TECHNISCHE UNIVERSITÄT WIEN

VIENNA UNIVERSITY OF TECHNOLOGY

MASTER'S THESIS

# Combinatorial Optimization for the Compression of Biometric Templates

Written at the Institute for
COMPUTER GRAPHICS AND ALGORITHMS
of the Vienna University of Technology

under supervision of
UNIV.PROF. DIPL.-ING. DR. TECHN. GÜNTHER RAIDL
UNIV.ASS. MAG. DIPL.-ING. ANDREAS CHWATAL

by

Olivia Dietzel, B.Sc.
Leebgasse 52/15
1100 Wien

5/5/08

# Abstract

A biometric template contains biometric traits belonging to a certain person, like e.g. fingerprints or the facial structure. Especially for verification purposes such human characteristics become ever more important. In order to recognize a person by means of his biometric traits a reference template must be available, which can be stored in a database and also on a RFID chip. With regard to mobile storage media, and thus only a small amount of memory, there is a need for the compression of biometric templates. This compression may be lossy, possible errors in the recognition however should be kept as small as possible. In this Master's Thesis in particular a new approach for the compression of *fingerprint templates* is developed. These templates contain information about the positions and orientations of the so-called minutiae, i.e. the endings and bifurcations of the dermal papillae. In turn this information is represented in the form of points of a $d$-dimensional coordinate system, and thus can be conceived as nodes of a graph. Hence, the focus of this thesis lies on the study of graph-based approaches. The basic idea is to store the difference vectors between always two points instead of the minutiae. For this purpose directed spanning trees allow an efficient encoding. Hence in the course of this thesis different approaches based on specific spanning trees, like e.g. the directed *minimum spanning tree*, the directed *minimum label spanning tree* and the directed *weight balanced spanning tree*, have been studied, and a compression of up to approximately 20% could be achieved.



# Zusammenfassung

Biometrische Templates enthalten die zu einer Person gehörenden biometrischen Daten, wie z. B. Fingerabdrücke oder Gesichtsmerkmale, die vor allem zu Verifikationszwecken immer mehr an Bedeutung gewinnen. Damit eine Person jedoch anhand ihrer biometrischen Merkmale erkannt werden kann, muss stets ein Referenztemplate zur Verfügung stehen. Dieses kann in einer Datenbank, aber z. B. auch auf einem RFID-Chip gespeichert sein. Gerade im Hinblick auf mobile Speichermedien und damit verbundene geringe Speicherkapazitäten besteht ein Bedarf an der Kompression von biometrischen Templates. Diese Kompression kann verlustbehaftet sein, sollte dabei jedoch mögliche Fehler bei der Verifikation so gering wie möglich halten. Im Speziellen wird in dieser Masterarbeit ein neuer Ansatz zur Kompression von *Fingerabdrucktemplates* entwickelt. Diese Templates enthalten dabei Informationen zur Lage und Orientierung der so genannten Minutien, d. h. den Endungen und Verzweigungen von Papillarlinien. Diese Informationen werden als Punkte in einem $d$-dimensionalen Koordinatensystem dargestellt und können auch als Knoten eines Graphen aufgefasst werden. Der Schwerpunkt dieser Arbeit liegt daher auf der Betrachtung von graphenbasierten Kompressionsansätzen. Die zugrunde liegende Idee dabei ist, anstelle von Punktkoordinaten die Differenzvektoren zwischen jeweils zwei Punkten zu speichern. Gerichtete Spannbäume ermöglichen dafür eine effiziente Kodierung. Im Zuge dieser Arbeit wurden daher Ansätze basierend auf speziellen Spannbäumen, wie z. B. dem gerichteten *Minimum Spanning Tree*, dem gerichteten *Minimum Label Spanning Tree* oder auch dem gerichteten *Weight Balanced Spanning Tree*, untersucht und somit eine Kompression von bis zu circa 20% erreicht.

# Acknowledgments

At this point I would like to thank those persons who made this Master's Thesis possible or even made a great contribution. In particular these are Günther Raidl and Andreas Chwatal from the Department of Algorithms and Data Structures of the Vienna University of Technology who provided me valuable ideas and an excellent support. Furthermore I would like to thank the people at m2n who finally gave me the needed time for finishing my studies. Of course, thanks to my significant other, Stefan, for sustaining, loving and enduring me in never ending times of work. And finally thanks to my parents for their lasting support and motivation and for never giving up faith in me.

# Contents

# Chapter 1 - Introduction

*"A long journey starts by taking the first step."*
Chinese Saying

Biometric identification stands for an automatic recognition of people by means of their characteristic traits or behaviors, such as body height, fingerprints or even the mode of speaking. These individualizing characteristics have been already known to the ancient peoples. Nowadays, we ever more frequently have to prove our identify to technical systems, which is mostly done with the aid of keys or passwords. However, by knowing a password or possessing a key, a person can never prove whether he is really the one who he claims to be. Thus biometric traits promise a higher certainty in people recognition since they are strongly bound to its owner, and cannot be removed under normal conditions.

Especially fingerprints have turned out to be an appropriate mean for identification purposes, since they do not change over time, and until now no two people having the same fingerprints have been found [Maltoni03]. Furthermore they have been extensively studied since the mid of the $19^{th}$ century, and since the beginning of the $20^{th}$ century they are applied for the identification of criminals. Due to the technological evolution, and thus the development of automated fingerprint recognition systems, they became interesting for a widespread use on the private sector, too. A recognition by means of fingerprints is in many cases based on comparing certain points of interest of the dermal papillae, the so-called minutiae. That are those points of the fingertips where the dermal ridges either end or bifurcate. In order to recognize a person an initial set of those minutiae has to be available, such that it can be compared to a freshly taken input sample of a fingerprint. This initial set is stored as a template, which in turn contains the absolute coordinates of the minutiae.

Although biometric characteristics are a secure mean for the recognition of people, they are not immune against misuse. Whereas passwords can be replaced by new ones, fingerprints cannot. Thus, for instance in order to make the new European biometric passports more forgery-proof, an additional embedding of both taken fingerprints into the digital photograph in the form of a digital watermark is considered. Such an embedding in turn must be resistant against intentional changes on the carrier image, and also against random bit flips. Thereto on the one hand redundancy is necessary, and on the other hand a strong embedding of the minutiae data into the carrier image. Such an embedding technique was introduced in [Jain02]. However, since the digital passport photographs are of very small size, an embedding of both fingerprint templates using this technique is not possible. Hence, the task of this thesis is to compress the fingerprint minutiae templates. Since the minutiae can be represented as $d$-dimensional points in absolute coordinates, thereto especially graph-theoretical approaches are going to be taken into account. The main idea is, to store the minutiae by means of a subset of their difference vectors, trusting that this representation uses a fewer number of bits than the original template.

Such a subset is for instance given by the edges of a spanning tree. In order to determine a spanning tree on the minutiae of a given template, different approaches were analyzed and implemented within this thesis. A first and very simple approach uses a directed minimum spanning tree. The second and main approach uses some kind of dictionary encoding, in which a small set of so-called reference vectors is going to be stored. Then

the edges of that tree can be represented by a reference to a dictionary entry together with a small correction vector. Since in this case the dictionary entries must be stored as well, a possibly small set of reference vectors is to be found. In order to determine the smallest possible set of dictionary entries from a large set of previously determined ones, a variant of the directed minimum label spanning tree [Chang96] is going to be applied. Finally the third and last approach turned out to be a special case of the second one, where only one dictionary entry is required, namely an offset vector. Hence, on the set of minutiae a tree is sought, in which the edges can be represented by the greatest common offset, and thus only some small correction vectors have to be stored. For solving this problem the directed weight balanced spanning tree [Gupta04] will be applied.

In order to analyze and implement these approaches, heuristic algorithms were developed and a framework using the C++ language was implemented. Whereas conventional compression techniques, such as ZIP, consistently enlarge the size of the given templates, the analyzed approaches achieve a reduction of up to approximately 20%.

The main ideas, algorithms and results are going to be presented within this thesis. Thereto the next chapter will give a short overview about biometrics, whereas in chapter 3 a more detailed description about fingerprint recognition will be given. Chapter 4 will refer to the concrete problem and will present the application background as well as common compression techniques, such that a differentiation to the approach of this thesis can be given. Then in chapter 5 the approaches will be introduced in detail, whereas in chapter 6 the implemented algorithms will be explicitly described. Moreover the implemented framework is going to be briefly introduced in chapter 7. Finally the test data and results are stated in chapter 8.

# Chapter 2 - Biometrics

*”A chain is only as strong as its weakest link.“*

Saying

Already in ancient Egypt when grain was delivered, or contracts were concluded, humans were confronted with the problem of authenticating unknown people. For this purpose visible body characteristics, such as eye color, body height, scars or even the complexion of a person were used. However other nations, like the Chinese or the Babylonian, were also familiar with these individualizing traits. Thus Assyrian potters marked their clay vases with their fingerprints.

Nowadays the task of recognizing people is no longer restricted to humans. As part of the process of the technological evolution we ever more frequently have to prove our identity to technical systems. Be it in order to withdraw at cashpoints, to order goods in the web, or even to get permission for entering a specially secured area. There exist various means to identify oneself to a human or a machine [BioBSI07]:

- *Knowledge*. Everything a person can know, such as passwords, personal identification numbers (PIN), and even watchwords, how they are often used by military, in order to distinguish between friends and enemies.
- *Possession*. Things a person can possess, e.g. keys, key cards or identity cards.
- *Being*. All physiological and behavioral characteristics of a human.

Currently, a lot of authentication methods are based on knowledge and possession. However, it can never be guaranteed that a person is really the one who he claims to be, since knowledge and possession can get into the wrong hands. Because biometric traits promise a greater certainty in people recognition, biometric systems get more and more attractive. An advantage of such systems consists in the fact that characteristics and behaviors are strongly bound to its owner and cannot be removed under normal conditions.

First biometric systems are available since the 60s in the form of fingerprint recognition systems. In the 70s hand geometry scanners followed, and since the 80s there is ongoing research on iris and retina scans [BioBSI07]. With continual development such systems gain more and more significance, even on the private sector.

The following section will give a short introduction to biometrics, biometric traits and the authentication process.

## 2.1 Definitions and Functions

Derived from the ancient Greek language the term *biometrics* specifies the ”counting and measuring of living organisms“ [Duden96]. In technical domains this term is often used instead of *biometric identification*. However, this is a process similar to the human recognition with the difference that the used sensors can capture characteristics, which humans cannot, e.g. fingerprints or the retina structure [Heumann06].

The underlying procedure is called *biometric process*. It covers both, a first collection of at least one biometric trait, called reference trait, using a compatible sensor, and a comparison of this reference trait with a fresh input sample.

Thus, a *biometric system* is nothing else than the combination of hard- and software for implementing the biometric process.

Finally, the goal of each biometric identification is to prove or disprove the identity of a person by means of biometric traits. For this purpose biometric systems can provide two possible functions:

- *Identification*. In this case a current input sample is compared to all reference traits that are stored in a database. If there exists exactly one reference trait that is sufficiently equal to the input sample with respect to a predefined threshold, a person is being considered recognized. Because of the number of comparisons the identification process is also called *one-to-many* comparison.
- *Verification.* In a verification process, also called *one-to-one* comparison, a current input sample is only compared to a single reference trait. If the person possesses this reference trait he just has to provide it to the biometric system. Otherwise he must identify himself to the system by means of knowledge or possession, like e.g. a password or a personal identification number. Then the system picks out the appropriate reference trait from its database. Thus, it is only checked, whether the person is really who he claims to be. One advantage of this function is the possibility of a decentralized storage of the reference traits.

Due to natural variations of human characteristics and inaccuracies in the respective measuring methods only a certain degree of similarity can be determined [Behrens01]. Hence in both cases a person is only being considered recognized if this degree exceeds a predefined threshold.

## 2.2 Biometric Traits

Biometric traits are the amount of all physiological and behavioral characteristics of a human.

- *Physiological* (or *static*) characteristics are for example the body height, fingerprints or the facial structure. These traits are generally inherited, but they can also result from randomized processes during the embryonal stage. Under normal conditions they are subject to only small temporal variations.
- *Behavioral* (or *dynamic*) characteristics are based on an active doing. Mostly they are semiskilled, but they can be inherited, too. The dynamics of a voice, the typing behavior on a keyboard, but also the signature are of this kind.

But not all characteristics are applicable for a biometric identification. Some are more eligible than others. Hence, there are different criteria, to pick out the best ones [Behrens01]. Basically, a biometric trait should be:

- *universal* – that is, that each person should have this specific characteristic,
- *unique* – this means that this characteristic should differ from person to person,
- *permanent* – the characteristic should be resistant against aging, and finally
- *collectable* – that is, that the specific characteristic should be easily ascertainable for measuring purposes.

A biometric trait is being rated as perfect, if it fulfills all these criteria. Especially due to temporal variations of dynamic characteristics, and a relatively easy possibility to imitate them, static characteristics seem to be more eligible to a biometric identification process.

## 2.3 Biometric Identification Process

Independently from the used biometric trait and the desired function, i.e. verification or identification, the biometric process can be split up into three subprocesses. In order to recognize a person, he must be already registered to the biometric system. This happens during the so-called *enrollment*. Thereby the desired biometric traits are captured and stored as reference traits. If this person must identify himself at a later date another *trait collection* takes place. Afterwards the stored trait and the freshly acquired input sample are compared. The result of this *matching* process indicates whether the person could be successfully identified or verified, respectively.

### 2.3.1 Trait Collection

During the trait collection process at least one physiological or behavioral characteristic is captured by a capable sensor. Depending on the further use, the obtained raw data (like e.g. an image) may either be used directly, or an optional feature extraction and template generation takes place.

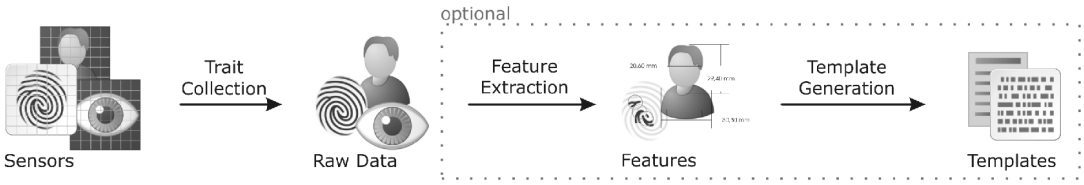



*Figure 1 - Trait collection: By means of a suitable sensor a biometric trait is acquired, that optionally can be scanned for characteristic features, which are in turn used for the template generation.*

Templates are mainly used if only marginal disk space is available. For a template generation the raw data is scanned for characteristic features, which are finally stored into a file. In case of fingerprints, such features are e.g. the number, position and type of certain points of the finger ridges.

### 2.3.2 Enrollment

The enrollment is the most important process of a biometric identification. It covers the initial trait collection on the one hand and the storage of the captured data on the other hand.


*Figure 2 - Enrollment: The acquired traits are stored e.g. in a database or on a smartcard.*

With respect to a good recognition rate, the biometric traits must be taken under best conditions, e.g. according to lighting. Mostly, they are captured more than once, because a later recognition should be even possible if the conditions during the respective collection process are much worse [Heumann06]. Subsequent to the initial collection phase the captured traits, the so-called *reference traits*, are stored for example in a database or on a smartcard. This depends on the further use. In case of an identification the data is most likely saved into a database. For verification purposes the reference trait may be stored on a smartcard.

### 2.3.3 Matching

The result of the matching process finally indicates whether a person could be successfully identified or verified, respectively. The process itself depends on the recognition function.

- *Identification*. Subsequent to the trait collection phase the captured input sample is compared to all reference traits in the database of the biometric system. If there is found exactly one sufficiently equal trait according to a predefined threshold the system returns a successful identification. Otherwise a recognition is not possible.

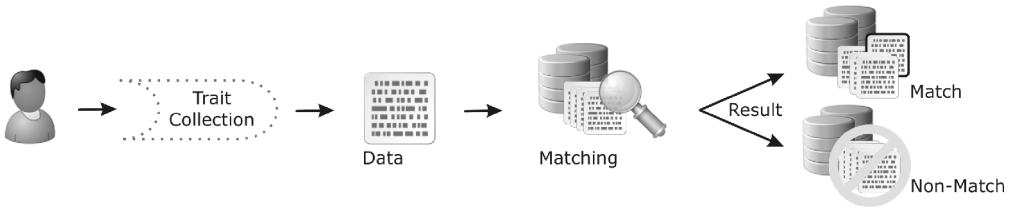



*Figure 3 - Matching (Identification): A fresh input sample is compared to all reference traits contained in database. The person is identified if exactly one corresponding trait is found.*

- *Verification*. Here the biometric system first has to obtain the reference trait belonging to the user. If the user for instance possesses a smartcard on which the reference trait is stored, he could provide it directly to the system. Otherwise he somehow has to unveil his identity, e.g. using a password or a PIN. In that case, the biometric system picks out the respective reference trait from its database. Finally, the obtained trait and the captured input sample are compared. If the computed similarity exceeds the predefined threshold, the user is being recognized successfully.

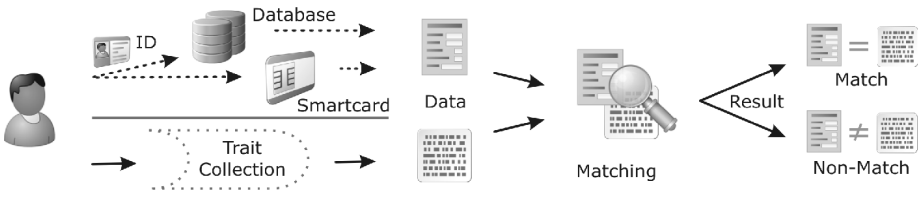



*Figure 4 - Matching (Verification): Parallel to the trait collection the user unveils his identity to the biometric system by means of knowledge or possession. In doing so, the system is able to acquire the reference trait belonging to the user, and hence only these both traits are compared.*

The predefined threshold is responsible for proving or disproving the identity of a person. But it is also responsible for the efficiency of a biometric system.

## 2.4 Requirements of Biometric Systems

How efficient a biometric system works, depends not only on the kind of the used physiological or behavioral characteristics but also on the recognition precision of the system itself. An exact match of two isomorphic images is not possible, since biometric characteristics are subject to temporal variations. And also changes in lighting conditions, make-up, carrying of a beard or a head coverage make each captured trait unique. As a result only a certain similarity can be determined. Depending on a threshold and the computed degree of similarity the biometric system proves or disproves the identity of a person. Due to this threshold, non-isomorphic characteristics can be recognized as identical, and vice versa. Hence, the efficiency of a biometric system can be determined according to some basic rates, which e.g. specify the number of incorrect or correct recognized persons. The most important are:

- FMR (*False Match Rate*) – which specifies the percentage of wrongly accepted persons, and
- FNMR (*False Non-Match Rate*) – which indicates the percentage of wrongly non-accepted persons.

Besides the *efficiency* there exist more requirements, that a biometric system should meet. Hence such a system should be [Behrens01]:

- *technically realizable* – it must be possible to distinguish a sufficiently large number of persons,
- *cost-effective* – the costs must be appropriate and sustainable,
- *resistant against circumvention* – the system should be robust against direct attacks that are able to outwit the system, and finally
- *acceptable* – the user must be willing to use the specific physiological or behavioral trait in a recognition process.

After giving a short survey about biometrics, the following chapter will give, with regard to the task of this thesis, a more detailed overview about the biometric recognition process by means of fingerprints.

# Chapter 3 - Fingerprint Recognition

*„The most secure locked door is that, which can be left open.“*

Chinese Saying

With regard to the biometric identification process fingerprints can be classified as perfect. Although they are nothing else than the impression of the epidermal ridges of a finger tip. These so-called dermal papillae evolve from randomized processes during the embryonal stage and remain a whole life long. Since lots of inspections on millions of fingerprints have been performed by experts (beneath them Faulds and Galton), they are being considered unique. But this assumption is only based on the fact, that until now no two individuals have been found, having the same fingerprints [Maltoni03]. Even identical twins are supposed to have different ones.

Under normal conditions every human (but also a huge amount of other mammals) has fingerprints. But there are exceptions that are for example based on genetical defects, skin diseases and also on exterior influences, such as cutting damages or abrasion by mechanical work.

The individuality of a fingerprint results from the texture, arrangement and orientation of the dermal papillae. These form on the one hand large-scale patterns, like e.g. arches, whorls or loops, and on the other hand these lines themselves are littered with tiny characteristics, the so-called minutiae. That are for example sudden ridge endings, ridge bifurcations and also sweat pores.

The usage of fingerprints for identification purposes reaches back to Babylonia, Assyria, China and Japan. And already in ancient India contracts were signed with fingerprints. However, a wide-spread use began not before the end of the 19th century.

## 3.1 History of Fingerprints

First scientific research on fingerprints was performed in the late sixteenth century by the English plant morphologist Nehemiah Grew, who published a paper in 1684 about his systematic studies on the ridge, furrow and pore structure of fingerprints. Further research on the anatomical formations of fingerprints by Mayer in 1788 and a first classification scheme by Purkinje in 1823 followed [Maltoni03].

But a British colonial civil servant named Sir William James Herschel, grandson of the astronomer Sir Friedrich Wilhelm Herschel, was the first European with the idea to distinguish people by means of their fingerprints. In 1858 he introduced the dactyloscopy (Greek: *daktylos* – ”finger“, *skopein* – ”to view“) in Calcutta in order to prevent identity swindle in the payment of pensions [Beavan01]. For that purpose a fingerprint of each pensionable Indian was collected. All future payments had to be signed by a new impression of the same finger. This procedure was also introduced in Indian prisons in order to guarantee the identity of exhibited convicts. In the course of time Herschel collected that way several thousand fingerprints. Due to the practical success of the dactyloscopy he tried to implement this procedure in the entire British empire. However, he remained without success.

Henry Faulds lived in Japan and in about 1870 he took notice of prehistoric potteries that were marked with fingerprints. He started to analyze those skin furrows. However, his research were focused on ethnological differences. Therefore he collected fingerprints of various nations, even from apes [Ihmor03]. In doing so he discovered different patterns in the center of each fingerprint, which he called *loops*, *arches* and *whorls*. Thereupon he developed a classification system, the so-called *Henry System*, that is still today used by experts to compare fingerprints. In 1880 he published an article in the scientific newspaper "Nature" where he discussed fingerprints as a mean for people identification. He also pointed out that fingerprints being left on a crime scene might convict offenders or disburden suspects. His efforts to convince criminal investigation departments from diverse countries of his idea, including the Scotland Yard, remained yet unsuccessful.

Also Francis Galton examined differences between fingerprints from different nations and races. Eventually he came to the conclusion that there exist none. In 1888 he was delegated by the British colonial government to develop an uncomplicated system for people identification. But before the dactyloscopy could be used for police purposes, some questions had to be answered [Galton03]:

- Are the epidermal ridges resistant against aging?
- Are the differences between all fingerprints sufficiently large so that thousands of criminals can be easily distinguished?
- Assuming that a fingerprint is already contained in a card index. A second one of the same finger is supposed to be inserted into that index, too. Is there a classification possible, so that the already in the index contained fingerprint may easily be found?

Finally, in 1892, with the help of the research results from Herschel and Faulds, Galton answered to all questions with YES.

A commission that was brought into being in 1887 by the Federal Ministry of Internal Affairs in India in order to prove different possibilities for the identification of criminals, also took Galton's work into account. In 1897 it came to the result that the dactyloscopy is most eligible. Thereupon the fingerprint procedure was introduced to entire India [Ihmor03].

Faulds, who afterwards was appointed to the position of the chief of the police to London, introduced the dactyloscopy in 1901 to England. Other countries followed gradually. Among them Saxony in 1903 and France in 1914.

In the 60s, first computer-based systems for detective purposes, the so-called AFIS systems (Automated Fingerprint Identification Systems), were designed for analyzing and comparing fingerprints. Optical finger sensors succeeded in the 80s. And also algorithmic enhancements have been performed, so that the dactyloscopy even became eligible for people recognition [Behrens01].

## 3.2 Distinctive Features of Fingerprints

Faulds supplied with his Henry System a possibility by which fingerprints can be roughly distinguished. Today's classification procedures are based on his system. However, they consider additional characteristics which result among others from the texture of the epidermal ridges. Thus, two certain lines in the center of a fingerprint – the *type*

*lines* (see figure 6) – constitute an area that is called *pattern area*. Within this area large-scale *patterns* as well as *singularities* can be found.

### 3.2.1 Singularities

These singularities are certain points within the pattern area which result from the arrangement of the epidermal ridges. Thereto belong for example the *delta* and the *core* [FingerBSI06].

- A *delta* may be configured by two diverging dermal papillae which form the Greek letter Δ (see figure 5A). But also a bifurcating epidermal ridge to whom another convex ridge comes from a third direction may form such a delta (see figure 5B and 5C)). Because of its location near to the margin of the pattern area a delta is often called *outer boundary*. A fingerprint may contain one, many, or no delta.
- The *core* of a fingerprint is commonly a freely chosen point between the type lines that represents the center of the respective pattern (see figure 5A + 5B). If there is more than one pattern, a point within the center of all is chosen (see figure 5C).

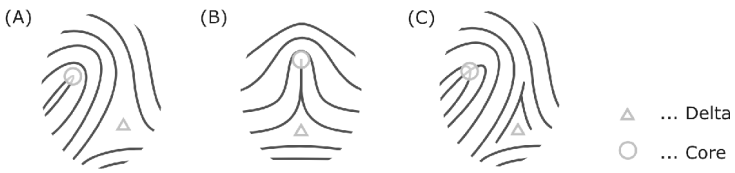



*Figure 5 - Some delta & core configurations.*

### 3.2.2 Patterns

Large-scale patterns, which were described by Faulds, can be classified into loops, whorls and arches [FingerBSI06].

- A *loop* is characterized by several epidermal ridges, which curve within the pattern area and return to the side from which they came. In doing so, the papillae either traverse the imaginary line between the core and the delta or they are tangent to it (see figure 6). Depending on the direction of a loop a distinction into a *left* or a *right loop* is performed. About 60 – 65% of all human fingerprints belong to this category.
- A *whorl* is characterized by two facts: On the one hand at least two deltas must be available. On the other hand the dermal papillae in the center of the pattern area must constitute a whirl. Depending on the declination and the number of whirls a distinction into *plain*, *central pocket* and *double loop whorls* can be made. To this category belong about 30 – 35% of all fingerprints.
- The *arch* pattern occurs in less than 5% of all cases. It can be subclassified into *plain* and *tended arches*. Plain arches are characterized by epidermal ridges that run almost parallel from one side of the impression to the other without forming a delta. However, the curve of the dermal papillae in a tended arch is so strong, such that a delta is formed.

Fingerprints can be categorized by means of singularities and patterns. That is why these features are especially used in systems, where a huge amount of fingerprints has to be compared (e.g. AFIS). However, in automated access control systems mostly a pure minutiae match is performed.

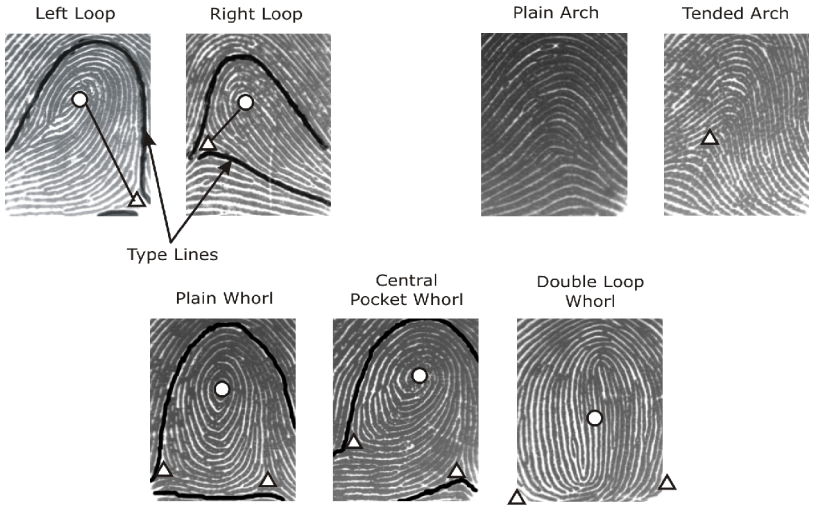



*Figure 6 - Various kinds of fingerprint patterns, type lines, deltas and cores. (Fingerprint images taken from the FVC2004 Database DB1_A, see [FVC04])*

### 3.2.3 Minutiae

Minutiae are those points which give individuality to each fingerprint. They are tiny characteristics of the epidermal ridges. The most frequent ones are:

- *Ridge Ending* and
- *Ridge Bifurcation*.

Both are certain points, where a dermal papilla either ends or bifurcates into two (or more) branches. But there are much more types, like e.g. *ridge crossings*, *bridges* or even *sweat pores*. A total of about 150 different kinds is known [Maltoni03]. A complete fingerprint typically contains between 40 and 100 of such minutiae.

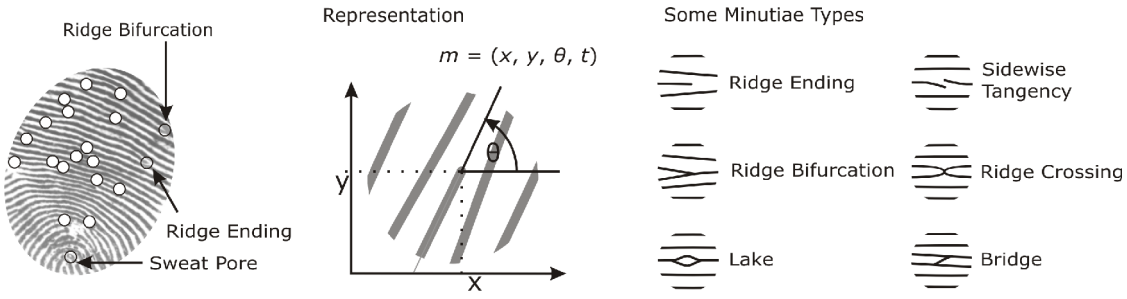



*Figure 7 - Minutiae representation and types*

A representation is usually given by a position, i.e. a point $p = (x, y)$ in a coordinate system, a type $t$ and an orientation angle $\theta$, which is measured counterclockwise between the horizontal $x$-axis and the tangent to the epidermal ridge in the point $p$. Hence, in a mathematical way a minutia can be understood as a vector $m = (x, y, \theta, t)$.

All minutiae together establish a template $T = (m_1, ..., m_n)$ that is used for the comparison of fingerprints. Templates are stored binary. For that purpose there are various standards, like BioAPI 1.1, that is based on the American standard ANSI/INCITS, or ISO/ICE 19794-2:2005 [ISO19794-2], which is based on DIN V66400 [DINV66400]. Therein 5 bytes are needed to store a single minutia. Another more compact storage for-

mat is also defined in the ISO standard, in which only 3 bytes are used. Additional data, such as the number of minutiae or the size of the template, are stored as metadata.

## 3.3 Fingerprint Analysis

The biometric identification process was already described in chapter 2. A recognition by means of fingerprints is very similar to this. First an image of the finger is gathered. All features, that are needed for the matching process are extracted from that image and afterwards compared with the stored reference traits. The choice of the capturing method as well as the following image processing techniques are significantly responsible for the quality of the extracted features, and hence for the result of the authentication process. Therefore the following sections will give a short introduction to some capturing techniques, feature extraction steps and the final matching process.

### 3.3.1 Image Acquisition

There are two basic techniques for acquiring an image of a fingerprint, *off-line* and *on-line*. In both cases a gray-scale image is produced, in which the ridges appear dark and the furrows light.

#### Off-line Sampling

For an off-line recognition the finger is coated with color and afterwards evenly unrolled on a document from one nail side to the other. This off-line print is subsequently digitalized with the aid of a scanner or a digital camera.

Due to the evenly unrolling of the finger the whole ridge information is visualized, including the patterns, singularities and minutiae. Nevertheless, this technique may lead to distortions. Also unavailable feedback possibilities for quality assurance purposes disqualify this technique for instance for automated access control systems.

#### On-line Sampling

The on-line sampling techniques are much more important for an automated biometric recognition. Here a person puts his finger on the surface of a sensor. Thereby only parts of a fingerprint are acquired. But quality assurance can be performed during the acquisition, due to a feedback to the image processing component is available. Analogously to the off-line sampling image distortions may occur. The quality of a sensor finally depends on its ability to deal with different skin properties, such as dryness, wetness or even dirt. According to [Maltoni03] current sensors can be subclassified into:

- *Optical Sensors*. This kind of sensor is currently the most popular one. The user puts his finger on the transparent glass layer (prism, fiberglass) of the sensor, such that the epidermal ridges are in direct contact with the sensor. A light-emitting source within the sensor illuminates the glass layer from beneath in a way, such that the rays of light are reflected only by the dermal papillae but not by the furrows. The returning rays are captured by light sensitive elements, like for instance photo diodes. All captured information is finally used for creating a gray-scale image of the fingerprint. Since the finger is always put directly on the sensor, the surface is frequently contaminated and hence must be cleaned from time to time. Another disadvantage is the

possibility to outwit the system by means of an artificial finger. However, images of a high resolution (up to 500dpi) are possible.

- *Solid-State Sensors*. Solid-state, or silicon sensors, are composed of a matrix of smaller sensors, where each one corresponds to a pixel in the final gray-scale image. These kind of sensors are able to measure various physical values. So they are subclassified for instance into:
  - *thermal sensors*, which capture the infra-red image of a finger,
  - *capacitive sensors*, where the sensor surface in combination with the skin of a finger generates a condenser, whose capacity differs according to the skin relief, and
  - *electric-field sensors*, which measure the local differences in the electric field at the surface of the skin. These differences occur, because the sensor emits small electric impulses.

  These kind of sensors come also into direct contact with fingers and hence must be frequently cleaned.
- *Ultrasound Sensors*. Ultrasound sensors measure the distance between the surfaces of finger and sensor by emitting an ultrasound signal which is reflected by the finger and finally captured by the sensor. The waves are not reflected by dirt. Furthermore a direct contact to the sensor is not necessary, thus ultrasound sensors are robust against contamination. The images being returned are of premium quality, but the acquisition process may take some seconds. And also because of the price and relative size these scanners are currently ineligible for a large-scale use.

