TECHNISCHE UNIVERSITÄT WIEN
Institut für Computergraphik und Algorithmen

Solving an Extended Minimum Label Spanning Tree Problem to Compress Fingerprint Templates

Andreas M. Chwatal and Günther R. Raidl and Karin Oberlechner

Forschungsbericht / Technical Report

TR–186–1–08–01

September 2008

Favoritenstraße 9-11 / E186, A-1040 Wien, Austria
Tel. +43 (1) 58801-18601, Fax +43 (1) 58801-18699
www.cg.tuwien.ac.at

Noname manuscript No.
(will be inserted by the editor)

Solving an Extended Minimum Label Spanning Tree Problem to Compress Fingerprint Templates

Andreas M. Chwatal · Günther R. Raidl · Karin Oberlechner

Received: dd. mm. yyyy / Accepted: dd. mm. yyy

Abstract We present a novel approach for compressing relatively small unordered data sets by means of combinatorial optimization. The application background comes from the field of biometrics, where the embedding of fingerprint template data into images by means of watermarking techniques requires extraordinary compression techniques. The approach is based on the construction of a directed tree, covering a sufficient subset of the data points. The arcs are stored via referencing a dictionary, which contains "typical" arcs w.r.t. the particular tree solution. By solving a tree-based combinatorial optimization problem we are able to find a compact representation of the input data. As optimization method we use on the one hand an exact branch-and-cut approach, and on the other hand heuristics including a *greedy randomized adaptive search procedure* (GRASP) and a *memetic algorithm*. Experimental results show that our method is able to achieve higher compression rates for fingerprint (minutiae) data than several standard compression algorithms.

Keywords combinatorial optimization · metaheuristics · GRASP · memetic algorithm · biometric template compression · fingerprint minutiae · unordered data set compression

1 Introduction

In this work, we describe a new possibility for compressing relatively small unordered data sets. Our particular application background is to encode fingerprint template data by means of watermarking techniques (see [13]) e.g. in images of identification cards as an additional security feature. Since the amount of information that can be stored by means of watermarking is very limited, extraordinary compression mechanisms are required in order to achieve reasonably small error rates when finally checking fingerprints against the encoded templates.

Institute of Computer Graphics and Algorithms
Vienna University of Technology
Vienna, Austria
E-mail: chwatal@ads.tuwien.ac.at, raidl@ads.tuwien.ac.at, karin.oberlechner@gmail.com

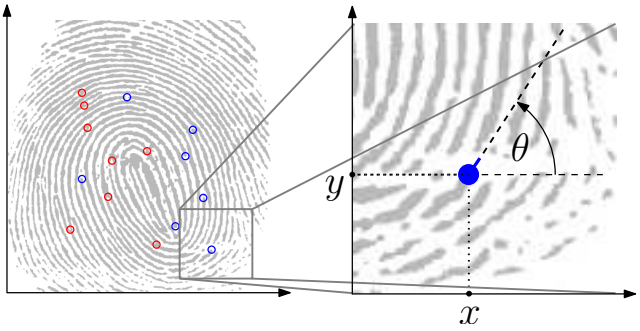

Fig. 1 Minutiae points of a fingerprint image. The right figure shows the cartesian coordinates x and y of a specific minutiae point, as well as its orientation θ. Together with its type (e.g. ridge ending, bifurcation, etc.) one minutiae point can be described as a 4-tuple (x, y, θ, t).

Having a scanned fingerprint image, traditional image processing techniques are applied for determining its *minutiae*, which are points of interest such as bifurcations, crossover, and ridge endings. Fingerprint matching algorithms are usually based on these minutiae data [17]. Typically, 15 to 80 minutiae are extracted from a single fingerprint, and for each we obtain as attributes its type t (e.g. ridge ending, bifurcation, etc.), x and y coordinates, and an angle θ (see Fig. 1). A minutia can thus be interpreted as a four-dimensional vector. The task we focus on here is the selection of an arbitrary subset of k minutiae in combination with its lossless encoding in a highly compact way, with k being a prespecified number.

