers.

For the normal formation of primer-dimers, we first make the reverse complement of the primer we are checking. The reason for this is that using the reverse complement does not have any impact on the trie. Otherwise we would have to use the normal reverse and check if `a` matches to `t` instead of itself. With this approach, we don't have to worry about hairpins any more, because the algorithm above has already filtered them out. Take for example the primer in Fig. 8.4; this is a primer that has formed a hairpin.

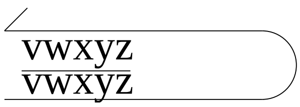

Figure 8.4: A primer in hairpin configuration

But in the algorithm, we have also tested the primer to itself in the configuration shown in Fig. 8.5, so there is no need to do it again.

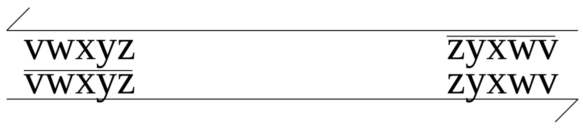

Figure 8.5: Two identical primers that could form a hairpin

8.5 Experiments

As an illustration of the methods described in this chapter, the pairwise alignment of many primers using a trie, we have done an experiment on a part of the human genome. The reason we only took a part is that in this case the numbers are somewhat easier to interpret.

In this experiment, we select pairs of primers as they are commonly used in pairs in real life experiments like PCR. Rather than just selecting a set of markers that are only tested for their uniqueness, we have chosen to implement also some restrictions that have a biochemical motivation. Of course, restricting the number of markers before computing their edit distance will reduce the computational effort for that last step.

The computer on which we ran our experiment is a Pentium IV based machine at 1,8 GHz with 256 Megabytes of internal memory. The operating system is SUSE Linux 9.2 with a 2.6.8 kernel.

We summarize the results of this experiment below. The last step of the experiment was performed using the algorithmic technique sketched before. In the remainder of the section we explain some additional details of our experiment, involving the selection of unique primers and filtering the primers based on combinatorial and biochemical properties.

The results are as follows:

Finding markers We took the human chromosome 1 from [69] (version June 26, 2006) which consists of 247,249,719 basepairs. In the first step of our analysis we extracted all unique strings of length 16. There were 123,685,514 of them (out of a grand total of 247,249,703). We need exactly 2 Gigabytes of memory to count all occurrences of length 16 and we ran our experiment on a computer with 256 Megabytes of memory. To be on the safe side, we chose the amount of memory available for our program to be 128 Megabytes, so we had to make 16 passes over our input data. The program was run as root to ensure memory locking. This process took 12 minutes and 49 seconds.

Filtering repeats Simple repeats are strings of low complexity, and in some sense not of much interest to biologists. We denote a repeat in the following way. By $(\underbrace{x \ldots x}_{s})^r$ we denote a repetition of r copies of a string of length s. So for instance, $(xxx)^2$ means a string of length 6 consisting of two equal halves of length 3, like *abbabb*; note that the character x may represent different characters. In this notation, we filtered out all of the following simple repeats: $(x)^5$, $(xx)^5$, $(xxx)^4$, $(xxxx)^3$, $(xxxxxxx)^2$, $(xxxxx)^3$, $(xxxxxxxx)^2$ and $(xxxxxx)^3$. This resulted in a set with 115,695,993 elements. This program was also run as root to ensure memory locking and took 2 minutes and 34 seconds.

`GC` content and temperature After that, we selected primer pairs based on a `GC` count between 20 % and 80 %, and a melting temperature between 60 ℃ and 63 ℃. The internal spacing of the primers was between 480 and 520 basepairs and the spacing between the primer pairs had a minimum of 9,800 basepairs. This resulted in 41,565 primers. This process took 28 seconds.

Edit distance After the Distance Selection (with respect to the formation of primer-dimers and normal alignment) with minimum edit distance 2, there were 35,781 primers left. This process took took 8 minutes and 47 seconds. If we set the edit distance to 3 only 7,530 primers remained.

This experiment shows that Distance Selection can make an enormous difference with respect to primer selection.

8.5.1 Finding markers: Determining unique substrings

We have chosen to count occurrences of length 16 strings, in a rather direct way, making multiple passes over the input data. At each pass we count the occurrences of a "block" of possible length 16 segments and write the result to disk afterwards. This algorithm is designed to use linear disk access as much as possible. All random access is done in memory only.

First, we convert the DNA data from ASCII to binary. This has three advantages. As the data is compressed by a factor of 4 (using only 2 bits per nucleotide), we need to use only one-fourth of the original memory. As a direct consequence, we can read data at 4 times the speed, for each pass. Finally, calculation with base-4 numbers is trivial compared to string operations with respect to the amount of calculation. The number of passes is directly related to the amount of data that fits in the memory.

Assigning the right two bit code to each letter, we can even take advantage of binary operators like the NOT operator, if we choose `a` and `t` as each others complement in the encoding (as they are in nature), as well as `c` and `g`. Hence we take the order `a`, `c`, `g`, `t` (which happens to be the alphabetical one), and code these nucleotides as 00, 01, 10, and 11 in binary.

Since we are interested in strings of length 16 we keep track of the last 16 nucleotides. When we read a new nucleotide, the string is shifted to the right by two bits, and the new two bits are shifted in (at the least significant end). This is a very inexpensive method to keep track of the last read string.

Of course there are some additional complications in preprocessing the data. The first one is that the data (as available via [69]) does not consist of only four letters. It has an alphabet of 10 letters. These are the normal letters, $\{\mathtt{a}, \mathtt{c}, \mathtt{g}, \mathtt{t}\}$, the normal letters written in capital, $\{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$, and the letters $\{\mathtt{n}, \mathtt{N}\}$. The capitals denote repeating segments or sequences of "low complexity" in the DNA (pieces of DNA that have a repetition with a period of 12 or less), which are biologically less interesting regions. The letters `n` and `N` denote the absence of data. We have decided to look for markers in the lower case segments of {`a`, `c`, `g`, `t`} only.

However, we cannot simply erase the other letters. For example, suppose we convert the string `ATATATttNttaatNnn` to `ttttaat`, then we will have found a string which is not part of the original DNA. One way of circumventing this problem is to remove all capitals and the letter `n`, and by keeping track of when in the output file we need to re-calculate the current number. We put these offsets in a separate table. In this way we avoid having to use additional escape sequences in the binary code of the DNA, making the code shorter and faster to read.

In our example we get two tables, one containing the binary equivalent of `tttta` and one containing the offsets 6, 2, 1, 5 and 3. To decode these tables, we take the offsets table and first write 6 `n`'s, then we decode 2 nucleotides, then we write 1 `n` and 5 more nucleotides. Finally we write 3 more `n`'s, which results in `nnnnnnttnttaatnnn`, the same string we had before, except all the capitals have been converted to `n`'s.

Our input data stores only a single strand of the DNA, but in practice one considers both strands. Each time we find the occurrence of a string, we also have to take into account the reverse complement of the string, as this is also present in the DNA. Rather than counting both strings however, we have chosen to take one of the two as a representative of the pair, incrementing only the counter of the smallest string (lexicographically ordered). Comparing strings can be done with the standard `<` operator, another advantage of the binary recoding. Of course, we keep track of the reverse complement on the fly, shifting and adding new bits (but in the other direction). Again this is a rather inexpensive operation.

In an additional experiment we have selected unique primers not only for the first chromosome, but also for the full genome. As an illustration, in Fig. 8.6 we see the number of occurrences of unique strings of length 12 for each consecutive series of 100,000 basepairs on the full human genome. The vertical dotted lines denote the chromosome boundaries; these are ordered as follows: 1 to 22, then X, Y, and finally the mitochondrial DNA. This mitochondrial DNA is so small that it does not show up in these graphs. The histogram shows the "hotspots" where many markers are found. The white bands (at offsets 1,300 and 15,900 for example) are due to unknown or unstable DNA. It is missing in our input (a large series of `n`'s).

8.5.2 Filtering out simple repeats

Strings containing short repetitive substrings are not of much interest to biologists, and that is why we filter out such "simple repeats".

This implies that we have to perform a certain pattern matching on the markers found. Since the markers all have a fixed length we do not have to

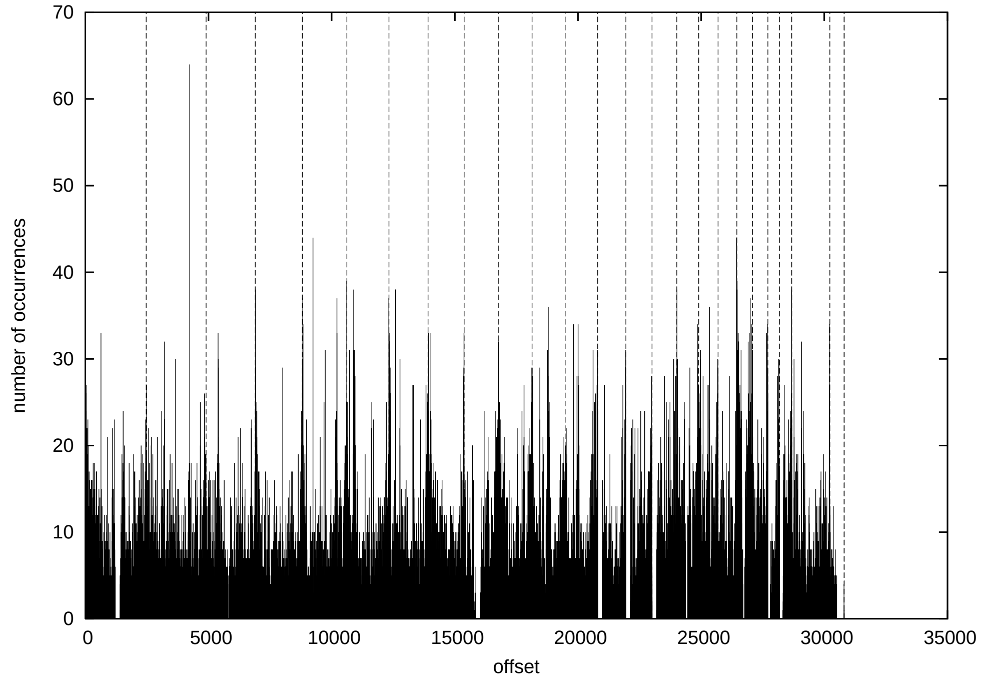

Figure 8.6: Occurrence of unique strings in the human genome

search for repetitions of unbounded length. Instead we can determine the unwanted substrings beforehand and search for them in the markers. We have stored the simple repeats in a trie, using the technique of Aho-Corasick [1], which is basically Knuth-Morris-Pratt [38] pattern matching, but applied to a set of patterns rather than a single pattern. This makes it possible to scan for all forbidden repeats in parallel.

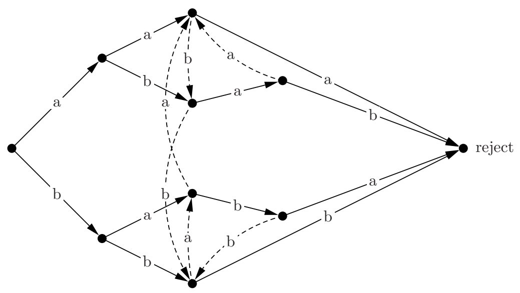

Figure 8.7: Wired repetition trie

After computing all simple repeats they are stored in a trie. As an illustration, Fig. 8.7 shows a trie to filter out the repetitions $(x)^3$ and $(xx)^2$ over the two letter alphabet $\{\mathtt{a},\mathtt{b}\}$, i.e., `aaa`, `bbb`, `abab` and `baba`. Note that `aaaa` and `bbbb` are already excluded. Whenever we reach the node at the right, we have found a simple repetition and we can disqualify that part of the input. This trie is “wired” by dashed edges that are added in a preprocessing phase, asserting that we always keep track of the longest prefix of one of the unwanted strings while scanning the marker. As usual with the Aho-Corasick method, we have to take special care when one of the unwanted repeats is a substring of another one, like `aa` and `baabaa` for the repeats $(x)^2$ and $(xxx)^2$.