The gray-scale images which finally emerge from the scanning process are mostly of varying quality. In order to provide a more or less homogeneous basis for feature extraction, different image processing steps are performed.

### 3.3.2 Image Processing Steps

The so-called *orientation image* is a basis for a large number of extraction steps. It represents the local orientations of the dermal papillae. To create such an image, the gray-scale image is partitioned into a certain number of blocks, which for example may consist of only one pixel. The local orientation of each block is computed by means of the epidermal ridge within the respective block, whereas the orientation is the angle between the horizontal $x$-axis and the tangent to the ridge. The set of all blocks finally represents the orientation image. Because of noise in the original image the orientation image can be very irregular. That is why a harmonization is performed, which adapts the orientations of the blocks at each other in a way, such that the orientations of neighboring blocks only differ slightly. This happens under the assumption, that the epidermal ridges underlie a certain regularity.

Orientation images can for example be used for filtering noise, such as sensor noise, cutting damages of the finger, or badly separated dermal ridges. Therefor a *frequency image* is created, which represents an estimation of the local density of the epidermal ridges. For every block of the orientation image a rectangle is constructed that is vertically oriented to the local orientation line. The frequency is represented by the number of ridges, which cross this rectangle.

There are more image processing steps, like e.g. the increase of contrast, or the *segmentation* which separates a fingerprint from the background, such that an extraction of features, which may emerge from background noise, can be avoided.

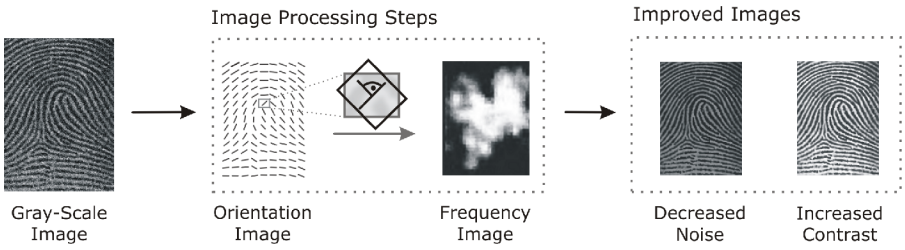



*Figure 8 - Image processing: A gray-scale image is transformed into an orientation image which is in turn used for creating a frequency image. Those and further improvements may for instance reduce noise or increase contrast. (Fingerprint image taken from [Maltoni03])*

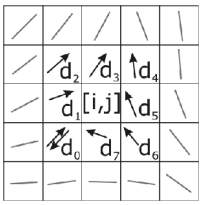
All these steps are performed in order to enhance the quality of an image and finally, generate a basis for a subsequent feature extraction.

### 3.3.3 Feature Extraction

The feature extraction is performed in order to find all individualizing features of a fingerprint. While patterns and singularities are mainly used for classification purposes, minutiae are needed for generating templates.

#### Pattern and Singularity Detection

One of the most commonly used technique for the determination of singularities and patterns is the *Poincaré* method which uses an orientation image as basis [Basler05]. For each block $[i, j]$ the *Poincaré index* is computed over all eight neighboring blocks $d_k$:




$$P(i,j) = \sum_{k=0}^{7} \sphericalangle(d_k, d_{(k-1) \bmod 8}) \qquad (1)$$

*Figure 9 - Computation of the Poincaré index: On the left side a cutout from an orientation image is shown, where the orientations of the neighboring blocks of [i, j] are already directed. The formula on the right represents the computation of the Poincaré index.*

Here $\sphericalangle(d_k, d_{(k-1) \bmod 8})$ defines each time the difference between the orientation angles of two clockwise viewed neighboring blocks. Since for the summation of the angle differences directed orientations are expected [Maltoni03] the direction of the first neighboring block is arbitrarily chosen. The direction of the next block is determined in a way, such that the resulting difference to the previous block is minimal. It has been proven that the Poincaré index on closed curves only adopts the values 0°, ±180° and ±360° [Maltoni03]. Hence for the pattern recognition holds:

- 0° ... the block neither belongs to any pattern nor contains any singularity
- 360° ... the selected block is part of a whorl pattern

- 180° ... the selected block is part of a loop pattern
- -180° ... the selected block contains or is part of a delta

There are more techniques for pattern and singularity detection, which e.g. are based on local characteristics or even on a partitioning of the orientation image. However, they will not be regarded in this thesis.

### Minutiae Detection

For the extraction of minutiae from a gray-scale image there exist mainly two possibilities. The minutiae are either extracted directly by tracking the epidermal ridges or the gray-scale image is used for creating a binary image, which is in turn used for extracting the minutiae [Maltoni03].

In case of a direct extraction from a gray-scale image the fingerprint is understood as a three-dimensional function. To each point $p = (x, y)$ in the image, $f(p) = z$ is the respective gray-scale value. An epidermal ridge is a set of local maxima of those gray-scale values. The minutiae are determined by tracking the dermal papillae. The tracking algorithm starts in an arbitrary point $p_i$ on an arbitrary epidermal ridge and covers a fixed distance into the direction of the orientation of $p_i$ (see figure 10).

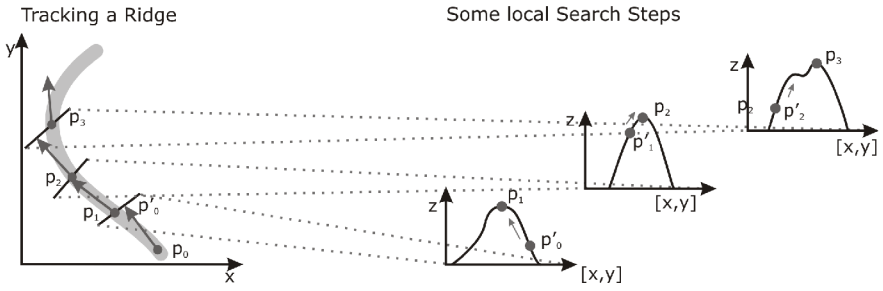



*Figure 10 - Extracting minutiae from gray-scale images: After covering a fixed distance starting in a point $p_i$ into the direction of its orientation a local search is performed, that looks for the local maximum $p_{i+1}$ on the vertical to that orientation. That point $p_{i+1}$ is starting point for the next iteration, which are repeated until the dermal papilla either ends or runs into another one.*

The reached point $p'_i$ defines the starting point for a local search that looks for the local maximum $p_{i+1}$ on the vertical line to the orientation of $p_i$, which is in turn the starting point for the next iteration. This is repeated until the examined papilla ends (ridge ending) or runs into an already analyzed one (ridge bifurcation). Iterated over all ridges, finally all minutiae should be detected.

In case of extracting the minutiae from a binary image, the existing gray-scale image is first converted to a black-and-white image. Afterwards a line thinning is performed, i.e. a normalization step that reduces the width of each ridge to exactly one pixel. Finally the minutiae extraction is done by means of the so-called *crossing number* $cn(p)$ of a pixel $p$, that specifies the number of lines which run into $p$. For determining the crossing number, the neighboring pixels $p_k$ are taken into account:

$$cn(p) = \frac{1}{2} \cdot \sum_{k=0}^{7} \left| val(p_k) - val(p_{(k-1) \bmod 8}) \right| \qquad (2)$$

Here the function *val*() specifies the color value of the respective pixel. "0" stands for white and "1" for black. A pixel $p$ is being recognized as a minutia, if the crossing number

equals 1 (ridge ending) or 3 (ridge bifurcation). Since previous normalization steps may have added additional artifacts to the binary image, the number of minutiae might be artificially enlarged. Therefore some succeeding steps are performed in order to filter the best-known wrong minutiae structures [Basler05].

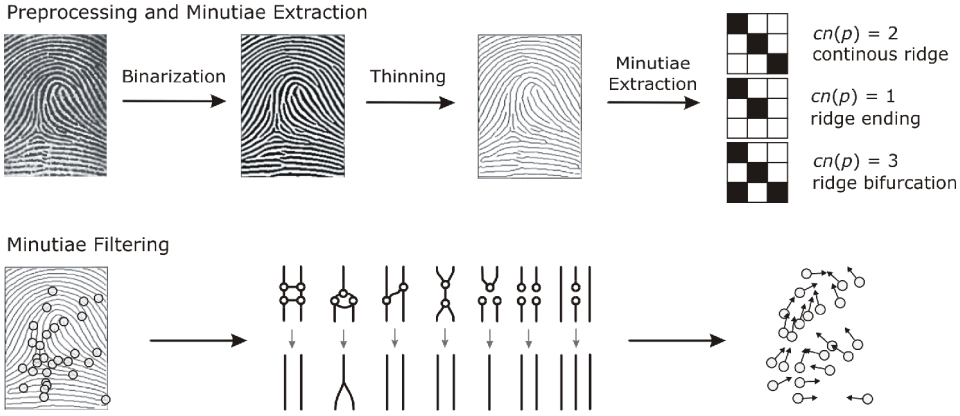



*Figure 11 - Extracting minutiae from binary images: After a binarization and thinning of the gray-scale image the minutiae are extracted with the aid of the crossing number. A subsequent filtering step removes wrong minutiae structures. (Fingerprint images taken from [Maltoni03])*

### 3.3.4 Matching

Bad image quality or errors resulting from the image processing steps, like e.g. additional or wrongly filtered minutiae, are some of the problematic facts which may occur during the matching process. Additionally every impression of a fingerprint is unique, since a finger can never be put twice on a sensor in exactly the same way. This means that a matching algorithm should also consider variations in the location or rotation of a fingerprint. But also distortions, differences in the resolutions of the images, or pollution of the finger that may lead to noisy fingerprint images, can finally influence the matching results. Currently, there are three common matching methods available [Maltoni03]:

- The *ridge feature-based* method is basically used in forensic institutes, because it is especially qualified for erroneous fingerprints and those of bad quality. The impressions are compared on the basis of characteristic features, such as orientation, frequency and form of the epidermal ridges. Both other methods can be conceived as sub-classes of this one.
- The *correlation-based* matching procedure performs a pixel comparison between the reference and the input image in various positions and rotations.
- The third method is the *minutiae-based* one. Here the minutiae are compared on the basis of their placements and orientations. Since this technique considers a lot of the problematic issues, it belongs to the most reliable ones, and thus will be shortly illustrated now.

As already mentioned, minutiae can be mathematically represented by means of vectors: $m = (x, y, \theta, t)$. In turn, templates are nothing else than a vector of minutiae. During the matching process the reference template $T^R = (m_1^R, \ldots, m_k^R)$ is compared to the input sample $T^I = (m_1^I, \ldots, m_n^I)$ with the aid of two distance measures:

- the *Spatial Distance* (*SD*)

$$sd(m_j^I, m_i^R) = \sqrt{(x_j^I, x_i^R)^2 + (y_j^I, y_i^R)^2} \leq r_0 \quad (3)$$

with $i = 1 \ldots k$, $j = 1 \ldots n$, and

- the *Direction Difference* (*DD*)

$$dd(m_j^I, m_i^R) = \min(|\theta_j^I - \theta_i^R|, 360° - |\theta_j^I - \theta_i^R|) \leq \theta_0 \quad (4)$$

If both results are located within the tolerance boxes $r_0$ and $\theta_0$, the minutiae are being considered mated. Since common minutiae matching algorithms do not consider the minutiae types, they are not mentioned here. However a comparison, that also checks the types on equality, is conceivable.

In order to determine, how many minutiae of an input sample coincide with those of a reference template, various mathematical transformations are performed on the position and orientation vectors of the input template. The most important ones are the relocation of the *x*- and *y*-coordinates as well as the rotation of the orientation vector, but also scaling and all further affine transformations. These transformations are performed until a maximum number of matching minutiae is found. Mathematically spoken:

$$\max_{\Delta x, \Delta y, \theta, P} \sum_{i=1}^{m} mm(map_{\Delta x, \Delta y, \theta}(m_{P(i)}^I), m_i^R) \quad (5)$$

At this, *map*() defines a function, which maps a minutia $m_j^I$ from the input sample into $m_j^{I'}$ according to the transformation rules mentioned above. Thereby $\Delta x$ and $\Delta y$ are the relocation of the *x* and *y* coordinates and $\theta$ is a rotation of the orientation vector. Furthermore *mm*() is an indicator function that returns "1", if $m_j^{I'}$ and $m_i^R$ match according to the distance measures *SD* and *DD*. $P(i) = j$ finally describes an unknown pairing function between the input sample and reference template, whereby a minutia $m_i^R$ of the reference template has either exactly one or no respective minutia $m_j^{I(')}$ in the (transformed) input sample. But a pairing does not necessarily mean, that both minutiae match according to the distance measures *SD* and *DD*.

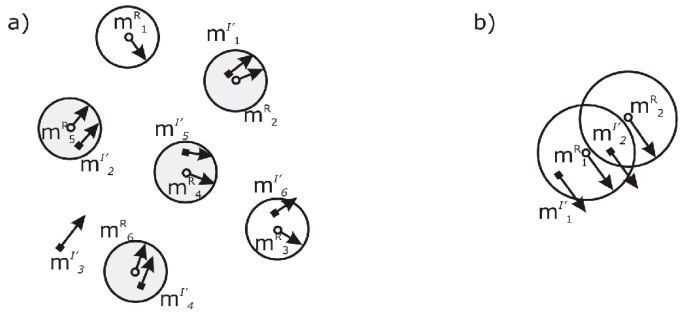



*Figure 12 - Minutiae matching: Minutiae from the input sample $T^I$ are transformed into the coordinates of the reference template $T^R$ by a mapping function, in order to find an optimal pairing. The pairing in figure 12a) is based on the minimum distance. The circles visualize the maximum spatial distance, i.e. the tolerance box $r_0$, and the gray circles denote successfully mated minutiae. But a pairing that is based on the minimum distance does not always lead to an optimal solution. If minutia $m_1^R$ in figure 12b) would have been paired with $m_2^{I'}$, $m_2^R$ would have remained unmated. Hence also other pairing strategies should be applied in order to comply with equation 5.*
*(Images adopted from [Maltoni03])*

Solving the matching problem is trivial, if either the pairing $P$ or the correct alignment $(\Delta x, \Delta y, \theta)$ is known. In both cases the respective alignment or pairing can be determined [Maltoni03]. In practice mostly none of them are given, and so the matching problem becomes a hard one, since testing all possible pairings and transformations would lead to an amount of solutions, that is exponential in the number of minutiae. However, there exist some brute force approaches, see for instance [Huvanandana00].

Approaches, which are based on the pattern matching problem, that is known from the area of pattern recognition, were extensively studied as well. Thereto belong for instance *relation methods* or *operational research solutions*. Beneath them the *Hough transform*-based approach is the most promising one, wherein the point pattern matching problem is converted to the problem of finding peaks in the Hough space of transformation parameters. Respective approaches for the minutiae matching were for example proposed by Ratha et al. in 1996 or by Chang et al. in 1997, in which, besides realignment of the $x$- and $y$-coordinates and rotation of $\theta$, a scaling is considered [Maltoni03].

Further approaches perform an absolute or relative pre-alignment, see e.g. [Jain97]. But also solutions, which do not use any alignments have been studied, since realignment takes a lot of time. Bazen and Gerez proposed a very promising approach, that uses an intrinsic coordinate system, whose axis run along hypothetical lines, which are defined by the local orientation of the fingerprint pattern [Bazen01]. The minutiae are therein defined with respect to their position in the orientation field. Translations, displacements and distortions move the minutiae with the orientation field, but do not change their intrinsic coordinates.

With the aid of the maximum number of matching minutiae it is eventually possible to determine a similarity between an input sample and the reference template, which in turn, if exceeding a predefined threshold, indicates whether two templates are being considered identical.

After dealing with the topic of biometric identification by means of fingerprints in this chapter, the next one will give more details about the application background and a general problem description.

# Chapter 4 - Problem Definition

*"If you wish to preserve your secret, wrap it up in frankness."*

Alexander Smith

In the course of the change to biometric passports two fingerprints will be stored of every person, in addition to a digital passport photo. Ensuring the integrity as well as the authenticity of the data however, is a sensitive topic. While a lost or stolen PIN or key can be replaced, fingerprints cannot. Thus in order to make biometric characteristics widespreadly usable, security of the stored data must be guaranteed. Furthermore must be ensured, that the biometric features stored in the passport indeed belong to that person to whom in turn the passport belongs. For that purpose digital watermarks seem to be an eligible mean. Thus, a hidden embedding of both fingerprint templates in the passport photograph could increase the fraud resistance on the one hand, and the authenticity of the data on the other hand [Jain02]. However the size of one photograph of 6 – 20 kilobytes [PassSpec06] as well as the memory requirements for a watermark currently allow to embed only one template. With regard to the embedding of both templates, different compression methods are to be studied, in order to finally reduce the size of the templates. Since minutiae as well as the nodes of a graph are represented by vectors, i.e. by absolute coordinates, the study of graph-based approaches seems to be very promising. Thereby the compression may be lossy, since two fingerprints of the same finger being captured at different times are never identical and hence their templates differ. The loss according to the error rates FMR and FNMR, introduced in section 2.4, should be kept small, in order to guarantee a secure verification.

The following chapter will give an overview about digital watermarking, common compression techniques as well as current fingerprint compression techniques, in order to finally present a general approach for solution.

## 4.1 Digital Watermarking

Digital watermarks can be used for providing digital media, like e.g. images, videos or text files, with additional information. In comparison with the embedding of metadata, the information is not simply appended but directly complected to the content. There exist two different kinds:

- *Visible watermarks,* which are for instance emblems that are visibly embedded into images or videos in order to ensure authorship.
- *Invisible watermarks* are embedded in a way such that no difference to the original data is recognizable, what means that even the presence of additional information is not certifiable. Hence, this type of watermarking can be understood as a kind of *steganography* (ancient Greek: "hidden writing"), that is also often called as "art and science of hiding secret information in harmless appearing data" [Pfitzmann00].

Since the digital photographs of the biometric passports shall be used among others for verification purposes [PassSpec06], it is necessary that visible changes are only minimal. Hence the fingerprint templates are supposed to be embedded in an invisible way, i.e. by steganographical techniques.

A steganographical embedding of information can be basically understood as adding some kind of noise, in which redundant or irrelevant parts of the carrier medium are completed or replaced by single bits from the information. An additional private or public key is used on the one hand to allocate those parts of the media in which the information is embedded, and on the other hand to increase the security in a way, such that no unauthorized person is able to encode the hidden message.


*Figure 13 - Watermarking: The single bits of a fingerprint template are embedded in those parts of a digital photograph, which are allocated by a key. With the aid of a respective key for the decoding the hidden information can be restored again.*

There are several techniques for hiding information in digital images. According to [Katzenbeisser00] thereto belong for example substitution and domain transformation systems.

### 4.1.1 Substitution

Substitution systems replace single bits of the carrier medium with single bits of the information that is supposed to be embedded. One of the best-known techniques regarding images is certainly the replacement of the *least significant bit* (*LSB*), whereby, as its name implies, the least significant bit of a byte is replaced with one bit of the information. This is possible, if the embedding changes the color information of a pixel only negligibly. Image formats that are based on color tables, like e.g. GIF, are hence not applicable for this method, unless the color tables are sorted in a way, such that similar colors are arranged close to each other.

Another substitution technique is based on *pseudo-random permutation*. Thereby a (pseudo) random key is generated which allocates arbitrary bits in the whole carrier medium, that are in turn replaced by the information bits.

In case of replacing the LSB, a JPEG image of 6 kilobyte size, i.e. 6144 bytes, would allow to embed up to 6144 bits of information, which conforms to 768 bytes. A fingerprint template with 40 minutiae, that is encoded by the compact ISO standard with 3 bytes per minutia, would need 148 bytes of disk space, including 28 bytes of header information [ISO19794-2]. Hence, up to 5 templates could be embedded into one photograph. Thus a need for compression seems not really obvious. Any subsequent manipulation of the image however, may destroy the watermarked information. And also unintended changes, like e.g. bit flips, can lead to a loss of information in the embedded templates. Although at least the latter case can be prevented by including error correction codes, like e.g. parity bits, into the embedded information, the substitution technique does not seem appropriate for hiding sensitive biometric data in images, since integrity cannot be guaranteed.

### 4.1.2 Domain Transformation

In comparison with the substitution method, the domain transformation replaces whole parts of the carrier image instead of single bits. So on the one hand the embedded information becomes more robust against subsequent intentional changes, like e.g. compression or converting, but on the other hand bit flips will affect parts of the information more probably.

In order to embed an information in a photograph by means of domain transformation, the image is partitioned into a number of blocks, which usually consist of $8 \times 8$ pixels. Each of them can hold exactly one bit of information. For this purpose the block is transformed into its frequency domain, for instance by means of a *discrete cosine transform* (DCT) or a *discrete Fourier transform* (DFT), wherein the changes are performed.

According to [Katzenbeisser00] one domain transformation method is to swap the values of two previously defined points $z_1 = f(x_1, y_1)$ and $z_2 = f(x_2, y_2)$ in the frequency image in a way, such that $z_1 \leq z_2$ holds if a "0" has to be encoded, and $z_1 > z_2$ otherwise. After embedding the information the whole block is transformed into its original domain again.

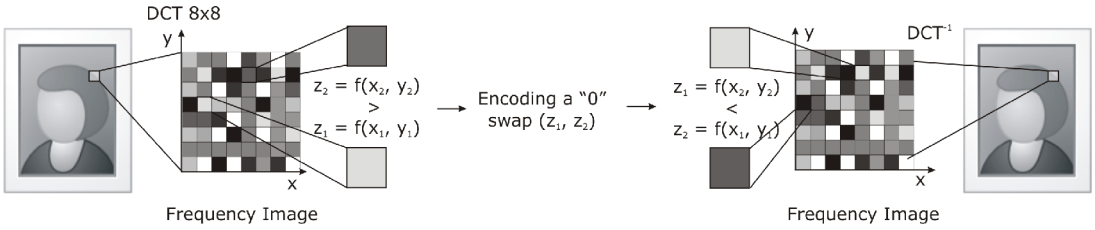



*Figure 14 - Domain transformation: A block of the image is transformed into its frequency domain. Since the value $z_2$ of the point $p_2 = (x_2, y_2)$ is larger than the respective value of point $p_1$, both values are swapped, in order to encode the information "0" and hence, to fulfill the condition $z_1 \leq z_2$. Afterwards the block is retransformed to its original domain.*

Since however only one bit per block can be encoded, the amount of information that fits into a photograph is very small. Considering the 6 kilobyte JPEG image again: With a color depth of 3 bytes the picture consists of 2048 pixels. A block size of $8 \times 8$ pixels would lead in an ideal case to exactly 32 blocks. Hence, exactly 32 bits of additional information might be hidden in that photograph. The encoding of a 148 byte template would require an enormous compression, without consideration of any error correction code. Hence also this embedding technique is not applicable for hiding fingerprint templates in digital photographs.

### 4.1.3 Hiding Minutiae Templates in Images

A watermarking technique, that can be directly applied in order to embed the fingerprint templates, was introduced by Anil Jain and Umut Uludag in [Jain02]. It is robust against subsequent intentional changes on the one hand, and able to minimize information loss due to bit flips on the other hand. Thereto an amplitude modulation is performed in the blue channel of a pixel, since the human eye recognizes changes in that spectrum least of all [Schmitz06]. With the aid of a key those bits of the image are picked out which eventually will contain the information. Similar to the domain transformation, the neighboring pixels of a pixel $p$ are considered within a $5 \times 5$ block. But the final changes are ex-

clusively performed on the selected pixel. Thus the watermarked pixel $P_{WM}(i,j)$ is computed using the following formula:

$$P_{WM}(i,j) = P(i,j) + (2s-1) \cdot P_{AV}(i,j) \cdot q \cdot \left(1 + \frac{P_{SD}(i,j)}{A}\right) \cdot \left(1 + \frac{P_{GM}(i,j)}{B}\right) \quad (6)$$

At this $P(i,j)$ is the selected pixel and $s \in \{0,1\}$ the bit that is going to be embedded. $P_{AV}(i,j)$, $P_{SD}(i,j)$ and $P_{GM}(i,j)$ are with regard to the $5 \times 5$ blocks the average deviation, the standard deviation and the gradient magnitude at position $(i,j)$, while $q$ is the embedding strength. Finally $A$, as well as $B$, are the respective weights for the standard deviation and the gradient magnitude. Every bit is redundantly embedded, like e.g. at 30 locations. This increases the correct decoding rate.

Decoding is done by determining the differences between the estimated and the real values of all pixels, in which the same watermark bit is encoded. These differences are averaged and afterwards used for computing an adaptive threshold, which indicates whether the watermarked bit is either a “0” or a “1”.

The 6 kilobyte JPEG image with its 2048 pixels could include the same number of information bits, what conforms to 256 bytes. Hence, nearly two templates without redundancy could be embedded into that picture. Thus, a compression is clearly necessary.

## 4.2 Data Compression

A data compression is applied, if either memory requirements or the volume for data transmissions shall be reduced. This is done by generating a better representation, in which for instance redundancies are removed or information is omitted. Basically there are two different forms of compression:

- *Lossless*. A compression is lossless, if and only if the original data can be faithfully reproduced after decompression. For this purpose, mainly redundant information is removed. Text and binary data are classical cases of application for this form of compression.
- *Lossy*. When data are compressed in a lossy way, a faithful reconstruction after decompression is not possible. At this, the reduction is achieved by omitting information. Multimedia data, such as music, videos or pictures, are typically reduced by lossy compression.

These types are not necessarily independent of one another: In order to increase the success of compression both forms can be combined. Since there are lots of standard compression techniques, some of them will below be shortly presented and analyzed of being applicable for compressing fingerprint templates.

### 4.2.1 Entropy Encoding

Entropy encoding is a kind of lossless compression. It is commonly used for text files, but e.g. also finds use in the compression of images. Depending on their frequency in a document single tokens are encoded by bit strings of varying length. Those lengths, i.e. the optimal number of bits, is defined by the information content $I = -\mathrm{ld}(p)$, whereby $p \in [0,1]$ is the relative occurrence frequency in the respective document. The less fre-

quent a token occurs, the larger its information content and the larger the number of bits, which are necessary for its encoding, whereas tokens, which occur more often, are represented by shorter strings. Thus, the total length of the resulting bit string is minimized. The performance of an entropy encoder can be finally determined with the aid of the entropy $H = \sum_{i=1} p_i \cdot I(p_i)$, i.e. the better the total number of bits, that are needed for encoding, approximates the entropy, the better the performance.

### Huffman Coding

Huffman encoders are classical representatives of the entropy encoders and are regarded as an improvement of the Shannon-Fano coding [Leweler87]. Both techniques create a prefix free code, which means that no code word is prefix of another one. Otherwise the original information could not be clearly restored. The character string "ABC" for instance, encoded with A = 10, B = 01 and C = 0, would result in the bit string "10010", which in turn could be decoded as "ACA". The Huffman code is created by means of a complete binary tree. The respective algorithm, presented below, constructs verifiably always the best prefix tree [Cormen07].

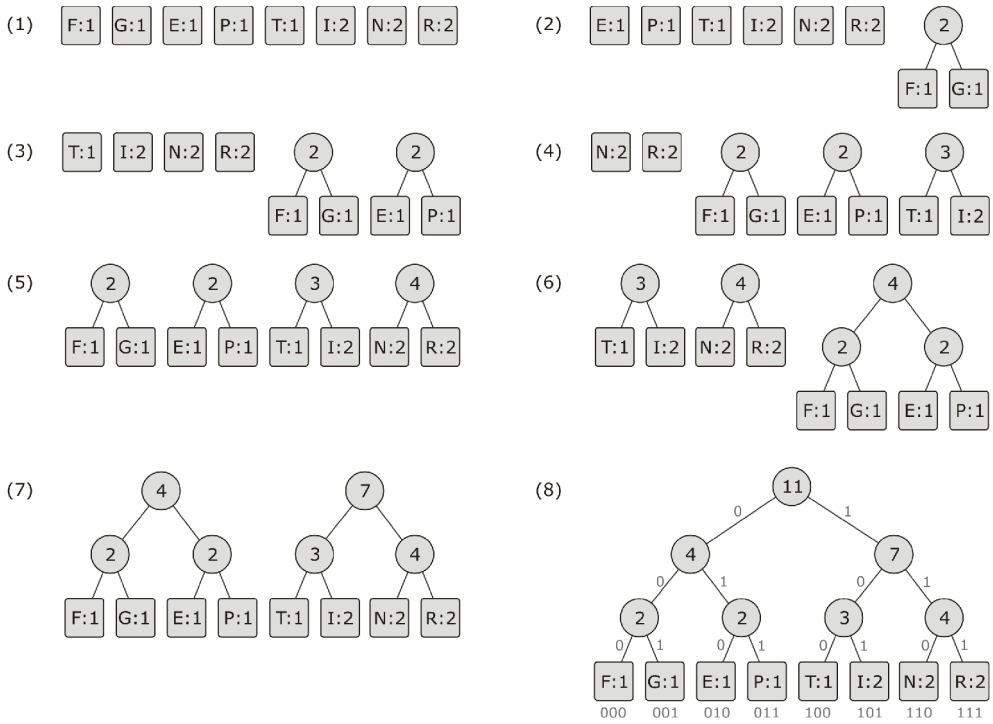



*Figure 15 - Huffman coding: Encoding the string "FINGERPRINT". Leafs are depicted by squares, labeled with the respective token and the absolute occurrence frequency. Inner and root nodes are represented by circles, labeled only with the frequency. After creating a node for each token and sorting them according to their occurrence frequencies, in every next step the two nodes with the smallest frequencies are selected and combined to a new tree, whose root node is labeled with the sum of the frequencies of both selected nodes. Finally a single prefix tree emerges. The entropy for this example is 2,914, while the Huffman algorithm creates a code that uses exactly 3 bits per token.*

In a first step the algorithm determines the occurrence frequencies of all tokens and creates nodes for all of them in the prefix tree. Furthermore links to those nodes are added to an increasingly sorted list. In every following step the two nodes with the smallest occurrence frequencies are determined with the aid of that list. Their links are removed, and the nodes are combined to a tree by adding a common root node, which in turn ob-

tains the sum of both occurrence frequencies as label and a respective link in the sorted list. These steps are repeated until the list consists of only one element, which is eventually the root node of the created prefix tree. In order to determine the code words, the edges are labeled. Commonly left branches obtain a “0” and right ones a “1”. The code word of a token emerges by following the branches to the respective leaf starting at the root node and connecting the labels of all visited edges to a bit string.

Huffman coding is perfectly applicable for text files but for binary documents, such as templates, it is less appropriate. Since the character set consists of only two elements, even different occurrence frequencies, like for instance 0 = 25% and 1 = 75%, would lead to the same number of bits: namely exactly 1. An optimal entropy encoder would only need $H = \sum_{i \in \{0,1\}} p_i \cdot I(p_i) = 0.81$ bit per token instead.

### Arithmetic Coding

Another kind of entropy encoding is the arithmetic coding, whereby the information is represented in the form of a number from the interval [0,1) [Leweler87]. Depending on the number of tokens this interval gets more and more restricted. Thus the information is not stored by means of single tokens but as a whole, which allows to approximate the entropy in a better way. Encoding is achieved by dividing the starting interval [0,1) in as much subintervals as different tokens exist, whereby the sizes of those intervals conform with the relative occurrence frequencies of the tokens they represent. Every succeeding step divides the subinterval of the currently to be encoded token in exactly the same way. Thereby the tokens are step by step processed in that order in which they occur in the message. The last token finally defines the interval from which an arbitrary number is chosen, which is in turn afterwards represented as a bit string.

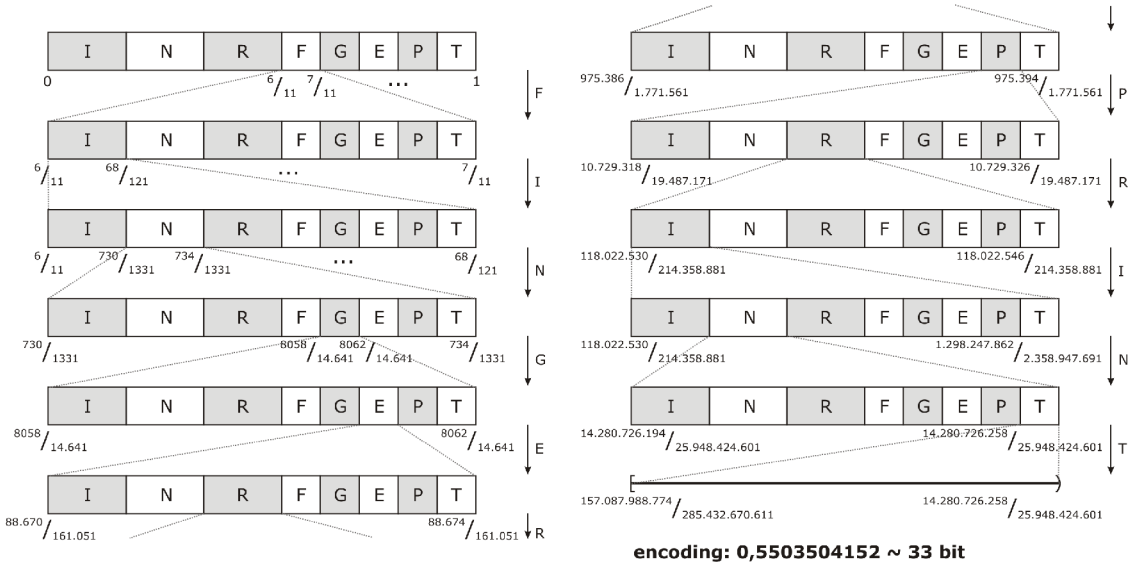



*Figure 16 - Arithmetic coding: Encoding the string “FINGERPRINT” again. The starting interval [0, 1) is divided in as much intervals as different tokens exist, while the sizes conform to the respective occurrence frequencies of the tokens they represent. The succeeding steps repeat doing that always for the subinterval of the token which is currently to be encoded. Averagely 3 bits per token are needed in this example.*

Just like Huffman coding arithmetic coding is perfectly applicable for text files. However, if a few amount of tokens occurs relatively often, this kind of coding is even better: A

document containing only zeros for instance could be stored with a single bit – the number to encode is "0", whereas the Huffman coding would need as much zeros as exist in the document. Insofar an encoding of templates by means of arithmetic encoding seems to be eligible if the occurrence frequencies of the bits differ enormously.