For this purpose we formulate the problem as a combinatorial optimization problem, in particular a variant of the well known *Minimum Label Spanning Tree (MLST) Problem* [3]. By finding optimal or near-optimal solutions to this problem, we can represent the minutia data by means of a directed tree spanning k nodes, where each edge is encoded by a reference to a small set of template arcs and a small correction vector.

The paper is organized as follows: After a review of related work in Section 2 we give a detailed and more formal problem description in Sections 3 and 4. Section 5 describes the preprocessing which actually computes the labels from the input data. Section 6 presents a branch-and-cut algorithm to solve the MLST problem variant to optimality. To achieve shorter running times for practical purposes metaheuristics are applied to solve the problem approximately. A greedy randomized adaptive search procedure and a memetic algorithm are described in detail in Section 7. In Section 8 we explicitly describe how to encode a solution on a binary level. Finally, we present the results from our computational experiments in Section 9, and conclusions are drawn in Section 10.

2 Previous Work

In general, data compression creates a compact representation of some input data by exhibiting certain structures and redundancies within these data. Various well established techniques exist for diverse fields of applications like text-, image-, and audio-compression.

For instance entropy based methods like *Huffman coding* or *arithmetic coding* are well approved in the field of lossless text compression. Alternatively, *dictionary coders*

like the well known LZ77 and LZ78 [27, 28] try to account for (repeating) structures in the text by means of a dictionary. The idea of a dictionary can also be found in other compression techniques specialized for image-, audio- or video-compression. For example, consider the lossy *vector quantization* method for image compression. Hereby the input data is grouped in blocks of length L, and the respective elements of such a block correspond to the components of a vector of size L. The image is represented by subsequent references to the most similar vector in the *codebook*, a list of typical and frequently occuring vectors. For a comprehensive review of data compression techniques see [22].

Our approach follows this general idea of using a dictionary or codebook, but its determination as well as its usage is very different to the existing methods. Before going into details we point out the limitations and peculiarities of our approach.

If k is equal to the number of input data points our approach encodes the input data in a lossless way; for lower values of k the method can be considered a special form of lossy compression.

As we are not interested in the respective order of the minutiae, the initial sequence need not to be preserved. In this case a theoretical bound for the encoding length of $O(\log k)$ exists [23]. As our encoding as directed k-node spanning tree does not preserve the relative order of the minutiae it can be interpreted as an attempt to benefit from the absence of the requirement to preserve the order.

In [6] we originally presented our compression model and outlined a GRASP algorithm to solve the associated optimization problem heuristically. An exact branch-and-cut approach was presented in [5, 20].

In this paper we describe the compression model in depth. Moreover we give a detailed description of the preprocessing method for the first time. In addition to a more extensive description of the exact branch-and-cut method and the GRASP approach we present a new memetic algorithm to solve our optimization problem. Finally we present a comprehensive evaluation of our results for the first time, including a detailed description of the test instances, an overview about the compression ratios, and algorithmic results including the respective running times.

3 Tree-Based Compression Model

More formally, we consider as raw data n d-dimensional points (vectors) $V = \{v_1, \ldots, v_n\}$ from a discrete domain $\mathbb{D} = \{0, \ldots, \tilde{v}^1 - 1\} \times \ldots \times \{0, \ldots, \tilde{v}^d - 1\}$ corresponding to our minutiae data ($d = 4$ in the above described application scenario). The domain limits $\tilde{v}^1, \ldots, \tilde{v}^d \in \mathbb{N}$ represent the individual sizes and resolutions of the d dimensions.

Our aim is to select k of these n points and to connect them by an outgoing arborescence, i.e. a directed k-node spanning tree. For this we start with a complete directed graph $G = (V, A)$ with $A = \{(u, v) \mid u, v \in V, u \neq v\}$ on which we search for the optimal arborescence by optimization. Each node in this complete graph corresponds exactly to one of the n points (vectors) and is therefore denoted by the same label $v_i, i \in [0, n]$. Consequently, each arc of the arborescence represents the relative geometric position of its end point in dependence of its starting point.