8.5.3 `GC` content and temperature

The practical use of primers in experiments is governed by bounds on the `GC` content of the primer and its melting temperature, and tools like Primer3 [60] allow the user to specify bounds. Keeping track of the `GC`-percentage is an easy task if we keep track of the symbols that are shifted in and out of the current substring of length 16, while reading the data. So when the least significant nucleotide of the reverse complement is `c` or `g`, we decrease the `GC`-count (it is shifted out in the next step). Then we shift in the new nucleotide, and if the least significant nucleotide on the original strand is `c` or `g`, we increase the `GC`-count. We actually use both the string itself and its reverse complement, because it is faster to extract the least significant bits.

A formula to calculate the melting temperature of a piece of DNA is given in [7]. We chose to implement this in an algorithm acting like the one calculating the `GC`-percentage.

The formula for the temperature is mainly based on the successive pairs occurring in the fragment, each of the pairs contributing to the bonding energy of the primer. We keep track of the two last nucleotides on the reverse complement (the first ones to be shifted out), and the next ones shifted in, and we can update the melting temperature by a simple table lookup. Internally we use integers to represent the temperature, to avoid rounding errors.

8.6 Conclusions and further research

We conclude that we can select primers and DNA markers that have a high chance of performing well in real laboratory experiments and microarrays.

As can be seen from our experiment in Section 8.5, most primers are disqualified because of the `GC` content and bonding energy, so intuitively one might think that this should be the first step in our analysis. This might be a good solution if the `GC` content and bonding energy are known in advance, which is often not the case in practice. With the current solution we can still tinker with the options, and that is the reason that the focus is primarily on unique strings.

Instead of working with tries we would like to experiment with *Compact Directed Acyclic Word Graphs* (CDAWG) [33]; this datastructure behaves like a trie (for our purposes), but is more compact. Furthermore the online algorithm stated in [33] is very attractive for these kinds of problems.

At present, the output of the Distance Selection algorithm is not used iteratively. We could for example still use the other primer in a primer pair if one primer is discarded. Since all these ideas are still very new, lots of experiments should be done.

Chapter 9

Substring Differences in Genomes

In this chapter, we introduce a new way of determining the difference between full genomes of different species, based upon the occurrence of small substrings in both genomes. Basically we count the number of occurrences of all substrings of a certain length and use that to determine to what extent two genomes are alike. Based on these numbers several difference measures can be defined, e.g., a Euclidean distance in the vector space that has the same dimension as the number of possible substrings of a certain length, a multiset distance, or other measures. Each of these measures can be applied for phylogenetic tree generation. We also pay attention to some visualisations and several statistics.

9.1 Introduction

Determining how one species relates to the other can be done in many ways. One of the many techniques is to look at the DNA. At this moment, many genomes can be downloaded from the internet [69], although not all genomes are complete yet. Also the genomes of many individuals of a given species become publicly available. In this chapter, we do not look for genes or markers in the genome, or any other annotation whatsoever, but just at the occurrence of substrings. Therefore the techniques described here can be used for a number of other problems, ranging from chromosome resemblance to the detection of plagiarism or document similarities for search engines.

In Section 9.2, we will describe a way to compare two long strings by counting rare substrings. This seemingly simple approach is highly non-trivial because the number of substrings is enormous. We have tried a number of ways to do the computation efficiently, like caching, using trees and hash tables, but all these methods need far too much memory. Note that if the substrings are small, these methods are just fine and solve the problem in linear time, but we deal with rather large substrings (length 14 and above). The only way to count large substrings was to use an exponential (in memory) approach, but with the adaptation that when the amount of available memory is not enough, we make multiple passes over the data; in each pass we search for all substrings with a pre-set prefix.

The count of all substrings is reintegrated in the genome data, so we know the exact position. Unfortunately, the DNA at a certain position in the genome of one species does not have to correspond to the DNA at the same position of the genome of another species. This annotated genome can be used in a large variety of applications. One of them, (using the co-occurrence of substrings as a metric) we shall discuss in this chapter.

We will introduce some elementary statistics and visualisations in Section 9.3, a distance measure for the comparison of two species in Section 9.4 and based on this the generation of phylogenetic trees in Section 9.5. In Section 9.6 we introduce a distance measure for multisets which we apply to our data. We conclude in Section 9.7.

9.2 Determining rare factors

The discovery of (almost) unique substrings of a given length n in a genome is not as trivial as it might seem. We use the following strategy:

- Convert the entire genome to a binary sequence, using a suitable encoding scheme.
- Use a sliding window to get all subsequences of length n.
- Count these subsequences and remember in which part of the genome each of the subsequences were found, by remembering the starting points.

9.2.1 Conversion

In Table 9.1 we give one of the possible binary encodings for nucleotides. However, these values were not chosen at random. Note that the complementary letters are also complementary in the binary encoding, i.e., A and T are complementary, and so are 00 and 11. There is an advantage in such a scheme, because the calculation of the complement of such a string is a very simple and fast operation. We shall see further on why this is important.

nucleotide	A	C	G	T
encoding	00	01	10	11

Table 9.1: Binary encoding of the nucleotides

Generally speaking, using a binary encoding scheme is beneficial because some operations can be done in parallel (the complement of sixteen nucleotides can be calculated with one operation on a 32-bit machine) and a binary encoding uses only one fourth of the memory it would usually take.

9.2.2 Sliding window

After we have converted the DNA to binary data, we use a sliding window to get all subsequences of a certain length. If we are searching for substrings of length n, we make an array of size 4^n. Each time we move the window we get another position in the array, and we simply increase the value of that array element. An advantage of this sliding window is that we only have to read one value to generate the next index. We simply shift the old index to the left and concatenate the newly read value to the end of the sequence.

The size of the array is the main difficulty in this approach. To give an indication: for $n = 16$, we have to make an array with 4^{16} entries, and if each entry consists of one byte, the array will be 4 Gigabytes large. The size of the input files have no influence on this array.

To make sure that all random access is done in memory, we use a memory locked part of the main memory. In practice, this will probably be smaller than the amount of required memory. Therefore, we make multiple passes over our input where in each pass a prefix is fixed. For example, if we require 4 Gibabytes of memory and we can only lock 2 Gigabytes, we make two passes over the input. In the first pass the fist bit is fixed and has the value 0. This means that in the first pass all substrings are counted that start with an `A` or `C`. This implies that the amount of physical memory used must always be a power of 2.

For substrings of length 18 and below, this is probably the most efficient data structure to work with, this is because the genome of most species is so large that most (if not all) combinations occur. For the human genome we know that about 95% of the substrings of length 18 are unique. If we put this data in a space-efficient data structure like a trie, we need about 650 Gigabytes (twice as much as one might expect, but this is because we need to use 64 bits pointers). If we use a PATRICIA tree [52], we still need about 100 Gigabytes (we base this assumption on the fact that the branching factor of the tree is very high).

The reason that we chose to use strings with a length between 12 and 18, is because using larger strings than 18 are not needed (most of the substrings of this length are unique) and below 12 there are too few unique substrings.

9.2.3 Counting

Since DNA is double stranded, we can not simply count all substrings, because the reverse complement of a string is essentially the same as the original string. Therefore we look at the other index as well. One of these two indexes has the smallest numerical value, and this one will be used as representative of the pair. Keeping track of the reverse complement is just as easy as keeping track of the original string. The difference is that we shift the old index to the right and insert the inverted newly read value to the beginning of the sequence. If we now encounter either of those sequences, we increase the value of the sequence with the lowest binary representation. This way both the sequence and its reverse complement are mapped to the same position in the counting table.

This results in a table where every possible subsequence is counted, however due to physical limitations (the size of the table in memory and the size of the annotated genomes that will be written to disk afterwards), we chose to count up to three, so if there is a three in this table, it means the corresponding substring is present three or more times. To be precise, each nibble in the counting table is used as an entry. This reduces the amount of memory for the table by a factor of two, but it complicates writing and reading in the table slightly; for an even substring length (ending in `A` or `G`) we do an AND with the value $0 \times 0F$ and in the other case we do a SHIFT RIGHT of 4 bits.

When counting the substrings in two species, we use the first half of the value of each element in the counting table for species A and the second half for species B (leaving only two bits for each species). This is quite convenient since we can look up the number of occurrences of a certain substring in both species at once.

9.3 Elementary statistics and visualisations

Now we can calculate a number of elementary statistics and do some visualisations. For example, we can give the number of unique strings of a given length and even the position of these strings. As a first example, in Table 9.2 the number of unique substrings of a certain length is shown. Note that each unique string of length n automatically accounts for $(m-n)+1$ unique strings of length m, if $m > n$ (a string of length n is a substring of $(m-n)+1$ strings of length m).

size	11	12	13	14	15	16
Human	210	47,668	1,335,256	15,412,176	85,793,791	346,600,204
Chimp	300	62,149	1,509,471	16,636,054	87,029,038	346,319,725

Table 9.2: Number of unique substrings in Human and Chimpanzee

Of course, this does not account for all unique strings of high length as can be seen in Table 9.2. The values grow far more rapid than the ones dictated by the formula above. Statistically, given a random string, the larger the length of the substring, the higher the chance is that a given substring is unique. This is the reason we find far more unique strings of higher length.

In Figure 9.1 we see the occurrence of unique strings in a Human. We visualise the number of unique strings of length 12, for each consecutive series of 100,000 base pairs. The vertical dotted lines denote the chromosome boundaries; these are ordered as follows: 1 to 22, then X, Y, and finally the mitochondrial DNA. This mitochondrial DNA is so small that it does not show up in these graphs. The white bands located at offset 1300 and 15900, for example, are due to unsampled or highly unstable DNA, and in either case it is missing from our input.

In Figure 9.2 we only plot the occurrences above 15. In Figure 9.3 we have plotted the number of occurrences of strings that are unique in the human

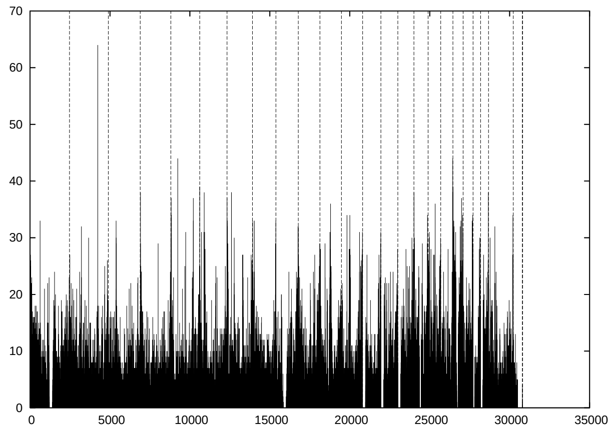

Figure 9.1: Occurrence of unique strings in Human

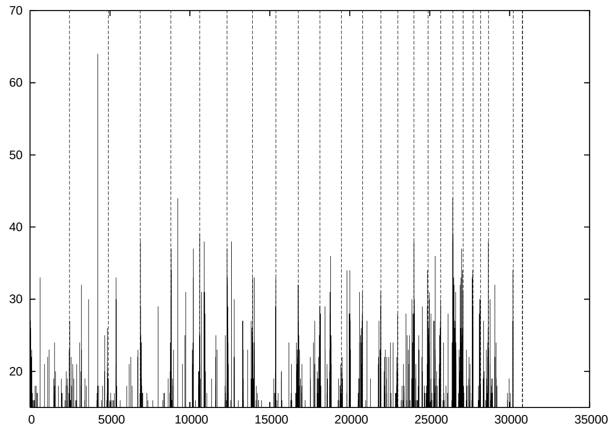

Figure 9.2: Occurrence of unique strings (length above 15) in Human

genome and not present in the genome of a chimp. Figure 9.4 shows unique strings for a chimp. The chromosomes are ordered 1 to 22, then Un, X, Y, and finally the mitochondrial DNA. This can for example be used to select markers to put on a *microarray* [63] to make a distinction between two (or more) species. In Figure 9.3 and 9.5, we see where these markers can be found. Another application is the selection of *primers* [23], commonly used in techniques like *Multiplex Ligation-dependent Probe Amplification* (MLPA) [64] and *Polymerase Chain Reaction*, or PCR [17]. We see the regions where the number of primers are abundant in Figure 9.2 for the human and in Figure 9.4 for the chimp.