### 4.2.2 Dictionary Coders

Dictionary coders are a further kind of lossless compression. As the name implies, a dictionary of strings and tokens is used. Instead of the real tokens the encoded message holds the respective dictionary entries and in many cases the dictionary itself as well. The basic idea for this method originates from Jacob Ziv and Abraham Lempel, who invented the algorithms LZ77 and LZ78 in 1977 and 1978. The latest one, LZW (Lempel-Ziv-Welch), is an improvement of those both, developed by Terry Welch in 1984, and is nowadays used for instance in the image format GIF. LZW is based on a dictionary, which is dynamically created during en- and decoding. Thus, additional disk space for the dictionary is omitted. Therefore the dictionary is initialized with an alphabet, like e.g. the ASCII tokens. Commonly each entry is composed of 12 bit, so one dictionary can hold a total of 4096 entries [Welch84]. The encoding algorithm looks as follows:

**Algorithm 1** - *LZW Encoding*($S$)

**Input**: $S$ – input string
**Output**: $S'$ – encoded string

```
 1 initialize dictionary
 2 prefix p ← read first token of input string S, S' ← ∅
 3 while (S contains more tokens T) do
 4    T ← read next token of S
 5    if (pT exists in dictionary) then p ← pT
 6    else
 7       S' ← S' + code(p)              // code(p) is the entry in the dictionary for p
 8       add pT to dictionary
 9       p ← T
10    end if
11 end while
12 S' ← S' + code(p)
13 return S'
```

The FINGERPRINT example, encoded by LZW, would result in the following dictionary and code string:

**String Table**

| Code | Entry | | |
|---|---|---|---|
| 0 | F | 9 | IN |
| 1 | I | 10 | NG |
| 2 | N | 11 | GE |
| 3 | G | 12 | ER |
| 4 | E | 13 | RP |
| 5 | R | 14 | PR |
| 6 | P | 15 | RI |
| 7 | T | 16 | INT |
| 8 | FI | | |

**Example: FINGERPRINT**

| S' | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 5 | 9 | 7 |
|---|---|---|---|---|---|---|---|---|---|---|
| | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ |
| Entry | F | I | N | G | E | R | P | R | IN | T |

*Figure 17 - LZW: Dictionary, code string and respective entries for the FINGERPRINT example.*

Since usually every dictionary entry is represented by 12 bit, also this compression technique is not perfectly fitting for the compression of biometric templates. However some other kind of dictionary in terms of references to minutiae vectors is imaginable, such that a few amount of minutiae may act as dictionary entries, and thus encoding is done by means of a reference together with a small correction vector.

### 4.2.3 Lossy Compression

A lossy compression is commonly used for videos, images and music in addition to a lossless compression [Schmitz06]. At this, mostly unnecessary information is omitted.

Since the human ear is only able to recognize frequencies between 20 Hz and 20 kHz [Fellbaum84], for the compression of music typically the frequency range is cut. The same happens with very silent sounds. But also quantization, i.e. the assignment of signal values to values from a finite domain, is a further possibility of compression, since the quantized values can finally be used as dictionary entries. There are two types of quantization:

- *Scalar quantization* means, that the original domain is first divided into intervals. Afterwards all values from one interval are mapped to a certain value from the finite domain, whereas that mapping might be either linear or non-linear.
- A *vector quantization* always involves several values at the same time and represents them as a vector. Hence instead of intervals the vectors are mapped to certain values, like for instance to an Euclidian distance measure.

Compressing images in a lossy way is similar to audio signal compression in terms of the weakness of human perception. Given that the human eye can only distinguish between up to 350.000 different colors [Fellbaum03], images can be compressed by quantizing the color information. And also the frequency domain can be compressed in this way. So for instance higher frequencies are quantized more coarsely, since a loss of information in fine image structures appears less disruptive [Schmitz06].

In order to compress videos a further technique is used in addition to the compression of the single pictures: Normally there are 25 frames broadcasted per second. Thus, it is assumable that successive frames only differ slightly, except of complete changes of scene of course. So roughly spoken only the differences between the successive images are stored. This happens with the aid of so-called *motion vectors*, which specify to what position an image block from one frame has moved in the next frame.

An adapted version of that motion vectors could be interesting for the compression of templates. Assuming that the distance values between those minutiae are smaller than their absolute coordinate values, it is conceivable to store those differences instead. But also the idea of vector quantization seems to be relevant. So for instance some distance vectors might be used as dictionary entries. But there are already existing approaches for the compression of fingerprints, as well, which will now be shortly presented.

### 4.2.4 Compression of Fingerprint Data

Fingerprints that are captured off-line, i.e. with the aid of ink on paper, are exclusively bicolored. These two colors are sufficient for representing all information of the fingerprints, namely the ridges, furrows, minutiae, patterns and singularities. Police authori-

ties, such as the Federal Bureau of Investigation (FBI), have collections of off-line prints consisting of approximately several millions of impressions [Sherlock96]. In order to make those prints comparable by means of computers, they get digitalized. But due to noise and other processes during the scanning procedure the bicolored impressions turn into gray-scale images [Zirkind07], that are generally stored with one byte per pixel [Fontenot02]. Since normally two colors are sufficient in order to distinguish between the ridges and the furrows, the FBI used black-and-white images for the comparison of fingerprints for a long time, and thus found out that removing the gray-scales rapidly worsens the image quality [Zirkind07]. So the gray-scales are necessary for maintaining the dermal ridge information of the scanned fingerprints. That is why the images are compressed and stored as JPEG with 256 shades of gray. JPEG compression is based on a discrete cosine transform and reaches a compression ratio up to 1:23, independently from the kind of the image. But specific features of fingerprint images are therein not considered and afford a starting point for further compression techniques.

So the compression of fingerprints is an active research area, whereby a large number of work falls back on wavelets, that are for instance used in the image format JPEG2000, which in turn is the current standard of AFIS systems. Wavelets are a mathematical tool for hierarchically representing and approximating functions [Grasemann05]. Commonly a wavelet $\psi_{j,i}$ is nothing else than a function, that can be adapted by means of translation ($j$) and dilatation ($i$) coefficients. Thus, by altering the coefficients, a huge amount of functions emerges. The process of dividing the function into wavelet coefficients is hence stated as *wavelet transform*. If those coefficients only adopt discrete values, the transform is also referred to as *discrete wavelet transform* (DWT). The quality of compression strongly depends on the choice of the wavelet, since wavelets, that achieve high compression results for photographs, must not do so for fingerprints. Hence, since the begining of the 90s there is a lot of research done on this area. Grasemann and Miikkulainen [Grasemann05] have run genetic algorithms on fingerprints in order to find suitable wavelets. Sherlock [Sherlock96] studied optimized wavelets, and also approaches that use several wavelets for compression, so-called *multiwavelets*, have already been studied [Sudhakar05].

A further approach, exclusively developed for the AFIS systems of the FBI, is called *GBP* compression and reaches a compression ratio of 1:92 by reducing the number of bits, that are necessary to store an image. This is done by saving the photographs in three colors instead of 256, namely black, white and gray. Thus instead of 8 bit per pixel only 2 bit are sufficient, whereby at least one necessary shade of gray remains. The tricolored images are afterwards compressed by a discrete cosine transform [Zirkind07].

But also other compression techniques which are based on the extraction of features, and hence do not consider the pixel information, have been studied [Chong92]. There the original gray-scale images are afterwards not restorable. Seen from this angle, templates that are composed of specific features – the minutiae – can be regarded as another kind of compression, although they are treated as some sort of notation for the fingerprint data [Jain02]. Hence, the approach being considered in this thesis will not be a conventional one, since the already very small templates shall be further compressed, in order to use them as digital watermarks and for verification purposes, as well.

## 4.3 General Approach

As already mentioned, a first idea is to treat the minutiae of a template as points of a graph and to store the preferably small difference vectors between two points instead of the absolute coordinates, trusting that those difference vectors can be represented by a smaller number of bits than the coordinates of the given points. Indeed, in a worst case one bit per dimension can only be saved if the domain borders of the edges are half as small as the domain borders of the nodes. Hence two bits per dimension are saved if the edge domain is quarter as small as the node domain, etc. Nevertheless, obtaining an optimal set of difference vectors is for example possible by means of a minimum spanning tree, where only those points get connected by an edge, which are close to each other according to some distance measure. Furthermore the resulting graph is connected, so all nodes are reachable by a path.

However the main approach of this thesis is based on some kind of dictionary in which a small set of so-called reference vectors is going to be stored, such that every difference vector between two points can be represented by a reference to a dictionary entry together with a small correction vector. Since also in this case only a subset of the difference vectors is going to be stored which in turn can be represented by a spanning tree, this thesis will deal with the compression of minutiae templates by means of directed spanning trees. Therefore the following chapter will present the different spanning tree-based approaches in detail.

# Chapter 5 - Formalization, Basics and Approaches

*"The beginning of all science is wondering why things are the way they are."*

Aristotle

As already known, fingerprint templates $T = (m_1, ..., m_n)$ consist of a set of minutiae $m = (x, y, \theta, t)$, which in turn characterize the positions, orientations and the types of certain points of interest on the dermal papillae of a fingertip. In order to obtain a more compact and hopefully optimal representation of the template data, the minutiae will be treated as points of a complete and directed graph. On that graph a spanning tree is going to be determined and afterwards converted into a bit string representation. This thesis will therefore deal with several approaches, each of them using another kind of directed spanning tree.

The first and probably simplest idea consists in the determination of a directed *minimum spanning tree* (MST). Starting at the root node, in this case the edges are going to be treated and stored as difference vectors between two points. The second and actually main approach will use some kind of dictionary, in which a subset of so-called reference vectors (in [Chwatal08] referred to as *template arcs*), is memorized. Representing an edge in that special context means, holding a link to the respective dictionary entry together with a small correction vector, so that the target node can be losslessly reconstructed. If treating the reference vectors as some kind of edge labels, an eligible set of dictionary entries can be determined with the aid of a *minimum label spanning tree* (MLST), which was first presented in [Chang96]. Finally the last solution to be dealt within this thesis, is to find a spanning tree where all difference vectors of the edges are as equal as possible with respect to their coordinate values. Hence a common offset vector can be obtained, such that all edges of that tree are representable by small correction vectors. This offset vector in turn can be viewed as a reference vector, wherefore this kind of solution can be understood as a special case of the MLST approach. However the applied algorithms, which are going to be used for solution, are based on the *weight balanced spanning tree* ( WBST) [Gupta04]. Hence besides a general formalization, in this chapter the three introduced approaches will be discussed in detail. But first some graph-related definitions are presented.

## 5.1 Graph-related Definitions

A graph $G$ is mathematically viewed a tuple of nodes $V$ and edges $E$. Although there is a distinction into *directed* and *undirected* graphs, for convenience within this section both variants will be denoted with $G = (V, E)$. At this $V = \{v_1, ..., v_n\}$ with $n \in \mathbb{N}$ represents the finite *set of nodes*. The *set of edges* $E \subseteq V \times V$ is a binary relation on $V$ with $E = \{e_1, \ldots, e_m\}$ or $E = \{e_{i,j} \mid e_{i,j} = \{v_i, v_j\}, v_i, v_j \in V \wedge v_i \neq v_j\}$. Note that for the directed set of edges $A \subseteq V \times V$ holds $A = \{a_{i,j} \mid a_{i,j} = (v_i, v_j), v_i, v_j \in V\}$, and that this notation will be used for directed arcs outside of this section.

In an undirected graph an edge $e_{i,j} = \{v_i, v_j\}$ is *incident* with the nodes $v_i$ and $v_j$, and both nodes are *adjacent*. The respective edge $e_{i,j} = (v_i, v_j)$ in a directed graph has a *source node* $v_i$ and a *target node* $v_j$. Hence $v_j$ is adjacent to $v_i$ but not vice versa, if there is no respective backward directed edge $e_{j,i}$. The *degree* $\delta(v)$ of a node $v$ conforms in the directed case

with the sum of all incoming (*indegree* $\delta^+(v)$) and outgoing (*outdegree* $\delta^-(v)$) edges and in the undirected case with the number of edges that are incident with $v$. An undirected graph is *complete*, if and only if all nodes $v_i \neq v_j$ with $v_i, v_j \in V$ are adjacent to each other. The same definition shall hold in the directed case. Furthermore, a graph $G' = (V', E') \subseteq G$ is a *subgraph* of $G$, if $V' \subseteq V$ and $E' \subseteq E$ with $\forall e_{i,j} \in E' \mid v_i, v_j \in V'$ holds.

A *path* $p$, starting at node $v$ and ending at $v'$, is a sequence of nodes $\langle v_0, ..., v_{|p|} \rangle$, for which $v = v_0$, $v' = v_{|p|}$ and $e_{i-1,i} = (v_{i-1}, v_i) \in E$ with $i = 1 ... |p|$ holds. Thereby the number of edges defines the length $|p|$. A path is also called a *cycle*, if $v_0 = v_{|p|}$ with $|p| \geq 1$ (directed) or $|p| \geq 3$ (undirected), respectively [Cormen07].

A *connected component* (CC) is a subgraph $G' \subseteq G$ of an undirected graph $G$, in which a path from every node $v \in G'$ to all other nodes $v' \in G'$ exists. If $G$ is in turn a directed graph, then the respective subgraph $G'$ constitutes a *strongly connected component* (SCC). Thus $G'$ is also referred to as *connected* or *strongly connected*, respectively.

An acyclic and undirected graph is also called a *forest*. If the graph is additionally connected, then it is a *tree* $T = (V, E)$. In that case it holds, that $|E| = |V| - 1$. A directed acyclic graph in turn, is also shortly called *DAG*. Finally, in a directed tree one node is marked as *root node*. It has no incoming edges. Nodes without outgoing edges are called *leafs*. All further nodes are simply *inner nodes*. A *spanning tree* is a graph where all nodes are connected by means of a tree, and a *directed spanning tree* has a marked root node. Finally an *arborescence* is a directed spanning tree on a directed graph, where there is a path from the root node to every further node.

Thus, a *minimum spanning tree* requires a weighted graph $G = (V, E, w(E))$, whereat $w: E \rightarrow \mathbb{R}$ is a function on the set of edges, which is also called *weighting function*. A minimum spanning tree is hence a spanning tree $T_S = (V, E_T)$, $E_T \subseteq E$ of a graph $G$ with a minimum overall weight [Cormen07]:

$$\text{minimize } w(T_S) = \sum_{e \in E_T} w(e) \tag{7}$$

Since all important definitions are introduced now a general formalization for the given problem can be stated.

## 5.2 General Formalization

Given is a directed Graph $G = (V, A)$ consisting of a set of points $V = \{v_1, \ldots, v_n\}$ with $n \in \mathbb{N}$. These points conform to the set of minutiae of a template $T$ and are part of the discrete domain $\mathbb{D} = \{0, \ldots, \tilde{v}^1\} \times \ldots \times \{0, \ldots, \tilde{v}^d\}$, where the values $\tilde{v}^1, \ldots, \tilde{v}^d \in \mathbb{N}$ with $d \in \mathbb{N}$ specify the domain borders, i.e. the largest values for every dimension $d$. The goal is to find a directed spanning tree on that graph and afterwards to transform the tree into an eligible bit string representation, which is in turn as compact as possible. Thus, a general solution consists of three steps:

- Creating a complete directed graph $G = (V, A)$ with $A = \{a_{i,j} \mid a_{i,j} = (v_i, v_j),\ v_i, v_j \in V \wedge v_i \neq v_j\}$.

- Determining a directed spanning tree $T_S = (V, A_T)$ with $A_T \subseteq A$. Thereto the set of edges $A$ is treated as a set of difference vectors $A_\Delta$ between all source and target nodes, i.e. $A_\Delta = \{\Delta a_{i,j} \mid \Delta a_{i,j} = v_j - v_i \wedge a_{i,j} \in A\}$.
- Transforming the spanning tree into a suitable bit string representation.

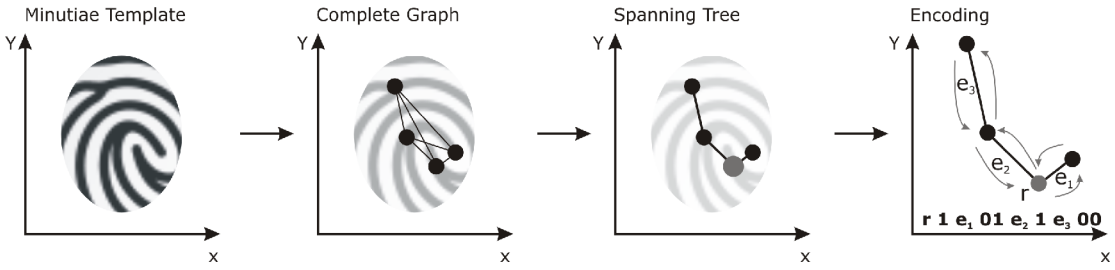



*Figure 18 - Template compression: After creating a complete directed graph on the set of minutiae, a directed spanning tree is computed. In order to translate that tree into a bit string, a depth first search is performed starting at the root node. Each time the search reaches a node that was not visited yet, the resulting bit string is first extended with a "1" and afterwards with the currently used, binary encoded edge. Otherwise, if the search returns to an already visited node, a "0" is appended .*

Since all the data points, that are contained in the resulting bit string, should be losslessly reconstructed, the following section will present the idea of en- and decoding.

### 5.2.1 Encoding and Decoding

In order to convert a directed spanning tree into a bit string, there are three things needed to be encoded – the *root node*, the *edges* and the *tree structure*. In this special case the root node $r$ is converted first of all into the bit string $S$ using a constant resolution. Afterwards the root node is used as starting point for a *depth first search* (DFS), that in turn helps to encode the tree structure and the edges. This is done by first appending a "1" to the bit string $S$ if the search reaches a non-visited node, and afterwards extending $S$ with the currently visited edge, which is also encoded in a constant resolution. If the search returns to an already visited node, a "0" is appended to the bit string. A recursive algorithm for the encoding of the tree structure and the edges hence would look as follows:

**Algorithm 2** - *encodeSpanningTree*$(T, v_i, S)$

**Input**: $T = (V, A_T)$ – spanning tree, $v_i$ – starting node for the DFS, $S$ – bit string
**Output**: $S$ – bit string

```
1 forall (a_{i,j} = {v_i, v_j} ∈ A_T) do
2     S ← S + "1"
3     S ← S + encode(Δa_{i,j})
4     encodeSpanningTree(T, v_j, S)
5     S ← S + "0"
6 end forall
7 return S
```

The method *encode*() for encoding the edges depends on the applied approach, so the concrete realization will be presented in the respective sections. In order to afterwards reconstruct all minutiae, the algorithm for decoding processes the bit string step by step. Since the root node is encoded with a constant number of bits, restoring is done by sim-

ply interpreting the first bits. Decoding all further nodes is shown in the algorithm below. Thereby, the method *decode*() is the opposite to *encode*() and computes the target node and the respective edge from a passed source node and bit string of constant size. Note, that the algorithm internally constructs a tree in order to always obtain the correct source node for the currently to be decoded edge.

**Algorithm 3** - *decodeDataPoints*($S$, $n_r$, $n_a$)

**Input**: $S$ – bit string, $n_r$ – number of bits for root node, $n_a$ – number of bits for edges
**Output**: $V$ – set of data points

```
1  V ← ∅, A_T ← ∅, T_S ← (V, A_T)
2  V ← {v}, whereby v is reconstructed from the first n_r bits of S
3  i ← n_r + 1
4  while (i ≤ |S|) do
5     if (S_i = 1) then                          // S_i ist the bit at position i
6        {v', (v, v')} ← decode(v, substring( S_{i+1}, S_{i+n_a} ))
7        V ← V ∪ {v'}, A_T ← {(v, v')}
8        i ← i + n_a + 1
9        v ← v'
10    else if (S_i = 0) then
11       v ← predecessor of v
12       i ← i + 1
13    end if
14 end while
15 return V
```

Since the original template as well as the encoded data are afterwards in a bit string representation, the success of compression can be expressed as ratio between both numbers of bits. However, we cannot assume that the original data is optimally encoded, because there might be further information, such as metadata or even redundant bits. So for the purpose of comparison some objective functions are going to be introduced.

### 5.2.2 Objective Functions

A general formula for determining the number of bits of the original data emerges from template specific metadata (MetaData), further minutiae-dependent information of constant size ConstData, such as offsets, and the minutiae themselves [Chwatal08]:

$$\lambda_{raw} = size(\text{MetaData}) + size(\text{ConstData}) + \; n \cdot \sum_{i=1}^{d} |\operatorname{ld} \tilde{v}^i| \tag{8}$$

Here the number of minutiae is given by $n$, $d$ is the number of dimensions, and $\tilde{v}^i$ specifies the greatest value for the specific dimension. Finally, *size*() is a function that returns the number of bits, that are necessary for encoding the passed data.

In turn, the respective formula for the encoded data is supposed to be the following:

$$\lambda = size(\text{MetaData}) + size(\text{ConstData}') + 2(n-1) + (n-1) \cdot size(encode(\Delta a)) \tag{9}$$

As already mentioned, the function *encode*() depends on the applied approach and will be later on described in detail. The size of the compressed data hence emerges from me-

tadata, the tree structure $(2 \cdot (n-1))$, the encoded edges as well as further information. MetaData as well as ConstData are at this information of constant size. Note that ConstData' must not be identical with ConstData, since the former one may for instance contain the encoded root node. What specific information belongs to that data depends on the one hand on the concrete template and on the other hand on the applied approach. Thus, more detailed descriptions will be given when introducing the test data and presenting the results in chapter 8.

### 5.2.3 Avoiding Negative Values

Since the edges will be treated as difference vectors between two nodes, i.e. $\Delta a_{i,j} = v_j - v_i = (\Delta a^1_{i,j}, \ldots, \Delta a^d_{i,j})$, the entries in general might adopt negative values. Hence, let the domain of the edges be $\mathbb{D}_{\Delta A} = \{-\tilde{v}^1, \ldots, \tilde{v}^1\} \times \ldots \times \{-\tilde{v}^d, \ldots, \tilde{v}^d\}$, where the domain borders arise from the given node domain $\mathbb{D}$. An encoding of those values must assure, that a distinction between positive and negative values can be made. So for that purpose an additional prefix bit would have to be stored for every value. But since this would unnecessarily enlarge the number of used bits, the following transform of the edge domain is going to be applied [Chwatal08]:

$$\Delta a_{i,j} = (v_j - v_i) \bmod \tilde{v} \tag{10}$$

Again the vector $\tilde{v} = (\tilde{v}^1, \ldots, \tilde{v}^d)$ represents the domain borders of the set of nodes $V$. Hence, after performing the modulo calculation the domain of the edges conforms exactly to the node domain, i.e. $\mathbb{D}_{\Delta A} = \mathbb{D} = \{0, \ldots, \tilde{v}^1\} \times \ldots \times \{0, \ldots, \tilde{v}^d\}$. Thus, transforming the edge domain at least will not increase the number of bits for encoding.

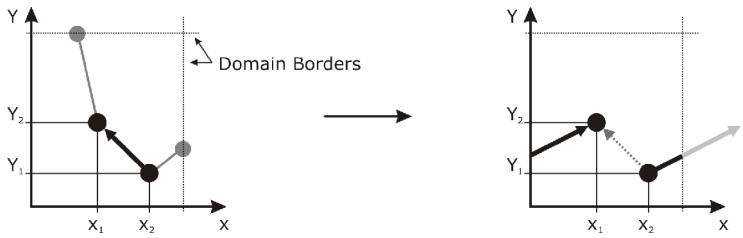



*Figure 19 - Transform of the edge domain: In order to avoid negative values in the difference vectors, the edge domain is transformed into the node domain by means of a modulo calculation. On the left side an original edge is shown, and on the right side an impression of the transformed edge is depicted.*

The approach so far dealt with a lossless compression of the minutiae data. Since however a lossy approach promises a stronger compression in any case, finally the common idea will be presented.

### 5.2.4 *k*-Node Spanning Tree

The main idea of lossy compressing the minutiae data is to find a suitable subset of nodes and to create a spanning tree up on them, instead of considering all nodes. Thus, a *k-node spanning tree* emerges, whereat $k \in \mathbb{N}$ is a predefined value. The computation is equivalent to the lossless case: After creating a complete directed graph over all data points, a directed spanning tree is sought and subsequently converted into a bit string representation. The difference is, that the algorithm stops as far as a subtree consisting of

at least $k$ nodes has been found. A compression by reducing the number of considered nodes however, can unnecessarily heighten the error rates FMR and FNMR (see section 2.4). Thus an acceptable tradeoff between compression and error ratio should be found.

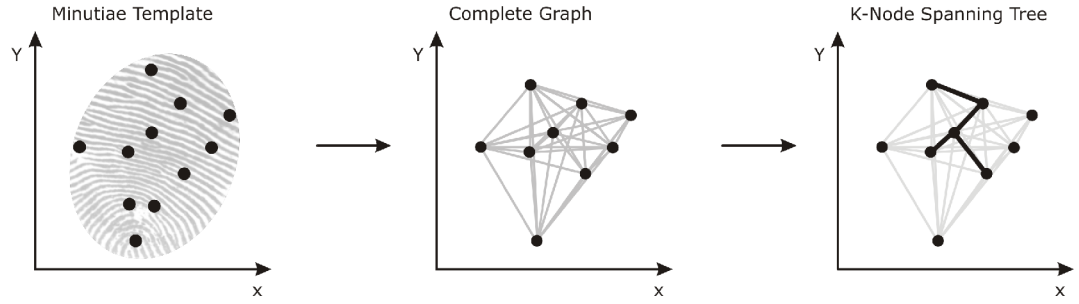



*Figure 20 - Lossy template compression: On the complete graph, that is induced by the set of minutiae, a spanning tree consisting of at least k nodes is computed.*

Because the minimized set of nodes can indeed be identical with the original set of nodes, all further considerations will refer to the idea of a $k$-node spanning tree. Hence, also the number of nodes $n$ in the objective functions will conform to $k$.

## 5.3 The Minimum Spanning Tree

A *minimum spanning tree* (MST) $T_S = (V, E_T)$ is as already mentioned an acyclic, connected subgraph of a connected, undirected graph $G = (V, E, w(E))$, that connects all nodes $v \in V$ by means of a tree. This tree is minimal, if and only if no other tree with a smaller overall weight $w(T_S) = \sum_{e \in E_T} w(e)$ exists on that graph. Finding such a tree is referred to as the *minimum spanning tree problem*. It occurs for example, when designing electronic circuits or connecting computers. In the graph theory the MST is also a good mean for solving more complex problems, such as the traveling salesperson route. For computation there are a few standard algorithms.

### 5.3.1 Algorithms for Solving the MST Problem

One possibility for determining a minimum spanning tree on a graph is to compute all possible spanning trees together with their overall weights, and to finally choose the best one. This brute-force method finds the optimum for sure but is quite inefficient on large graphs. Hence for the computation so-called *greedy algorithms* are used, which successively create a global solution by always choosing the best local one. This means in the case of a MST, that beginning with an empty set of edges exactly one edge is added in every step. Beneath of all possible edges, each time the locally best one is chosen. Greedy algorithms in general do not guarantee a globally optimal solution but in many cases they do [Cormen07], so for example the two best-known ones for solving the MST problem: The algorithms from Kruskal and from Prim.

#### Kruskal's Algorithm

For determining a minimum spanning tree on an undirected graph Kruskal's algorithm starts with a forest of trees, each of them having size one. By adding an edge to the solution in every iteration step always two trees are combined to a new one. Therefore be-

neath of all edges that are able to connect two trees, that one having the smallest edge weight is selected, and thus always the best local solution is taken.

**Algorithm 4** - *Kruskal-MST*($G$)

**Input**: $G = (V, E, w(E))$ – an undirected, connected graph
**Output**: $E_T$ – a set of edges inducing a minimum spanning tree

```
 1 E_T ← ∅
 2 ∀v ∈ V: make_set(v)
 3 sort all edges e ∈ E increasingly by their weights w(e)
 4 forall ((e_{i,j} = {v_i, v_j}) ∈ E) do
 5    if (find_set(v_i) ≠ find_set(v_j)) then
 6       E_T ← E_T ∪ {e_{i,j}}
 7       union(v_i, v_j)
 8    end if
 9 end forall
10 return E_T
```

In the above algorithm a *union-find data structure* is used, in order to determine whether an edge is capable of connecting two trees. Thereby each set contains exactly one tree. This is made sure by first assigning every node to a separate set using the function *make_set*(), and afterwards successively inserting those edges that connect nodes of different sets. Otherwise there would be cycles formed within the components. The function *find_set*() thereby returns the representor nodes of the sets, to which $v_i$ and $v_j$ belong to. Both affected sets are afterwards united to a new combined set by means of the *union*() function. Sorting the edges according to their weights causes their processing in the respective chronological order. Thus always the best local edge can be chosen. The currently best-known implementation of Kruskal's algorithm provides a runtime of $O(m \log n)$ [Cormen07].

### Prim's Algorithm

Prim's algorithm is another greedy algorithm. Compared with Kruskal's one however, the greedy strategy consists in successively extending an existing tree instead of always merging two components. Therefore in every step a non-cycle forming edge with minimal edge weight is added to the solution until finally a spanning tree emerges. The algorithm (5) below, taken from [Cormen07], uses a priority queue for the purpose of an efficient implementation.

First of all the priority queue is initialized with all nodes of the graph, and their respective key values are set to $\infty$, except for the root node, which obtains the value 0. Thus, the root node is the first one being processed. Thereby all nodes adjacent to $v_i$, that are still contained in the priority queue and whose key values are larger then the weight of their respective edge, get a predecessor node and a new key value assigned. Eventually, if the priority queue is empty, every node, except the root node, has obtained exactly one predecessor node. Hence the set of minimum spanning tree edges emerges from $\{(v, \pi(v)) \mid v \in V \setminus \{v_r\}\}$. An efficient implementation of the priority queue, like e.g. as Fibonacci heap, results in a runtime of $O(m + n \log n)$.

**Algorithm 5** - *MST-Prim*($G$, $r$)

**Input**: $G = (V, E, w(E))$ - an undirected, connected graph, $v_r \in V$ – root node
**Output**: $E_T$ – a set of edges defining the MST

```
 1 forall (v ∈ V) do
 2    key[v] ← ∞                          // key[v] is a weight
 3    π(v) ← NIL                          // π(v) is the predecessor node
 4 end forall
 5 key[v_r] ← 0
 6 Q ← V                                  // Q is a priority queue
 7 while (Q ≠ ∅) do
 8    v_i ← extract_min(Q)
 9    forall (v_j ∈ V | e_{i,j} = {v_i, v_j} ∈ E) do
10       if (v_j ∈ Q ∧ w(e_{i,j}) < key(v_j)) then
11          π(v_j) ← v_i
12          key(v_j) ← w(e_{i,j})
13       end if
14    end forall
15 end while
16 return E_T ← {(v, π(v)) | v ∈ V \ {v_r}}
```

Both algorithms expect a weighting function. However the minutiae templates only consist of a set of $d$-dimensional points. Thus the edge weights still have to be defined.

### 5.3.2 Edge Weights

With regard to the above algorithms a minimum spanning tree is determined by successively adding those edges to the solution, whose weight is as small as possible. Concerning the minutiae templates those edges are sought, which connect in terms of their spatial distance close together lying minutiae. Since that distance depends on $d$ dimensions a one-dimensional representation should be found. In order to map the $d$-dimensional difference vectors to a one-dimensional edge weight, beside of the *maximum metric* with $\max\{|v_2^1 - v_1^1|, \ldots, |v_2^d - v_1^d|\}$ for instance *distance functions* or even *similarity measures* seem to be an eligible mean.

#### Distance Functions

A distance function maps the features of two objects to a non-negative, real number [Schmitt05]. The distance is "0" if both objects exactly conform to each other. In case of $d$-dimensional points the features are the individual coordinates. A classical representative for such a distance function is the *Minkowski metric* [Ferber03]:

$$mm(v_1, v_2) = \sqrt[m]{\sum_{i=1}^{d} \left| v_2^i - v_1^i \right|^m} \tag{11}$$

The *Manhattan distance* emerges if setting $m = 1$, whereas $m = 2$ results in the *Euclidian distance*. Note that the Minkowski metric exclusively considers the absolute value of the spatial distance between a source and a target node, which means that the difference vector $\Delta a_{i,j} = (5, 1)$ for example has the same distance as $\Delta a_{i,k} = (1, 5)$, although it is differently oriented. Thus in order to minimize the total number of encoding bits the orien-

tation of the edges might be of interest as well. Edge $a_{i,l}$ with $\Delta a_{i,l} = (6,1)$ instead of $a_{i,k}$ would for instance lead to a higher compression rate. Hence for considering equally directed edges similarity measures might be an appropriate edge weight, too.

**Similarity Measures**

A similarity measure is a function, which assigns a real number from the interval [0,1] to a pair of objects or points, respectively. Thereby "0" stands for no and "1" for a complete match. Information retrieval is the major field of application, where the feature vectors of documents or other objects are compared in order to determine the degree of similarity. Typical measures are the *cosine*, *Dice, overlap* and *Jaccard measure* [Ferber03]. For use as edge weight the *pseudo-cosine measure* seems to be most capable. Here those edges are being considered similar, which are almost equally directed [Ferber03]:

$$pcos(v_1, v_2) = \frac{\sum_{i=1}^{d} v_1^i \cdot v_2^i}{(\sum_{i=1}^{d} v_1^i) \cdot (\sum_{i=1}^{d} v_2^i)} \tag{12}$$

Thus, the similarity of two nodes is basically defined by their scalar product, while their spatial distance is not considered. However, in order to use the pseudo-cosine measure as weighting function in the MST algorithms, $w(a_{i,j})$ must be computed as follows: $w(a_{i,j}) = 1 - pcos(a_{i,j})$. After introducing the minimum spanning tree, algorithms for determination and possible weighting functions, the formerly given general formalization of the compression problem will be adapted to the minimum spanning tree.

### 5.3.3 Compressing Templates Using a Minimum Spanning Tree

For a given complete and directed graph $G = (V,A)$ the solution consists in

- an eligible weighting function $w(A)$, and
- a preferably optimal directed minimum spanning tree $T_S = (V,A_T)$ on $G$ with $A_T \subset A$.