In addition, we use a small set of specially chosen *template arcs*. Instead of storing for each tree arc its length in any of the d dimensions, we encode it more tightly by

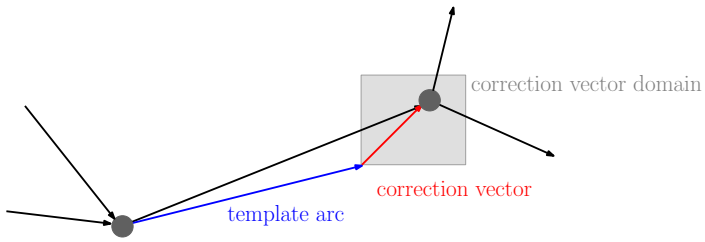

Fig. 2 Encoding of some tree arc by means of a template arc and a (small) correction vector

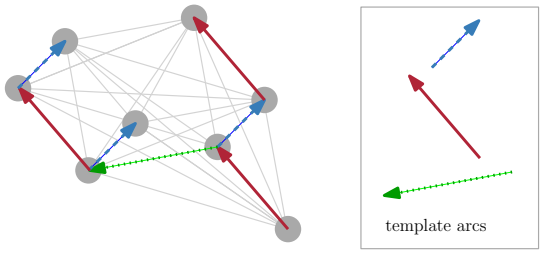

Fig. 3 Illustration to the encoding of points via a directed spanning tree using a codebook of template arcs; correction vectors are omitted

a reference to the most similar template arc and a so called *correction vector* from a small domain (see Fig. 2). Thus, the set of template arcs acts as a *codebook* (Fig. 3).

In order to achieve a high compression rate, we optimize the selection of the k encoded points, the tree structure, and the used template arcs simultaneously. The domain for the correction vectors is pre-specified, while the number of template arcs is the objective to be minimized. Another possibility would be to prespecify the number of template arcs and minimize the correction vector domain. This approach, however, is not part of this work and could be a topic of further research.

Having solved this optimization problem, we finally store as compressed information the template arc set and the tree. The latter is encoded by traversing it with depth-first search; at each step we write one bit indicating whether a new arc has been traversed to reach a yet unvisited node or backtracking along one arc took place. When following a new arc, a reference to its template arc plus the (small) correction vector are additionally written.

More formally, a solution to our problem consists of

1. a vector of template arcs $T = (t_1, \ldots, t_m) \in \mathbb{D}^m$ of arbitrary size m representing the codebook, and
2. a rooted, outgoing tree $G_T = (V_T, A_T)$ with $V_T \subseteq V$ and $A_T \subseteq A$ connecting precisely $|V_T| = k$ nodes, in which each tree arc $(i, j) \in A_T$ has associated
   - a template arc index $\kappa_{i,j} \in \{1, \ldots, m\}$ and
   - a correction vector $\delta_{i,j} \in \mathbb{D}'$ from a pre-specified, small domain $\mathbb{D}' \subseteq \mathbb{D}$ with $\mathbb{D}' = \{0, \ldots, \tilde{\delta}^1 - 1\} \times \ldots \times \{0, \ldots, \tilde{\delta}^d - 1\}$.

For any two points v_i and v_j connected by a tree arc $(i, j) \in A_T$ the relation

$$v_j = (v_i + t_{\kappa_{i,j}} + \delta_{i,j}) \bmod \tilde{v}, \qquad \forall (i, j) \in A_T, \tag{1}$$

must hold; i.e. v_j can be derived from v_i by adding the corresponding template and correction vectors. The modulo-calculation is performed in order to always stay within a finite ring, so there is no need for negative values and we do not have to explicitly consider domain boundaries.