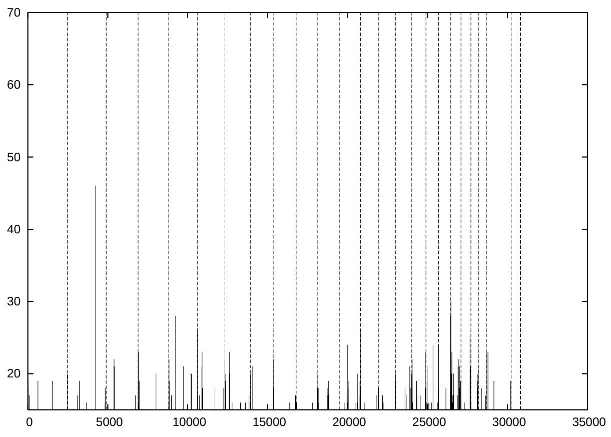

Figure 9.3: Occurrence of unique strings present in Human and not in Chimp

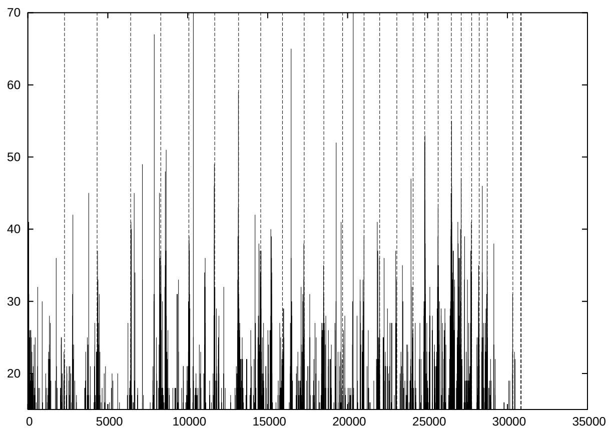

Figure 9.4: Occurrence of unique strings in Chimp

9.4 Distances and weights

After the substrings of length n have been counted, we make a matrix where two species are represented by counting the number of strings that occur a times in species A and b times in species B for $0 \leq a, b \leq 3$. This is a method that loses a lot of information, but we have a very small matrix left to work with and as we shall see further on, the information in this matrix is still sufficient to make a difference between species. The 4×4 matrix M, referred to as the *counting*

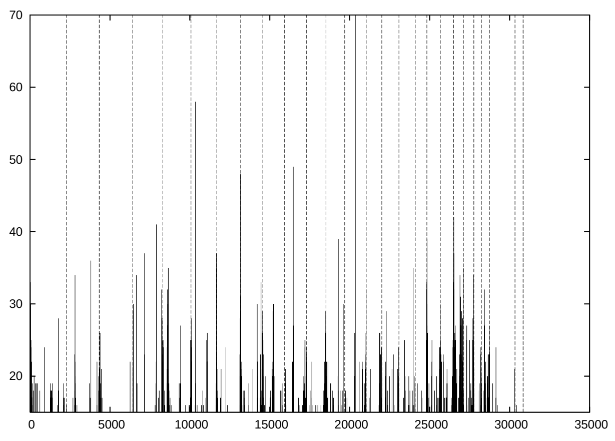

Figure 9.5: Occurrence of unique strings present in Chimp and not in Human

matrix, contains the following data:

$$M = M(A,B) = (m_{i,j}) = \begin{pmatrix} m_{0,0} & \underline{m_{0,1}} & \underline{m_{0,2}} & \underline{m_{0,3}} \\ \underline{m_{1,0}} & m_{1,1} & \underline{m_{1,2}} & \underline{m_{1,3}} \\ \underline{m_{2,0}} & \underline{m_{2,1}} & m_{2,2} & \underline{m_{2,3}} \\ \underline{m_{3,0}} & \underline{m_{3,1}} & \underline{m_{3,2}} & m_{3,3} \end{pmatrix}$$

Where $m_{i,j}$ denotes how many substrings are present i times in species A and j times in species B. As mentioned before, we only count up to three, so, e.g., the element $m_{1,3}$ is the amount of substrings that are present once in species A and three or more times in species B. All elements that contribute to the difference are underlined to indicate the relevant elements of the matrix.

We want to use the following distance formula:

$$\text{dist}(S,T) = \frac{|S \backslash T| + |T \backslash S|}{|S \cup T|}, \qquad (9.1)$$

where S and T are sets. We divide the symmetrical difference of set S and T by the maximum value of the numerator. If both S and T are the empty set, we let $\text{dist}(S,T) = 0$. The reason we choose for this particular distance measure is because it takes the sizes of the sets into account (also see [24]). To adjust this formula to work with our matrix, we have to rewrite it as follows (for species A and B):

$$\text{dist}(A,B) = \frac{\sum_{i=1}^{3}(m_{0,i} + m_{i,0})}{4^{\ell} - m_{0,0}}, \qquad (9.2)$$

where $M = M(A,B)$ is the counting matrix of the pair (A,B) and $4^{\ell} - m_{0,0}$ is the total number of substrings that occur in at least one of A and B.

One of the shortcomings of this distance is that only the absolute differences are used, i.e., only the substrings present in one of the species. Another shortcoming is that all differences are weighted equally, although it is perhaps reasonable to assume that a substring that is present once has less significance than one that is present more than three times.

To compensate for these shortcomings, we use a *weighting matrix* W:

$$W = (w_{i,j}) = \begin{pmatrix} 0 & \alpha_3 & \alpha_4 & \alpha_5 \\ \alpha_3 & 0 & \alpha_0 & \alpha_2 \\ \alpha_4 & \alpha_0 & 0 & \alpha_1 \\ \alpha_5 & \alpha_2 & \alpha_1 & 0 \end{pmatrix}$$

The values of $\alpha_0, \dots, \alpha_5$ are weights applied to the matrix M. They are ordered in ascending order of significance, e.g., we assume that the value of $m_{2,1}$ is less significant than $m_{3,2}$, and therefore we should set α_0 to a lower value than α_1. We base this assumption on the fact that if a substring is present zero times in species A and two times in species B, this is a more significant difference than once in species A and two times in species B for example. We now define

$$\text{dist}_W(A, B) = \frac{\sum_{i,j} w_{i,j} m_{i,j}}{\max(\alpha_0, \dots, \alpha_5)(4^\ell - m_{0,0})}, \tag{9.3}$$

where W is the weighting matrix and M is the counting matrix. We calculate the weighted sum of the relevant matrix elements and divide by the maximum possible difference. Note that this is a generalization of Equation 9.2, if we choose $\alpha_0 = \alpha_1 = \alpha_2 = 0$ and $\alpha_3 = \alpha_4 = \alpha_5 = 1$, then we get Equation 9.2 again.

9.5 Experiments and results

We have done two types of experiments. The first one is the comparison of a pair of species, the second one is extracting a distance from the first experiments and to combine a number of species in a distance matrix.

9.5.1 Raw data

For the following results, we have chosen to look at sequences of length $n = 14$.

Table 9.3 is the raw comparison matrix of a human genome and that of a chimp. Notice that the number at position $(0, 0)$ is huge and non-informative. The other numbers on the main diagonal are also relatively large. This might mean that there are lots of similarities between the two species. The number at $(3, 3)$ is also very large, but that is because it is actually the sum of all points (x, y) with $x, y \geq 3$. Actually, all points $(3, x)$ and $(x, 3)$ with $x \in \mathbb{N}$ are less informative than the other numbers in this matrix, because we can not be sure if the 3 “is actually” a 3.

We will compare these figures with the difference between a cow and yeast. In Table 9.4, we see a quite different picture. The matrix is even less symmetrical. We see two reasons. Firstly a cow and yeast are quite different species, and

		Human			
		0	1	2	≥ 3
	0	150,783,349	4,486,933	1,216,093	498,090
	1	3,212,656	7,352,318	3,737,739	2,333,341
Chimp	2	602,927	2,621,970	4,011,169	4,907,515
	≥ 3	145,530	950,955	2,697,230	78,877,641

Table 9.3: Differences between Human and Chimp

secondly the genome of yeast is a lot shorter than that of a cow. Therefore a given random string is more likely to be present in a cow, so the matrix is what we would expect it to be.

		Yeast			
		0	1	2	≥ 3
	0	153,248,529	544,363	21,518	5,229
	1	15,023,538	548,614	25,707	5,624
Cow	2	11,361,444	489,848	26,124	5,459
	≥ 3	78,706,409	7,293,480	876,010	253,560

Table 9.4: Differences between Yeast and Cow

9.5.2 Visualisation of the raw data

For the following results, we have chosen to look at sequences of length $n = 16$. In Figure 9.6 we have plotted an interpolation of the values in the matrix M

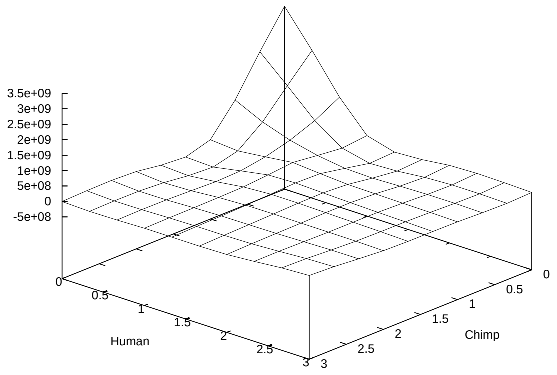

Figure 9.6: Human-Chimp raw data

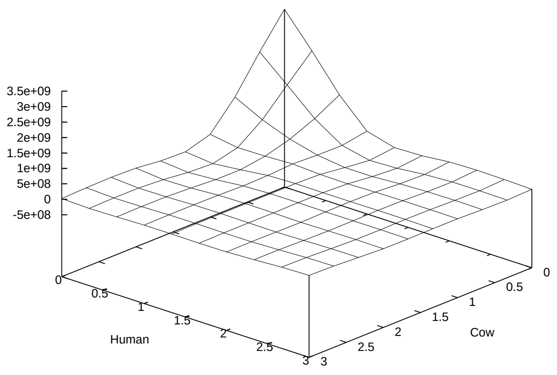

Figure 9.7: Human-Cow raw data

of the human genome and that of the chimp. Notice that although the matrix contains lots of information, the graph is almost symmetric, this is also the case for the almost identical Figure 9.7. This is because the similarities between the two species are much larger than the differences. Another disturbing factor is the point at $(0, 0)$: this is where all substrings that are not present in either species are. If the length of the substrings becomes too large (like here), this peak will be enormous. This is why we chose to leave out similarities in the next two pictures. In Figure 9.8 the difference between the human genome and that

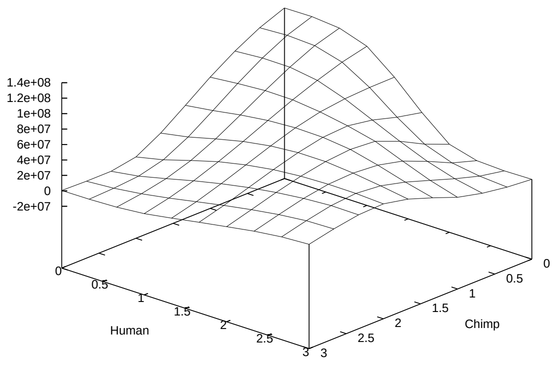

Figure 9.8: Human-Chimp raw data, main diagonal removed

of a chimp is plotted. The main diagonal has been removed from the data to emphasize the differences (the values at these positions are interpolated). The same technique is used in Figure 9.9, where the difference between human and cow is plotted.