For implementation beside of the maximum metric the Minkowski metric with $m = 2$ and $m = d$ will be analyzed as weighting functions. Furthermore the following objective function will hold for the compressed data:

$$size(encode(\Delta a)) = \left\lceil \sum_{i=1}^{d} \left( \chi_i \cdot (1 + \mathrm{ld}\ \tilde{a}^i) \ + \ (1 - \chi_i) \cdot \mathrm{ld}\ \tilde{v}^i \right) \right\rceil \tag{13}$$

Note that $\tilde{a}^i$ and $\tilde{v}^i$ define the respective domain borders of the difference vectors and nodes at coordinate $i$. Also note that the domain borders of the difference vectors conform to $\mathbb{D}_{\Delta A}$, see section 5.2.3, which means that in this case negative edge values are going to be allowed. Furthermore $\chi_i$ is a *characteristic function* [Chwatal08] which specifies, whether dimension $i$ is considered for compression ($\chi_i = 1$) or not ($\chi_i = 0$). In the latter case the respective coordinate value of the target node must be additionally encoded. Finally must be mentioned that the above algorithms for determining minimum spanning trees can only be applied to undirected graphs. Thus some adaptations are necessary. How these will look like in particular, is going to be described in the subsequent

algorithm chapter. But first the main approach of this thesis, which is based on the minimum label spanning tree problem, is going to be presented.

## 5.4 The Minimum Label Spanning Tree

Sometimes a spanning tree is sought that in a certain manner is as uniform as possible. A typical example therefore is the design of communication networks where many different transmission media can be used. Besides copper-cored cables such media are for instance fiberglass or even microwaves. A node of that network may communicate with other nodes using different transmission media. If treating this network as a graph, the communication media represent the edges that are depending on their types differently labeled. In order to save costs for construction and in order to minimize the network complexity it is very useful to use a minimum of different media, i.e. retaining the network regarding the transmission media as uniform as possible. Thus, Chang and Leu presented in 1996 the *minimum label spanning tree* (MLST) which can be used for solving such problems [Chang96].

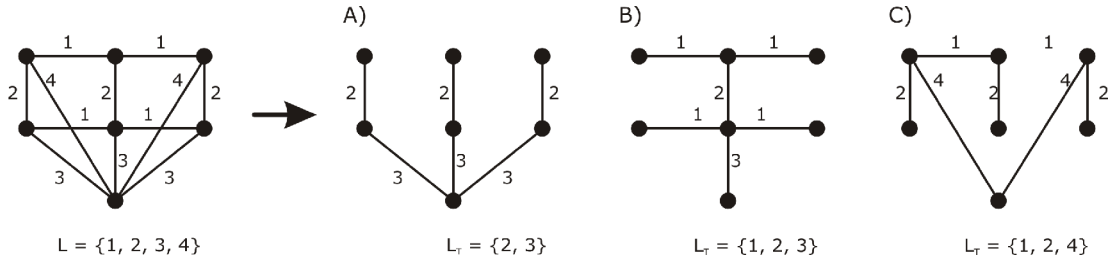



*Figure 21 - Minimum label spanning tree: On the left side a labeled graph is depicted, on which a MLST is going to be determined. Figures A), B) and C) show different solutions, whereat only variant A) is optimal. (Figures adopted from [Xiong04])*

Hence formally speaking, an undirected graph $G = (V, E, l(E))$ is given, where $l{:}E \to L$ with $L = \{l_1, \ldots, l_k\}$ is a function, that assigns at least one label from $L$ to every edge $e \in E$ of $G$. On that graph a spanning tree $T_L = (V, E_T, L_T)$ is sought with $E_T \subseteq E$, $L_T \subseteq L$ and $L_T \neq \emptyset$, such that the number of elements in $L_T$ is minimal and every edge $e \in E_T$ has assigned at least one label $l \in L_T$. There are already various algorithms for solving that problem, and thus some major ones will be shortly presented.

### 5.4.1 Algorithms for Solving the MLST Problem

The minimum label spanning tree problem is considered being *NP-complete* [Chang96]. Hence exact algorithms can only perform well on small graphs. That is why lots of algorithms have been already studied that do not necessarily return an optimal but a sufficiently good solution in an acceptable time. Thereto belong besides heuristic algorithms for instance also genetic ones.

#### An Optimal Algorithm

In [Chang96] the authors introduced an optimal algorithm based on an *A* search*, which is a special kind of the *best first search algorithms* on a graph $G$. In this special case the next label $l \in L$ to be added to the solution is selected with the aid of an *evaluation func-*

*tion* $f(l) = g(l) + h(l)$ [Russel04], whereat $g(l)$ specifies the number of already used labels. The heuristic function $h(l)$ in turn computes the estimated number of labels that are at least still necessary for obtaining a spanning tree. Insofar the heuristic function does not overestimate the number of labels, the $A^*$ algorithm is considered being complete and optimal [Russel04].

**Algorithm 6** - *OptMLST*($G$)

**Input**: $G = (V, E, l(E))$ – an edge-labeled, undirected graph with $l{:}E \rightarrow L = \{l_1, \ldots, l_k\}$
**Output**: $T_L = (V, E_T, L_T)$ – the obtained minimal label spanning tree with $E_T \subseteq E$ and $L_T \subseteq L$

```
 1 l_B ← l_s, L_T ← ∅                    // l_s = ∅ is the initial set of labels
 2 OPEN ← {l}                            // the set of generated but unexpanded labels
 3 CLOSED ← ∅                            // the set of processed labels
 4 repeat
 5    OPEN ← OPEN \ {l_B}
 6    CLOSED ← CLOSED ∪ {l_B}
 7    forall (l' ∈ L\L_T) do
 8       if (l' ∉ OPEN ∧ l' ∉ CLOSED) then
 9          compute h(l')
10          g(l') ← g(l_B) + 1
11          f(l') ← g(l') + h(l')
12          OPEN ← OPEN ∪ {l'}
13       end if
14    end forall
15    l_B ← the label with smallest f(l')
16    L_T ← L_T ∪ {l_B}
17    E_T ← E_T ∪ {e ∈ E | l(e) = l_B}
18 until (T_L is connected)
19 T_L ← arbitrary spanning tree after deleting cycle-forming edges from E_T
20 return T_L
```

Note, that in the above algorithm the initial solution is an empty set of labels for which holds $g(l_s) = 0$. In turn for that label $l_t$, that eventually produces a spanning tree, holds $h(l_t) = 0$. Experimental results have shown, that the presented algorithm works efficiently and that the heuristic function $h(l)$ can be computed in polynomial time [Chang96].

### Maximum Vertex Cover Algorithm

In addition to the optimal algorithm Chang and Leu introduced two heuristics. One of them, the *Maximum Vertex Cover Algorithm* (MVCA), is a construction heuristic, where in every iteration step a label with its corresponding edges is added to the solution. The runtime of that algorithm is $O(lmn)$, whereat $l$ is the number of labels, $m$ the number of edges and $n$ the number of nodes in the considered graph $G$. In [Krumke98] the authors presented an approximation algorithm that is based on that MVCA heuristic. They found a logarithmic performance guarantee of $(2 \ln n + 1)$, which Wan et al. improved to $(\ln (n - 1) + 1)$ in [Wan02]. A tighter bound which depends on the label frequency of a graph bounded by $b$ was obtained by Xiong et al. in [Xiong05]. In worst case the bound of the MVCA is the $b^{\text{th}}$ harmonic number $H_b = \sum_{i=1}^{b} \frac{1}{i}$.

**Algorithm 7** - *ApproximationMVCA*($G$)

**Input**: $G = (V, E, l(E))$ – an edge-labeled, undirected graph with $l{:}E \to L = \{l_1, \ldots, l_k\}$
**Output**: $T_L = (V, E_T, L_T)$ – the obtained minimal label spanning tree with $E_T \subseteq E$ and $L_T \subseteq L$

```
1  L_T ← ∅, E_T ← ∅
2  ∀v ∈ V: make_set(v)                    // create a union-find set for every node
3  repeat
4     forall (l ∈ L \ L_T) do
5        determine the number of CC's when adding all edges covered by l to E_T
6     end forall
7     l_B ← label with the smallest number of connected components
8     L_T ← L_T ∪ {l_B}
9     forall (e_{i,j} = {v_i, v_j} ∈ E | l(e) = l_B) do
10       E_T ← E_T ∪ {e_{i,j}}
11       union(v_i, v_j)
12    end forall
13 until (T_L is connected)
14 T_L ← arbitrary spanning tree after deleting cycle-forming edges from E_T
15 return T_L
```

This algorithm uses a union-find data structure for managing the connected components. In every iteration step a label $l$, that does not belong to the solution yet, is temporarily added together with its corresponding edges. Afterwards the number of connected components is determined, such that that label, which reduces the number of connected components most, can be obtained and eventually permanently added to the solution. Finally when the graph is connected, all superfluous edges are to be removed in order to obtain a spanning tree.

### Further Algorithms

The other heuristic introduced by Chang and Leu has turned out being inefficient [Chang96]. Therein an arbitrary initial spanning tree is created on the graph $G$. In every succeeding iteration step one edge of the solution is replaced by another valid one, which reduces the number of used labels. The main idea of this algorithm is based on a *local search*, i.e. some sort of *improvement heuristic*, which is in turn often applied subsequent to a construction heuristic.

In [Brüggemann02] the authors have investigated more efficient neighborhoods for local search algorithms, which swap $k$ of the available labels instead of the edges. Furthermore, Xiong et al. in 2004 introduced a genetic algorithm which seizes die idea of the MVCA heuristic, and whose results even outperform MVCA in many cases [Xiong04]. Another three genetic algorithms were proposed and analyzed in [Nummela06]. One of them includes a local search in its evaluation function, and thus also outperforms the results of the MVCA heuristic. Xiong et al. furthermore implemented modified versions of MVCA in [Xiong06] focussing on the label to be added initially.

The *Pilot Method*, a greedy heuristic developed by Duin and Voß in [Voss99], was applied to the MLST problem by Cerulli et al. in [Cerulli05]. Compared to other metaheuristics, such as *reactive tabu search*, *simulated annealing* and *variable neighborhood search*, best results were achieved in most cases, although the runtime is very large.

Consoli et al. proposed a *greedy randomized adaptive search procedure* and different versions of a variable neighborhood search in [Consoli07]. Compared to the results of an exact algorithm their results are optimal or near-optimal and can be quickly obtained.

Eventually the following section will show how the idea of the minimum label spanning tree problem can find use in the compression of fingerprint templates.

### 5.4.2 Compressing Templates Using a Minimum Label Spanning Tree

A complete and directed graph $G = (V, A)$ consisting of the set of nodes $V$ and the set of edges $A = \{a_{i,j} \mid a_{i,j} = (v_i, v_j),\ v_i, v_j \in V \wedge v_i \neq v_j\}$ is given again. Hence, a solution to the compression problem consists in [Chwatal08]:

- a set of reference vectors $R = \{r_1, \ldots, r_l\}$, whereat $R \subset \mathbb{D}$,
- an arborescence, i.e. a directed spanning tree $T_L = (V_T, A_T)$ on $G$ with $V_T \subseteq V$ and $A_T \subset A$,
- an assignment of all edges $a_{i,j} \in A_T$ to always exactly one reference vector $r \in R$, denoted by the index $\kappa_{i,j} \in \{1, \ldots, l\}$,
- an assignment of always one correction vector $\delta_j \in \mathbb{D}'$ to every node $v_j \in V_T$ except the root node, such that

$$v_j = (v_i + r_{\kappa_{i,j}} + \delta_j) \bmod \tilde{v} \tag{14}$$

holds, whereat $\mathbb{D}' = \{0, \ldots, \tilde{\delta}^1 - 1\} \times \ldots \times \{0, \ldots, \tilde{\delta}^d - 1\}$ with $\mathbb{D}' \subseteq \mathbb{D}$.

Thus, every node $v_j \in V_T$ but the root node is effectively stored by an index $\kappa_{i,j}$ to the respective reference vector and a small correction vector $\delta_j$, similar to a dictionary compression technique. Hence, with respect to a high compression ratio the domain borders $\tilde{\delta}^i$ with $i = 1, \ldots, d$ of the correction vectors as well as the number of reference vectors should be as small as possible. Therefore different minimization problems arise.

#### Minimization Problems

As mentioned above, beside of minimizing the number of reference vectors also minimizing the domain of the correction vectors is conceivable, in order to reduce the total number of bits for storing a template. This leads to the following minimization problems [Chwatal08]:

- *Min-l*: For a given correction vector domain $\tilde{\delta}$ a solution is sought, in which the *number of reference vectors* is minimal.
- *Min-* $\tilde{\delta}$ : If the number of reference vectors $l$ is specified, the solution must minimize the *domain* $\tilde{\delta}$ *of the correction vectors*.
- *Min-*$\lambda$: For minimizing the *total number of bits* neither the number of reference vectors nor the domain of the correction vectors is stated. Thus, a capable tradeoff between those values must be found.

For solving the Min-*l* minimization problem the minimum label spanning tree is appropriate. Therefore a set of so-called candidate reference vectors $R_C$ will correspond to the set of available labels $L$, from which a smallest possible subset, the set of reference vectors $R$, is going to be determined, which constitute a spanning tree on a graph $G$. But just like the weighting function of the MST the set of candidate reference vectors is not given and thus must be determined at first.

### Candidate Reference Vectors

Let $\mathbb{D} = \{0, \ldots, \tilde{v}^1\} \times \ldots \times \{0, \ldots, \tilde{v}^d\}$ be the set of vectors, that are induced by the domain borders $\tilde{v}$ of a given set of nodes $V$ from a graph $G = (V, A)$. Then the set of *candidate reference vectors* $R_C \subseteq \mathbb{D}$ is a subset of $\mathbb{D}$, which fulfills formula (14) with respect to the correction vector domain $\tilde{\delta}$. Furthermore let the set of *reference vectors* $R \subseteq R_C$ be the smallest possible subset from the set of candidate reference vectors, that is able to constitute a spanning tree on $G$. So the *number of reference vectors* will be denoted by $l = |R|$.

In section 5.2 the set of *difference vectors* for a graph $G$ was already defined by $A_\Delta = \{\Delta a_1, \ldots, \Delta a_\mu\} = \{\Delta a_{i,j} \mid \Delta a_{i,j} = (v_j - v_i) \bmod \tilde{v} \wedge (v_i, v_j) \in A\}$. Then let $A'_\Delta \subseteq A_\Delta$ be an arbitrary subset of $A_\Delta$ with $A'_\Delta \neq \emptyset$. Furthermore let $\rho(A'_\Delta) = (\rho^1(A'_\Delta), \ldots, \rho^d(A'_\Delta))$ with $\rho^i(A'_\Delta) = \operatorname{argmax}_{j=1,\ldots,|A'_\Delta|} \Delta a^i_j - \Delta a^i_{j-1}$ and $i = 1, \ldots, d$ be the standard reference vector of $A'_\Delta$, which represents all vectors from $A'_\Delta$ with respect to the correction vector domain $\tilde{\delta}$. Finally let $D(r) \subseteq \mathbb{D}$ be the set of all vectors from $\mathbb{D}$, that a certain reference vector $r \in R$ can represent with respect to $\tilde{\delta}$. Then $R_C$ can be represented by the standard reference vectors of all possible subsets $A'_\Delta$ of $A_\Delta$ [Chwatal08]:

$$R_C = \{\rho(A'_\Delta) \mid A'_\Delta \subseteq A_\Delta, A'_\Delta \neq \emptyset \wedge A'_\Delta \subseteq D(\rho(A'_\Delta))\} \tag{15}$$

However, according to that formula $R_C$ can still contain reference vectors $r \in R$ that are dominating further reference vectors $r' \in R$, i.e. $A_\Delta(r') \subseteq A_\Delta(r)$, whereby $A_\Delta(r)$ is the set of all difference vectors from $A_\Delta$, that $r$ is able to represent with respect to $\tilde{\delta}$. In order to further reduce $R_C$, from all equal reference vectors for which $A_\Delta(r) = A_\Delta(r')$ holds, only one is kept. In turn, for those reference vectors $r$ that are covering further ones, those ones are removed, that are dominated by $r$, i.e. for which $\exists r: A_\Delta(r') \subset A_\Delta(r)$ holds. Thus $|R_C| = 1$ emerges as a lower bound for the number of possible candidate reference vectors. The upper bound in turn depends on the number of dimensions $d$ as well as on the number of vectors in the set $A_\Delta$, which means that $|R_C| = O(\mu^d)$ [Chwatal08].

In [Chwatal08] there is also an algorithm given that computes $R_C$ with the aid of a restricted enumeration of all subsets $A'_\Delta \subseteq A_\Delta$. It will be used for implementation as a preprocessing step, see chapter 6.

### Objective Function

In contrast to the MST, in the MLST approach the edges are exclusively considered within a ring domain. Therefore only one objective function arises as size for encoding the edges:

$$size(encode(\Delta a)) = \left\lceil \operatorname{ld} l \; + \; \sum_{i=1}^{d} \left( \operatorname{ld} \tilde{\delta}^i \cdot \chi_i \; + \; \operatorname{ld} \tilde{v}^i \cdot (1 - \chi_i) \right) \right\rceil \tag{16}$$

At this $l$ is the number of reference vectors again, while $\tilde{\delta}^i$ and $\tilde{v}^i$ are the domain borders at coordinate $i$ of the correction vectors and nodes, respectively. Due to additionally storing the reference vectors in this approach, the global objective function from formula (9) must be extended, as well:

$$\begin{aligned}\lambda = {} & size(\text{MetaData}) + size(\text{ConstData}') + 2(n-1) \\ & + (n-1) \cdot size(encode(\Delta a)) + l \cdot \sum_{i=1}^{d} \lceil \operatorname{ld} \tilde{v}^i \rceil \cdot \chi_i\end{aligned} \tag{17}$$

Note that also most of the MLST algorithms can only be applied to undirected graphs. Thus those algorithms must to be adapted for implementation, too. But before these adaptations will be presented, the WBST approach, which can be regarded as extension of the MLST approach, but however solves the Min- $\tilde{\delta}$ minimization problem, is going to be introduced.

## 5.5 The Weight Balanced Spanning Tree

Similar to the idea of the MLST the *weight balanced spanning tree* (WBST), sometimes also referred to as *balanced spanning tree*, represents a tree on a graph, whose edges are as equal as possible. However, not the number of different labels or edges is minimized, but the difference between the smallest and largest edge weight. Thus the weight balanced spanning tree requires just like the MST an edge-weighted graph $G = (V, E, w(E))$ with $w{:}E \rightarrow \mathbb{R}$. Again, a typical example for this kind of trees are computer networks. If for instance information packets are to be transferred from one node to all others, the buffering requirements within the single nodes should be kept small, in order to save memory capacity requirements. Therefore it makes sense to minimize the difference between the transmission rates of the affected communication media. In a network, consisting of 10, 100 and 1000 Mbps transmission rate media, the MLST approach could for example deliver a spanning tree, that only consists of 10 and 1000 Mbps transmission lines. Indeed this would be a valid solution, but not in terms of the WBST problem, since the difference between the smallest (10 Mbps) and the largest (1000 Mbps) edge weight is not minimal. This section will restate the compression problem in terms of the WBST, but first algorithms for computation are given.

### 5.5.1 Algorithms for Solving the WBST Problem

For solving the WBST problem in literature only one algorithm can be found [Gupta04]. After sorting the edges with respect to their weights, they are successively added to $G'$, until this graph is connected. Afterwards the edges, starting with those having the smallest weight, are successively removed again as long as $G'$ remains connected. Thus, the range of the weights is reduced. Since furthermore all possible ranges of weight are covered by stepwise shifting the indices *low* and *high*, the algorithm is correct and always finds an optimal solution. If efficiently checking the connectedness in line 8, using for instance a depth first search with runtime $O(n + m)$, and since the repeat loop takes time $O(2m)$ according to alternately increasing *high* and *low* by 1, a runtime of $O(2m \cdot (n + m))$ and thus $O(m^2)$ emerges.

**Algorithm 8** - *WBST*($G$)

**Input**: $G = (V, E, w(E))$ – a connected and undirected graph with $w{:}E \rightarrow \mathbb{R}$
**Output**: $T_W = (V, E_T)$ – a weight balanced spanning tree with $E_T \subseteq E$

```
 1 sort the edges increasingly according to their weights, such that w(e_1) ≤ ... ≤ w(e_m)
 2 diff ← w(e_m) − w(e_1)
 3 low ← 1, high ← 1
 4 low_B ← 1, high_B ← m
 5 repeat
 6    G' ← (V, E'), E' ← ∅
 7    E' ← {e_i ∈ E | low ≤ i ≤ high}
 8    if (G' is connected) then
 9       if (diff > w(e_high) − w(e_low)) then
10          diff ← w(e_high) − w(e_low)
11          high_B ← high
12          low_B ← low
13       end if
14       low ← low + 1
15    else high ← high + 1
16 until (high = m ∧ low = m)
17 E_T ← {e_i ∈ E | low_B ≤ i ≤ high_B}
18 return T_W
```

Eventually the question, how fingerprint templates can be compressed by means of the weight balanced spanning tree, is to be answered.

### 5.5.2 Compressing Templates Using a Weight Balanced Spanning Tree

Once again a complete and directed graph $G = (V, A)$ is given. A solution this time consists of:

- an eligible weighting function $w(A)$,
- a directed spanning tree $T_W = (V_T, A_T)$ with $V_T \subseteq V$ and $A_T \subseteq A$,
- an offset vector $r_O$, and
- an assignment of a correction vector $\delta_j$ to every node $v_j \in V_T$, such that

$$v_j = (v_i + r_O + \delta_j) \bmod \tilde{v} \tag{18}$$

holds, whereat $v_i \in V_T$ and $\tilde{v}$ is the node domain, again.

Thus, analogously to the MLST approach every node $v_j \in V_T$ except the root is represented by an offset vector $r_O$ and a hopefully small correction vector $\delta_j$. However in contrary, the storage space for the index $\kappa_{i,j}$ can be dropped due to the existence of only one single reference vector. This vector emerges dynamically from the computed minimal coordinate values of the edges belonging to $T_W$. Hence, for the encoding size of the edges the following objective function arises:

$$size(encode(\Delta a)) = \left\lceil \sum_{i=1}^{d} (\chi_i \cdot \mathrm{ld}\ \tilde{\delta}^i \ + \ (1-\chi_i) \cdot \mathrm{ld}\ \tilde{v}^i) \right\rceil \tag{19}$$

Remember that $\tilde{\delta}$ and $\tilde{v}$ are the domain borders of the correction vectors and the nodes, and that $\chi_i$ is a characteristic function, which specifies whether dimension $i$ is considered for solution, or not. So if the difference between the smallest and the largest edge weight is minimal, the domain of the correction vectors should be relatively small as well. Thereto the weighting function should reflect the similarity between the source and target nodes with respect to their coordinate values, such that those edges are selected for solution which do only differ slightly in their coordinates. Thus for implementation similarity measures and distance functions (see section 5.3.2) are going to be examined of being an eligible weighting function. Furthermore the WBST algorithm is alike the other approaches only applicable for undirected graphs. Hence for implementation again some adaptations are necessary.

Since all approaches are introduced now, the implemented algorithms can eventually be presented.

# Chapter 6 - Algorithms

*"Beware of bugs in the above code – I have only proved it correct, not tried it."*

Donald Knuth

The last chapters have given a review about the fingerprint compression problem, and further presented in detail the different approaches for solution. Since none of the introduced algorithms can be exactly applied, this chapter will describe the adaptations for implementing the discussed ideas. Thereto, starting with some methods used by most of the algorithms, the specifically adapted implementations are going to be explicitly described.

## 6.1 Global Algorithms

Independently from the applied approaches for solution there are algorithms needed for pre- and post-processing graphs. That are for instance methods for creating complete graphs, assigning edge weights or labels, and also removing unnecessary edges and nodes. But one of the most important methods, which is used in every approach, is the test whether a graph contains a directed spanning tree. Thus, this algorithm is first of all presented in detail.

### 6.1.1 Looking for a Directed Spanning Tree

A simple way to determine whether a directed graph contains a directed spanning tree of size $k$, is to run a depth first search from every node and thus, to look whether the desired number of nodes is reachable. In the worst case the search indeed is performed from every node in the graph, which leads to a runtime of $O(n \cdot (n+m))$, which conforms to $O(n^3)$). Some of the algorithms that are going to be presented, especially the MVCA based minimum label spanning tree algorithms, perform this test in every iteration. Hence this task is very time critical. Thus it makes sense to minimize the number of depth first search calls. This is on the one hand possible if only those graphs are explicitly examined that have a real chance of containing an arborescence, and on the other hand by running the DFS only from that nodes that come into question as possible root nodes. A graph can only contain a directed spanning tree, if

- the number of edges is sufficiently large, i.e. $|A| \geq k-1$, and if
- there are enough nodes with incoming edges. Thus, if the number of nodes with indegree zero is larger than $|V| - k + 1$, then the graph for sure cannot contain a directed spanning tree.

In both cases no depth first search calls are to be started. Furthermore, as possible root nodes only that ones are appropriate which have an outdegree larger than zero. Thus the depth first search calls may only be started from those nodes.

A special case occurs if a spanning tree is sought on the whole graph, i.e. if $k = |V|$ holds, and if there exists exactly one node without incoming edges. Then this node must be the root node. Therefore the DFS is only started from that single node and if not all further nodes are reached, then the graph does not contain an arborescence [Oberlechner08]. Thus the implemented algorithm looks like follows:

**Algorithm 9** - *containsArborescence*$(G,k)$

**Input**: $G=(V,A)$ – a directed graph, $k$ – number of nodes to connect
**Output**: **true** – if $G$ contains a directed spanning tree of size $k$, **false** – otherwise

```
 1  V_{δ0+} ← nodes having indegree 0
 2  V_{δ>0-} ← nodes having an outdegree larger than 0
 3  if (|A| < (k − 1)) then return false          // there are not enough edges
 4  if ( |V_{δ0+}| > |V| − k + 1) then return false   // to many nodes have indegree zero
 5  if (|V| = k ∧ |V_{δ0+}| = 1) then
 6     v ← is the node from V_{δ0+}               // this must be the root node
 7     V_DFS ← DFS(G, v)                          // V_DFS the set of reached nodes
 8     if (|V_DFS| ≥ k) then return true
 9  else
10     forall (v ∈ V_{δ>0-}) do
11        V_DFS ← DFS(G, v)
12        if (|V_DFS| ≥ k) then return true
13     end forall
14  end if
15  return false
```

First of all is checked whether the graph contains a sufficiently large number of edges, and nodes with incoming edges. If not the algorithm stops. In a next step is tested whether a spanning tree over all nodes is sought, and whether there exists a single node with indegree zero. Then the depth first search is called from that node, and afterwards the algorithm returns. Only then if none of the above cases can be applied, the depth first search calls are performed from all nodes with outgoing edges, and as far as an arborescence is found the algorithm stops. Thus in a worst case the runtime of $O(n^3)$ does not change, but actually the implemented algorithm runs clearly faster.

### 6.1.2 Global Pre-processing

The most important step of pre-processing is to create a complete and directed graph, which is simply done by inserting the forward and backward directed edges between all pair of nodes. Thus the implemented function *createCompleteDirectedGraph*$(V)$ creates a graph $G=(V,A)$ from the set of nodes $V=\{v_1,\ldots,v_n\}$ by inserting the following edges: $A=\{a_{i,j}\,|\,a_{i,j}=(v_i,v_j), v_i,v_j\in V\wedge v_i\neq v_j\}$. Further pre-processing steps additionally have to assign either edge weights or labels to the created edges. But since those assignments depend on the applied approaches, they are presented in the respective sections.

### 6.1.3 Global Post-processing

According to the implemented strategies of the MST, MLST and WBST algorithms the graph, which eventually is constituted, must not necessarily be a directed spanning tree consisting of $k$ nodes. Nevertheless it contains such a tree. This means, that the superfluous nodes and redundant edges still have to be removed.

The respective algorithm successively removes every edge from the solution and checks whether the graph still contains a $k$-node arborescence. If not the respective edge is inserted again. Note, that this check is necessary in order to find and finally remove all

separated nodes. Thus eventually exactly $k-1$ edges remain. If $k$ does not equal the number of nodes in the graph, furthermore redundant nodes, i.e. the separated nodes having no incoming and no outgoing edges, stay over and are afterwards deleted as well.

**Algorithm 10** - *removeRedundantEdgesAndNodes*$(G,k)$

**Input**: $G=(V,A)$ – a graph containing a directed spanning,
$k$ – number of nodes to connect
**Output**: $G=(V,A)$ – a directed spanning tree consisting of $k$ nodes and $k$ - 1 edges

```
 1 forall (a ∈ A) do                                  // remove redundant edges
 2   A ← A \ {a}
 3   if not (containsArborescence(G, k)) then
 4     A ← A ∪ {a}
 5   end if
 6 end forall                                         // remove redundant nodes
 7 if (|V| ≠ k) then
 8   forall (v ∈ V) do
 9     if (δ(v) = 0) then V ← V \ {v}
10   end forall
11 end if
12 return G
```

Due to the *containsArborescence*() calls for every removed edge a runtime of $O(m \cdot n^3 + n)$ and thus $O(n^5)$ emerges. Since all fundamental algorithms are introduced now, the following sections will present the single approaches for solution, starting with the algorithms for determining a directed minimum spanning tree.

## 6.2 Directed Minimum Spanning Tree Algorithm

The basic algorithm for the approach based on the directed minimum spanning tree is the following:

**Algorithm 11** - *DirectedMST*$(V,k)$

**Input**: $V=\{v_1,\ldots,v_n\}$ – a set of data points, $k$ – number of nodes to connect
**Output**: $T_S=(V_T,A_T)$ – a directed minimum spanning tree

```
1 G = (V, A) ← createCompleteDirectedGraph(V)
2 G = (V, A, w(A)) ← assignEdgeWeights(G)
3 T_S = (V, A_T) ← computeSolution(G, k)
4 T_S = (V_T, A_T) ← removeRedundantEdgesAndNodes(T_S, k)
5 return T_S
```

The function *assignEdgeWeights*() assigns an edge weight to all edges in the graph $G$. Note that the maximum metric, Minkowski metric with $m=2$ and $m=d$ are considered for implementation. A solution is then computed using the method *computeSolution*(), which returns the subgraph $T_S$ of $G$, that in turn contains the hopefully optimal directed spanning tree consisting of $k$ nodes. Nevertheless $T_S$ may still contain superfluous edges and nodes that subsequently are to be removed.

Since the MST algorithms presented in chapter 5 cannot be directly applied to directed graphs in order to compute a solution some adaptations are necessary. Although Prim's algorithm is supposed to perform better on dense graphs, it seems to be inadequate for this problem because a directed spanning tree of size $k$ is sought and no predefined root node is given. In order to obtain an optimal $k$-node spanning tree therefore an adapted version would have to be started from every node, resulting in a runtime of $O(n \cdot (m + n \log n))$. Existing algorithms for solving the directed minimum spanning tree, for instance presented in [Chu65] and [Edmonds67] have runtime $O(\min\{m \log n, n^2\})$. However in both cases a root node is needed again and a tree containing all nodes of the given graph is sought, i.e. that adaptations would heighten the runtime by factor $n$. Thus Kruskal's algorithm was chosen to be modified. In order to maintain strongly connected components instead of only adding the forward directed edges to the graph, the backward directed edges are added as well. Obviously this leads to cycles, which however are removed during post-processing. Note that this strategy is only applicable if the forward and backward directed edges have the same weights, which holds in this case according to the computation of the edge weights, see section 5.3.2. A second adaptation concerns the early termination of the algorithm as far as a strongly connected component (SSC) consists of at least $k$ nodes. Since the forward and backward directed edges only differ in their signs but not in their absolute distances, this algorithm is optimal, alike the original algorithm of Kruskal.

**Algorithm 12** - *SimpleMST::computeSolution*$(G, k)$

**Input**: $G = (V, A, w(A))$ – a complete, directed and edge-weighted graph
$k$ – number of nodes to connect
**Output**: $T_S = (V, A_T)$ – a graph containing the directed spanning tree

```
1  A_T ← ∅
2  ∀v ∈ V: make_set(v)                          // initialize union-find data structure
3  sort all edges a ∈ A according to their edge weights, such that w(a_1) ≤ … w(a_m)
4  forall ((a_{i,j} = (v_i, v_j)) ∈ A) do
5     if (find_set(v_i) ≠ find_set(v_j) then
6        A_T ← A_T ∪ {a_{i,j}, a_{j,i}}
7        union(v_i, v_j)
8        if (|find_set(v_i)| ≥ k) then return T_S = (V, A_T)
9     end if
10 end forall
11 return T_S = (V, A_T)
```

The runtime conforms to the undirected case with $O(m \cdot \log n)$, since the only adaptations are a further *find_set*() operation and the insertion of a further edge in every iteration. Thus, this algorithm provides a very simple and fast approach for solving the template compression problem.

## 6.3 Directed Minimum Label Spanning Tree Algorithms

In order to solve the fingerprint template compression problem by means of a MLST, the realization focuses on variations of the MVCA heuristic. Thereto two different versions are implemented. A GRASP algorithm based on the second MVCA approach was imple-

mented as well. All these algorithms will be described on the following pages, but first some considerations on upper bounds regarding the runtime are made.

### 6.3.1 Considerations on Upper Bounds

Let $n = |V|$ be the number of nodes in a graph $G$. Thus in a directed and complete graph the number of edges is $m = |A| = n \cdot (n-1)$, what conforms to an upper bound of $O(n^2)$. Each of those edges can be represented by a difference vector $\Delta a_{i,j} = v_j - v_i$, that indeed can be identical for different edges, but must not be. Hence the upper bound for the number of difference vectors $\mu = |A_\Delta|$ in a graph is stated by the number of edges, i.e. $O(m)$, or even $O(n^2)$. Those difference vectors furthermore are used for determining the candidate reference vectors, for which in turn an upper bound of $\ell = |R_C| = O(\mu^d)$ was declared in section 5.4.2. With $\mu = O(m)$ this conforms to $O(m^d)$ or $O(n^{2d})$, respectively. Finally the number of reference vectors $l = |R|$ depends on the number of edges in a tree, which is defined with by $n-1$, and thus $O(n)$ is an upper bound for $l$. Since all important values for subsequent runtime considerations are defined now, the MLST algorithms will be presented starting with some global pre-processing steps.