Our main objective is now to find a feasible solution with a smallest possible codebook size, i.e. which requires a minimal number m of template arcs.

4 Reformulation as a Minimum Label k-Node Subtree Problem

We approach the problem of finding a smallest possible codebook of template arcs together with a feasible tree as follows: First we derive a large set T^c of *candidate template arcs* (see Section 5); then we assign to each arc $(i, j) \in A$ all template arcs $T^c(a_{ij}) \subseteq T^c$ that are able to represent it w.r.t. equation (1). Secondly we optimize the codebook by selecting a minimal subset $T \subseteq T^c$ allowing a feasible tree encoding.

The remaining problem in the second part of this approach is related to the *Minimum Label Spanning Tree (MLST) Problem*, first introduced in [3]. In this problem an undirected graph is given and each edge has associated a label from a discrete label set. The objective is to find a spanning tree whose edges correspond to a minimum set of labels. The problem is known to be NP-hard, which can be easily shown by a reduction from the *set cover* problem [3]. In [14] the authors show that the MLST problem is not approximable within a constant factor.

In our problem the candidate template arcs T^c correspond to the labels. Major differences are, however, that we have to consider complete directed graphs, multiple labels may be assigned to an arc, and the labels come up with certain geometric properties.

Altough we do not have a proof yet, there are strong hints that the problem remains NP-hard in this version. Consider the situation, where an arborescence is prespecified and its optimal labelling should be found by optimization. Due to the geometric properties of the arcs and the labels this problem is equivalent to the *rectangle covering problem*, which is known to be NP-complete [11].

Another major extension to the MLST problem is the fact that not all nodes but only an arbitrary subset of size k shall be connected in general. We call the resulting version of the MLST problem *k-node Minimum Label Spanning Arborescence* (k-MLSA) problem.

5 Preprocessing

The preprocessing step is to derive a set of candidate template arcs from which the codebook will be choosen as a subset. This set of candidate template arcs has to be sufficiently large to allow an overall optimal solution, i.e. a minimal codebook.

In the following we will use the terms arc and vector equivalently, as each arc (i, j) in our graph represents the geometric information of the vector $(v_j - v_i) \bmod \tilde{v}$. To describe the preprocessing in more detail we have to introduce further notation:

- $B = \{v_{ij} = (v_j - v_i) \bmod \tilde{v} \mid (i, j) \in A\} = \{b_1, \ldots, b_{|B|}\}$, the set of different vectors we eventually have to represent.

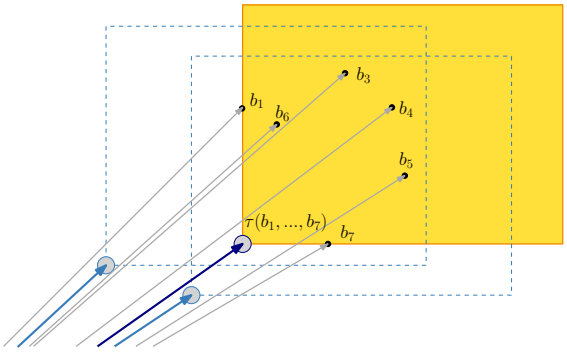

Fig. 4 The big gray dots are three of the possible representants for the tree arcs $b_1, \ldots, b_7$, but the standard template arc τ is the lower left point of the shaded rectangle. The rectangles depict the $\tilde{\delta}$-domain.

– $D(t) \subseteq \mathbb{D}$, the subspace of all vectors a particular template arc $t \in \mathbb{D}$ is able to represent when considering the restricted domain $\mathbb{D}'$ for the correction vectors, i.e.

$$D(t) = \{t^1, \ldots, (t^1 + \tilde{\delta}^1 - 1) \bmod \tilde{v}^1\} \times \ldots \times \{t^d, \ldots, (t^d + \tilde{\delta}^d - 1) \bmod \tilde{v}^d\}. \quad (2)$$

– $B(t) \subseteq B$, $t \in \mathbb{D}$, the subset of vectors from B that a particular template arc t is able to represent, i.e. $B(t) = \{b \in B \mid b \in D(t)\}$.