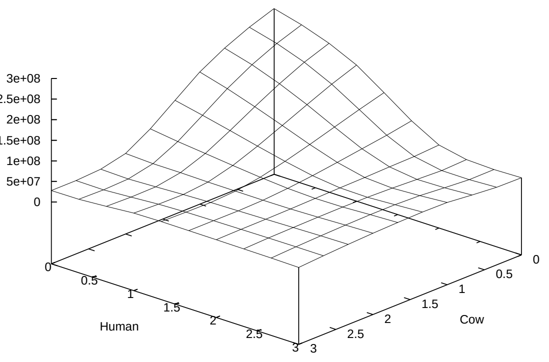

Figure 9.9: Human-Cow raw data, main diagonal removed

9.5.3 Comparison of many species

We have taken the genomes of the species shown in Table 9.5 from [69].

species	abbreviation	genome length
Bee	B	$5.13 \cdot 10^8$
C_elegans	Ce	$1.02 \cdot 10^8$
Chicken	Ci	$1.13 \cdot 10^9$
Chimp	C	$3.15 \cdot 10^9$
Cow	Co	$3.60 \cdot 10^9$
Dog	D	$2.58 \cdot 10^9$
Drosophila_melanogaster	Dm	$1.35 \cdot 10^8$
Human	H	$3.15 \cdot 10^9$
SARS	S	$3.69 \cdot 10^4$
Yeast	Y	$1.24 \cdot 10^7$

Table 9.5: Species

In Table 9.6 we give the distance matrix, where we took $\alpha_i = 1$ for all i. Notice that since the metric is symmetric, we do not have to show the upper half of the matrix, because we can just mirror it in the main diagonal. Also note that although we have not done any weighting, some things are already remarkable, for instance: SARS is at distance 0.999 to all of the other species (as expected) and the lowest distance (0.446) is the one between a human and a chimp.

In Table 9.7 we took $\alpha_0 = 1, \alpha_1 = 2, \alpha_2 = 4, \alpha_3 = 10, \alpha_4 = 20$ and $\alpha_5 = 1$. These values are taken quite arbitrary. The reason we chose for this particular set of values, is because if we assume that the two genomes are normal random strings, this would be a nice weighting scheme.

	Y	S	H	Dm	D	Co	C	Ci	Ce	B
Y	.000									
S	.999	.000								
H	.997	.999	.000							
Dm	.990	.999	.979	.000						
D	.997	.999	.740	.977	.000					
Co	.997	.999	.744	.977	.750	.000				
C	.997	.999	.442	.977	.748	.752	.000			
Ci	.995	.999	.834	.968	.833	.833	.830	.000		
Ce	.988	.999	.984	.959	.982	.983	.982	.973	.000	
B	.991	.999	.971	.957	.969	.969	.969	.959	.953	.000

Table 9.6: Distance matrix for $\alpha_i = 1$ for $i = 0, 1, 2, 3, 4, 5$

	Y	S	H	Dm	D	Co	C	Ci	Ce	B
Y	.000									
S	.507	.000								
H	.442	.443	.000							
Dm	.517	.525	.431	.000						
D	.463	.464	.293	.450	.000					
Co	.456	.457	.294	.443	.301	.000				
C	.459	.461	.142	.447	.300	.301	.000			
Ci	.514	.518	.340	.491	.349	.348	.346	.000		
Ce	.513	.524	.435	.494	.454	.447	.450	.495	.000	
B	.519	.526	.433	.494	.451	.445	.448	.488	.491	.000

Table 9.7: Distance matrix with weight (large α_3 and α_4, see text)

From these distance matrices we can make a *phylogenetic tree* [11]. We chose to make two visualisations, one rooted tree in which the distances are not preserved and one unrooted tree where distances are preserved as much as possible. Of course, since this is only a projection of the actual data, more trees can be drawn apart from these ones. In Figure 9.10 and 9.11 we see a rooted tree in which distances are not preserved. This is only to give the reader a global view of the distances between the given species. In Figure 9.12 and 9.13 we see an unrooted tree with partially preserved distances. The path from yeast to SARS for example is shorter than the path from yeast to cow. We see a difference in the warm-blooded animals when we compare these trees, the triple Dog, Cow, Chicken seem to be affected by our choice of weights. These differences can be observed in both the rooted and the unrooted trees.

Figure 9.10, 9.11, 9.12 and 9.13 are constructed by means of the *Fitch-Margoliash* [21] algorithm. They are visualisations of the matrices in Table 9.6 and 9.7.

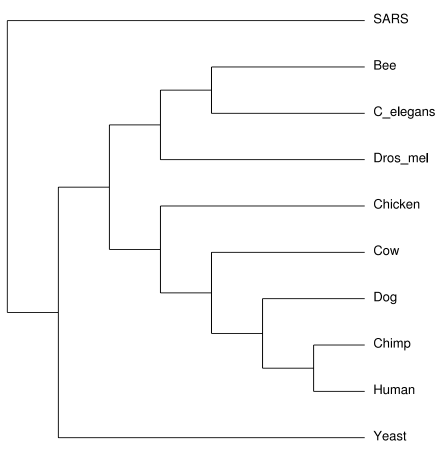

Figure 9.10: Phylogenetic tree of Table 9.6

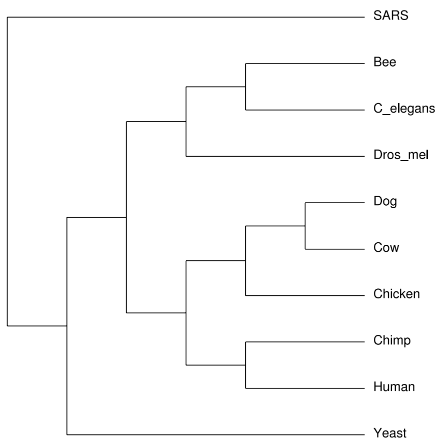

Figure 9.11: Phylogenetic tree of Table 9.7

9.6 A multiset distance measure

In this section we use a distance measure designed for multisets (see Chapter 6). This metric is parametrised by a function f that, given a few restrictions, will give a valid metric. (We shall adhere to these restrictions.)

The distance measure is defined as follows:

$$d_f(X,Y) = \frac{\sum_{i=1}^{n} f(x_i, y_i)}{|S(X) \cup S(Y)|}.$$

The numerator is the sum of values of a function f, that indicates the difference between the number of elements in one category. In this case, the difference in occurrences of a particular piece of DNA. The denominator is the number of

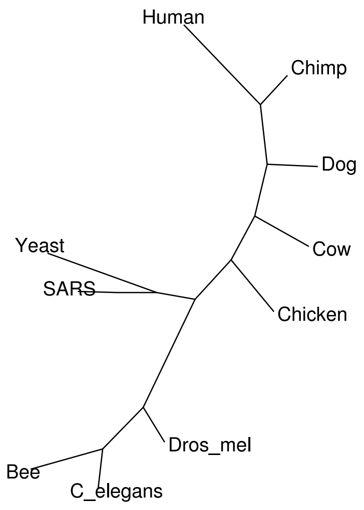

Figure 9.12: Unrooted phylogenetic tree of Table 9.6

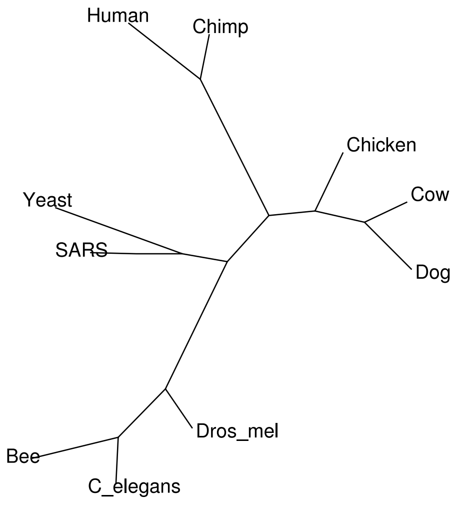

Figure 9.13: Unrooted phylogenetic tree of Table 9.7

categories, in this case, the number of strands of DNA present in either of the samples.

The function we use is:

$$f(x, y) = \frac{|x - y|}{(x + 1)(y + 1)}.$$

The reason for using this function is quite intuitive. If a particular strand of DNA is present once in one of the samples, and not in the other sample, the function will return distance 1/2. But when this strand is present in one sample once and twice in the other, the distance will be 1/6. In other words, the fact that two samples share a piece of DNA or not, is more important than the number of occurrences, though the latter is included.

	Y	S	H	Dm	D	Co	C	Ci	Ce	B
Y	.000									
S	.505	.000								
H	.618	.623	.000							
Dm	.511	.519	.582	.000						
D	.610	.615	.315	.574	.000					
Co	.613	.618	.320	.577	.323	.000				
C	.611	.616	.150	.574	.320	.325	.000			
Ci	.581	.587	.389	.542	.388	.390	.386	.000		
Ce	.516	.528	.590	.490	.582	.584	.582	.549	.000	
B	.532	.542	.571	.493	.563	.565	.562	.531	.491	.000

Table 9.8: Distance matrix as calculated with the multiset metric

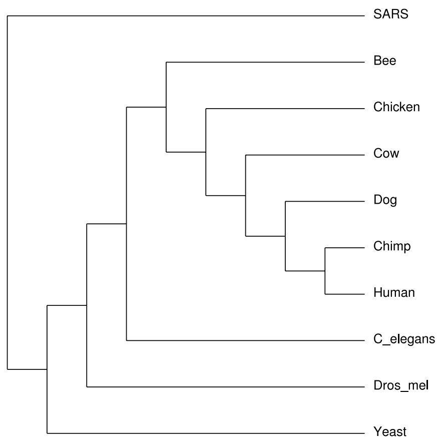

Figure 9.14: Phylogenetic tree of Table 9.8

Using the metric described above, we obtain the distance matrix shown in Table 9.8. The rooted and unrooted phylogenetic trees are shown in Figure 9.14 and 9.15.

9.7 Conclusions and further research

We have shown that determining (rare) substrings in a genome is possible up to a certain length. With the result we can make an annotated genome from which we can extract lots of data. The cumulative count of strings that occur n times

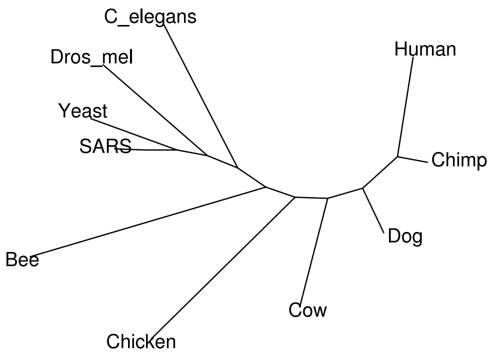

Figure 9.15: Unrooted phylogenetic tree of Table 9.8

in species A and m times in species B, where n, m are at most 4, still contains enough data to make a phylogenetic tree.

The techniques described in this chapter could also be used to discover *Single Nucleotide Polymorphisms* or SNP's [55] by using two individuals of the same species as input.

For further research we could make a distance measure based on (some of) the unique strings themselves, not the amount of them. This way we could make a very accurate distinction between species or individuals.

Chapter 10

Visualising Genomes in 3D using Rauzy Projections

We propose a novel visualisation method for DNA and other long sequences over a small alphabet, which is based on the construction of the family of Rauzy fractals for infinite words. We use this technique to find repeating structures of widely varying length in the input string as well as the identification of coding segments. Other properties of the input can also come to light using this technique.

10.1 Introduction

Projections of high dimensional structures onto a low dimensional surfaces (e.g. 2D, 3D) are commonly used to make structures in the data insightful.

Recognising patterns in long sequences is difficult for humans. The longer the sequence, the harder it gets. Furthermore, if the alphabet used for this sequence is small, the task gets even harder. If there are (small) deviations allowed in the patterns to be recognised, it will be nearly impossible without the aid of some sort.

In this chapter, we introduce a visualisation technique for DNA sequences using a projection onto a surface to investigate patterns in the DNA.

In Section 10.2 we explain the underlying idea of making fractals out of infinite words, which we adapt for our purposes in Sections 10.3. In Section 10.4 we show the visualisation results. Finally, related work is discussed in Section 10.5 and we conclude in Section 10.6.

10.2 Background

In this section, we describe the approach of Rauzy [59] to construct a fractal from an infinite word.