### 6.3.2 MLST specific Pre-processing

Subsequent to the global pre-processing, i.e. the creation of a complete and directed graph, the MLST specific pre-processing is performed. Thereby all candidate reference vectors $R_C$, which are specific for the given graph $G$ and also depend on the specific correction vector domain $\tilde{\delta}$, are determined and afterwards assigned to the edges of $G$ by means of the labeling function $l{:}A \rightarrow R_C$. Thereto in the respective algorithm for every difference vector $\Delta a \in A_\Delta$ each candidate reference vector $r \in R_C$ is checked of being capable of representing $\Delta a$. If this is the case, $r$ is added to the set of labels $l(a)$ of the respective edge $a$. Thus the assignment has a runtime of $O(m\ell)$.

But before the candidate reference vectors can be assigned to the respective edges they have to be determined first, following the idea presented in section 5.4.2 from [Chwatal08]:

**Algorithm 13** - *determine*$R_C(A_\Delta)$

**Input**: $A_\Delta = \{\Delta a_1, \ldots, \Delta a_\mu\}$ – the set of difference vectors for a graph $G$
**Output**: $R_C$ – the set of determined candidate reference vectors

1 $checkedBB \leftarrow \emptyset$, $S \leftarrow \emptyset$, $N \leftarrow \emptyset$, $R_C \leftarrow \emptyset$
2 $\Omega \leftarrow \{1, \ldots, \mu\}$
3 $R_C \leftarrow determineR_C(R_C, checkedBB, S, 0, 0, N, \Omega)$
4 **return** $R_C$

The set of candidate reference vectors is determined using a restricted enumeration of all possible subsets $A_\Delta' \subseteq A_\Delta$. These sets $A_\Delta'$ in turn can be represented by their *bounding boxes* (BB), i.e. by two vectors $r$ and $\hat{r}$ that span the smallest possible range, in which all vectors of $A_\Delta'$ are located. In the algorithm thereto three disjoint sets, $S$, $N$ and $\Omega$, are used which at each point in time represent a partitioning of the difference vectors $A_\Delta$, i.e.

$A_\Delta = S \cup N \cup \Omega$, each of them holding indices to the respective vectors of $A_\Delta$. Thereby $S$ represents those difference vectors that are located within the currently considered bounding box. In turn $N$ holds those vectors that are actively excluded from further considerations. Finally $\Omega$ represents the still opened, i.e. not considered vectors of $A_\Delta$. So the actual enumeration algorithm is the following:

**Algorithm 14** - *determine*$R_C$($R_C$, var *checkedBB*, var $S$, $r$, $\hat{r}$, var $N$, var $\Omega$)

**Input**: $R_C$ – the set of already found candidate reference vectors
*checkedBB* – the already checked bounding boxes
$S$, $N$, $\Omega$ – sets of indices to vectors from $A_\Delta$, $r$, $\hat{r}$ – vectors spanning a BB
**Output**: $R_C$ – the set of candidate reference vectors

```
 1 I ← ∅, C ← ∅
 2 if (S = ∅) then C ← {1, ..., μ}
 3 else
 4    checkedBB ← checkedBB ∪ {(r, r̂)}
 5    forall (j ∈ Ω) do                                  // find new vectors in the BB
 6       if ( Δa_j^i ∈ { r^i, ..., r̂^i mod r̃^i }, ∀i = 1, ..., d) then I ← I ∪ {j}
 7    end forall
 8    S ← S ∪ I, Ω ← Ω \ I
 9    forall (j ∈ Ω) do                                  // find addable vectors
10       if ( Δa_j^i ∈ {( r̂^i − δ̃^i + 1) mod r̃^i , ..., (r + δ̃^i − 1) mod r̃^i }, ∀i = 1, ..., d) then
11          C ← C ∪ {j}
12       end if
13    end forall
14 end if
15 if (C = ∅) then
16    if not (∃j ∈ N | Δa_j^i ∈ {( r̂^i − δ̃^i + 1) mod r̃^i , ..., (r + δ̃^i − 1) mod r̃^i }) then
17       R_C ← R_C ∪ {r}
18    end if
19 else
20    forall (j ∈ C) do
21       S ← S ∪ {j}, Ω ← Ω \ {j}
22       (r', r̂') ← updateBB(r, r̂ ,Δa_j)
23       if ((r', r̂') ∉ checkedBB) then
24          R_C ← determineR_C(R_C, checkedBB, S, r', r̂' , N, Ω)
25       end if
26       S ← S \ {j}, N ← N ∪ {j}
27    end forall
28 end if
29 S ← S \ I, N ← N \ C, Ω ← Ω ∪ I ∪ N
30 return R_C
```

In a first step a set of indices $I$ is determined, which represents those vectors of $A_\Delta$ that are covered by the currently considered bounding box $(r, \hat{r})$. Those indices are afterwards moved from $\Omega$ to $S$, and a further set of indices $C$ is obtained, holding those vectors of $A_\Delta$ that can be added to $S$ by expanding the bounding box. Finally, all indices $j$ of $C$ are moved from $\Omega$ to $S$. Afterwards the bounding box is updated, and if it was not considered yet, *determine*$R_C$() is called recursively in order to find further reference vectors.

As far as $C$ is empty no further vector can be added to $S$ and thus the recursion terminates, but first is checked whether $r$ is dominated by another candidate reference vector already contained in $R_C$. If not $r$ is added to $R_C$. With the end of each recursion the considered $j \in C$ is moved from $S$ to $N$ and eventually all sets are reset to their original states. The update of the bounding box before the recursion calls is performed as follows:

**Algorithm 15** - *updateBB*($r$, $\hat{r}$ , $\Delta a_j$)

**Input**: $r$, $\hat{r}$ – vectors spanning a BB, $\Delta a_j$ – the difference vector currently added to $S$
**Output**: $r'$, $\hat{r}'$ – vectors spanning the new BB

1 **if** ($\delta = 0$) **then**
2 $r' \leftarrow \Delta a_j$, $\hat{r}' \leftarrow \Delta a_j$
3 **else**
4 **for** ($i = 1, \ldots, d$) **do**
5 **if** ( $\Delta a_j^i \in \{(\hat{r}^i - \tilde{\delta}^i + 1) \bmod \tilde{v}^i, \ldots, r^i\}$) **then** $r'^i \leftarrow \Delta a_j^i$ **else** $r'^i \leftarrow r^i$ **end if**
6 **if** ( $\Delta a_j^i \in \{(\hat{r}^i, \ldots, (r^i + \tilde{\delta}^i - 1) \bmod \tilde{v}^i)$ **then** $\hat{r}'^i \leftarrow \Delta a_j^i$ **else** $\hat{r}'^i \leftarrow \hat{r}^i$ **end if**
7 **end for**
8 **end if**
9 **return** ($r'$, $\hat{r}'$ )

According to [Chwatal08] *determine*$R_C$() has a runtime of $O(\mu^d)$ which conforms to $O(\ell)$, see section 6.3.1. Hence the total runtime of pre-processing is $O(\ell + m\ell)$, and thus $O(m\ell)$. For large delta values therefore not neglectable runtimes may arise, however for small delta values the computation of the candidate reference vectors is sufficiently fast.

### 6.3.3 MLST specific Post-processing

In addition to superfluous edges and nodes, after termination of the MLST algorithms the solution may contain also redundant reference vectors. In order to not worsen the success of compression they ought to be removed, too. However, finding the redundant reference vectors is supposed to be a hard problem, and thus their determination happens heuristically:

**Algorithm 16** - *removeRedundantLabels*($G$)

**Input**: $G = (V, A, l(A))$ – a directed, edge-labeled graph with $l{:}A \rightarrow R$
**Output**: $G = (V, A', l'(A))$ – a cleaned graph with $A' \subseteq A$ and $l'{:}A' \rightarrow R'$, whereat $R' \subseteq R$

1 **forall** ($r \in R$) **do**
2 $A(r) \leftarrow \{a \in A \mid r \in l(a)\}$ // all edges covered by $r$
3 $\forall r' \in R{:}\ A(r) \leftarrow A(r) \setminus \{a \in A \mid r' \in l(a)\}$ // edges covered by $r'$
4 $A \leftarrow A \setminus A(r)$
5 **if not** (*containsArborescence*($G, k$)) **then** $A \leftarrow A \cup A(r)$ **end if**
6 **end for**
7 **return** $G$

Every reference vector $r \in R$ together with its corresponding edges is successively removed from $G$. Subsequently is checked whether $G$ still contains a directed spanning tree of size $k$. If not, $r$ cannot be removed from the solution, and the respective deleted edges are to be restored. However, there can be edges that are not only covered by $r$ but

also by further reference vectors from $R$. Those ones should not be considered on removal. In the respective algorithm presented above in every iteration step the set of edges $A(r)$ is determined, that contains those edges which are covered by $r$. That ones which are also covered by further reference vectors are subsequently removed from $A(r)$. Thus, according to the *containsArborescence*() calls with runtime $O(n^3)$ for all reference vectors a total runtime of $O(l \cdot n^3)$ emerges, which according to the considerations in 6.3.1 conforms to $O(m^2)$.

Since all elementary MLST specific algorithms for pre- and post-processing are introduced now, we can proceed to the MLST algorithms starting with the MVCA based construction heuristics.

### 6.3.4 MVCA based Construction Heuristics

Since the maximum vertex cover algorithm returns good solutions for the undirected MLST problem, the idea was adapted within this thesis and extended for the directed case. Thereto two different approaches were implemented – the *MvcaMLST* and the *GreedyMLST*. The structure of both algorithms is identical, they only differ in their applied greedy strategies. The overall procedure is described in the following algorithm.

**Algorithm 17** - *DirectedMLST*$(V, k)$

**Input**: $V = \{v_1, \ldots, v_n\}$ – a set of data points, $k$ – number of nodes to connect
**Output**: $T_L = (V_T, A_T, R)$ – a directed minimum label spanning tree

1 $G = (V, A) \leftarrow$ *createCompleteDirectedGraph*$(V)$
2 $G = (V, A, l(A)) \leftarrow$ *assignEdgeLabels*$(G)$, with $l{:}A \rightarrow R_C$
3 $T_L = (V, A_T, R) \leftarrow$ *computeSolution*$(G, k)$
4 $T_L = (V, A_T, R) \leftarrow$ *removeRedundantLabels*$(T_L, k)$
5 $T_L = (V_T, A_T, R) \leftarrow$ *removeRedundantEdgesAndNodes*$(T_L, k)$
6 **return** $T_L$

Analogously to the MST algorithms first of all a complete and directed graph is created, and afterwards the respective edge labels, i.e. the candidate reference vectors, are assigned. Additionally redundant reference vectors are removed during post-processing. This is necessary because with respect to the computed set of edges there can exist reference vectors that are redundant with respect to other reference vectors of the solution.

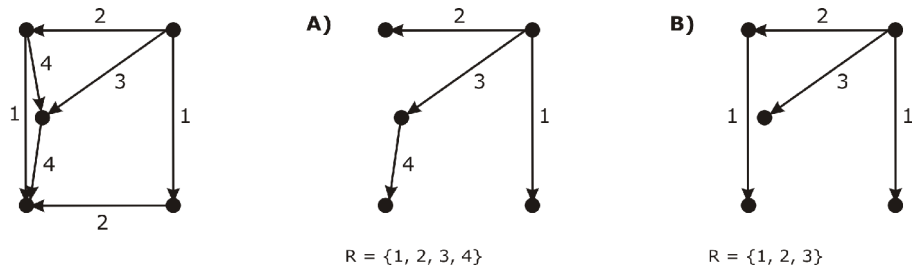



*Figure 22 - Post-processing: Let the left graph G be a computed solution. Figure A) shows a possible result, if first removing all redundant edges and nodes. The number of reference vectors cannot be decreased, although the solution is not optimal for G. Figure B) in turn depicts one of the both optimal solutions which arises if first removing all redundant reference vectors. Note that for convenience the modulo calculation of the edges is not considered within this and the following images.*

Important at this point is that the removal happens before the global post-processing, i.e. the removal of the redundant edges and nodes. Since the removal of redundant edges and nodes does not respect the reference vectors but only successively removes edges until a spanning tree of size $k$ remains, an unnecessarily higher number of reference vectors could remain. Thus it seems to be more suitable to first remove the redundant reference vectors, although according to the heuristic approach of the *removeRedundantLabels*() method indeed an optimal solution is not guaranteed.

### A first MVCA inspired Algorithm

The *MvcaMLST* algorithm uses a very simple greedy strategy: In every step that candidate reference vector is added which covers the greatest number of edges not belonging to the solution yet.

**Algorithm 18** - *MvcaMLST*::*computeSolution*$(G, k)$

**Input**: $G = (V, A, l(A))$ – a complete, directed and edge-labeled graph with $l{:}A \to R_C$
$k$ – number of nodes to connect
**Output**: $T_L = (V, A_T, R)$ – a graph containing the minimum label spanning tree

```
1 A_T ← ∅, R ← ∅
2 while not (containsArborescence(G, k)) do
3    r ← label from R_C covering the highest number of edges not belonging to A_T yet
4    A(r) ← {a ∈ A \ A_T | r ∈ l(a)}        // edges covered by r and not belonging to A_T
5    A_T ← A_T ∪ A(r)
6    R ← R ∪ {r}, R_C ← R_C \ {r}
7 end while
8 return T_L
```

Thereto, as long as no directed spanning tree of size $k$ emerged those edges, that are covered by the candidate reference vector $r$ covering the highest number of edges not belonging to the solution yet, are added to the solution, and $r$ is moved from $R_C$ to $R$. Thus, according to the *containsArborescence*() calls, the while loop and finding the candidate reference vector with the highest number of covered edges in line 3 with a linear runtime of $O(\ell)$ a total runtime of $O(m \cdot (\ell + n^3))$, and thus $O(m \cdot \ell)$ emerges.

### A MVCA based Greedy Algorithm

In the modified MVCA algorithm by [Krumke98] in every iteration step that label is added to the solution, which reduces the number of connected components most. However, the *MvcaMLST* mentioned above only considers the number of newly added edges without taking into account the number of newly connected nodes. Of course considering connected components in a directed graph makes little sense. Nevertheless the greedy strategy can be adapted insofar as the next label $r$ to be added to the solution can be chosen according to the number of edges, that after insertion of $r$ are still necessary for spanning a directed tree of size $k$. So let that number of edges in the following be denoted by $\epsilon$. The main idea of the *GreedyMLST* consists in treating the graph, which is induced by the set of reference vectors $R$, by means of its strongly connected components (*SCC*). If shrinking all nodes of a strongly connected component to a single node, and combining all parallel edges between two different components to a single edge, then by

ignoring all edges within one and the same component a DAG emerges. With the aid of that DAG $\epsilon$ can be determined, which in turn is defined by the number of strongly connected components minus one, the root node.

As long as no directed spanning tree of size $k$ exists, in the *GreedyMLST* algorithm that candidate reference vector $r_B$ is added to the solution, which reduces $\epsilon$ most. In order to find this $r_B$ each candidate reference vector $r$ from $R_C$ is temporarily added to the graph, and afterwards the strongly connected components are determined. Since the union-find data structure offers no possibility for determining the degree of a strongly connected component, $\epsilon$ must be computed by first finding all those components having incoming edges, and by afterwards subtracting their number from the number of all components.

**Algorithm 19** - *GreedyMLST*::*computeSolution*$(G, k)$

**Input**: $G = (V, A, l(A))$ – a complete, directed and edge-labeled graph with $l{:}A \to R_C$
$k$ – number of nodes to connect
**Output**: $T_L = (V, A_T, R)$ – a graph containing the directed spanning tree

```
1  A_T ← ∅, R ← ∅
2  while not (containsArborescence(G, k)) do
3     r_B ← ∅                                  // best reference vector found so far
4     ε ← ∞                                    // number of SCCs still to be connected
5     forall (r ∈ R_C) do
6        A(r) ← {a ∈ A\A_T | r ∈ l(a)}, A_T ← A_T ∪ A(r)
7        SCC ← computeSCC(T_L)  // SCC is a union find structure holding the SCCs
8        SCC+ ← ∅                              // set of SCCs with incoming edges
9        forall (a_ij ∈ A_T) do                // get all SCCs having incoming edges
10          if (find_set(v_i) ≠ find_set(v_j)) then SCC+ ← SCC+ ∪ {find_set(v_j)}
11       end forall
12       if ((|SCC| − |SCC+| − 1) < ε) then
13          ε ← |SCC| − |SCC+| − 1             // compute ε
14          r_B ← r
15       end if
16       A_T ← A_T\A(r)
17    end forall
18    A_T ← A_T ∪ {a ∈ A | r_B ∈ l(a)}
19    R ← R ∪ {r_B}, R_C ← R_C \ {r_B}
20 end while
21 return T_L
```

Again, *containsArborescence*() in line 2 takes time $O(n^3)$. Furthermore computing the strongly connected components in line 7 has runtime $O(n + m)$. Finding those components with incoming edges (lines 9 – 11) takes $O(m \cdot \alpha(n))$ due to the *find_set*() operations having runtime $O(\alpha(n))$, whereas $\alpha(n) \leq 4$ is a very slowly growing function, see [Cormen07]. Although a newly added reference vector may decrease the number of strongly connected components, it must not necessarily decrease the number of those components having indegree zero. Hence the while loop has runtime $O(m)$. In total, this leads to $O(m \cdot (n^3 + \ell \cdot ((n + m) + (m \cdot \alpha(n)))))$, and thus is $O(\ell \cdot m^2 \cdot \alpha(n))$.

In order to further improve the results a GRASP algorithm was implemented too, whereat this greedy algorithm serves as a basis for the respective construction phase.

### 6.3.5 A GRASP for the Directed Minimum Label Spanning Tree

The GRASP meta heuristic, shortly for *Greedy Randomized Adaptive Search Procedure*, is a multi-start algorithm in which every iteration consists of two parts, a *construction* and a *local search*. During the construction phase a valid solution is computed by means of a greedy randomized construction algorithm. The subsequent local search is for optimizing the solution with regard to a local optimum. The goal is to find a global optimum with the aid of different local optima [Resende02].

**Algorithm 20** - *GRASP*($I$)

**Input**: $I$ – maximal number of iterations
**Output**: $S$ – the solution

```
1 S ← ∅
2 forall (i = 1,...,I) do
3    S' ← greedyRandomizedConstruction()
4    S' ← localSearch(S')
5    if (S' is better than S) then S ← S'
6 end forall
7 return S
```

In order to create different initial solutions, it is important that the greedy construction heuristic randomizes its solution.

**Algorithm 21** - *greedyRandomizedConstruction*()

**Input**: –
**Output**: $S$ – a randomized initial solution

```
1 S ← ∅
2 while (S is not a valid solution) do
3    determine CL          // a candidate list of possible extensions for the solution
4    determine RCL ⊆ CL    // a restricted candidate list of promising extensions
5    chose e ∈ RCL         // a randomly chosen element from RCL
6    S ← S ∪ {e}
7 end while
8 return S
```

The *candidate list* (*CL*) contains all possible extensions for the current solution. The *restricted candidate list* (*RCL*) in turn holds only the promising extensions, like for instance in case of a minimum spanning tree problem a certain number of edges that have the smallest edge weights. From this set in every iteration one element is randomly chosen and added to the solution, wherefore the generated solution is neither locally nor globally optimal. For finding at least a locally optimal solution the local search is subsequently performed.

The efficiency of the local search strongly depends on some aspects: the initial solution, the choice of the neighborhood, how the neighborhood is browsed, etc. Usually simple neighborhoods are chosen [Resende02], like for instance a $k$-switch or an insertion neighborhood. But also variable neighborhoods are possible.

**Algorithm 22** - *localSearch*($S$)

**Input**: $S$ – an initial solution
**Output**: $S_L$ – a locally optimal solution

```
1 repeat
2    chose S ∈ N(S)                    // N(S) is the neighborhood of a solution S
3    if (S is better then S_L) then S_L ← S
4 until break condition
5 return S_L
```

For searching the neighborhood there are different step functions available. Some typical ones are for example:

- *Random Neighbor*: The next solution is randomly chosen from the neighborhood.
- *Next Improvement*: The neighborhood is processed in a strict order and the first neighboring solution, which is better then current one, is adopted.
- *Best Improvement*: The neighborhood is completely searched, and the best solution, as far as better as the current one, is adopted for the next local search step.

According to the global GRASP structure from above, the greedy randomized adaptive search procedure for the directed minimum label spanning tree problem is implemented as follows:

**Algorithm 23** - *GraspMLST*($V, k, I$)

**Input**: $V = \{v_1, \ldots, v_n\}$ – a set of points, $k, I$ – number of nodes to connect and iterations
**Output**: $T_L = (V_T, A_T, R)$ – a graph containing the directed spanning tree

```
1  G = (V, A) ← createCompleteDirectedGraph(V)
2  G = (V, A, l(A)) ← assignEdgeLabels(A)
3  while (i = 1, ..., I) do
4     G' = (V, A', R') ← GreedyRandomizedMLST::computeSolution(G, k)
5     G' = (V', A', R') ← removeRedundantEdgesAndNodes(G', k)
6     G' ← localSearch(G', G, k)
7     G' ← removeRedundantLabels(G', k)
8     if (|R'| < |R|) then T_L ← G'
9  end while
10 return T_L
```

As greedy randomized construction for the initial solution an adapted version of the *GreedyMLST* algorithm is applied. For the subsequent local search phase in turn two different variants were implemented and are going to be introduced below.

### Greedy Randomized Construction Heuristic

As already mentioned the *GreedyMLST* algorithm serves as basis for this construction heuristic. There are only some slight adaptations in order to obtain the restricted candidate list *RCL* from the set of all candidate reference vectors. In every iteration first of all $\epsilon$, i.e. the current number of strongly connected components having indegree zero minus one, is determined. Afterwards, as in the *GreedyMLST* algorithm, all candidate reference vectors are successively temporarily added to the solution, the number of strongly connected components having indegree zero minus one is determined, and if this number at

least conforms to $\epsilon$ or is even smaller, than the restricted candidate list is extended by the actual candidate reference vector. From this constructed *RCL* in every iteration eventually one candidate reference vector is randomly chosen and permanently added to the solution.

**Algorithm 24** - *GreedyRandomizedMLST*::*computeSolution*$(G, k)$

**Input**: $G = (V, A, l(A))$ – a complete, directed and edge-labeled graph with $l{:}A \to R_C$
$k$ – number of nodes to connect
**Output**: $T_L = (V, A_T, R)$ – a graph containing the directed spanning tree

```
1  A_T ← ∅, R ← ∅
2  while not (containsArborescence(T_L, k) do
3     RCL ← ∅
4     ε ← the current number of SCCs with indegree zero, minus 1
5     forall (r ∈ R_C) do
6        A(r) ← {a ∈ A\A_T | r ∈ l(a)}, A_T ← A_T ∪ A(r)
7        SCC ← computeSCC(T_L), SCC⁺ ← ∅
8        forall (a_{i,j} ∈ A_T) do
9           if (find_set(v_i) ≠ find_set(v_j)) then SCC⁺ ← SCC⁺ ∪ {find_set(v_j)}
10       end forall
11       if ((|SCC| − |SCC⁺| − 1) ≤ ε) then RCL ← RCL ∪ {r}
12       A_T ← A_T\A(r)
13    end forall
14    r ← chose randomly from RCL
15    A_T ← A_T ∪ {a ∈ A | r ∈ l(a)}
16    R ← R ∪ {r}, R_C ← R_C \ {r}
17 end while
18 return T_L
```

Since every reference vector, resulting in a number of strongly connected components having indegree zero that is smaller than or equal to $\epsilon$, is added to the restricted candidate list the constructed solution is not necessarily a good one. However, if additionally restricting the *RCL* to a constant size, like for instance to five candidate reference vectors, the algorithm delivers fast results. Nevertheless the worst case runtime conforms to the *GreedyMLST* algorithm with $O(\ell \cdot m^2 \cdot \alpha(n))$.

### First Local Search Method

As already mentioned above for the local search two different variants were implemented. The first one uses the $k$ nodes returned from the construction phase as a neighborhood in a broader sense.

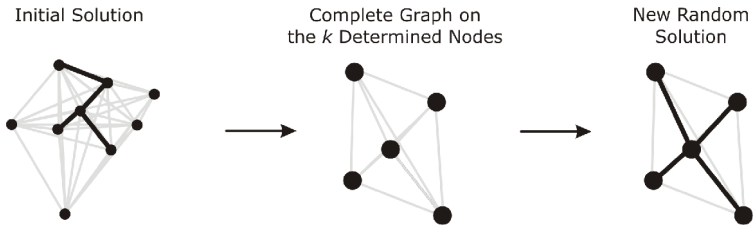



*Figure 23 - Local search 0: On the computed set of nodes a complete and directed graph is created, and by once again applying the randomized greedy construction a new solution is created.*

In particular, the $k$ nodes are used for creating again a complete and directed graph, on which afterwards the greedy randomized construction algorithm is iteratively applied.

**Algorithm 25** - *localSearch0*$(T_L, G, k, I_L)$

**Input**: $T_L = (V_T, A_T, R)$ – an initial solution, $G = (V, A, l(A))$ – the original graph, $k$ – number of nodes to connect, $I_L$ – number of iterations for this local search
**Output**: $T_B = (V_T, A_B, R_B)$ – the best found subgraph of $G$

```
1 T_B ← T_L
2 G' = (V_T, A')← createCompleteDirectedGraph(V_T)
3 ∀a' ∈ A' := l(a') ← l(a)
4 for (i = 1, ..., I_L) do
5    T'_L = (V_T, A'_T, R')← GreedyRandomizedMLST::computeSolution(G', k)
6    if (|R'| < |R_B|) then T_B ← T'_L
7 end for
8 T_B ← removeRedundantEdgesAndNodes(T_B, k)
9 return T_B
```

In order to avoid a complete recomputation of the difference and candidate reference vectors, the original graph $G$ is delivered to the local search, too. Thus the edges of the new created graph $G'$ obtain those reference vectors which are assigned to the edges of the original graph. In the subsequent loop thus $G'$ is passed to the construction heuristic, which returns a new random minimum label spanning tree $T'_L$. If its number of reference vectors is smaller than those of the currently best known tree $T_B$, then $T'_L$ is memorized as best tree. According to the repetitive calls of the *GreedyRandomizedMLST* algorithm the runtime of this local search is $O(I_L \cdot \ell \cdot m^2 \cdot \alpha(n))$, whereat $I_L$ specifies the number of local search iterations. However due to randomly constructing a new tree on only a subset of the nodes this algorithm is not very efficient.

### Second Local Search Method

The second local search algorithm uses a reference vector insertion neighborhood. Therefore, successively every candidate reference vector is temporarily added to the solution, and thereupon redundant reference vectors are removed.

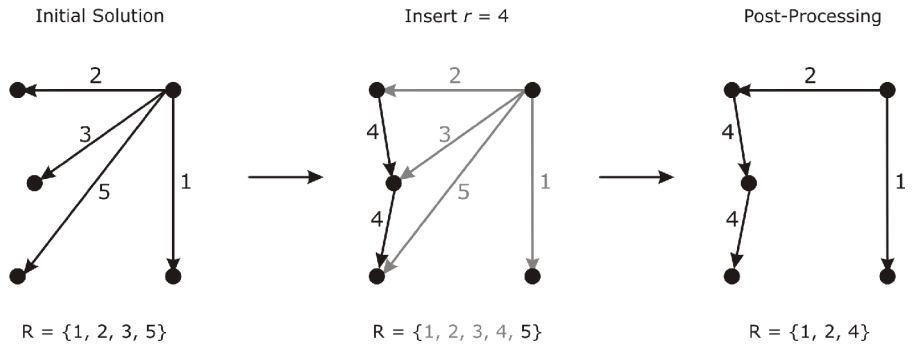



*Figure 24 - Local search 1: The initial solution is extended with a new candidate reference vector. Subsequently all redundant reference vectors are removed.*

The goal is to find a candidate reference vector $r \in R_C$ that dominates at least two reference vectors of the current solution, and thus makes them redundant. Since $r$ most likely will not cover more than two of the existing reference vectors, a next improvement step

function is used here. Thus, on the other hand also the runtime will not be unnecessarily heightened. In order that the new reference vector covers a preferably large number of the current edges or inserts lots of new edges, respectively, the remaining candidate reference vectors $R_R$ are in the beginning sorted according to the number of edges they are covering. Thus only a promising subset, namely $|R_R|/c$ of the remaining candidate reference vectors whereas $c$ is a variable number, can be examined. This is based on the fact that those reference vectors that cover only one or two of the edges will not be able to dominate other reference vectors. Then within the for loop every considered candidate reference vector $r_i$ is added to the solution and afterwards the redundant reference vectors are removed. If the number of remaining reference vectors is smaller than the number of vectors in $R_B$ then the new created minimum label spanning tree $T'$ is adopted as new best solution $T_B$, $r_i$ is removed from $R_R$, and a new local search step is started.

**Algorithm 26** - *localSearch1*$(T_L, G, k)$

**Input**: $T_L = (V_T, A_T, R)$ – an initial solution, $G = (V, A, l(A))$ – the original graph, $k$ – number of nodes to connect
**Output**: $T_B = (V_T, A_B, R_B)$ – the best found subgraph of $G$

```
 1  T_B ← T_L
 2  R_R ← R_C \ R                          // remaining candidate reference vectors
 3  sort R_R decreasingly according to the number of edges they represent, i.e. r_1 ≥ … ≥ r_ℓ
 4  for (i = 1, …, |R_R|/c) do             // insert r_i and perform post-processing
 5     G' ← (V, A', R') with R' ← R_B ∪ {r_i} and A' ← {a ∈ A | ∃r ∈ R': r ∈ l(a)}
 6     T' = (V, A', R') ← removeRedundantLabels(T', k)
 7     if (|R'| < |R|) then
 8        T_B ← T'
 9        R_R ← R_R \ {r_i}
10        i ← 0
11     end if
12  end for
13  T_B ← removeRedundantEdgesAndNodes(T_B, k)
14  return T_B
```

Since creating the set of edges $A'$ for the graph $G'$ in line 5 takes time $O(m \cdot l)$ and removing redundant labels in line 6 takes $O(m^2)$ according to algorithm (16), the total runtime for this local search is $O(I_L \cdot \ell \cdot m^2)$, whereat $I_L$ is the number of iterations within the local search.

Finally, in order to solve the directed MLST problem different algorithms were implemented: the MVCA based greedy construction heuristics *MvcaMLST* and *GreedyMLST* and a GRASP, which in turn is based on the *GreedyMLST*. For the GRASP two different local searches were implemented. Beneath of them the second variant is the more promising one with respect to the results and the worst case runtime. But how the runtime indeed affects the quality of the solutions will be shown in chapter 8. Before that however, the directed weight balanced spanning tree algorithms are going to be introduced.

## 6.4 Directed Weight Balanced Spanning Tree Algorithms

For a one-dimensional cost function the WBST algorithm presented in section 5.5 always finds the optimum spanning tree, i.e. that the difference between all edge weights is minimal. In order to obtain possibly small delta values for the template compression problem, it makes sense that the weighting function describes the similarity of the difference vectors with respect to their single coordinate values. In general the difference vector itself can be treated as a $d$-dimensional cost function. Hence, the classical WBST algorithm would find the optimal spanning tree for a single dimension but not for all dimensions together. In order to solve this problem, three different approaches were analyzed. In the first algorithm, the *SimpleWBST*, the edge weights are computed analogously to the MST using distance functions and similarity measures, respectively. In particular the maximum metric, the Minkowski metric with $m = 2$ and $m = d$, and furthermore the pseudo-cosine measure are anylzed. By transforming the $d$-dimensional difference vectors to one-dimensional edge weights, a weight balanced spanning tree can be computed, which is optimal with respect to the considered edge weights. In the second approach in turn, the *MultipleWBST*, the optimal weighting range for every single dimension is computed, which results in an optimal bounding box. Subsequently a tree is constructed, in which the edge weights are located within this bounding box or do only differ slightly from it. Finally the third algorithm for solving the WBST problem is a GRASP. Once again in all cases the main structure of the algorithms is similar.

**Algorithm 27** - *DirectedWBST*$(V, k)$

**Input**: $V = \{v_1, \ldots, v_n\}$ – a set of data points, $k$ – number of nodes to connect
**Output**: $T_W = (V_T, A_T, r_O)$ – a weight balanced spanning tree

1 $G = (V, A) \leftarrow$ *createCompleteDirectedGraph*$(V)$
2 $T_W \leftarrow$ *computeSolution*$(G, k)$
3 $T_W \leftarrow$ *removeRedundantEdgesAndNodes*$(T_W, k)$
4 $r_O \leftarrow$ the minimal values of every edge dimension
5 $A_T \leftarrow$ subtract offset $r_O$ from all edges
6 **return** $T_W$

After creating a complete and directed graph a solution is computed and all redundant edges and nodes are finally removed. Conclusively the offset vector $r_O$ is determined and subtracted from all edges.

### 6.4.1 A Directed WBST with one-dimensional Edge Costs

The *SimpleWBST* algorithm is a straight implementation of the optimal algorithm presented in section 5.5.1. Therefore all edges obtain a one-dimensional weight, which is either computed by a distance function or a similarity measure. However instead of checking the connectedness, in this version a directed spanning tree of size $k$ is sought on the graph. In the algorithm below, after assigning the weights the edges are sorted according to their weights in an increasing way. Subsequently the optimal weighting range is determined and the respective edges are added to the resulting graph $T_W$.

**Algorithm 28** - *SimpleWBST*::*computeSolution*$(G, k)$

**Input**: $G = (V, A)$ – a complete, directed graph, $k$ – number of nodes to connect
**Output**: $T_W = (V, A_T)$ – a graph containing the weight balanced spanning tree

```
1  G = (V,A,w(A)) ← assignEdgeWeights(A)
2  sort the edges increasingly according to their edge weights, i.e. w(a_1) ≤ ... ≤ w(a_m)
3  diff ← w(a_m) − w(a_1)
4  low ← 1, high ← 1
5  low_B ← 1, high_B ← m
6  repeat
7     G' ← (V,A') with A' ← {a_i ∈ A | low ≤ i ≤ high}
8     if (containsArborescence(G',k) then
9        if (diff > w(a_high) − w(a_low)) then
10          diff ← w(a_high) − w(a_low)
11          high_B ← high
12          low_B ← low
13       end if
14       low ← low + 1
15    else high ← high + 1
16 until (high = m ∧ low = m)
17 A_T ← {a_i ∈ A | low_B ≤ i ≤ high_B}
18 return T_W
```

Since looking for an arborescence in every iteration and successively increasing the variables *high* and *low* the runtime of the *SimpleWBST* algorithm conforms to $O(m \cdot n^3)$.