Furthermore, let $B' \subseteq B$, $B' \neq \emptyset$, be some subset of vectors from B. For each dimension $l = 1, \ldots, d$ assume the l-th elements (coordinates) of the vectors in B' are labeled by indices in a non-decreasing way, i.e. $b'^l_1 \leq b'^l_2 \leq \ldots \leq b'^l_{|B'|}$. Let $b'^l_0 = b'^l_{|B'|} - \tilde{v}^l$ for convenience. (Note that b'^l_0 can be negative.)

For such a B', we define the *standard template arc* (see also Fig. 4)

$$\tau(B') = (\tau^1(B'), \ldots, \tau^d(B')) \quad (3)$$

where

$$\tau^l(B') = b'^l_{i^*_l} \text{ with } i^*_l = \operatorname{argmax}_{i=1,\ldots,|B'|} b'^l_i - b'^l_{i-1} \quad \forall l = 1, \ldots, d. \quad (4)$$

The subspace $BB(B') = \{b^1_{i^*_1}, \ldots, b^1_{i^*_1 - 1} \bmod \tilde{v}^1\} \times \ldots \times \{b^d_{i^*_d}, \ldots, b^d_{i^*_d - 1} \bmod \tilde{v}^d\}$ is the smallest *bounding box* including all vectors from B' with respect to the ring structure.

To denote the limits of the bounding box $BB(B')$ in a simpler way, we further define $\hat{\tau}(B') = (b^1_{i^*_1 - 1}, \ldots, b^d_{i^*_d - 1})$, i.e. $\hat{\tau}(B')$ represents the corner point of the bounding box opposite to $\tau(B')$.

These definitions lead to the following lemma.

Lemma 1 *If a subset $B' \subseteq B$ of vectors can be represented by a single template arc, then the standard template arc $\tau(B')$ always is such a template arc.*

Proof This directly follows from the definition of $\tau(B')$, since this is the corner point with the smallest coordinates of the smallest bounding box of all vectors in B'. □

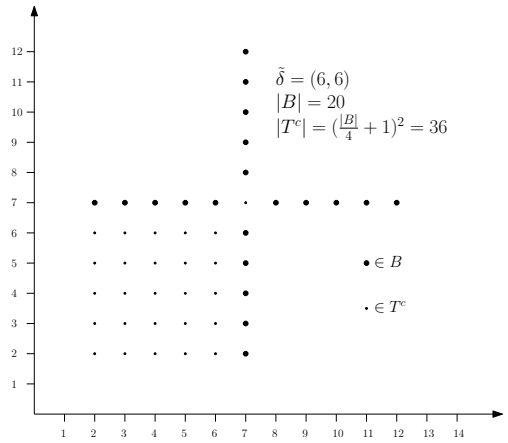

Fig. 5 Example for $|T^c| = \Theta(|B|^d)$ with $d = 2$.

We therefore can restrict all further considerations to the set of standard template arcs induced by all nonempty subsets of vectors that can be represented by a single template arc, i.e.

$$T = \{\tau(B') \mid B' \subseteq B, B' \neq \emptyset \wedge B' \subseteq D(\tau(B'))\}. \tag{5}$$

Lemma 2 *A set $B' \subseteq B$ can be represented by a single template arc, thus in particular by $\tau(B')$, if*

$$\tilde{v}^l - (b^l_{i^*_l} - b^l_{i^*_l - 1}) < \tilde{\delta}^l, \quad \forall l = 1, \ldots, d. \tag{6}$$

Proof Case 1: $i^* = 1$. In this case we have $(\tilde{v}^l - (b^l_1 - (v^l_{|B'|} - \tilde{v}^l)) = v^l_{|B'|} - b^l_1 < \tilde{\delta}^l, \forall l = 1, \ldots, d$. Case 2: $i^* > 1$. In this case the bounding box associated to $\tau(B')$ goes accross the domain border and the condition is $(\tilde{v}^l - (b^l_{i^*} - v^l_{i^*-1})) < \tilde{\delta}^l, \forall l = 1, \ldots, d$. □

Definition 1 (Domination of template arcs) Let $t' = \tau(B')$ and $t'' = \tau(B'')$, $B' \subseteq B, B'' \subseteq B$. Standard template arc t' dominates t'' if and only if $B'' \subset B'$.