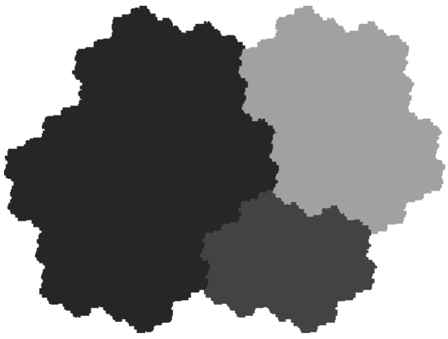
Figure 10.1: Standard Rauzy fractal

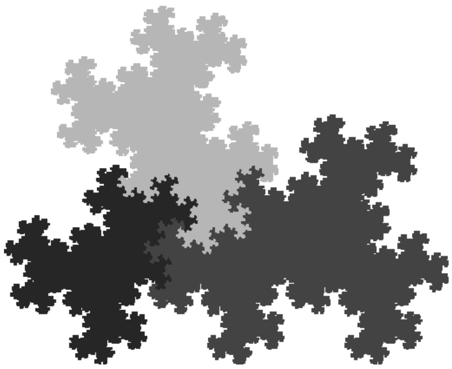
Figure 10.2: A "Rauzy" fractal using a different substitution

Given a finite alphabet Σ, we denote the set of all finite strings over this alphabet as Σ^*. In this section we take $\Sigma = \{0, 1, 2\}$.

Rauzy investigated the so-called tribonacci substitution:

$$\sigma : \begin{cases} 0 & \rightarrow & 01 \\ 1 & \rightarrow & 02 \\ 2 & \rightarrow & 0 \end{cases}$$

This substitution induces a *homomorphism*, again denoted by σ, from Σ^* to Σ^*. It is uniquely extended to Σ^* by requiring $\sigma(u \cdot v) = \sigma(u) \cdot \sigma(v)$ for all $u, v \in \Sigma^*$, where $\cdot$ denotes concatenation of strings.

Since 0 is a prefix of $\sigma(0)$, the following holds: $\sigma^n(0)$ is a prefix of $\sigma^{n+1}(0)(n = 1, 2, \ldots)$. Also $|\sigma^n(0)| \geq n \rightarrow \infty$ when $n \rightarrow \infty$. Therefore $(\sigma^n(0))_{n\in\mathbb{N}}$ defines a unique infinite word that has $\sigma^n(0)$ as finite prefix for each $n \in \mathbb{N}$. This word is invariant under the substitution (where we simultaneously substitute each letter). We call this word an accumulation point or fixed point of this substitution. For the given substitution σ, we get the word $0102010010201010201 00\ldots$ as fixed point.

Rauzy used this infinite word to make a plot in 3-dimensional space, starting in $(0, 0, 0)$ and doing one step in the x-direction whenever a 0 occurs, a step in the y-direction when a 1 occurs and a step in the z-direction when a 2 occurs. All steps are of equal length. This results in a so-called *broken halfline* which "approximates" a halfline ℓ starting in the origin. The existence of this halfline is discussed in [59].

Now we can take the plane through the origin which is perpendicular to the line ℓ and project the broken halfline onto this plane. This results in the *Rauzy fractal*, depicted in Figure 10.1. The colour of the projected points depends on whether the next step from the original point on the broken halfline is in the x-, y-, or z-direction.

When a different substitution is used, fractals can be made in the same manner. We see such a fractal in Figure 10.2. The substitution in applied here is:

$$\sigma : \begin{cases} 0 & \rightarrow & 01 \\ 1 & \rightarrow & 2 \\ 2 & \rightarrow & 0 \end{cases}$$

10.3 Application to DNA

We can apply a similar technique on a DNA sequence. First we have to make a plot in 4-dimensional space by associating each nucleotide (`A`, `C`, `G` and `T`) with one of the spacial dimensions and analogous to the construction described above building a broken halfline, but now in four dimensions.

Then we can make a 2- or 3-dimensional projection by either choosing a plane or a hyperplane through the origin and by projecting the broken halfline upon that (hyper)plane.

One big difference with the Rauzy approach in three dimensions is that the choice of the line ℓ which approximates the broken halfline is not pre-determined. A DNA string does not have the nice mathematical properties that the tribonacci fixed point possesses, so there is no clear preference for ℓ. One choice could be the line going through the begin- and endpoint of the broken halfline. This is possible since a DNA string is finite. However, this will result in different choices for ℓ for different DNA strings. This might not be the best choice. Therefore, we shall look (only) at a solution that uses a fixed ℓ for every input.

If ℓ is chosen well, we expect to find the following in the projected image:

- A non-predictable walk for information rich parts of the DNA.
- A true random walk for random parts.
- Lines (or approximate lines) for repeating parts of the DNA.
- Large copies of substrings in the DNA, that can be easily visualised.

The term *non-predictable walk* should be taken loosely, because we know that coding DNA (and therefore information rich DNA) has a higher `GC`-content than other parts of the DNA. Therefore, the walk will tend to go into the `GC`-direction, and second, DNA is clearly not random.

Since we have so much freedom in the choice of ℓ, we can also look at the projection in the following way: In principle, we can choose four vectors in the plane and use these four vectors to make the projection. Associate the first vector with the nucleotide `A`, the second one with `C`, and so on. Now, to make an insightful projection, we want the four vectors to have the following properties:

- The vectors should be of comparable length.

- The four vectors should add up to 0.
- Every subset of three vectors should be independent.

By adhering to these properties, we arrive in the same point if we consecutively plot a combination of all four letters, this would thus indicate a perfect repeat of those four letters, or a repeat of all four letters in any order.

When using an interactive program like `gnuplot` to visualise the data, the angle between the vectors can easily be adjusted (especially when making 2-D projections) by stretching the image. We must however, make sure that the vectors are not chosen parallel to the axes, otherwise the stretching will only result in the alteration of the length of the vectors. This property of interactive programs, along with the ability to zoom in, makes the exploration of dense areas in our visualisation possible.

We made an interactive web application available via [44]. This demonstration has access to the first 1,000,000 base pairs of the Human chromosome 1 and can visualise any substring. For performance reasons, we added a *step size* to make the rendering of large substrings possible. The data is located on the webserver and it is kept small for practical purposes. Making more data available will have no effect on the visualisation application.

10.4 A number of DNA sequence visualisations

In this section, we shall make several projections, on both two and tree dimensional hyperplanes. In each case, we choose a set of vectors for the nucleotides `A`, `C`, `G` and `T`. We denote these vectors by $v_{\mathtt{A}}$, $v_{\mathtt{C}}$, $v_{\mathtt{G}}$ and $v_{\mathtt{T}}$ respectively.

10.4.1 Projections in three dimensions

For three dimensions, there is, apart from symmetry, a natural choice of the projection line ℓ and therefore the resulting vectors. This set is the one that defines a tetrahedron (centred at the origin), as shown in Table 10.1. In other words, the convex hull of the set of endpoints of these vectors forms a tetrahedron. All vectors in this set have the same length, the four of them sum up to 0 and each three-element subset is independent. Furthermore, the vectors are uniformly distributed, i.e., the angle between each pair of vectors is equal.

$$\begin{pmatrix} 1 & -1 & -1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \end{pmatrix}$$

Table 10.1: Vertices of a tetrahedron

We associate the first row with $v_{\mathtt{A}}$, the second row with $v_{\mathtt{C}}$ and so on. In the remainder of this section, we shall use these vectors for our projections.

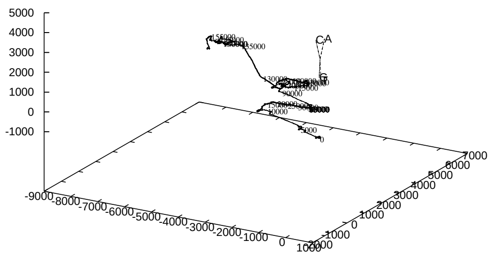
Figure 10.3: The first 160,000 nucleotides of the human Y-chromosome

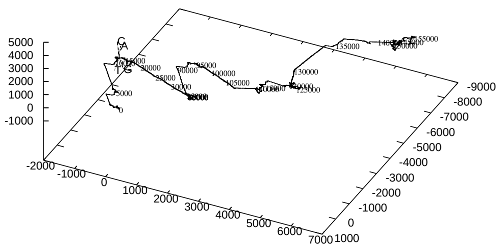
Figure 10.4: The first 160,000 nucleotides of the human Y-chromosome

As input for our three dimensional projection, we use the first 160,000 nucleotides of the human Y-chromosome [69] (build 18). This results in the picture shown in Figure 10.3. For clarity, we include the four vectors and the associated nucleotides in the visualisation.

In Figure 10.4, we see the exact same data and projection, but shown from a different angle.

This figure is a better representation of the data, more structures can be seen directly from this angle. We shall discuss the findings in detail in Section 10.4.2.

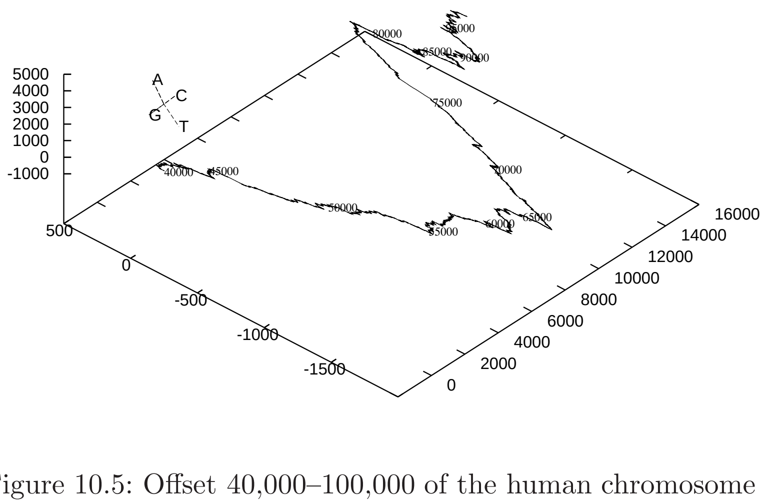

Figure 10.5: Offset 40,000–100,000 of the human chromosome 1

In Figure 10.5, we see a part of the first human chromosome, again, we shall discuss the findings in detail in Section 10.4.2.

10.4.2 Projections in two dimensions

Since, in two dimensions, there is no way to choose four vectors of equal length, of which all subsets are independent, we must choose the vectors in such a way that the lengths differ, to satisfy the independency constraint. Therefore we can choose e.g., the following vectors for `A`, `C`, `G` and `T` respectively: $(5, 7)$, $(-7, 6)$, $(8, -4)$ and $(-6, -9)$. By choosing these vectors, we have the property that every pair is independent, and the lengths are comparable. For practical purposes, we have chosen integer coordinates.

As input we use the first 160,000 nucleotides of the human Y-chromosome [69] (build 18). This results in the following projection:

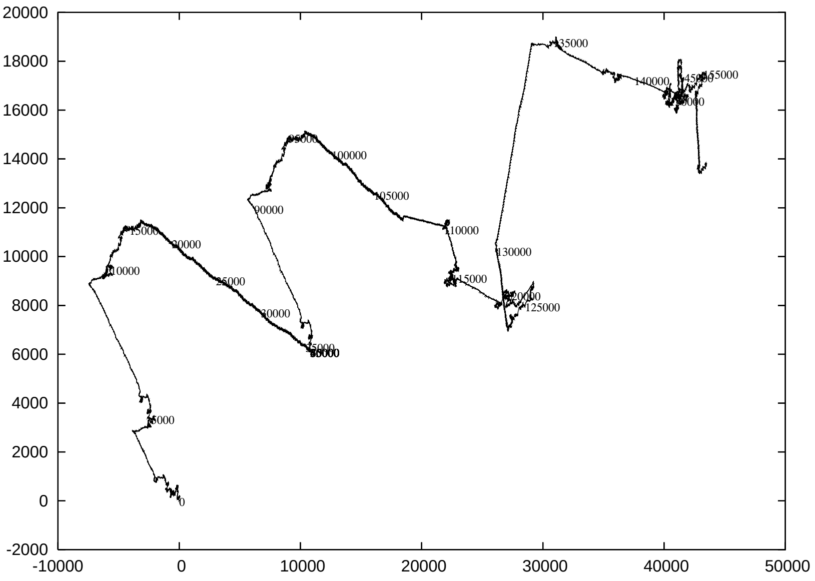

Figure 10.6: The first 160,000 nucleotides of the human Y-chromosome

The numbers in this plot denote the offset in the DNA, the plot starts in $(0, 0)$. We immediately see that two of the three assumptions from Section 10.3 can be verified for this input data. There are lines, which denote (approximate) repeats. For example, the line that starts somewhere near $(-3000, 4500)$ and ends in $(-7500, 9000)$ contains a large number of approximate copies of the string `CCCCGCTCCTCCCCTCGGGACCACCCCAGA`. In the region near $(23000, 9000)$, marked by the offset 115000, we see a part where the walk seems random. Moreover, we can see an extremely large substring from $(-2500, 3000)$ to $(5500, 8500)$ (approximately offset 5000 to 25000) that repeats itself from $(11000, 6000)$ to $(18500, 11500)$ (approximately offset 93000 to 107000).