### 6.4.2 A Directed WBST with *d*-dimensional Edge Costs

Instead of finding an eligible one-dimensional edge weight representation for the $d$-dimensional difference vectors of the edges, the *MultipleWBST* algorithm in a first step determines the optimal weighting range $(low_B(i), high_B(i))$ for every single dimension $i$.

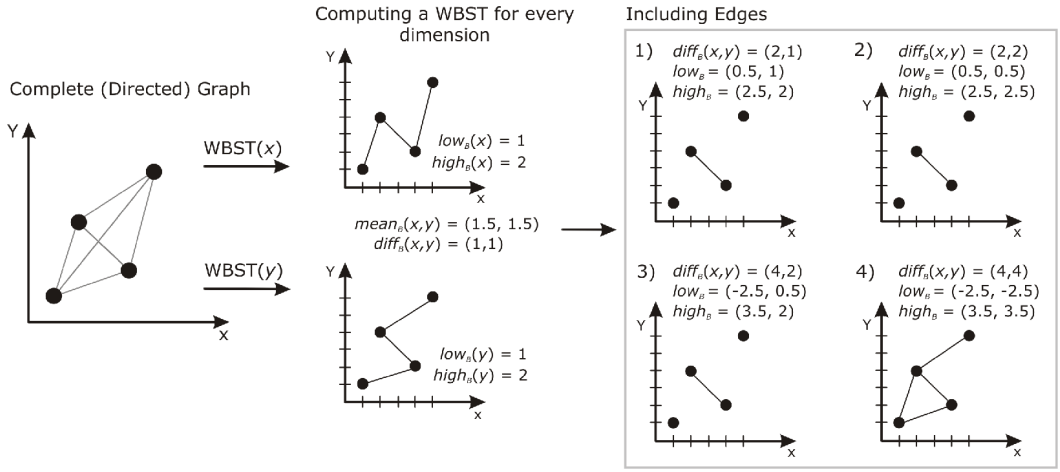



*Figure 25 - Schematic representation of the MultipleWBST algorithm: For a complete (directed) graph in a first step the optimal weighting ranges ($low_B(i)$, $high_B(i)$) for every dimension i are computed. Note that $diff_B$ at this is the difference $high_B - low_B$, which should always adopt values that are a power of two. These ranges in turn constitute two vectors, which span a bounding box that in an ideal case contains all edges for spanning a tree of size k. Since in most cases this number of edges will not be sufficient, the bounding box is stepwise expanded around its center point $mean_B$ until such a spanning tree exists.*

Together these ranges constitute two vectors, $low_B$ and $high_B$, which in turn span a bounding box. In an ideal case a solution of the *MultipleWBST* thus only consists of those edges that are located within that bounding box. However, since probably in most cases this set of edges will not be enough for spanning a tree of size $k$, in the *MultipleWBST* algorithm this bounding box is successively expanded until such a spanning tree exists. Therefore, the greatest difficulty consists in finding an appropriate update strategy for the bounding box, because this in turn should not become too large. This first idea of an update strategy increases the range values of the bounding box, i.e. the differences $high_B - low_B$, in every dimension to powers of two.

**Algorithm 29** - *MultipleWBST*::*computeSolution*$(G, k)$

**Input**: $G = (V, A)$ – a complete and directed graph, $k$ – number of nodes to connect
**Output**: $T_W = (V, A_T)$ – a graph containing the optimal weight balanced spanning tree

```
 1 for (i = 1, ..., d) do
 2    T^i = (V^i, A^i) ← SimpleWBST::computeSolution(G^i, k) // opt. WBST for dimension i
 3    low(i)_B, high(i)_B ← lowest and highest values of A_Δ^i
 4    diff(i)_B ← high(i)_B − low(i)_B
 5    mean(i)_B ← low(i)_B + ⌈(diff(i)_B) / 2⌉          // center of BB(low(i)_B, high(i)_B)
 6    low(i)_B ← mean(i)_B − (2^⌈ld(diff(i)_B)⌉) / 2     // expand BB range to a power of 2
 7    high(i)_B ← mean(i)_B + (2^⌈ld(diff(i)_B)⌉) / 2
 8 end for
 9 A_T ← {a ∈ A | a ∈ BB(low_B, high_B)}
10 while not (containsArborescence(T_W, k)) do
11    i ← dimension having the smallest value of diff_B
12    diff(i)_B ← 2^(⌈ld(diff(i)_B)⌉ + 1)               // update diff(i)_B to a next power of 2
13    low(i)_B ← mean(i)_B − diff(i)_B / 2              // update bounding box
14    high(i)_B ← mean(i)_B + diff(i)_B) / 2
15    A_T ← {a ∈ A | a ∈ BB(low_B, high_B)}
16 end while
17 return T_W
```

For determining the weighting ranges for the single dimensions the *MultipleWBST* algorithm uses the *SimpleWBST*. Therefore before computing the optimal WBST for dimension $i$, the *SimpleWBST* algorithm below has to assign as edge weights the respective coordinate values of the difference vectors at the considered dimension, i.e. $\Delta a^i$. The $mean_B$ vector in this algorithm represents the center point of the bounding box. Around this point the bounding is expanded within the while loop until eventually a spanning tree of size $k$ emerges. For expansion always that dimension $i$ is chosen, in which the range $high(i)_B - low(i)_B$ of the bounding box is minimal. Thus, the bounding box effectively is evenly expanded to all dimensions. Computing the WBST for every dimension takes time $O(d \cdot m \cdot n^3)$ according to the *SimpleWBST* algorithm. As long as the bounding box is updates less then $d \cdot m$ times, see lines 10 – 15, $O(d \cdot m \cdot n^3)$ is also the total worst case runtime for the *MultipleWBST* algorithm, due to $O(n^3)$ is the runtime for the *containsArborescence*() method.

Further update strategies, that for instance expand one dimension right up to the global domain border $\tilde{v}$ before considering the next one, are possible as well, however will not be part of this thesis anymore.

### 6.4.3 A GRASP for the Directed Weight Balanced Spanning Tree

Since the *SimpleWBST* and the *MultipleWBST* both not necessarily deliver an optimal solution except for the one-dimensional case, a conclusive GRASP algorithm was implemented. Therein, in order to compare the quality of the single solutions, the total encoding length according to formula (19), see section 5.5.2, is considered.

**Algorithm 30** - *GraspWBST*::*computeSolution*$(G, k)$

**Input**: $G = (V, A)$ – a complete, directed graph, $k$ – number of nodes to connect
**Output**: $T_W = (V_T, A_T, r_O\}$ – a weight balanced spanning tree

```
 1 T_W ← ∅, λ ← ∞                                  // λ is the total encoding length
 2 repeat
 3    G' = (V, A', r_O') ← GreedyRandomizedWBST::computeSolution(G, k)
 4    G' ← localSearch(G, G', k)
 5    if (λ < λ(G')) then
 6       T_W ← G'
 7       λ ← λ(G')
 8    end if
 9 until break condition
10 return T_W
```

Furthermore the greedy randomized construction heuristic uses the *SimpleWBST*. Indeed the *SimpleWBST* is no greedy algorithm, however it returns the optimal range for the edge weights with respect to the considered weighting function. This range is used in turn for determining the restricted candidate list, which emerges by expanding the weighting range with $2k$ values into both directions and including the respective edges. The value $2k$ is chosen, in order to make sure that the *RCL* always contains a valid solution, which however is already guaranteed by the computed range, and furthermore a more or less random solution can be constructed. Afterwards always one edge from the *RCL* is randomly chosen and inserted into the graph $T_W$ until eventually a spanning tree of size $k$ emerges. As a conclusive step a post-processing is performed in order to remove all redundant edges and nodes.

**Algorithm 31** - *GreedyRandomizedWBST*::*computeSolution*$(G, k)$

**Input**: $G = (V, A)$ – a complete, directed graph, $k$ – number of nodes to connect
**Output**: $T_W = (V_T, A_T, r_O)$ – a weight balanced spanning tree

```
 1 G' = (V', A', r) ← SimpleWBST::computeSolution(G, k)
 2 low_B, high_B ← lowest and highest values of A_Δ'
 3 if ((high_B + 2k) < m) then high_B ← high_B + 2k else high_B ← m // only valid values,
 4 if ((low_B – 2k) > 0) then low_B ← low_B – 2k else low_B ← 0        // i.e. 0 ≤ … ≤ m
 5 RCL ← {a_i ∈ A | low_B ≤ i ≤ high_B}
 6 T_W ← (V, A_T) with A_T ← ∅
 7 while not (containsArborescence(T_W, k)) do
 8    a ← randomly chosen from RCL
 9    A_T ← A_T ∪ {a}, RCL ← RCL \ {a}
10 end while
11 T_W ← removeRedundantEdgesAndNodes(T_W, k)
12 return T_W
```

Since the *SimpleWBST* algorithm in line 1 has runtime $O(m \cdot n^3)$, *containsArborescence*() in line 7 takes $O(n^3)$ and the while loop in the worst case is repeated for all edges $m$, the total runtime for the *GreedyRandomizedWBST* conforms to $O(m \cdot n^3)$.

The respective local search algorithm for the *GraspWBST* uses a 2-switch neighborhood, in which always one edge of the solution is replaced.

**Algorithm 32** - *GraspWBST*::*localSearch*$(G, T_W, k)$

**Input**: $G = (V, A, l(A))$ – the original graph, $T_W = (V_T, A_T, r_O)$ – an initial solution, $k$ – number of nodes to connect
**Output**: $T_B = (V_B, A_B, R_B)$ – the best found weight balanced spanning tree

```
 1 T_W ← (V, A_T)
 2 repeat
 3    a^i_W ← edge conforming to max( ã^1 , …, ã^d ), whereas ã is the current edge domain
 4            and i the respective dimension
 5    A_T ← A_T \ {a_W}, A_T' ← A \ A_T
 6    forall (a ∈ A_T') do
 7       if (a^i < a^i_W ) then          // checking the coordinate value at dimension i
 8          pot_b ← true                 // indicates whether a is potentially better than a_W
 9          for (j = 1, …, d) do
10             if (a^j > ã^j ) then pot_b ← false
11          end for
12          if (pot_b = true) then
13             A_T ← A_T ∪ {a}           // add a to A_T and look for directed spanning tree
14             if (containsArborescence(T_W, k)) then a_W ← a
15             A_T ← A_T \ {a}
16          end if
17       end if
18    end forall
19    A_T ← A_T ∪ {a_W}
20 until no better edge was found
21 T_B ← removeRedundantEdgesAndNodes(T_B, k)
22 return T_B
```

First of all the dimension $i$ of the current edge domain $\tilde{a}$ with the highest coordinate value is sought, and afterwards the respective edge $a_W$ is determined. This is the edge to be replaced, hence this edge is removed and a new edge $a$ with a smaller coordinate value at dimension $i$ is looked for. If this new edge is potentially better than $a_W$, which means that the values of $a$ in all other dimensions do not exceed the values of $\tilde{a}$ , it is added to the solution. Hence if this new set of edges again spans a tree of size $k$, then a better edge and thus a better solution was found. The best of all these better edges, i.e. that with the smallest value at dimension $i$, is finally added permanently to the solution, and the next local search step is started. Thus with respect to the number of search iterations, the number of edges and the *containsArborescence*() calls in line 14, a runtime for the local search algorithm below of $O(I_L \cdot m \cdot n^3)$ emerges.

All the MST, MLST and WBST algorithms presented within this chapter were implemented together with a framework, which is going to be described in the following chapter.

# Chapter 7 - Implementation

*"In general, an implementation should be conservative in its sending behavior, and liberal in its receiving behavior."*

Jonathan Postel

For implementing the algorithms of the template compression problem, in the course of this thesis a framework was developed, which provides functions for reading and writing templates, results and reference vectors, for compression and of course for the computation of the respective spanning trees. This *Biometric Template Compression* (BTC) framework is going to be presented on the following pages in more detail. Beside of information to the applied programming language and used libraries, also an overview about the packages, classes and important functions will be given. Furthermore the main program and its usage will be introduced.

## 7.1 Programming Language and Frameworks

The main requirement when choosing the programming language was object orientation and reusability. In order to furthermore keep the amount of programming within a limit, also already existing frameworks should be integrable. Since for the template compression problem also genetic algorithms were developed, which are based on the *EAlib2* library [Wagner05] for meta heuristics, developed at the Institute for Computer Graphics and Algorithms of the Vienna University of Technology, eventually C++ using the gcc-4.1 compiler was chosen. Additionally for dealing with graphs and classical graph algorithms the *LEDA* library version 5.1 [Leda] was integrated, which was developed since the end of the 80s by the Max Planck Institute for Computer Science, and which is distributed by the Algorithmic Solutions Software GmbH since 2001. Beside of graph algorithms therein also classical data structures and algorithms for network and optimization problems are provided. Furthermore the CPLEX 11 library for integer and linear programming was applied.

## 7.2 Biometric Template Compression Framework

The *Biometric Template Compression* Framework consists of five packages: *io*, *graph*, *algorithms*, *compression* and *tools*.

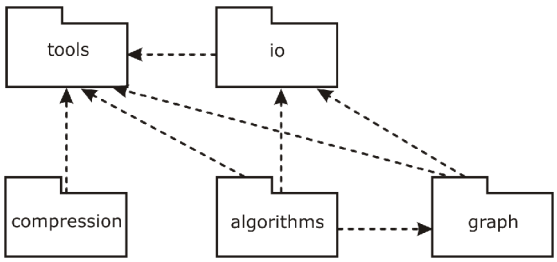



*Figure 26 - Package overview: The dotted arrows depict the usage dependency relation for the single packages.*

As its name implies the *io* package is responsible for reading and writing data from and to the file system, like for instance templates or stored reference vectors. The *graph* and

*algorithm* packages provide interfaces and implementations for solving the different spanning tree problems presented in chapter 6. Transforming the computed spanning trees into the respective bit string representations is done with the aid of the *compression* package. Finally the *tools* package provides some globally useful functions, like for instance converting numbers into bit strings, mathematical conversions, or cost function computations.

### 7.2.1 Package *io*

The classes of the *io* package provide methods for reading and writing files from and to the file system.

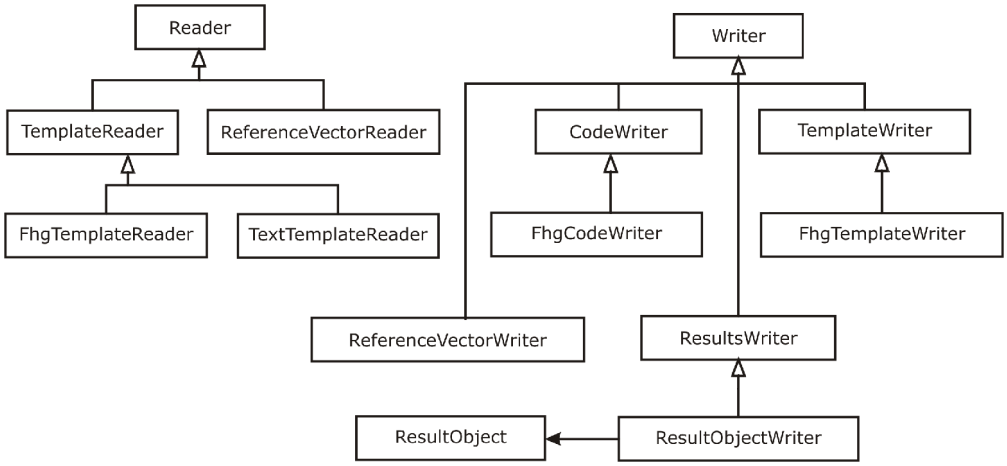



*Figure 27 - Class overview for the io package: The arrows ending with a hollow triangle depict the generalization relation and the arrow with the solid triangle depicts the usage dependency.*

Thereto two abstract class types are available: *Reader* and *Writer*. The *Reader* classes are responsible for reading files. All implementing classes have to pass a filename to the *Reader* class, and have to provide the method *readAll*() which reads in the data from the specified file. Currently the following subclasses are available:

- *TemplateReader* is the main abstract class for reading minutiae templates. In order to read in templates of a special type, this class has to be derived and the methods *getNoOfPoints*() and *getPoints*() are to be provided, which return the number of points and the data points themselves, respectively. Current implementations are the *FhgTemplateReader*, which reads in minutiae templates encoded in the Fraunhofer template format (see chapter 8), and the *TextTemplateReader* for getting comma separated minutiae data.
- The *ReferenceVectorReader* class is for reading reference vectors, which have been saved into a file. This class is used by the MLST algorithms.

The *Writer* classes in turn provide functions for writing data into files. Analogously to the *Reader*, all implementing classes have to pass a filename to the *Writer* class, and furthermore have to provide the method *writeAll*(), which writes a passed string into the previously specified file. Here the following subclasses are available at the moment:

- The *ReferenceVectorWriter* class writes those reference vectors that have been computed by the MLST algorithms into a file, such that other MLST algorithms can use them at a future date.

- *CodeWriter* is an abstract class for writing the compressed templates into a file. Currently the only implementation is the *FhgCodeWriter* that takes the computed tree and an *Encoder* class (see compression package, section 7.2.4) as input, encodes the tree with the aid of the encoder and writes the resulting bit string extended with the meta data from the original template into a file.
- The *TemplateWriter* class in turn is for encoding and writing the minutia data in a specific format, like e.g. the Fraunhofer minutiae template format. This may be used for instance if having computed a *k*-node spanning tree, and thus to check whether the matching results of the reduced minutiae template still conform to those of the original one. Thus, the only implementation for the moment is the *FhgTemplateWriter*.
- The *ResultWriter* is responsible for writing important results of the spanning tree algorithms into a file. These results are to be passed as a string. The implementing class *ResultObjectWriter* therefore creates a string from a passed *ResultObject*, transfers it to the *ResultWriter* and thus the result is written into a file.

Finally the *ResultObject* class is created within all spanning tree algorithms and contains all important approach specific results. It is mainly used in order to make the results from the different algorithms comparable.

### 7.2.2 Package *graph*

The *graph* package provides data structures for dealing with minutiae and miscellaneous data points, which are representable in the form of vectors, such as difference and reference vectors. Furthermore main interfaces for the realization of the different spanning tree algorithms are contained within this package.

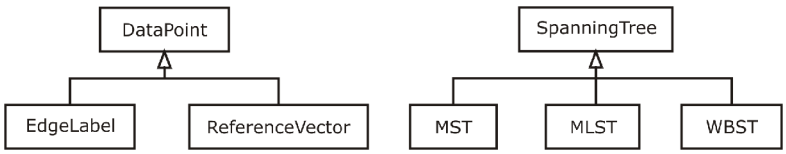



*Figure 28 - Class overview for the graph package*

For dealing with data points and vectors respectively, the classes *DataPoint, EdgeLabel* and *ReferenceVector* are used.

- *DataPoint* is the basic class for representing points and minutiae, which both in turn are nothing else than vectors. Within this class the single coordinate values are contained, and furthermore functions for modifying vectors, such as adding, subtracting, comparing with respect to the finite ring structure, etc. are provided. In addition there are functions available for checking whether these vectors are located within a bounding box, and whether vectors are covered or dominated by other ones, respectively.
- *EdgeLabel* is a special implementation of the *DataPoint* class. In general it conforms to the definition of a difference vector $\Delta a_{s,t}$. Hence, an *EdgeLabel* has a source node $v_s$ and a target node $v_t$, whereas the coordinate values for the single dimensions $i$ conform to the distance values $v_t^i - v_s^i$. Furthermore an edge weight can be assigned and also a set of reference vectors, that are able to represent this specific difference vector or edge, respectively.

- *ReferenceVector* is another special implementation of the *DataPoint* class, which how its name implies corresponds to the definition of a reference vector. Thus every specific reference vector stores the set of difference vectors it is able to represent.

As already mention this package contains furthermore some interfaces for realizing the spanning tree algorithms presented in chapter 6.

- *SpanningTree* is the main representation of a spanning tree and thus provides some major functions for the specific implementations. Thereto belong the global pre- and post-processing functions for creating a complete and directed graph from a passed set of nodes, and for removing redundant edges and nodes. Furthermore the function *containsArborescence*() is contained in here. All implementing classes thus, have at least to provide the *computeSolution*() method.
- *MST*, *MLST* and *WBST* are the interfaces to the actual implementations of the introduced algorithms. Therein mainly the specific pre- and post-processing functions of the different approaches are to be found.

Finally the concrete implementations of the presented algorithms are located within the *algorithms* package.

### 7.2.3 Package *algorithms*

The *algorithms* package contains all the implemented algorithms which were introduced in the previous chapter.

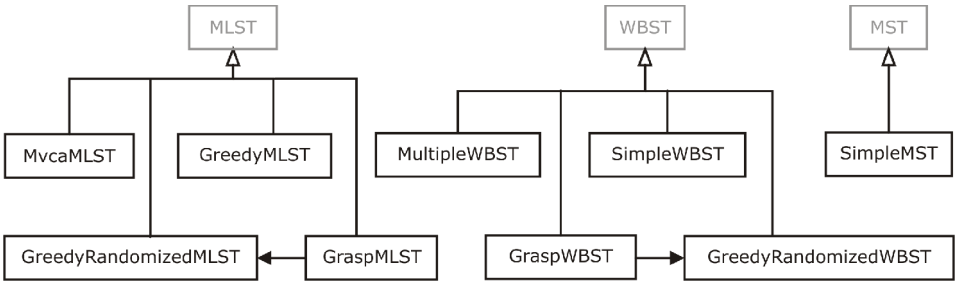



*Figure 29 - Class overview for the algorithms package: Note that the classes MST, MLST and WBST, depicted in a gray color, belong to the graph package, and that they are illustrated here in order to show the hierarchy of inheritance.*

Since the algorithms, whose names conform to the class names depicted ahead, were already presented in detail in chapter 6, no further description is given here. Only note, that all these classes provide the method *computeSolution*() which has to be called after class initialization in order to obtain the respective resulting spanning tree.

### 7.2.4 Package *compression*

In order to encode the computed spanning trees, i.e. transforming them into the respective bit string representation presented in chapter 5, mainly the *compression* package is responsible.

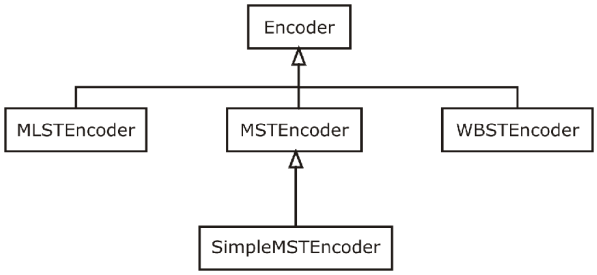



*Figure 30 - Class overview for the compression package*

Here the abstract class *Encoder* is the main compression object, which contains the resulting bit string, the spanning tree and the assigned root node. Hence, the implementing classes have at least to provide the method *encode*() which eventually creates a bit string from the passed tree. The current implementations *MSTEncoder* with its subclass for the *SimpleMST* results, *MLSTEncoder* as well as *WBSTEncoder* are rough realizations for the compression of the respective spanning trees, which in general conform to the objective functions presented in chapter 5. However they are not well tested yet, and thus are to be taken with a pinch of salt.

### 7.2.5 Package *tools*

Within the *tools* package on the one hand some globally useful functions and classes are to be found, and on the other hand the different cost function implementations are located in here.

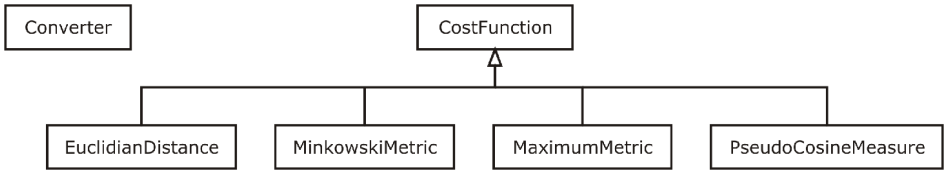



*Figure 31 - Class overview for the tools package*

The *Converter* class contains functions for converting integer values into bit strings of a certain length and vice versa, furthermore for computing the logarithm dualis for (non-) negative integer values, for translating bit strings into byte arrays, and also for getting integer vectors from a comma separated list of numbers.

In turn the abstract class *CostFunction* represents an interface for those algorithms that need a cost or weighting function on their edges. Here the implementing classes at least have to provide the method *computeCosts*() that should return the costs for either a passed difference vector or two passed points, respectively. Current subclasses are the *PseudoCosineMeasure*, *MaximumMetric*, *EuclidianDistance* and *MinkowskiMetric*. Note that the latter one returns the Minkowski metric for $m = d$.

## 7.3 Main Program

In order to run one of the presented spanning tree algorithms, the main program has to be invoked from the console using the *FPC* command. Thereto some parameters exist, that are going to be introduced now.

*f*: The name of the template file to be processed.

*a*: The algorithm to be called. Currently the following algorithms are available: *SimpleMST*, *MvcaMLST*, *GreedyMLST*, *GraspMLST*, *SimpleWBST*, *MultipleWBST* and *GraspWBST*.

*d*: The delta values for the MLST algorithms. For passing these values, they have to be written in a comma separated style without any blanks, e.g. *-d* 10,10.

*n*: The number of dimension to be considered during computation, i.e. $n \in \{1, \ldots, d\}$ whereas $d$ is the number of all dimensions. This parameter is only interesting for the MST and WBST algorithms, since in the MLST case the number is implicitly given by the number of passed delta values.

*t*: The template reader to be used. Hereby "fhg" stands for Fraunhofer templates and "txt" for simple comma separated template files. Note that if no template reader is set, by default the Fraunhofer template reader will be used.

*c*: The cost function for the MST and WBST algorithms. The Euclidian distance is selected by "ed", the Minkowski metric with $m = d$ by "mm", the maximum metric by "max", and finally the pseudo-cosine measure by "pcm". Also in this case a default value is set, namely the Euclidian distance.

*v*: Specifies the verbose mode. At this a value $v \in \{0, \ldots, 3\}$ can be set. Thereby 0 stands for no output, 1 for some important messages, 2 for more detailed and 3 for all possible output messages.

*k*: The number of nodes to connect. If no value is passed, all nodes will be considered.

*i*: The number of search iterations for the GRASP algorithms. Like with the delta values, a comma separated list of integer values is expected, e.g. *-i* 100,100,1. The first number specifies the number of GRASP iterations and the second one the number of local search iterations. Since there are two local search types for the *GraspMLST* algorithm available the third number is only for the MLST case and defines the local search type, whereas the number conforms to the names presented in chapter 6. The default values in this case are 5,100,1.

*e*: The domain size of the raw data. This is mainly used for computing the total encoding length according to the formulas presented in chapter 5, but also finds use in the actual encoding algorithms within the compression package. Also here the values are to be passed in the form of a comma separated list. If this parameter is not specified, the Fraunhofer domain size with 9,9,9,1 (see chapter 8) is set by default.

Thus typical calls for the different algorithms could look as follows:

- MST: FPC *-f* <file> *-a SimpleMST -n* 2 *-t* fhg *-c* max *-v* 0 *-k* 15
- MLST: FPC *-f* <file> *-a GreedyMLST -d* 20,20 *-t* txt *-v* 2 *-k* 20 *-e* 12,12,9,1
  FPC *-f* <file> *-a GraspMLST -d* 15,15,15 *-t* fhg *-v* 1 *-i* 100,50,1
- WBST: FPC *-f* <file> *-a MultipleWBST -n* 3 *-t* fhg *-c* mm *-v* 1 *-k* 30

Finally the framework with its classes, packages and usage was described. Thus, the test data and results are going to be presented conclusively.

# Chapter 8 - Results

*"The gratification comes in the doing, not in the results."*
James Dean

The previously presented approaches and algorithms were all tested on a small set of minutiae that was provided by the Fraunhofer Institute for Production Systems and Design Technology. All tests were run on a machine with a 1,83 gigahertz Intel Core 2 Duo processor, having 3 gigabyte of RAM and 2 megabyte of L2 cache. The respective test data and results are going to be presented in detail on the next pages. In order to make all introduced approaches comparable to one another, the evaluation will focus on the compression ratio, which is computed as follows:

$$\textit{compression ratio}\ [\text{in } \%] = 100 - \frac{100 \cdot \lambda_{raw}}{\lambda} \tag{20}$$

According to formula (8) $\lambda_{raw}$ specifies the size of the raw data. In turn $\lambda$ conforms to the objective function depending on the applied approach. A positive compression result will state the value by which the original data could be reduced, whereas a negative ratio specifies that value by which the data actually were increased.

Additionally, in case of the minimum label spanning tree algorithms also runtime and the computed number of reference vectors will be examined. But first the provided test data are going to be introduced.

## 8.1 Fraunhofer Test Data

The Fraunhofer test data set consists of 20 templates belonging to four fingers. Thus, for every finger there are five impressions. Each of them contains between 15 and 40 minutiae. The main template structure is the following:

- 48 byte of MetaData. These are template specific information, such as resolution, number of minutiae, etc.
- 4 byte of ConstData. An offset for the *x* and *y* coordinates.
- 4 byte per minutia. Every minutia consists of four dimensions, see section 3.2.3, whereby the *x* and *y* coordinates both are encoded with 10 bits, the angle with 9 bits and the type with 3 bits.

A more detailed analysis of these templates revealed that the minutiae resolution is not completely exhausted, which means that actually only 9 bits for the *x* and *y* coordinates and 1 bit for the type are used. For comparing the test results this "real" encoding size will be used as raw data $\lambda_{raw}$ and no MetaData is going to be considered. Furthermore the ConstData of the original templates will be neglected as well. Hence, if a template contains 15 minutiae, only these minutiae each with 28 bits resolution will be treated as original encoding size, in order to not obtain a compression result which is based on the omission of redundancy.

## 8.2 Minimum Spanning Tree Results

The minimum spanning tree approach *SimpleMST* provides a very simple and fast approach for solving the fingerprint template compression problem. In all cases the algorithm runtime was smaller than 1 second, and a compression ratio of up to 17% could be achieved. Remember that according to the formulas (9) and (13), see chapter 5, the following objective function holds for the *SimpleMST* algorithm:

$$\lambda = size(\text{ConstData}') + 2(n-1) + (n-1) \cdot \left\lceil \sum_{i=1}^{d} \left( \chi_i \cdot (1 + \text{ld } \tilde{a}^i) + (1-\chi_i) \cdot \text{ld } \tilde{v}^i \right) \right\rceil \quad (21)$$

As already mentioned MetaData is set to zero. In turn ConstData' contains the root node in the original encoding size of 28 bits. Furthermore the domain of the edges regarding the considered number of dimensions, and for the remaining dimensions the domain of the nodes is saved. Thus further 28 bits belong to ConstData'.

In order to analyze the *SimpleMST* results, the compression ratio will be considered with respect to different parameters: the *cost function*, the number of considered *dimensions* and the number of *nodes to connect*.

### *SimpleMST* compared by Cost Function

In chapter 5 three different cost functions were introduced: The Minkowski metric with $m = 2$ (Euclidian distance) and $m = d$, the maximum metric and the pseudo-cosine measure. In the figure below these cost functions except for the pseudo-cosine measure are compared over all test data and considered dimensions with respect to their mean compression ratios.

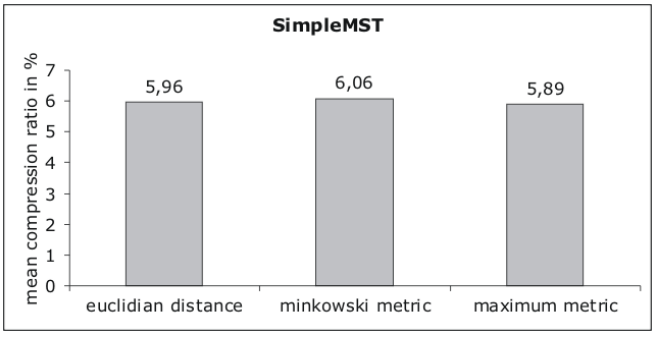



*Figure 32 - SimpleMST compared by cost function. The compression ratios are a mean value over all test data and considered dimensions.*

Obviously the Euclidian distance, Minkowski metric and maximum metric perform nearly identical. However note, that in the one-dimensional case by definition the Minkowski metric, Euclidian distance and maximum metric must lead to the same compression results. The same holds in the two-dimensional case for the Euclidian distance and the Minkowski metric. Thus the only differences might occur for three and four considered dimensions.

### *SimpleMST* compared by the Number of considered Dimensions

Altogether the results for the three different cost functions, Euclidian distance, Minkowski metric and maximum metric, are very similar. In all three cases the best results

were achieved by considering three of the four available dimensions for compression, i.e. if considering *x* and *y* coordinates and the angle.

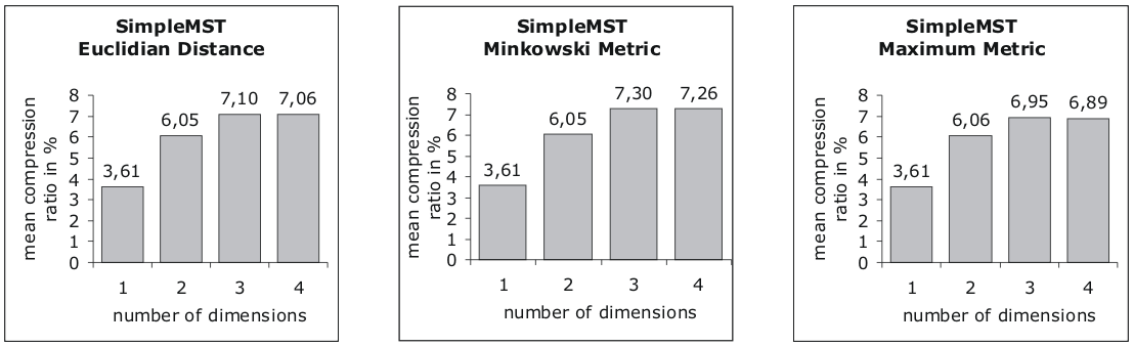


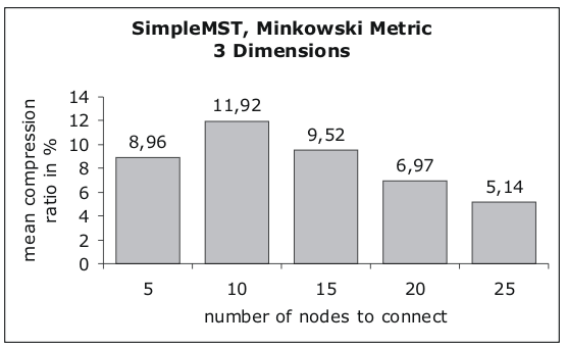

*Figure 33 - SimpleMST compared by number of considered dimensions.*

However the best overall results were obtained by applying the Minkowski metric. Since eventually the total number of used bits shall be minimized the metric for determining the edge costs must consider the values of the dimensions in a respective manner. The maximum metric thereby considers only that dimension holding the greatest value, which leads to the fact that the vector (3,0,0) for instance obtains the same costs as the vector (3,1,2), although actually more bits are needed for representing the second one. The Minkowski metric in turn computes the costs by weighting the greatest values stronger than smaller ones, whereat the weighting furthermore intensifies with a higher number of dimensions. Thus the Minkowski metric seems to be the most appropriate weighting function for representing the length of a *d*-dimensional edge, at least for this algorithm.

Since furthermore the number of nodes to connect *k* was modified, the three-dimensional results computed by applying the Minkowski metric are examined in more detail with respect to *k*.

### *SimpleMST* compared by the Number of Nodes to connect

As already mentioned the test data templates consist of 15 to 40 minutiae. In order to perform a lossy compression different values of *k* were analyzed, in particular 5, 10, 15, 20 and 25 nodes. For these values the following compression results arose.




| *k* | Max in % | Min in % | Std. Dev. |
|---|---|---|---|
| 5 | 11,69 | 3,90 | 2,25 |
| 10 | 16,67 | -1,70 | 4,16 |
| 15 | 16,13 | 0,00 | 3,88 |
| 20 | 10,80 | 0,87 | 3,09 |
| 25 | 11,48 | 1,40 | 2,94 |

*Figure 34 - SimpleMST compared by the number of nodes to connect. The ratios are a mean value over all 20 templates, and the table on the right states the maximum and minimum compression ratio, as well as the standard deviation from the mean compression ratio for every considered k.*

By connecting 10 nodes a mean compression ratio averaged over all 20 templates of nearly 12% could be achieved. Thereby the greatest compression ratio conforms to nearly 17%. If connecting more than 10 nodes the compression ratio consistently reduces. However 10 nodes might not be sufficient for verification purposes.

### *SimpleMST* Matching Results

Since the Fraunhofer Institute furthermore provided a matching algorithm, it can be examined whether the obtained subset of minutiae is sufficiently high for verification purposes. Remember that the test data consist of 20 templates from four fingers. In particular there are five templates for every finger. Thus it can be checked, whether a *k*-node template can still be successfully mated or non-mated to all other templates.

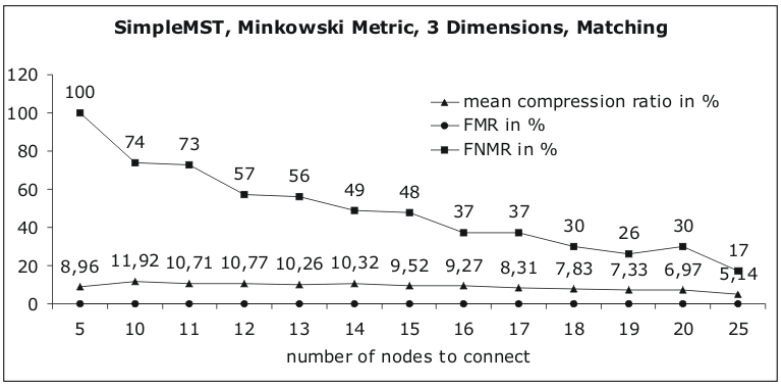



*Figure 35 - Matching results for the SimpleMST algorithm using the Minkowski metric.*

The figure above depicts the FMR, FNMR (see section 2.4) and the mean compression ratio averaged over all templates with respect to the number of nodes to connect. It turned out, that a number *k* smaller than 10 leads to a false non-match rate of 100%. Thus, 10 is the smallest number of minutiae by which a verification is still possible, at least for the Fraunhofer templates and the used matching algorithm. However, although the FMR conforms to 0% in all cases, the number of falsely non-mated templates is extremely high. When connecting 10 nodes the FNMR conforms to 74%. For a higher value of *k*, like e.g. 25, the FNMR decreases to 17%, which seems to be an acceptable ratio when considering that the original templates have a FNMR of 5%. However, by connecting 25 minutiae the compression ratio shrinks to approximately 5%.

## 8.3 Minimum Label Spanning Tree Results

The computational experiments for the minimum label spanning tree algorithms were performed for various parameter values of the *correction vector domain* $\tilde{\delta}$, the number of *considered dimensions* and the number of *nodes to connect k*. Since the analysis focuses on the compression ratio, remember that the following objective function according to formulas (16) and (17) is used for computation:

$$\lambda = size(\text{ConstData}') + 2(n-1) + (n-1) \cdot \left\lceil \operatorname{ld} l + \sum_{i=1}^{d} \left( \operatorname{ld} \tilde{\delta}^i \cdot \chi_i + \operatorname{ld} \tilde{v}^i \cdot (1-\chi_i) \right) \right\rceil + l \cdot \sum_{i=1}^{d} \lceil \operatorname{ld} \tilde{v}^i \rceil \cdot \chi_i \tag{22}$$

In this case ConstData' contains beside of the root node with 28 bits the node domain with further 28 bits for a faithful reconstruction of all nodes. Furthermore the number of computed reference vectors must be stored, using further 4 bits.

The three minimum label spanning tree algorithms *MvcaMLST*, *GreedyMLST* and *GraspMLST* delivered varying results regarding the compression ratio, runtime and the number of computed reference vectors. The *MvcaMLST* was thought as a first and rough implementation of a MVCA-based algorithm, whereat the compression results look quite poor. The *GreedyMLST* delivers much faster and also better compression ratios of up to 9%. Finally, with up to 12% the results of the *GraspMLST* are even better. However note, that the runtime for the candidate reference vector determination for values of delta larger than 30, and a relatively high number of minutiae in a template, e.g. larger than 35, could last up to a few hours. However for small delta values the respective runtime is sufficiently fast.

### 8.3.1 Results for the *MvcaMLST* Algorithm

The image below depicts the mean compression ratios depending on the delta values and the considered dimensions for the *MvcaMLST* algorithm.

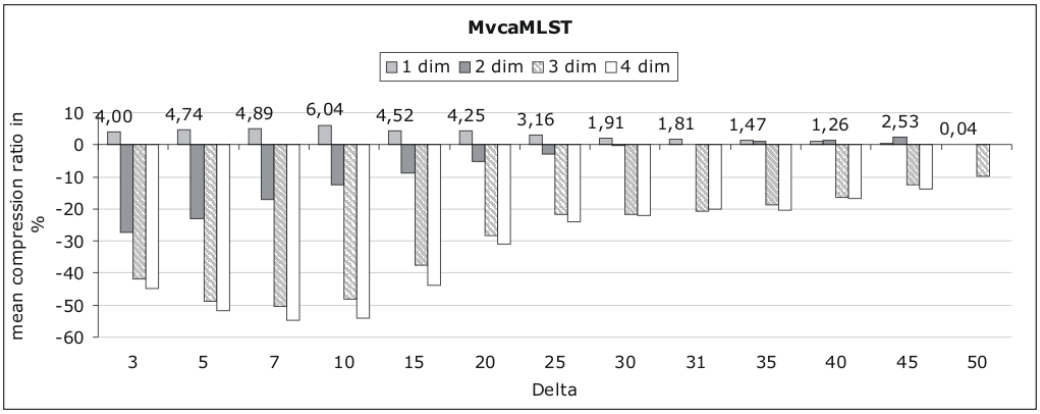



*Figure 36 - Results for the MvcaMLST algorithm analyzed with respect to the considered dimensions and delta values. Furthermore the highest mean compression ratios for every delta value are depicted.*

If considering only one dimension for compression the results are exclusively positive, although not very high. At this, especially small delta values between 5 and 15 lead to the best results. In turn for a higher number of considered dimensions the results differ enormously. Nearly all delta values result in an increased template size. Hence, at least for this algorithm only the one-dimensional case could be of interest.

However, as already mentioned, the *GreedyMLST* results are better than this, and thus the *MvcaMLST* results will not be examined in detail.

### 8.3.2 Results for the *GreedyMLST* Algorithm

The runtime of the *GreedyMLST* algorithm in most cases took not even a second. However for delta values larger than 40 in a few cases it could last up to 5 seconds. Note that these values do not contain the computation of the candidate reference vectors, but only the computation of the solution for a given complete, directed and edge-labeled graph.

Nevertheless from a general overview the results for this algorithm look already promising.

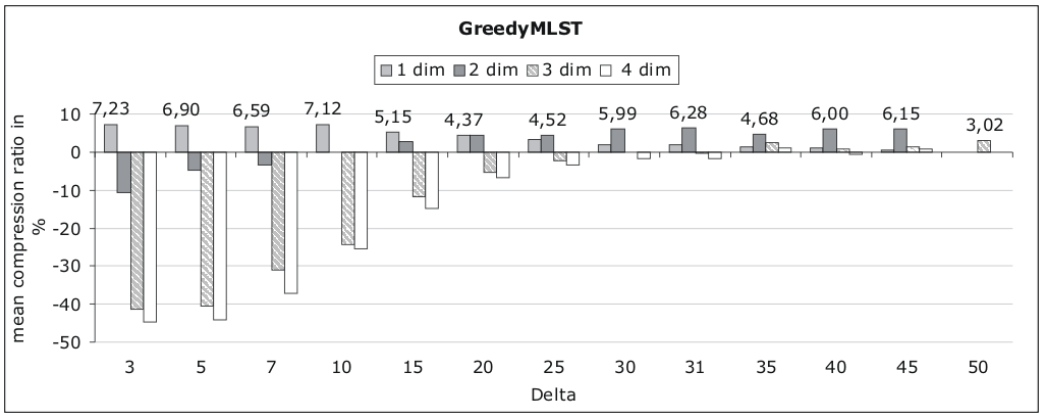
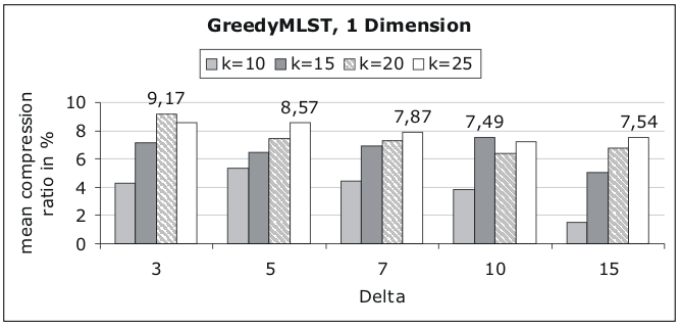



*Figure 37 - Results for the GreedyMLST algorithm analyzed with respect to the considered dimensions and delta values. For every delta value the highest mean compression ratio is depicted.*

Again the compression ratios for the one-dimensional case are exclusively positive. However this time the results for the two-dimensional case with up to 6,28% for a delta value of 31 look good, as well. In turn the results for the three- and four-dimensional cases are not very sufficient. Nevertheless, the results with respect to all considered dimensions are going to be analyzed in more detail.

### *GreedyMLST* Results for one considered Dimension

Regarding the runtime and compression ratio, especially for very small deltas the one-dimensional case could be of interest, i.e. for delta values that are smaller than 20. Hence, the following cases are depicted in the figure below.



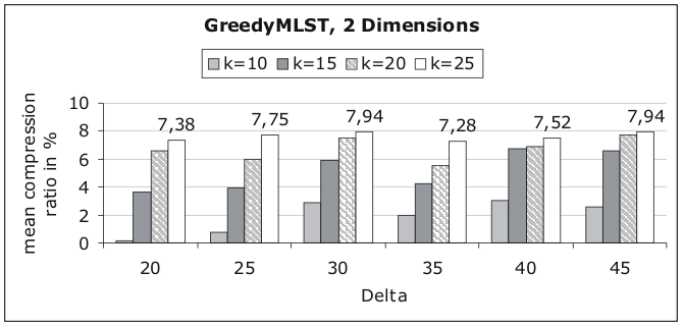

| $\tilde{\delta}$ | Max % | Min % | Std. Dev. |
|---|---|---|---|
| 3 | 10,50 | 0,00 | 1,59 |
| 5 | 13,82 | 0,00 | 1,22 |
| 7 | 13,03 | 0,00 | 2,44 |
| 10 | 13,03 | 3,06 | 1,40 |
| 15 | 9,66 | 0,00 | 0,00 |

*Figure 38 - GreedyMLST results for one considered dimension. The table on the right states the lowest and highest compression ratios for the considered delta values, and the standard deviation for that k resulting in the highest mean compression ratio.*

The best mean compression results for the one-dimensional case could be obtained for a delta value of 3 and if connecting 20 nodes, although the highest overall compression results of 13,82% were obtained for a delta of 5. Hence, if considering only one dimension and small delta values for compression fast results and a mean compression of about 9% can be achieved for the Fraunhofer templates.

### *GreedyMLST* Results for two considered Dimensions

In contrast to the one-dimensional case better compression results were obtained here for larger values of delta. In particular delta values larger than 15 are analyzed in detail.



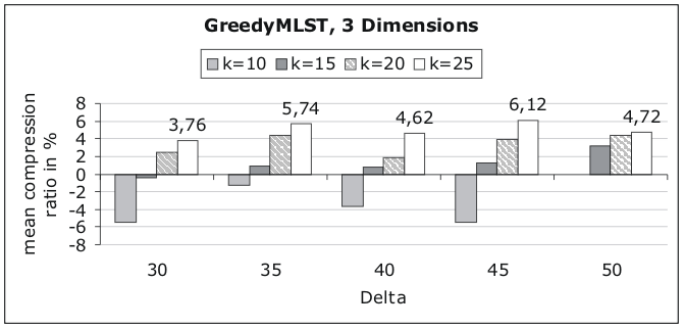

| $\tilde{\delta}$ | Max % | Min % | Std. Dev. |
|---|---|---|---|
| 20 | 12,61 | -6,12 | 2,89 |
| 25 | 12,61 | -3,06 | 1,62 |
| 30 | 13,59 | -3,06 | 2,45 |
| 35 | 15,13 | -6,12 | 2,57 |
| 40 | 11,76 | 0,46 | 2,63 |
| 45 | 11,76 | 0 | 3,61 |

*Figure 39 - GreedyMLST results for two considered dimensions. The table on the right states the lowest and highest compression ratios for the considered delta values, and the standard deviation for that k resulting in the highest mean compression ratio.*

With $\tilde{\delta} = (30, 30)$ and $\tilde{\delta} = (45, 45)$, both with $k = 25$, the best results for two considered dimensions were achieved having a mean compression ratio of 7,94%. For $\tilde{\delta} = (35, 35)$ actually a compression of maximally 15,13% could be reached. In turn for this delta value also the lowest ratio of -6,12% occurred. However delta values larger than 30 already have a relatively high computation time for pre-processing. Hence the two-dimensional results for $\tilde{\delta} = (25, 25)$ or $\tilde{\delta} = (20, 20)$ and $k = 25$ are indeed slightly worse but better regarding the runtime, and thus lead to a better tradeoff between compression ratio and runtime.

### *GreedyMLST* Results for three considered Dimensions

Neither regarding the compression ratio nor the runtime the three-dimensional results are better than for one or two considered dimensions.




| $\tilde{\delta}$ | Max % | Min % | Std. Dev. |
|---|---|---|---|
| 30 | 13,03 | -9,18 | 2,8 |
| 35 | 12,24 | -12,24 | 3,53 |
| 40 | 6,3 | -15,31 | 1,88 |
| 45 | 10,08 | -6,12 | 1,9 |
| 50 | 12,21 | -6,12 | 3,87 |

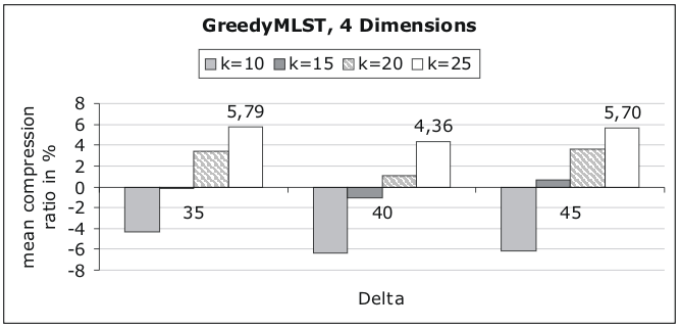
*Figure 40 - GreedyMLST results for three considered dimensions. The table on the right states the lowest and highest compression ratios for the considered delta values, and the standard deviation for that k resulting in the highest mean compression ratio.*

Although the highest overall compression ratio of 13,03% was achieved for a delta value of 30, the best mean compression results of 6,12% were obtained for $\tilde{\delta} = (45, 45, 45)$ and $k = 25$. Thus the results are about 2% poorer than the two-dimensional, and about 3 – 4% worse than the one-dimensional ones. Since the best results in this case were obtained for delta values of 45 or 35, where the computation of the candidate reference vec-

tors is not sufficiently fast anymore, this number of considered dimensions is not very promising.

### *GreedyMLST* Results for four considered Dimensions

Finally the results for four considered dimensions are yet a bit poorer than the three-dimensional ones.




| $\tilde{\delta}$ | Max % | Min % | Std. Dev. |
|---|---|---|---|
| 35 | 9,24 | -22,79 | 2,48 |
| 40 | 5,88 | -16,33 | 1,91 |
| 45 | 9,80 | -6,80 | 1,94 |

*Figure 41 - GreedyMLST results for four considered dimensions. The table on the right states the lowest and highest compression ratios for the considered delta values, and the standard deviation for that k resulting in the highest mean compression ratio.*

Analogously to the three-dimensional case the best results with a mean ratio of 5,70% were obtained for 25 nodes and $\tilde{\delta} = (45, 45, 45, 3)$, which also led to the highest overall result. However also this time the computation time of pre-processing for that delta values can not be neglected.

### Best *GreedyMLST* Results

The last sections have given an independent insight to the compression results of the *GreedyMLST* algorithm. Hence a conclusive overview about the best results is given here. In particular only the one- and two-dimensional results are considered. Since furthermore the runtime and the number of computed reference vectors was not mentioned before, the respective numbers are stated here as well.

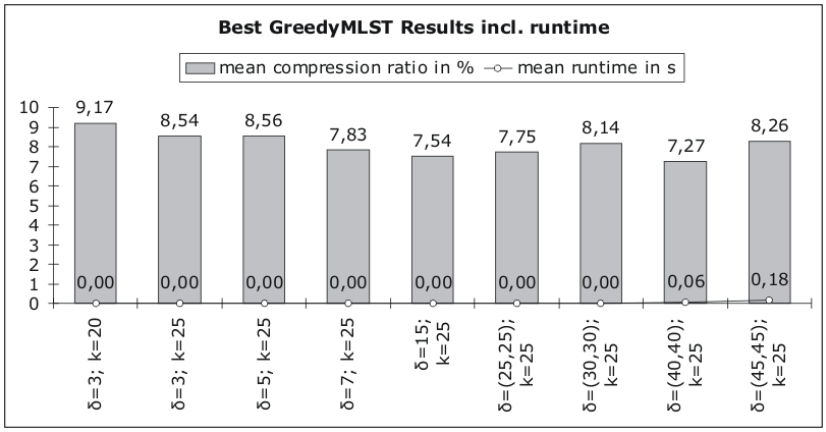



*Figure 42 - Runtime for the best GreedyMLST results. Beside of the mean compression ratios the mean runtime over all 20 templates for the given correction vector domain and k is depicted.*

The one-dimensional case provides a very fast approach and already relatively high compression results. By increasing the number of dimensions at least for the considered pa-

rameters no better results could be achieved. However the runtime without considering the determination of the candidate reference vectors is in all cases clearly fast. For a higher number of considered dimensions and a larger correction vector domain it rises only very slightly.

The number of computed reference vectors in turn for the parameter constellations depicted below is always located between one and four. Thereby, regarding the average number of reference vectors, a greater correction vector domain leads to a lower number of used reference vectors but not necessarily to a better compression result, since greater delta values of course need a higher number of bits for encoding.

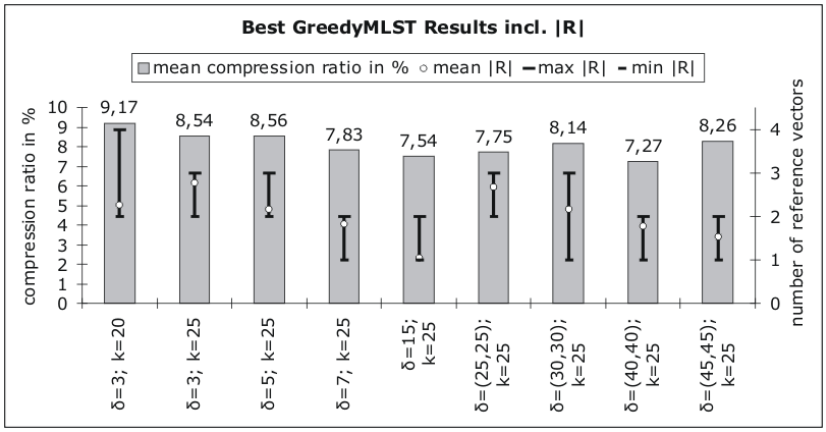



*Figure 43 - Number of reference vectors for the best GreedyMLST results. Beside of the mean compression ratios the maximum, minimum and mean numbers of computed reference vectors over all 20 templates for the given correction vector domain and k are depicted.*

Thus for one considered dimension and $k = 25$ delta values between 3 and 5 lead to the highest compression results and for two considered dimensions delta values of 25 result in the best tradeoff between compression and runtime if considering the determination of candidate reference vectors.

### 8.3.3 Results for the *GraspMLST* Algorithm

The *GraspMLST* algorithm was only examined for those parameters, which returned the best results in the *GreedyMLST* case. Therefore the following delta values were analyzed for $k = 25$: (3), (5), (7), (15), (25, 25), (30, 30), (40, 40) and (35, 35). In section 6.3.5 different types of local searches were presented. The figure below shows the respective compression results and the mean runtime.

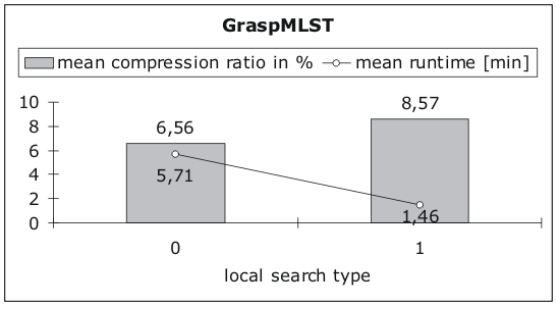



*Figure 44 - GraspMLST compared by the type of the applied local search. Beside of the mean compression ratio furthermore the mean runtime is stated.*

Note that the *GraspMLST* was run with 20 iterations for the local search 0, which in turn had 100 local iterations. The local search 1 was run with 200 GRASP iterations. Although the local search 0 was started with the smallest number of GRASP iterations it had the longer runtime with averagely 5,71 minutes and with a mean of 6,56% poorer compression results. Applying the local search 1 led to more promising results regarding the runtime and the compression ratio. Thus, it is going to be analyzed in more detail.

### *GraspMLST* Results for the Local Search 1

The local search 1 uses some kind of reference vector insertion neighborhood, i.e. that in every iteration one candidate reference vector is added to the solution. Afterwards a post-processing is performed in order to remove all redundantly gotten reference vectors. Using this strategy for those parameters that returned the best results for the *GreedyMLST* algorithm, the results depicted in the image below occur.

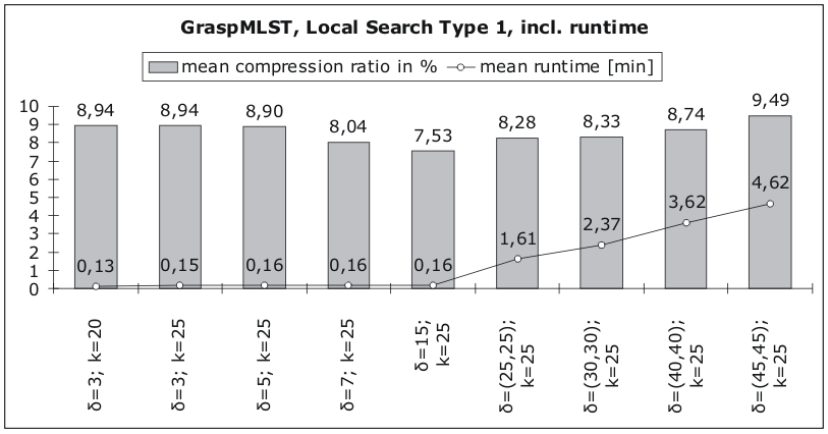



*Figure 45 - Runtime for the GraspMLST using local search 1. Beside of the mean compression ratios the mean runtimes over all 20 templates for the given correction vector domain and k are depicted.*

The results became only slightly better as for the *GreedyMLST* algorithm, especially for two considered dimensions, compare to figure (42). Furthermore the runtime for 200 GRASP iterations is depicted above, which pends in a mean between a few seconds and up to nearly five minutes for a greater correction vector domain and a higher number of considered dimensions. Below the used number of reference vectors is depicted.

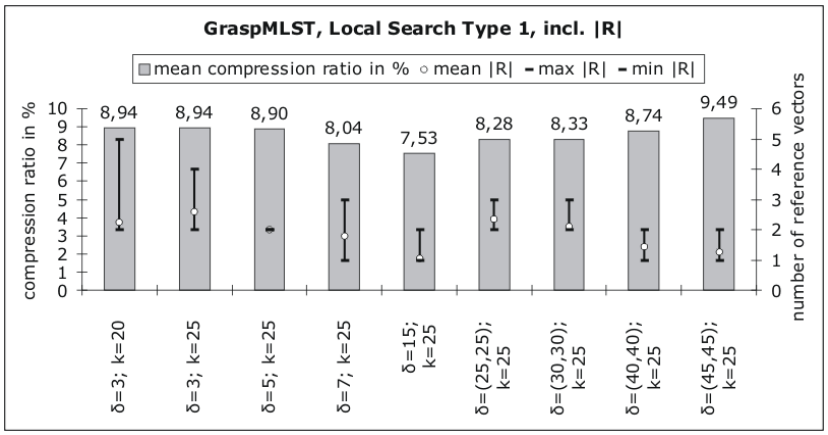



*Figure 46 - Number of reference vectors for the GraspMLST using local search 1. Beside of the mean compression ratios the maximum, minimum and mean numbers of computed reference vectors over all 20 templates for the given correction vector domain and k are depicted.*

Also the mean number of used reference vectors decreased only slightly. Thus the results are not significantly better. This is especially dependent on the fact, that the *GreedyMLST* algorithm in many cases already returns the optimal results with respect to the given correction vector domain. Hence, the following section will give a short comparison of the *GreedyMLST* and *GraspMLST* results with respect to the optimal results computed by an exact algorithm.

### Compare of Greedy and GRASP Results with Respect to the optimal Results

The following table gives a small overview about the results of 10 from the 20 given templates. The *GreedyMLST* and *GraspMLST* algorithms were run for the parameter values $\tilde{\delta} = (25, 25)$ and $\tilde{\delta} = (30, 30)$, both with $k = 20$ and $k = 25$, and compared to the results computed by an exact *branch-and-cut* algorithm [Chwatal08].

| **Template** | $\delta$ | $k$ | **GreedyMLST** | | **GraspMLST** | | **B&C** | |
|---|---|---|---|---|---|---|---|---|
| | | | \|R\| | t | \|R\| | t | \|R\| | t |
| 1 | **25,25** | **20** | **2** | **0 s** | **2** | **1 m** | **2** | **51 s** |
| | | **25** | **2** | **0 s** | **2** | **1 m** | **2** | **57 s** |
| | *30,30* | *20* | *2* | *0 s* | *1* | *1 m* | *1* | *1 h* |
| | | **25** | **2** | **0 s** | **2** | **2 m** | **2** | **1 m** |
| 2 | **25,25** | **20** | **2** | **0 s** | **2** | **53 s** | **2** | **16 s** |
| | | 25 | 3 | 0 s | 3 | 54 s | 2 | 16 s |
| | **30,30** | **20** | **2** | **0 s** | **2** | **66 s** | **2** | **55 s** |
| | | *25* | *3* | *0 s* | *2* | *1 m* | *2* | *47 s* |
| 3 | **25,25** | **20** | **2** | **0 s** | **2** | **2 m** | **2** | **3 m** |
| | | *25* | *3* | *0 s* | *2* | *4 m* | *2* | *2 m* |
| | **30,30** | **20** | **2** | **0 s** | **2** | **3 m** | **2** | **11 m** |
| | | **25** | **2** | **0 s** | **2** | **4 m** | **2** | **22 m** |
| 4 | **25,25** | **20** | **3** | **0 s** | **3** | **14 s** | **3** | **6 s** |
| | 30,30 | 20 | 2 | 0 s | 3 | 19 s | 2 | 0 s |
| 5 | **25,25** | **20** | **2** | **0 s** | **2** | **3 m** | **2** | **57 h** |
| | | **25** | **2** | **0 s** | **2** | **4 m** | **2** | **10 m** |
| | *30,30* | *20* | *2* | *0 s* | *1* | *5 m* | | |
| | | **25** | **2** | **0 s** | **2** | **6 m** | **2** | **33 m** |
| 6 | *25,25* | *15* | *3* | *0 s* | *2* | *3 s* | *2* | *51 s* |
| | 30,30 | 15 | 2 | 0 s | 2 | 3 s | 1 | 1 h |
| 7 | *25,25* | *20* | *3* | *0 s* | *2* | *36 s* | *2* | *16 s* |
| | | 25 | 3 | 0 s | 3 | 34 s | 2 | 16 s |
| | *30,30* | *20* | *3* | *0 s* | *2* | *51 s* | *2* | *55 s* |
| | | 25 | 3 | 0 s | 3 | 52 s | 2 | 47 s |
| 8 | *25,25* | *20* | *3* | *0 s* | *2* | *29 s* | *2* | *3 m* |
| | | *25* | *3* | *0 s* | *2* | *29 s* | *2* | *2 m* |
| | **30,30** | **20** | **2** | **0 s** | **2** | **38 s** | **2** | **11 m** |
| | | **25** | **2** | **0 s** | **2** | **36 s** | **2** | **22 m** |
| 9 | *25,25* | *20* | *3* | *0 s* | *2* | *45 s* | | |
| | | **25** | **3** | **0 s** | **3** | **32 s** | **3** | **6 s** |
| | **30,30** | **20** | **2** | **0 s** | **2** | **44 s** | **2** | **0 s** |
| | | *25* | *3* | *0 s* | *2* | *47 s* | | |
| 10 | *25,25* | *20* | *3* | *0 s* | *2* | *60 s* | *2* | *57 h* |
| | | 25 | 3 | 0 s | 3 | 1 m | 2 | 10 m |
| | 30,30 | 20 | 2 | 0 s | 2 | 1 m | | |
| | | *25* | *3* | *0 s* | *2* | *1 m* | *2* | *33 m* |

*Figure 47 - Compare of the GreedyMLST and GraspMLST algorithms. The table states the results for the GreedyMLST, GraspMLST and an exact Branch-and-Cut algorithm for 10 of the 20 given templates. Beside of the number of reference vectors also the runtime is depicted. The gray, bold lines depict those parameter constellations for which both, the GreedyMLST and the GraspMLST, deliver optimal results. The italic lines in turn depict those parameter values for which the GRASP improved the results of the greedy algorithm.*

The *GreedyMLST* algorithm in many cases already delivers optimal or at least nearly optimal results for the given parameter values, and also the runtime taking not even a second is clearly fast. Whereas in turn the runtime for the branch-and-cut algorithm could last up to a few hours, also the *GraspMLST* with a runtime of up to 6 minutes was sufficiently fast. Furthermore in many cases the GRASP algorithm could improve the results of the greedy algorithm up to the optimum. Hence both algorithms, the *GreedyMLST* and *GraspMLST* deliver high quality results that are in many cases optimal or at least nearly optimal. However finally still has to be checked whether the compressed templates during mating still lead to sufficient matching rates.

### 8.3.4 MLST Matching Results

Since the approach of compression is identical for all MLST algorithms, i.e. treating the edges as part of a ring structure and representing them by a reference vector together with a small correction vector, only the GRASP results are going to be mated. It is assumable that a different number of considered dimensions and also different delta values may lead to differently distributed resulting minutiae, and thus matching is performed here for a one- and a two-dimensional example.

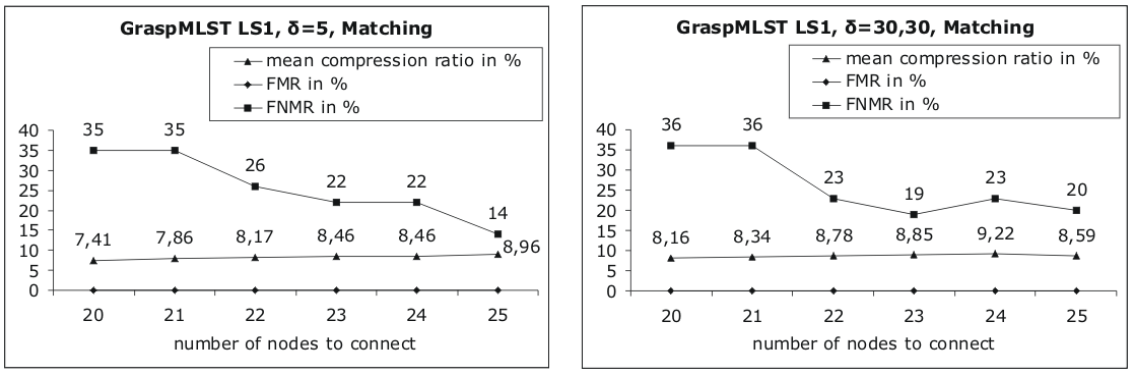



*Figure 48 - Matching results for the GraspMLST for different numbers of considered dimensions. Furthermore the FMR and FNMR is depicted.*

Analogously to the MST algorithm the false match rate in all cases conforms to 0%. However the false non-match rate differs for the different numbers of considered dimensions. Of course it must be mentioned that if running the tests again the results may differ, due to the random computation of the GRASP algorithm.