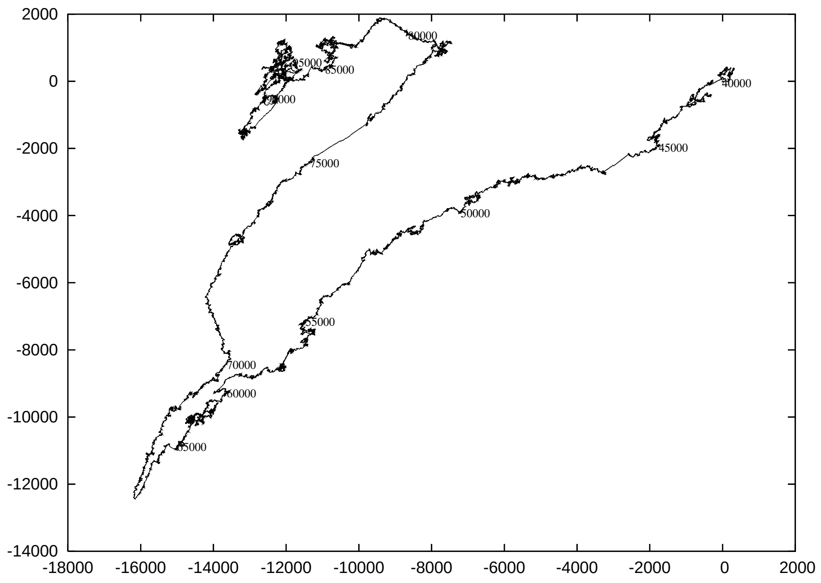

Figure 10.7: Offset 40,000–100,000 of the human chromosome 1

In Figure 10.7 we see a part of the human chromosome 1. No large repeats are noticeable in this part of the genome, but we can clearly see some short repeating sequences. The string `TTC` for example, is repeated a number of times from position $(-2600, -2200)$ to $(-3200, -2700)$. Another obvious short repeat `AAG`, can be seen from position $(-11200, -2200)$ to $(-9700, -1200)$, approximately.

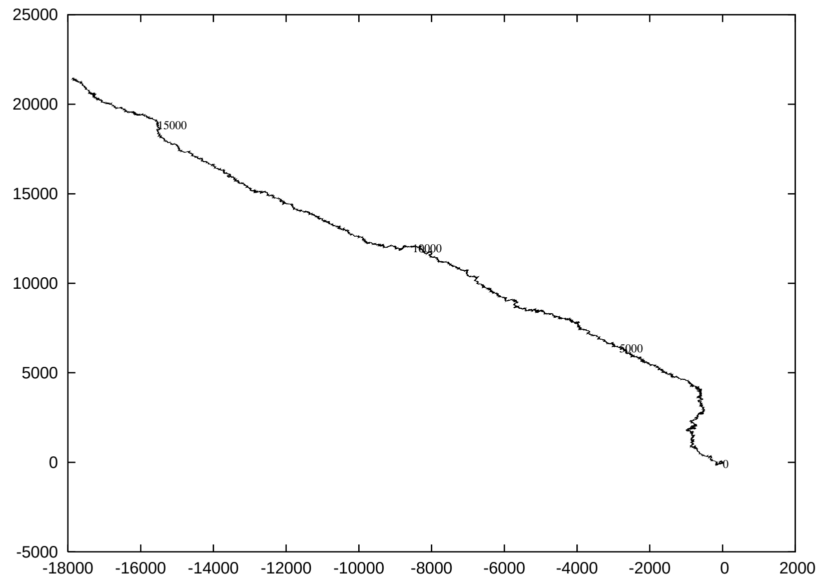

Figure 10.8: The complete mitochondrial DNA

In Figure 10.8, we see the DNA of mitochondria found in the human species.

The most noticeable is the drift in the C-direction. Further analysis show that this is due to the relatively low content of G-nucleotides in the mitochondrial DNA (13% versus 25–31% for the other nucleotides).

10.5 Related work

In the DNA-rainbow [3] project, plots of DNA were made by assigning a colour to a nucleotide; green for Adenine, red for Thymine, white for Guanine and blue for Cytosine. Undetermined positions were given a grey colour. Then the nucleotides were plotted to a standard bitmap with a width of 770 pixels. The resulting pictures, viewable with any picture viewer and/or editor, give a colourful impression of the DNA. Most of the pictures appear random, but in some parts, repeating parts of the genome can be seen in the form of diagonal lines. A nice property of this approach is that GC-rich areas (which are associated with encoding DNA), can be spotted right away.

An obvious shortcoming of this approach is the fixed width of the picture. Short repeating sequences which have a total length of under 770, will not stand out. Very long repeating sequences will not stand out either, since it will look like a random block that is repeated. However, for the detection of short repeating sequences (that are repeated for a large number of times), this approach seems very well suited.

In [62], several visualisation methods are investigated. One of them is essentially the same as in the previously described project, but here the width of the columns can be changed to detect (small) repeats of different length. Other visualisation methods include information hiding by using less colours, and usage of expert knowledge to emphasise some subsequences, like start and stop codons. Furthermore, a translation to amino acids can also be made with this technique. According to the authors, their proposed method is useful for sequences up to a length of 2,000 base pairs.

In [53], several visualisation techniques are reviewed: the “random walk” visualisation, as well as a fractal visualisation and a visualisation based on entropy-like parameters which are calculated within a sliding window. The “random walk” resembles the visualisation discussed in this chapter, with the exception that the directions that are associated with the nucleotides are fixed and the technique is limited to two dimensions.

10.6 Conclusions and further research

We have shown that all our hypothesis were confirmed; the simple repeats indeed show up as lines in our visualisation. What we did not expect were the large repeats (the thick lines mentioned in Section 10.4), although it is a rather nice result. Furthermore, we detected a large approximate repeat on the first part of the Y-chromosome. This repeat is already known by genetic experts, but it is nice to have detected it with no excessive calculation, like the alignment of two large sequences, provides hope for further exploration of DNA in this way.

As recommendations for further research, we suggest using colours as an extra coding scheme. By doing this, we can see the direction of a line in our visualisation. For example, it is hard to distinguish between simple repeats `AAG` and `CTT` because they are almost each others inverse. The lines resulting from these repeats will have approximately the same slope (see Figure 10.7 for an example of this), although their contents is different. The only ways to distinguish them at this point is to measure the exact slope, or by looking at the offsets and thereby finding the orientation of the line. By using a colour coding scheme, the colour of the line will represent the orientation directly.

An other interesting extension would be to make the line ℓ along which we project the 4-D structure onto a surface a parameter that can be changed in real time. A potential user could try to find a projection, better suited for his or her purposes.

Bibliography

[1] A. Aho and M. Corasick. Efficient string matching: An aid to bibliographic search. *Communications of the ACM*, 18:333–340, 1975.

[2] A.N. Arslan. Efficient approximate dictionary look-up for long words over small alphabets. *Lecture Notes in Computer Science*, 3887:118–129, 2006. LATIN, Theoretical Informatics: 7th Latin American Symposium.

[3] K. Bierkandt and J. Bierkandt. DNA rainbow. 2009. Website available via `http://www.dna-rainbow.org/contact.html`, Retrieved April 3, 2009.

[4] C.M. Bishop. *Pattern Recognition and Machine Learning.* Springer, 2006.

[5] I. Borg and P. Groenen. *Modern Multidimensional Scaling: Theory and Applications.* Springer, 1997.

[6] J.R. Bray and J.T. Curtis. An ordination of the upland forest communities of southern Wisconsin. *Ecol. Monogr.*, 27:325–349, 1957.

[7] K.J. Breslauer, R. Frank, H. Blöcker, and L.A. Marky. Predicting DNA duplex stability from the base sequence. *Proc. Natl. Acad. Sci. USA*, 83:3746–3750, 1986.

[8] J. Broekens, T.K. Cocx, and W.A. Kosters. Object-centered interactive multi-dimensional scaling: Ask the expert. In *Proceedings of the 18th Belgium-Netherlands Conference on Artificial Intelligence (BNAIC 2006)*, pages 59–66, 2006.

[9] J.S. de Bruin, T.K. Cocx, W.A. Kosters, J.F.J. Laros, and J.N. Kok. Data mining approaches to criminal career analysis. In *Proceedings of the Sixth IEEE International Conference on Data Mining (ICDM 2006)*, pages 171–177. IEEE, 2006.

[10] J.S. de Bruin, T.K. Cocx, W.A. Kosters, J.F.J. Laros, and J.N. Kok. Onto clustering of criminal careers. In *Proceedings of the Workshop on Practical Data Mining: Applications, Experiences and Challenges (ECML/PKDD-2006)*, pages 90–93. IEEE, 2006.

[11] P.Y. Chan, T.W. Lam, and S.M. Yiu. A more accurate and efficient whole genome phylogeny. *Asia-Pacific Bioinformatics*, pages 337–351, 2006.

[12] T.K. Cocx and W.A. Kosters. A distance measure for determining similarity between criminal investigations. In *Advances in Data Mining, Proceedings of the Industrial Conference on Data Mining 2006 (ICDM2006)*, volume 4065 of *LNAI*, pages 511–525. Springer, 2006.

[13] T.K. Cocx, W.A. Kosters, and J.F.J. Laros. Enhancing the automated analysis of criminal careers. In *SIAM Workshop on Link Analysis, Counterterrorism, and Security 2008 (LACTS2008)*. SIAM Press, 2008.

[14] T.K. Cocx, W.A. Kosters, and J.F.J. Laros. Temporal extrapolation within a static clustering. In *Foundations of Intelligent Systems, Proceedings of ISMIS 2008*, volume 4994 of *LNAI*, pages 189–195. Springer, 2008.

[15] T.K. Cocx, W.A. Kosters, and J.F.J. Laros. Temporal extrapolation within a static clustering. In *Proceedings of the 20th Belgium-Netherlands Conference on Artificial Intelligence (BNAIC 2008)*, pages 295–296, 2008.

[16] M.L. Davison. *Multidimensional Scaling.* John Wiley and Sons, New York, 1983.

[17] C.W. Dieffenbach and G.S. Dveksler. *PCR Primer: A Laboratory Manual.* CSHL Press, Cold Spring Harbor, USA, 1995.

[18] T. Duchamp and W. Stuetzle. Geometric properties of principal curves in the plane. *Lecture Notes in Statistics*, 109:135–152, 1996.

[19] P. Eades. A heuristic for graph drawing. *Congressus Numerantium*, 42:149–160, 1984.

[20] I. Fischer. Similarity-based neural networks for applications in computational molecular biology. *Lecture Notes in Computer Science*, 2810:208–218, 2003.

[21] W.M. Fitch and E. Margoliash. Construction of phylogenetic trees. *Science*, 155:279–284, 1967.

[22] I.K. Fodor. A survey of dimension reduction techniques. technical report. *Lawrence Livermore National Laboratory*, 2002.
`http://www.llnl.gov/tid/lof/documents/pdf/240921.pdf`.

[23] G. Gibson and S. Muse. *A Primer of Genome Science.* Boyle Biochemistry and Molecular Biology Education, 2nd edition, 2005.

[24] G. Glazko, A. Gordon, and A. Mushegian. The choice of optimal distance measure in genome-wide datasets. *Bioinformatics*, 21:iii3–iii11, 2005.

[25] E.H. de Graaf, J.N. Kok, and W.A. Kosters. Clustering improves the exploration of graph mining results. In C. Boukis, A. Pnevmatikakis, and L. Polymenakos, editors, *Proceedings of the 4th IFIP International Conference on Artificial Intelligence Applications and Innovations (AIAI07)*, pages 13–20. Springer, 2007.

[26] D. Gusfield. *Algorithms on Strings, Trees and Sequences.* Cambridge University Press, 1997.

[27] S. Haas, M. Vingron, A. Poustka, and S. Wiemann. Primer design for large scale sequencing. *Nucleic Acids Research*, 26(12):3006–3012, 1998.

[28] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning.* Springer, 2001.

[29] T.J. Hastie and W. Stuetzle. Principal curves. *J. Amer. Statist. Assoc.*, 84:502–516, 1989.

[30] G. Hinton and S.T. Roweis. Stochastic neighbor embedding. *Advances in Neural Information Processing Systems*, 15:833–840, 2003.

[31] D.S. Hirschberg. Algorithms for the longest common subsequence problem. *Journal of the ACM*, 24(4):664–675, 1977.

[32] H.J. Hoogeboom, W.A. Kosters, and J.F.J. Laros. Selection of DNA markers. *IEEE Transactions on Systems, Man, and Cybernetics Part C*, 38:26–32, 2008.

[33] S. Inenaga, H. Hoshino, A. Shinohara, M. Takeda, S. Arikawa, G. Mauri, and Pavesi G. On-line construction of compact directed acyclic word graphs. *Discrete Applied Mathematics*, 146(2):156–179, 2005.

[34] P. Jaccard. Lois de distribution florale dans la zone alpine. *Bull. Soc. Vaud. Sci. Nat.*, 38:69–130, 1902.

[35] I.T. Jolliffe. *Principal Component Analysis.* Springer, 2nd edition, 2002.

[36] T. Kadota, M. Hirao, A. Ishino, M. Takeda, A. Shinohara, and F Matsuo. Musical sequence comparison for melodic and rhythmic similarities. In *Proceedings of the Eighth International Symposium on String Processing and Information Retrieval*, pages 111–122. IEEE, 2001.

[37] T. Kämpke, M. Kieninger, and M. Mecklenburg. Efficient primer design algorithms. *Bioinformatics*, 17(3):214–225, 2001.

[38] D. Knuth, J. Morris, and V. Pratt. Fast pattern matching in strings. *SIAM J. Computing*, 6:323–350, 1977.

[39] T. Kohonen. *Self-Organizating Maps.* New York: Springer-Verlag, 1997.

[40] W.A. Kosters and J.F.J. Laros. Metrics for mining multisets. In *Research and Development in Intelligent Systems XXIV, Proceedings of AI-2007*, pages 293–303. Springer, 2007.

[41] W.A. Kosters and J.F.J. Laros. Metrics for mining multisets. In *Proceedings of the 20th Belgium-Netherlands Conference on Artificial Intelligence (BNAIC 2008)*, pages 329–330, 2007.

[42] W.A. Kosters and J.F.J. Laros. Visualisation on a closed surface. In *Proceedings of the 19th Belgium-Netherlands Conference on Artificial Intelligence (BNAIC 2007)*, pages 189–195, 2007.

[43] W.A. Kosters and M.C. van Wezel. Competitive neural networks for customer choice models, in e-commerce and intelligent methods. *Studies in Fuzziness and Soft Computing*, 105:41–60, 2002.

[44] J. F. J. Laros. Visualising DNA with Rauzy projections, 2009. Web application available via `http://www.liacs.nl/home/jlaros/dnavis/` [retrieved April 3, 2009].

[45] V.I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Doklady Akademii Nauk SSSR*, 163(4):845–848, 1965. English translation in Soviet Physics Doklady 10(8):707–710, 1966.

[46] Q. Ma, J.T.L. Wang, D. Shasha, and C.H. Wu. DNA sequence classification via an expectation maximization algorithm and neural networks: A case study. *IEEE Transactions on Systems, Man, and Cybernetics Part C — Applications and Reviews*, 31(4):468–475, 2001.

[47] M.S. Madhusudhan, M.A. Marti-Renom, R. Sanchez, and A. Sali. Variable gap penalty for protein sequence-structure alignment. *Protein Engineering Design and Selection*, 19(3):129–133, 2006.

[48] P.C. Mahalanobis. On the generalised distance in statistics. *Proceedings of the National Institute of Science of India*, 12:49–55, 1936.

[49] H. Mannila and P. Ronkainen. Similarity of event sequences. In *Proceedings of the International Symposium on Temporal Representation and Reasoning*, pages 136–139. IEEE Computer Society, 1997.

[50] C. McCue. *Data Mining and Predictive Analysis: Intelligence Gathering and Crime Analysis*. Butterworth-Heinemann, 1st edition, 2007.

[51] J. Mena. *Investigative Data Mining for Security and Criminal Detection*. Butterworth-Heinemann, 1st edition, 2003.

[52] D.R. Morrison. PATRICIA — Practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM*, 15(4):514–534, 1968.

[53] A. Mylläri, T. Salakoski, and A. Pasechnik. On the visualization of the DNA sequence and its nucleotide content. *SIGSAM Bull.*, 39(4):131–135, 2005.

[54] S.B. Needleman and C.D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *J. Mol. Biol.*, 48:443–453, 1970.

[55] A. Oliphant, D.L. Barker, J.R. Stuelpnagel, and M.S. Chee. SNPs: Discovery of markers for disease. *BioTechniques*, 32:S56–S61, 2002.

[56] S.K. Pal, S. Bandyopadhyay, and S.S. Ray. Evolutionary computation in bioinformatics: A review. *IEEE Transactions on Systems, Man, and Cybernetics Part C — Applications and Reviews*, 36(5):601–615, 2006.

[57] K. Pearson. On lines and planes of closest fit to systems of points in space. *Phil. Mag.*, 2(6):559–572, 1901.

[58] G. Raddatz, M. Dehio, T.F. Meyer, and C. Dehio. Primearray: Genome-scale primer design for DNA-microarray construction. *Bioinformatics*, 17(1):98–99, 2001.

[59] G. Rauzy. Nombres algébriques et substitutions. *Bull. Soc. Math. France*, 110:147–178, 1982.

[60] S. Rozen and H.J. Skaletsky. Primer3, 1998. `http://www-genome.wi.mit.edu/genome_software/other/primer3.html`.

[61] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, 2nd edition, 2002.

[62] P. Rutherford, C. Churcher, and J. McCallum. An interactive visualisation for investigating DNA sequence information. In *APVis '04: Proceedings of the 2004 Australasian Symposium on Information Visualisation*, pages 101–107. Australian Computer Society, Inc., 2004.

[63] M. Schena, D. Shalon, R.W. Davis, and P.O. Brown. Quantitative monitoring of gene expression patterns with a complementary DNA microarray. *Science*, 270(5235):467–470, 1995.

[64] J.P. Schouten, C.J. McElgunn, R. Waaijer, D. Zwijnenburg, F. Diepvens, and G. Pals. Relative quantification of 40 nucleic acid sequences by multiplex ligation-dependent probe amplification. *Nucleic Acid Research*, 30(12, e57):1–13, 2002.

[65] J. Setubal and J. Meidanis. *Introduction to Computational Molecular Biology.* Thomson Learning, 1997.

[66] T. Smith and M. Waterman. Identification of common molecular subsequences. *J. Mol. Biol.*, 147:195–197, 1981.

[67] P.-N. Tan, M. Steinbach, and V. Kumar. *Introduction to Data Mining.* Addison-Wesley, 2005.

[68] Cocx T.K., W.A. Kosters, and J.F.J. Laros. An early warning system for the prediction of criminal careers. In *MICAI 2008: Advances in Artificial Intelligence*, volume 5317 of *LNAI*, pages 77–89. Springer, 2008.

[69] UCSC Genome Bioinformatics. `http://genome.ucsc.edu/`, version April 21, 2006 [retrieved April 3, 2009].

[70] P.M. Vallone and J.M. Butler. Autodimer: A screening tool for primer-dimer and hairpin structures. *BioTechniques*, 37:226–231, 2004.

[71] C. Varotto, E. Richly, F. Salamini, and D. Leister. GST-PRIME: A genome-wide primer design software for the generation of gene sequence tags. *Nucleic Acids Research*, 29(21):4373–4377, 2001.

[72] M. Vingron and P.A. Pevzner. Multiple sequence comparison and consistency on multipartite graphs. *Adv. Appl. Math.*, 16:1–22, 1995.

[73] A.A. Westen, A. S. Matai, J.F.J. Laros, H.C. Meiland, M. Jasper, W.J.F. de Leeuw, P. de Knijff, and T. Sijen. Tri-allelic SNP markers enable analysis of mixed and degraded DNA samples. *Forensic Science International: Genetics*, 3(4):233–241, 2009.

Nederlandse Samenvatting

Deze samenvatting is als volgt opgebouwd. In de eerste drie delen beschrijven we de hoofdlijnen van dit proefschrift, in het laatste deel geven we een overzicht van de verschillende hoofdstukken.

Data Mining

Data Mining is informeel gesproken het extraheren van voorheen onbekende en vooral interessante patronen uit data. In het algemeen wordt dit gerealiseerd door het gebruik van een scala aan technieken, waarbij elk van deze technieken een ander licht werpt op de data. Omdat we te maken hebben met een ware data-explosie, is er meer vraag naar dit soort methoden. De ontwikkeling van snellere hardware stelt ons ook in staat deze technieken toe te passen.

DNA

Desoxyribonucleïnezuur (DNA) is een groot molecuul dat genetische informatie bevat. Het bestaat uit vier letters (A, C, G, T) of nucleotiden. Deze nucleotiden vormen grote lineaire structuren die chromosomen heten.

De laatste decennia zijn methodes ontwikkeld die deze woorden efficiënt kunnen verwerken; deze vallen onder de naam *sequencing*. Deze technieken lezen het DNA en geven lange woorden als uitvoer. Door deze te analyseren is het mogelijk om verschillen tussen soorten en zelfs individuen te vinden. Ook is het mogelijk om, zonder de specifieke verschillen te kennen, een evolutionaire boom te maken op basis van korte deelwoorden.

Metrieken

In dit proefschrift wordt een nieuwe afstandsmaat of metriek gebruikt. Deze afstandsmaat, ontworpen voor *multisets*, is in grote mate configureerbaar. Er is een speciale functie nodig, die in het algemeen door een expert op een bepaald domein wordt gekozen. Deze functie moet het verschil tussen twee voorkomens van een element binnen een multiset uitdrukken. Ter illustratie, het verschil

tussen een persoon die geen fietsen steelt en iemand die er één heeft gestolen is verdedigbaar groter dan het verschil tussen iemand die 100 fietsen steelt en iemand die er 101 steelt. Dit verschil moet worden uitgedrukt in een functie.

Deze afstandsmaat was eigenlijk ontworpen voor de analyse van criminele activiteiten, maar bleek door het vervangen van de expert-functie ook goed toepasbaar in andere domeinen. Dit komt in de latere hoofdstukken terug.

Overzicht

Dit proefschrift is opgebouwd uit drie delen. In het eerste deel richten we ons op de toepassing van Data Mining in de wetshandhaving, met name de toepassing van het *deeltjesmodel* in dit gebied. Het deeltjesmodel is een dimentiereductietechniek, waarbij elk object in de invoer geassocieerd wordt met een punt in een ruimte. De punten worden in eerste instantie willekeurig in de ruimte gezet en, afhankelijk van de onderlinge afstand (gedefinieerd op de objecten waarmee ze geassocieerd zijn) naar elkaar toe verplaatst of van elkaar af geduwd, al naar gelang ze te ver van elkaar af staan, of te dicht bij elkaar staan.