The one-dimensional case was analyzed for $\tilde{\delta} = 5$. There the best compression result was obtained for $k = 25$, having a mean ratio of 8,96%. For this number of nodes also the FNMR of 14% is furthermore sufficiently small. In turn for two considered dimensions delta values of 30 were analyzed. Thereby the compression ratio is best for 24 nodes with 9,22%, whereas the FNMR of 23% is relatively high. However if connecting 25 nodes, which results in a mean compression ratio of 8,59%, the FNMR decreases to 20%.

Thus altogether, if connecting more than 22 nodes the FNMR of maximally 23% is relatively high, but could be acceptable, because remember that the original uncompressed templates have a FNMR of 5%.

## 8.4 Weight Balanced Spanning Tree Results

The weight balanced spanning tree approach is based on the idea that one reference vector might be sufficient for representing all tree edges, such that finally only small correction vectors have to be stored. The implemented algorithms *SimpleWBST*, *MultipleWBST* and *GraspWBST* thereby led to varying results.

Remember that according to the formulas (9) and (19) the following objective function holds:

$$\lambda = size(\text{ConstData}') + 2(n-1) + (n-1) \cdot \left\lceil \sum_{i=1}^{d} (\chi_i \cdot \text{ld}\ \tilde{\delta}^i + (1-\chi_i) \cdot \text{ld}\ \tilde{v}^i) \right\rceil \quad (23)$$

ConstData' in this case contains the root node with 28 bits, the domain of the nodes with 28 bits, the domain of the correction vectors and the offset vector for the considered dimensions, whereas the respective values are encoded using the original encoding size.

The parameters that were considered for analysis conform to the MST approach – the *cost function*, the *number of considered dimensions* an the *number of nodes to connect*.

### 8.4.1 Results for the *SimpleWBST* Algorithm

A first overview about the *SimpleWBST* results states a poor image. Except for the Minkowski metric the mean compression ratios are negative.

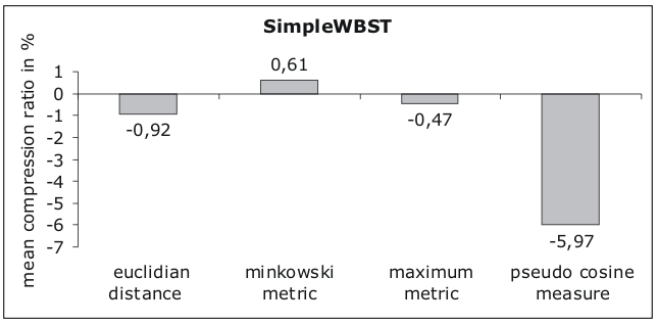



*Figure 49 - SimpleWBST compared by cost function. The compression ratios are a mean value over all test data, considered dimensions and the following numbers of nodes: 10, 15, 20 and 25.*

Note that this time also the pseudo-cosine measure has been analyzed of being an appropriate weighting function. However the respective results look not too promising, which might be based on the fact that instead of the absolute value only the orientation of the edges is considered, and thus a vector $(1,1)$ obtains the same weighting as a vector $(10,10)$. Thus it seems that at least for this case the pseudo-cosine measure is not a good choice for a weighting function. In turn the mean compression ratio for the Minkowski metric of 0,61% is not very promising, too. However the three best cost functions Euclidian distance, Minkowski metric and maximum metric are going to be analyzed in more detail.

#### *SimpleWBST* compared by the Number of considered Dimensions

At least for one considered dimension the results for the *SimpleWBST* algorithm are surprisingly high, which means that the results with a mean compression ratio of 11,79% are better than for the MST and MLST algorithms, see figures (33) and (38).

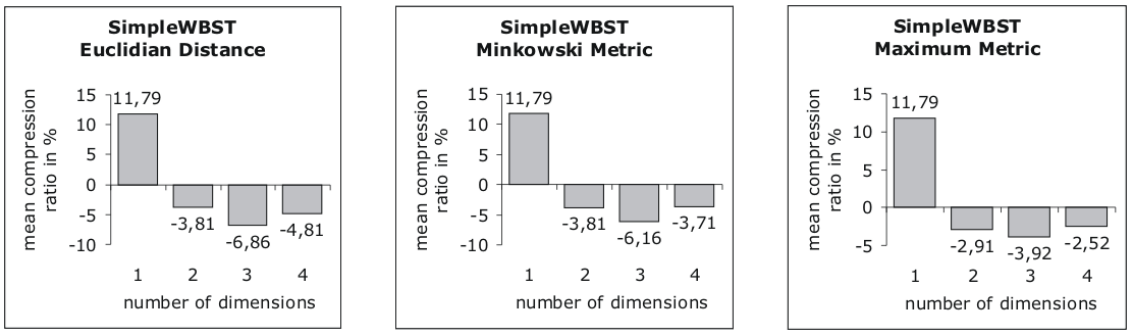
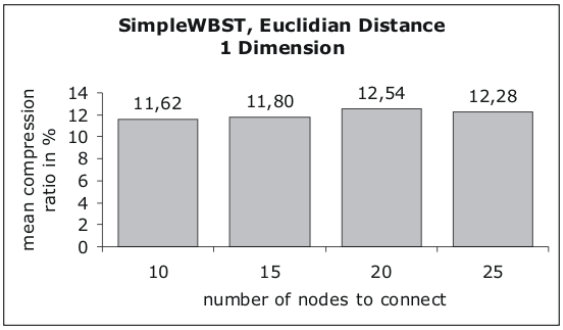



*Figure 50 - SimpleWBST compared by number of considered dimensions.*

Indeed the results for the one-dimensional case are even optimal, because the *SimpleWBST* algorithm delivers the optimal spanning tree for a one-dimensional cost function, and the costs given by the Euclidian distance, the Minkowski metric and the maximum metric directly conform to the distance values of the edges. However for a higher number of dimensions the results are not satisfyingly anymore. Nevertheless the runtime in all cases was less than one second.

### *SimpleWBST* compared by the Number of Nodes to connect

Analyzing the one-dimensional results in more detail by the number of nodes to connect leads to the following results.




| *k* | Max in % | Min in % | Std. Dev. |
|---|---|---|---|
| 10 | 13,61 | 10,54 | 1,46 |
| 15 | 14,06 | 10,83 | 1,48 |
| 20 | 15,85 | 9,23 | 1,07 |
| 25 | 13,59 | 10,22 | 1,64 |

*Figure 51 - SimpleWBST compared by the number of nodes to connect. The ratios are a mean value over all 20 templates, and the table on the right states the maximum and minimum compression ratio, as well as the standard deviation from the mean compression ratio for every considered k.*

The highest compression ratio of 12,54% averaged over all 20 templates can be reached if connecting 20 nodes, whereat the maximum compression ratio conforms to 15,85%. In case of $k = 25$ thereby the correction vector domain is between 5 and 10. Compared to the results of the currently available MLST results this seems to be very promising. Nevertheless the results of the MLST approach should be identical, due to the WBST approach is just a special case of the MLST, in which the number of used labels conforms to one. That means that at least for one considered dimension in the optimal case delta values between 5 and 10 would also in the MLST approach lead to one reference vector.

### *SimpleWBST* Matching Results

The matching results for the *SimpleWBST* do not look too satisfyingly, due to also for larger values of *k* the false non-match rate remains relatively high.

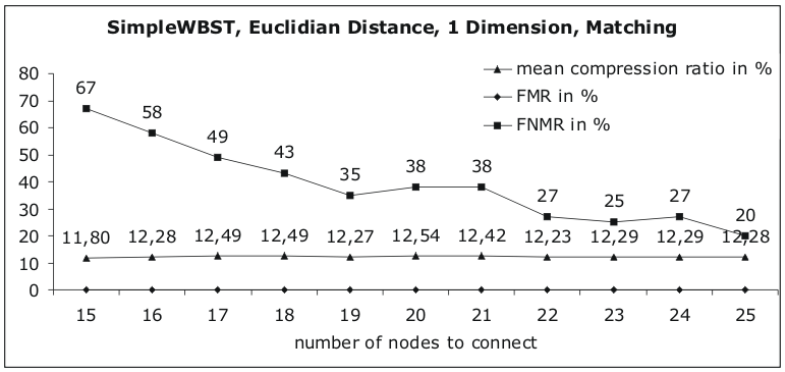



*Figure 52 - Matching results for the SimpleWBST algorithm.*

However for $k = 25$ the FNMR again is equal to the MLST results and thus should be acceptable. Furthermore like for all other matching results the FMR conforms to 0%.

### 8.4.2 Results for the *MultipleWBST* Algorithm

Since the one-dimensional case can already be optimally solved using the *SimpleWBST* algorithm, for evaluating the *MultipleWBST* results this case is not considered anymore. However the results of this algorithm were surprising too, because they do not depend on the choice of the cost function, anyway.

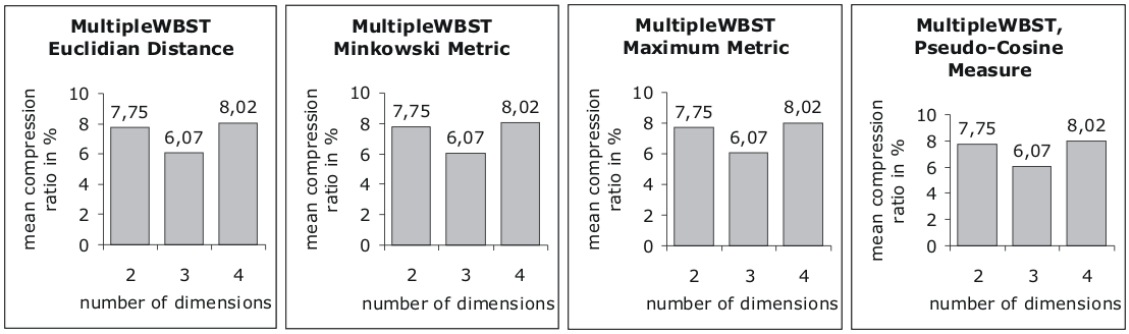


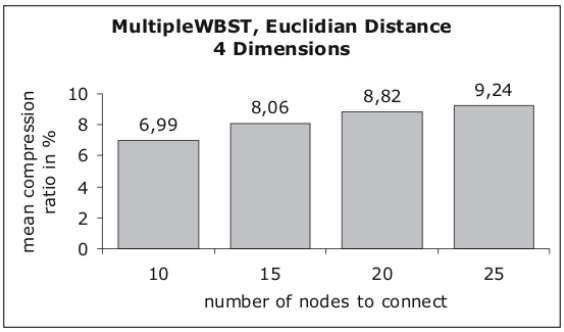

*Figure 53 - SimpleWBST compared by number of considered dimensions.*

The results for the different numbers of considered dimensions are identical for every analyzed cost function. At least for the Minkowski metric and the maximum metric this is based on the fact that in the algorithm first of all an optimal bounding box is computed with respect to the optimal weighting ranges for every single dimension. Afterwards an expansion of that bounding box takes place in order to finally obtain a spanning tree whose edges are located within this bounding box. With regard to this expansion the choice of the weighting function in the end becomes more or less unimportant. Nevertheless eventually relatively high compression results were obtained. In the two-dimensional case a mean compression of nearly 7,75% was achieved. However, the best results for this algorithm were obtained for four considered dimensions with a mean compression ratio of 8%. Furthermore in all cases the runtime was less than 4 seconds.

#### *MultipleWBST* compared by the Number of Nodes to connect

Analogously to all other results, in which the edges are treated as part of a ring domain a higher compression ratio is achieved if connecting a higher number of minutiae. In par-

ticular the best results for this algorithm with a mean compression ratio of 9,24% could be obtained for 25 nodes to connect.




| *k* | Max in % | Min in % | Std. Dev. |
|---|---|---|---|
| 10 | 19,39 | 1,02 | 4,38 |
| 15 | 12,90 | 3,23 | 2,60 |
| 20 | 12,54 | 2,61 | 2,61 |
| 25 | 14,29 | 4,20 | 2,57 |

*Figure 54 - MultipleWBST compared by the number of nodes to connect. The ratios are a mean value over all templates, and the table on the right states the maximum and minimum compression ratio, as well as the standard deviation from the mean compression ratio for every considered k.*

Although the maximum compression ratio of 19,39% was achieved for 10 nodes, the respective matching results for such a small number of nodes would not be sufficient. Nevertheless, a maximum compression ratio of 14,29% for 25 nodes to connect looks still good. However note, that the correction vector domain computed by the *MultipleWBST* algorithm is relatively high, namely in most cases between 60 and 160 for the *x* and *y* coordinates and the angle. The respective domains were not analyzed for the MLST algorithms due to the large runtime of pre-processing. From this point of view, the WBST approach is a good alternative to the MLST, since relatively good compression results can be obtained in a short runtime. Of course with respect to the update strategy of the bounding box the results for this algorithm must not be optimal.

### *MultipleWBST* Matching Results

In contrary to the *SimpleWBST* matching results, the respective results for the *MultipleWBST* look quite better.

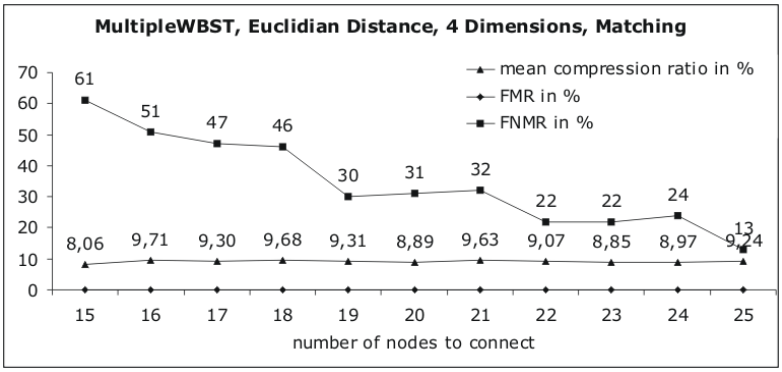



*Figure 55 - Matching results for the MultipleWBST algorithm.*

Although the highest mean compression result of 9,71% for this algorithm was achieved for 16 connected nodes, a ratio of 9,24% for 25 nodes and a respective FNMR of 13% look very satisfyingly.

### 8.4.3 Results for the *GraspWBST* Algorithm

The *GraspWBST* algorithm was introduced in order to state a further approach for improving the results of the *SimpleWBST* algorithm for a higher number of considered dimensions. Although this results are better than the *SimpleWBST* results, they are neither very satisfying nor is the approach a very fast one.

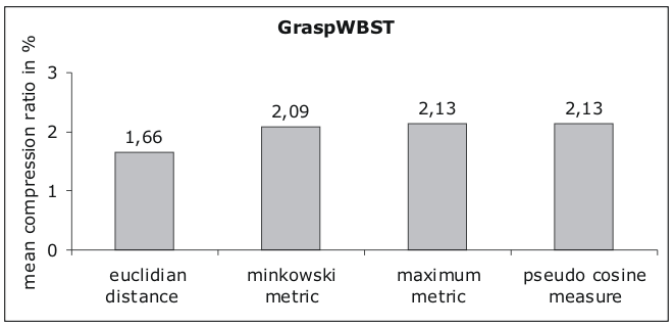



*Figure 56 - GraspWBST compared by the applied cost function.*

For all analyzed cost functions the mean compression ratios conform to approximately 2%. Although the *SimpleWBST* algorithm could surprise with optimal compression ratios for the one-dimensional case, some further evaluation of the *GraspWBST* revealed that the best results were reached using the pseudo-cosine measure for 2 dimensions, and if connecting 25 nodes. However, since this ratio of 7,5% is smaller than for the *MultipleWBST* algorithm and for the MLST algorithms, a further analysis will not be given.

## 8.5 Some Conclusive Remarks

Altogether the results for the Fraunhofer templates are already very promising. The MST algorithm as a very initial idea delivered compression results of up to 17%, however the respective matching results are not satisfying. In turn the MLST algorithms delivered optimal or nearly optimal results with respect to the number of used reference vectors, and compression results of up to 13%. And also the false non-match rate seems to be acceptable. However delta values larger than 30 for more than one dimension lead to very high runtimes based on the determination of the candidate reference vectors. Finally the WBST approach is actually a special case of the MLST approach, but instead of determining the reference vectors the correction vector domain is computed, and thus compression ratios of up to 14% can be reached in a very short runtime. However note that the mentioned compression ratios do not reflect the actual ratios regarding the original template sizes, since for computation of $\lambda_{raw}$ only $k$ nodes instead of all are considered.

| algorithm and parameters | | compression ratio in % | FNMR in % |
|---|---|---|---|
| *SimpleMST*, Minkowski metric, 3 dimensions | k = 20 | 45,62 | 30 |
| | k = 25 | 33,11 | 17 |
| *GraspMLST*, $\delta$ = (30,30) | k = 20 | 46,35 | 36 |
| | k = 25 | 35,63 | 20 |
| *SimpleWBST*, Minkowski metric, 1 dimension | k = 20 | 49,07 | 38 |
| | k = 25 | 38,26 | 20 |

*Figure 57 - Real compression ratios. The table above depicts the real mean compression ratios with respect to the real template size, which is in a mean 1012 bits for all 20 templates. Note that only those algorithms and parameters are considered that returned the best results.*

With respect to the application background indeed the compression ratios are much higher. Thus, for those parameter constellations and algorithms, which previously delivered the best results, the real mean ratios depicted above can be obtained. With regard to an acceptable false non-match rate thus compression ratios of 30 to 40% can be reached. If considering the 6 kilobyte image from section 4.1, in which 256 byte of additional information can be embedded, and if furthermore considering the mean template size of 1012 bits of the Fraunhofer templates, which conforms to approximately 127 byte, without compression 2 templates could be embedded. A compression of 30% however would eventually allow to embed up to 3 templates, which however is still not sufficient for the regarded compression technique from [Jain02] introduced in section 4.1.3, because the necessary redundancy cannot be ensured. However for slightly larger images, like for instance such of 10 kilobyte size, at least a double embedding of two templates becomes possible.

# Chapter 9 - Conclusions

"*In the end, everything is a gag.*"

Charly Chaplin

In order to embed fingerprint templates in the form of digital watermarks in digital passport photographs by means of a secure and robust embedding technique introduced in [Jain02] a compression of these templates is necessary. Such fingerprint templates contain information about the positions and orientations of the so-called minutiae, i.e. those points of a fingertip where the dermal papillae either end or bifurcate. Since this information is represented as points of a *d*-dimensional coordinate system, and thus can be conceived as points of a graph, especially graph-based approaches are of interest for this thesis. The basic idea is to store a subset of the difference vectors between the minutiae instead of the given absolute coordinate values. For this purpose directed spanning trees provide a compact encoding format. Thus if starting at a root node by performing a depth first search all further nodes can be reached and encoded. Since the compression can be lossy as well, the idea was adapted to a *k*-node directed spanning tree. Within this thesis different approaches based on various directed spanning trees are introduced. In particular, the directed minimum spanning tree, the directed minimum label spanning tree and the directed weight balanced spanning tree are considered. For testing and analyzing a small set of templates and a respective matching algorithm was provided by the Fraunhofer Institute for Productions Systems and Design Technologies.

The first and very simple approach introduced in this thesis applies the directed minimum spanning tree. The presented algorithm *SimpleMST* is an adapted version of Kruskal's algorithm and already provides relatively good results. With respect to acceptable matching results the *SimpleMST* algorithm delivers compression ratios of averagely 5%.

The main approach of this thesis uses some kind of dictionary for compression in which a small subset of the difference vectors between all pairs of nodes is stored. The edges are represented by a reference to the respective dictionary entry, a so-called reference vector, together with a small correction vector. In order to compute the smallest possible subset of dictionary entries from a pre-computed set of candidate reference vectors with respect to a given correction vector domain the directed minimum label spanning tree is applied. All implemented algorithms, i.e. the *MvcaMLST*, *GreedyMLST* and *GraspMLST* are greedy heuristics or metaheuristics respectively, based on an adapted version of the maximum vertex cover algorithm introduced in [Chang96]. The presented algorithms were analyzed with respect to a different number of considered dimensions for compression and different correction vector domains. It turned out that if considering one or even two dimensions the highest compression results of up to 15% could be achieved. At this, the *GraspMLST* algorithm performed averagely up to 1,5% better than the *GreedyMLST*, whereat the latter one in many cases already returned optimal results with respect to the computed number of reference vectors. However the respective domain of the correction vectors with values of up to 35 leads especially for greater template instances to a relatively high runtime of the pre-processing, which delivers the candidate reference vectors. Thus if considering a smaller correction vector domain a mean compression ratio of 8% in a sufficiently fast runtime is achievable. Independently from this thesis also other algorithms applying the minimum label spanning tree were developed at the Institute for Computer Graphics and Algorithms of the Vienna University of Tech-

nology, in particular an exact branch-and-cut algorithm and genetic algorithms, see [Chwatal08].

A third and last approach is a special case of the second one. However instead of determining the smallest possible subset of reference vectors one common offset vector is sought. For solving this problem a directed version of the weight balanced spanning tree is applied, that determines a spanning tree in which the difference between the smallest and largest edge weight is minimal. If considering only one dimension for compression the introduced algorithm *SimpleWBST* delivers fast and optimal results with a mean compression ratio of up to 13%. Both other implemented algorithms, the *MultipleWBST* and the *GraspWBST* deliver mean compression results of 8% and 5%, respectively.

If considering the real template size instead, which is not bound to the number of nodes to connect, much higher compression ratios of up to 40% in a mean and in a few cases up to 80% in total can be reached. However only a small amount of data was tested and the modifiable parameters were not exhaustively varied. Thus still higher compression ratios are possible. Since common compression techniques, such as ZIP, moreover consistently increase the size of the considered templates, the techniques presented within this thesis seem to be a highly promising approach.

# List of Abbreviations

| | |
|---|---|
| AFIS | Automated Fingerprint Identification System |
| ASCII | American Standard Code for Information Interchange |
| BTC | Biometric Template Compression |
| CC | Connected Component |
| CL | Candidate List |
| DAG | Directed Acyclic Graph |
| DD | Direction Difference |
| DFS | Depth First Search |
| DFT | Discrete Fourier Transform |
| DWT | Discrete Wavelet Transform |
| FBI | Federal Bureau of Investigation |
| FMR | False Match Rate |
| FNMR | False Non-Match Rate |
| GIF | Graphics Interchange Format |
| GRASP | Greedy Randomized Adaptive Search Procedure |
| ISO | International Organization for Standardization |
| JPEG | Joint Photographic Experts Group |
| LSB | Least Significant Bit |
| LZW | Lempel-Ziv-Welch |
| MLST | Minimum Label Spanning Tree |
| MST | Minimum Spanning Tree |
| MVCA | Maximum Vertex Cover Algorithm |
| PIN | Personal Identification Number |
| RAM | Random Access Memory |
| RCL | Restricted Candidate List |
| RFID | Radio Frequency Identification |
| SCC | Strongly Connected Component |
| SD | Spatial Distance |
| WBST | Weight Balanced Spanning Tree |

# List of Figures

# List of Algorithms

# References


[Maltoni03] Davide Maltoni, Dario Maio, Anil K. Jain, Salil Prabhakar: Handbook of Fingerprint Recognition, Springer-Verlag, New York, Berlin, Heidelberg 2003, ISBN: 0-387--95431-7

[Jain02] Anil J. Kain, U. Uludag: Hiding Fingerprint Minutiae in Images, 2002, http://biometrics.cse.msu.edu/Publications/SecureBiometrics/JainUludag_HidingFpMina_AutoID02.pdf

[Chang96] Ruay-Shiung Chang, Shing-Jiuan Leu: The minimum labeling spanning trees, Information Processing Letters 63, 277-282, 1996

[Gupta04] Rajarshi Gupta: Graph Theory in Networks, Lecture Notes, University of California, Berkeley, 2004

[BioBSI07] Biometrie, Bundesamt für Sicherheit in der Informationstechnik, 2007, http://www.bsi.bund.de/fachthem/biometrie/index.htm

[Duden96] Duden, Rechtschreibung der deutschen Sprache, 21., völlig neu bearbeitete und erweiterte Auflage, Dudenverlag, Dudenredaktion, Mannheim, Leipzig, Wien, Zürich 1996, ISBN: 3-411-040-11-4

[Heumann06] Björn Heumann: Whitepaper Biometrie, 2006, http://www.heumann-webdesign.de/pages/biometrie/Whitepaper_Biometrie.pdf

[Behrens01] Michael Behrens: Biometrische Identifikation, 1. Auflage, Vieweg & Sohn Verlagsgesellschaft mbH, Richard Roth, Braunschweig/Wiesbaden 2001

[Beavan01] Colin Beavan: Fingerprints - The Origins of Crime Detection and the Murder Case..., Hyperion, New York 2001, ISBN: 978-0786866076

[Ihmor03] Heinrich Ihmor: Wird das Rad neu erfunden?, Bundesamt für Sicherheit in der Informationstechnik, KES – Die Zeitschrift für Informationssicherheit, 2003/05, http://www.bsi.de/fachthem/biometrie/dokumente/geschichte.pdf

[Galton03] Gavan Tredoux: Henry Faulds - The Invention of a Fingerprinter, 2003, http://galton.org/fingerprints/faulds.htm

[FingerBSI06] Fingerabdruckerkennung, Bundesamt für Sicherheit in der Informationstechnik, 2006, http://www.bsi.de/fachthem/biometrie/dokumente/Fingerabdruckerkennung.pdf

[FVC04] Fingerprint Verification Competition 2004, http://biometrics.cse.msu.edu/fvc04db/index.html

[ISO19794-2] International Standard ISO/IEC 19494-2, Information Technology - Biometric Data Interchange Format – Part 2: Finger Minutiae Data, ISO/IEC, 2005

[DINV66400] Dr. Robert Müller: Finger Minutiae Encoding Format and Parameters for On-Card Matching, 2002

[Basler05] Georg Basler, Andreas Schutt: Fingerabdrucksysteme, Technical Report, 2005, http://www2.informatik.hu-berlin.de/Forschung_Lehre/algorithmenII/Lehre/SS2004/Biometrie/04Fingerprint/fingerabdrucksysteme.pdf

[Huvanandana00] Sanpachai Huvanandana, Changick Kim, Jenq-Neng Hwang: Reliable and Fast Fingerprint Identification for Security Applications, Proceedings of the IEEE, 2000

[Jain97] Anil Jain, Lin Hong, Sharath Pankanti, Ruud Bolle: An Identity Authentication System Using Fingerprints, Proceedings of the IEEE, Vol. 85, No. 9, Sept. 1997, 1997, http://www.research.ibm.com/ecvg/pubs/sharat-proc.pdf

[Bazen01] Asker M. Bazen, Sabih H. Gerez: An Intrinsic Coordinate System for Fingerprint Matching, 2001, http://www.springerlink.com/content/j1pr9h3d68gmlp1j/

[PassSpec06] Kommission der europäischen Gemeinschaften: Entscheidung der Kommission über die Spezifikationen von EU-Pässen, 2006

[Pfitzmann00] Andreas Pfitzmann: Sicherheit in Rechnernetzen, TU Dresden, 2000

[Katzenbeisser00] Stefan Katzenbeisser, Fabien A.P. Petitcolas: Information Hiding Techniques for Steganography and Digital Watermarking, Artech House Publishers, 2000, ISBN: 978-1580530354

[Schmitz06] Roland Schmitz, Roland Kiefer, Johannes Maucher, Jan Schulze, Thomas Suchy: Kompendium Medieninformatik, Mediennetze, Springer, Berlin, Heidelberg, New York 2006, ISBN: 978-3-540-30224-7

[Leweler87] Debra A. Leweler, Daniel S. Hirschberg: Data Compression, ACM, Computing Surveys, Vol. 19, No. 3, 1987

[Cormen07] Th. H. Cormen, Ch. E. Leiserson, R. Rivest, C. Stein: Algorithmen – Eine Einführung, 2. Auflage, Oldenbourg, München, Wien 2007, ISBN: 978-3-486-58262-8

[Welch84] Terry A. Welch: A Technique for High-Performance Data Compression, Sperry Research Center, 1984

[Fellbaum84] Klaus-Rüdiger Fellbaum, Hans Marko: Sprachverarbeitung und Sprachübertragung, Springer-Verlag GmbH, 1984, ISBN: 978-3540133063

[Fellbaum03] Klaus-Rüdiger Fellbaum, Hans-Jörg Ullman: Grundzüge der Multimedia-Kommunikation, TU Cottbus, Lehrstuhl Kommunikationstechnik, Elektrotechnik und Informationstechnik, Skriptum, 2003

[Sherlock96] B. G. Sherlock, D. M. Monro: Optimized Wavelets for Fingerprint Compression, University of Bath, School of Electronic and Electrical Engineering, Proceedings of the ICASSP,1996

[Zirkind07] Givon Zirking: AFIS Data Compression, ACM SIGSOFT Software Engineering Notes, Volume 32, Number 6, 2007

[Fontenot02] Mark Fontenot: A Wavelets Introduction, McNeese State University, 2002

[Grasemann05] Uli Grasemann, Misto Miikkulainen: Effective Image Compression using Evolved Wavelets, The University of Texas at Austin, Department of Computer Sciences, 2005

[Sudhakar05] R. Sudhakar, S. Jayaraman: Fingerprint Compression Using Multiwavelets, International Journal of Signal Processing, Volume 2, Number 1, 2005

[Chong92] Michael M. S. Chong et al.: Automatic representation of fingerprints for data compression by b-spline functions, Pattern Recognition, Volume 25, Issue 10, 1992

[Chwatal08] (in preparation) Andreas Chwatal, Günther Raidl, Olivia Dietzel, Karin Oberlechner: Compressing Fingerprint Templates by Solving an Extended Minimum Label Spanning Tree Problem, Vienna University of Technology, Institute of Computer Graphics and Algorithms, 2008

[Schmitt05] Ingo Schmitt: Ähnlichkeitssuche in Multimedia – Datenbanken, Retrieval, Suchalgorithmen und Anfragebehandlung, Oldenbourg Wissenschaftsverlag, München 2005, ISBN: 348657907X

[Ferber03] Reginald Ferber: Information Retrieval, Suchmodelle und Data-Mining-Verfahren für Textsammlungen und das Web, 1. Auflage, dpunkt.verlag GmbH, Heidelberg 2003, ISBN: 3-89864-213-5

[Xiong04] Yupei Xiong, Bruce Golden, Edward Wasil: A One-Parameter Genetic Algorithm for the Minimum Label Spanning Tree Problem, IEEE Transactions On Evolutionary Computation, Vol. 9, No. 1, 2004

[Russel04] Stuart Russel, Peter Norvig: Künstliche Intelligenz, Ein moderner Ansatz, 2. Auflage, Pearson Studium, München 2004, ISBN: 3-8273-7089-2

[Krumke98] Sven O. Krumke, Hans-Christoph Wirth: On the minimum label spanning tree problem, Information Processing Letters 66, 81-85, 1998

[Wan02] Yiunyu Wan, Guoliang Chen, Yinlong Xu: A note on the minimum label spanning tree, Department of Computer Science and Technology, University of Science and Technology of China, Information Processing Letters 84, 99-101, 2002

[Xiong05] Y. Xiong, B. Golden, E. Wasil: Worst Case Behaviour of the mvca heuristic for the Minimum Label Spanning Tree problem, Operations Research Letters, 33 (1), 2005

[Brüggemann02] Tobias Brüggemann, Jerome Monnot, Gerhard J. Woeginger: Local Search for the minimum label spanning tree problem with bounded color classes, Department of Mathematics, University of Twente, The Netherlands, Operations Research Letters 31, 195 - 201, 2002

[Nummela06] Jeremiah Nummela, Bryant A. Julstrom: An effective genetic algorithm for the minimum label spanning tree problem, 2006, http://portal.acm.org/citation.cfm?id=1144097

[Xiong06] Yupei Xiong, Bruce Golden, Edward Wasil: Improved Heuristics for the Minimum Label Spanning Tree Problem, IEEE Transactions On Evolutionary Computation, Vol. 10, 2006

[Voss99] Cees Duin, Stefan Voß: The Pilot Method – A Strategy for Heuristic Repetition with Application to the Steiner Problem in Graphs, Networks, Vol. 34, No. 3, 1999

[Cerulli05] R. Cerulli, A. Fink, M. Gentili, S. Voß: Metaheuristics comparison for the minimum labelling spanning tree problem, The Next Wave on Computing, Optimization, and Decision Technologies, Vol. 29, New York 2005, ISBN: 978-0-387-23528-8

[Consoli07] S. Consoli, K. Darby-Dowman, N. Mladenovic, J. A. Moreno Perez: Solving the minimum label spanning tree problem using hybrid

local search, 2007

[Oberlechner08] Karin Oberlechner: A Genetic Algorithm for the Minimum Label Spanning Tree Problem, Praktikumsbericht, 2008

[Chu65] Y. J. Chu, T. H. Liu: On the shortest arborescence of a directed graph, Science Sinica, 14, 1965

[Edmonds67] J. Edmonds: Optimum branchings, J. Research of the National Bureau of Standards, 71B, 1967

[Resende02] Mauricio G.C. Resende, Celso C. Ribeiro: Greedy Randomized Adaptive Search Procedures, 2002

[Wagner05] Daniel Wagner: Eine generische Bibliothek für Metaheuristiken und ihre Anwendung auf das Quadratic Assignment Problem, Diploma Thesis, Vienna University of Technologiy, Insitute for Computer Graphics and Algorithms, 2005

[Leda] Algorithmic Solutions Software GmbH: LEDA Description, 2008, http://www.algorithmic-solutions.com/