In het tweede deel nemen we de metrieken die in het eerste deel worden genoemd onder de loep. Het derde deel is gericht op DNA. We laten met name zien dat de gebruikte metrieken ook van toepassing zijn op het gebied van de moleculaire genetica.

In Hoofdstuk 2 geven we een uitgebreid overzicht over het deeltjesmodel en zijn toepassingen. In Hoofdstuk 3 bekijken we een specifieke variant van het deeltjesmodel, namelijk degene waarin we een torus als uitvoeroppervlak gebruiken. In Hoofdstuk 4 introduceren we een nieuwe manier om fouten die gemaakt worden in dimensiereductietechnieken op te sporen en in kaart te brengen. In Hoofdstuk 5 gebruiken we de Levenshtein-afstand tussen twee carrières van criminelen om een overeenkomst in de geschiedenis van deze carrières te vinden. Ook gebruiken we gelijkende carrières om voorspellingen te doen.

In Hoofdstuk 6 gaan we in op de afstandsmaat voor multisets. De restricties op de expertfunctie, die een parameter is voor deze afstandsmaat, worden besproken. Hoofdstuk 7 houdt zich bezig met de uitbreiding van de afstandsmaat voor multisets tot die van een afstandsmaat voor sequenties van multisets.

In Hoofdstuk 8 gaan we in op het probleem waarbij we korte, unieke deelwoorden willen vinden in grote woorden. Hoofdstuk 9 behandelt een nieuwe manier om de afstand tussen twee genomen te geven. Hoofdstuk 10 gaat in op een nieuwe manier om DNA te visualiseren, waarbij gebruik wordt gemaakt van Rauzy-projecties.

Curriculum Vitae

Jeroen Laros is geboren in Den Helder, Noord-Holland, op 27 juli 1977. Van 1989 tot 1997 doorliep hij achtereenvolgens MAVO, HAVO en VWO op de scholengemeenschap Oscar Romero te Hoorn en het Bornego College te Heerenveen. Van 1997 tot 2005 studeerde hij Informatica aan de Universiteit Leiden, waar hij al vanaf zijn eerste jaar betrokken was bij het systeembeheer, eerst bij de helpdesk, en daarna als systeembeheerder. Vanaf 2005 was hij verbonden aan het Leiden Institute of Advanced Computer Science als promovendus, waar hij zijn promotieonderzoek uitvoerde binnen het NWO-project DALE. Eveneens gaf hij in die jaren verschillende werkgroepen. Sinds het voorjaar van 2009 is hij werkzaam als postdoc bij het Leids Universitair Medisch Centrum (LUMC).

Titles in the IPA Dissertation Series since 2005

E. Ábrahám. *An Assertional Proof System for Multithreaded Java - Theory and Tool Support-* . Faculty of Mathematics and Natural Sciences, UL. 2005-01

R. Ruimerman. *Modeling and Remodeling in Bone Tissue*. Faculty of Biomedical Engineering, TU/e. 2005-02

C.N. Chong. *Experiments in Rights Control - Expression and Enforcement*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-03

H. Gao. *Design and Verification of Lock-free Parallel Algorithms*. Faculty of Mathematics and Computing Sciences, RUG. 2005-04

H.M.A. van Beek. *Specification and Analysis of Internet Applications*. Faculty of Mathematics and Computer Science, TU/e. 2005-05

M.T. Ionita. *Scenario-Based System Architecting - A Systematic Approach to Developing Future-Proof System Architectures*. Faculty of Mathematics and Computing Sciences, TU/e. 2005-06

G. Lenzini. *Integration of Analysis Techniques in Security and Fault-Tolerance*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-07

I. Kurtev. *Adaptability of Model Transformations*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-08

T. Wolle. *Computational Aspects of Treewidth - Lower Bounds and Network Reliability*. Faculty of Science, UU. 2005-09

O. Tveretina. *Decision Procedures for Equality Logic with Uninterpreted Functions*. Faculty of Mathematics and Computer Science, TU/e. 2005-10

A.M.L. Liekens. *Evolution of Finite Populations in Dynamic Environments*. Faculty of Biomedical Engineering, TU/e. 2005-11

J. Eggermont. *Data Mining using Genetic Programming: Classification and Symbolic Regression*. Faculty of Mathematics and Natural Sciences, UL. 2005-12

B.J. Heeren. *Top Quality Type Error Messages*. Faculty of Science, UU. 2005-13

G.F. Frehse. *Compositional Verification of Hybrid Systems using Simulation Relations*. Faculty of Science, Mathematics and Computer Science, RU. 2005-14

M.R. Mousavi. *Structuring Structural Operational Semantics*. Faculty of Mathematics and Computer Science, TU/e. 2005-15

A. Sokolova. *Coalgebraic Analysis of Probabilistic Systems*. Faculty of Mathematics and Computer Science, TU/e. 2005-16

T. Gelsema. *Effective Models for the Structure of pi-Calculus Processes with Replication*. Faculty of Mathematics and Natural Sciences, UL. 2005-17

P. Zoeteweij. *Composing Constraint Solvers*. Faculty of Natural Sciences,

M.E. Warnier. *Language Based Security for Java and JML*. Faculty of Science, Mathematics and Computer Science, RU. 2006-16

V. Sundramoorthy. *At Home In Service Discovery*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-17

B. Gebremichael. *Expressivity of Timed Automata Models*. Faculty of Science, Mathematics and Computer Science, RU. 2006-18

L.C.M. van Gool. *Formalising Interface Specifications*. Faculty of Mathematics and Computer Science, TU/e. 2006-19

C.J.F. Cremers. *Scyther - Semantics and Verification of Security Protocols*. Faculty of Mathematics and Computer Science, TU/e. 2006-20

J.V. Guillen Scholten. *Mobile Channels for Exogenous Coordination of Distributed Systems: Semantics, Implementation and Composition*. Faculty of Mathematics and Natural Sciences, UL. 2006-21

H.A. de Jong. *Flexible Heterogeneous Software Systems*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-01

N.K. Kavaldjiev. *A run-time reconfigurable Network-on-Chip for streaming DSP applications*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-02

M. van Veelen. *Considerations on Modeling for Early Detection of Abnormalities in Locally Autonomous Distributed Systems*. Faculty of Mathematics and Computing Sciences, RUG. 2007-03

T.D. Vu. *Semantics and Applications of Process and Program Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-04

L. Brandán Briones. *Theories for Model-based Testing: Real-time and Coverage*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-05

I. Loeb. *Natural Deduction: Sharing by Presentation*. Faculty of Science, Mathematics and Computer Science, RU. 2007-06

M.W.A. Streppel. *Multifunctional Geometric Data Structures*. Faculty of Mathematics and Computer Science, TU/e. 2007-07

N. Trčka. *Silent Steps in Transition Systems and Markov Chains*. Faculty of Mathematics and Computer Science, TU/e. 2007-08

R. Brinkman. *Searching in encrypted data*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-09

A. van Weelden. *Putting types to good use*. Faculty of Science, Mathematics and Computer Science, RU. 2007-10

J.A.R. Noppen. *Imperfect Information in Software Development Processes*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-11

R. Boumen. *Integration and Test plans for Complex Manufacturing Systems*. Faculty of Mechanical Engineering, TU/e. 2007-12

A.J. Wijs. *What to do Next?: Analysing and Optimising System Behaviour in Time*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2007-13

C.F.J. Lange. *Assessing and Improving the Quality of Modeling: A Series of Empirical Studies about the UML*. Faculty of Mathematics and Computer Science, TU/e. 2007-14

T. van der Storm. *Component-based Configuration, Integration and Delivery*. Faculty of Natural Sciences, Mathematics, and Computer Science,UvA. 2007-15

B.S. Graaf. *Model-Driven Evolution of Software Architectures*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2007-16

A.H.J. Mathijssen. *Logical Calculi for Reasoning with Binding*. Faculty of Mathematics and Computer Science, TU/e. 2007-17

D. Jarnikov. *QoS framework for Video Streaming in Home Networks*. Faculty of Mathematics and Computer Science, TU/e. 2007-18

M. A. Abam. *New Data Structures and Algorithms for Mobile Data*. Faculty of Mathematics and Computer Science, TU/e. 2007-19

W. Pieters. *La Volonté Machinale: Understanding the Electronic Voting Controversy*. Faculty of Science, Mathematics and Computer Science, RU. 2008-01

A.L. de Groot. *Practical Automaton Proofs in PVS*. Faculty of Science, Mathematics and Computer Science, RU. 2008-02

M. Bruntink. *Renovation of Idiomatic Crosscutting Concerns in Embedded Systems*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-03

A.M. Marin. *An Integrated System to Manage Crosscutting Concerns in Source Code*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-04

N.C.W.M. Braspenning. *Model-based Integration and Testing of High-tech Multi-disciplinary Systems*. Faculty of Mechanical Engineering, TU/e. 2008-05

M. Bravenboer. *Exercises in Free Syntax: Syntax Definition, Parsing, and Assimilation of Language Conglomerates*. Faculty of Science, UU. 2008-06

M. Torabi Dashti. *Keeping Fairness Alive: Design and Formal Verification of Optimistic Fair Exchange Protocols*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2008-07

I.S.M. de Jong. *Integration and Test Strategies for Complex Manufacturing Machines*. Faculty of Mechanical Engineering, TU/e. 2008-08

I. Hasuo. *Tracing Anonymity with Coalgebras*. Faculty of Science, Mathematics and Computer Science, RU. 2008-09

L.G.W.A. Cleophas. *Tree Algorithms: Two Taxonomies and a Toolkit*. Faculty of Mathematics and Computer Science, TU/e. 2008-10

I.S. Zapreev. *Model Checking Markov Chains: Techniques and Tools*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-11

M. Farshi. *A Theoretical and Experimental Study of Geometric Networks*. Faculty of Mathematics and Computer Science, TU/e. 2008-12

of Science, Mathematics and Computer Science, RU. 2009-01

M. de Mol. *Reasoning about Functional Programs: Sparkle, a proof assistant for Clean.* Faculty of Science, Mathematics and Computer Science, RU. 2009-02

M. Lormans. *Managing Requirements Evolution.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-03

M.P.W.J. van Osch. *Automated Model-based Testing of Hybrid Systems.* Faculty of Mathematics and Computer Science, TU/e. 2009-04

H. Sozer. *Architecting Fault-Tolerant Software Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-05

M.J. van Weerdenburg. *Efficient Rewriting Techniques.* Faculty of Mathematics and Computer Science, TU/e. 2009-06

H.H. Hansen. *Coalgebraic Modelling: Applications in Automata Theory and Modal Logic.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-07

A. Mesbah. *Analysis and Testing of Ajax-based Single-page Web Applications.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-08

A.L. Rodriguez Yakushev. *Towards Getting Generic Programming Ready for Prime Time.* Faculty of Science, UU. 2009-9

K.R. Olmos Joffré. *Strategies for Context Sensitive Program Transformation.* Faculty of Science, UU. 2009-10

J.A.G.M. van den Berg. *Reasoning about Java programs in PVS using JML.* Faculty of Science, Mathematics and Computer Science, RU. 2009-11

M.G. Khatib. *MEMS-Based Storage Devices. Integration in Energy-Constrained Mobile Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-12

S.G.M. Cornelissen. *Evaluating Dynamic Analysis Techniques for Program Comprehension.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-13

D. Bolzoni. *Revisiting Anomaly-based Network Intrusion Detection Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-14

H.L. Jonker. *Security Matters: Privacy in Voting and Fairness in Digital Exchange.* Faculty of Mathematics and Computer Science, TU/e. 2009-15

M.R. Czenko. *TuLiP - Reshaping Trust Management.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-16

T. Chen. *Clocks, Dice and Processes.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-17

C. Kaliszyk. *Correctness and Availability: Building Computer Algebra on top of Proof Assistants and making Proof Assistants available over the Web.* Faculty of Science, Mathematics and Computer Science, RU. 2009-18

R.S.S. O'Connor. *Incompleteness & Completeness: Formalizing Logic and Analysis in Type Theory.* Faculty