From the set T we only need to keep the nondominated template arcs for our purpose, and call the resulting set T^c (candidate template arcs).

5.1 Bounds for the Number of Candidate Template Arcs

A lower bound on $|T^c|$ obviously is 1: in the best case, one template arc is able to represent all $b \in B$.

An upper bound is given by $O(|B|^d)$: Each standard template arc $t \in T^c$ is composed of d coordinates that are adopted from up to d vectors from B. This bound is tight as the worst-case example in Fig. 5 shows for $d = 2$. Bold dots represent the vector set B, small dots the nondominated standard template arcs T^c. Obviously, $|T^c| = (|B|/4+1)^2 = \Theta(|B|^2)$. The example can be extended to higher dimensions d and larger $|B|$ in a straight-forward way. In practice, however, we expect $|T^c| \ll \Theta(|B|^d)$.

Table 1 Basic data structures of the preprocessing algorithm.

Symbol	Purpose
C	Covered vectors (by current bounding box $(t, \hat{t})$)
E	Actively excluded vectors (must not be covered)
Ω	Open vectors
N	Newly covered vectors
F	Feasibly addable vectors

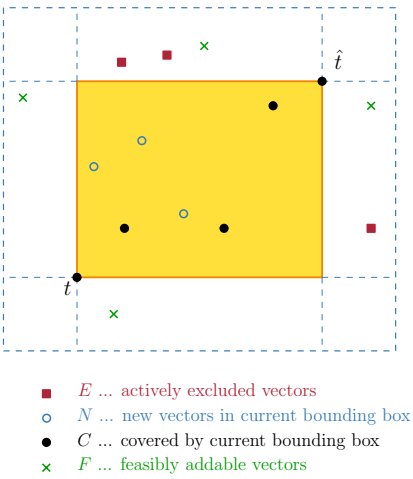

Fig. 6 Partitioning of the nodes during w.r.t. the current bounding box, defined by $(t, \hat{t})$.

5.2 An Algorithm for Determining T^c

We determine the set of candidate template arcs T^c by performing a restricted enumeration of all subsets $B' \subseteq B$, $B' \neq \emptyset$ that can be represented by their standard template arc $\tau(B')$. The algorithm maintains three disjunct index sets $C,\ E,\ \Omega \subseteq \{1, \ldots, |B|\}$ that represent at all time a partitioning of B, i.e. $B = B(C) \cup B(E) \cup B(\Omega)$, $C \cap E = \emptyset,\ E \cap \Omega = \emptyset,\ C \cap \Omega = \emptyset$. Hereby $B(S)$ is considered to be (arbitrarily) ordered and denotes the vectors in B referenced by the indices in S. Set C contains the indices of the vectors which are covered by a current bounding box represented by vectors t and $\hat{t}$, set E refers to the vectors that have been actively excluded and must not be covered, and Ω refers to the remaining, still "open" vectors. Table 1 summarizes these and a few local data structures.

The candidate template arc determination is started with the procedure `determine-`T^c (Algorithm 1), which performs the initialization of the global data structures and then calls the recursive procedure `recursive-`T^c`(var` C`,` t`,` $\hat{t}$`, var` E`, var` Ω`)` (Algorithm 2). The keyword `var` denotes that the respective variables are passed by *call-by-reference*. The overall procedure follows the subsequent principle.

1. find further vectors to be added to the current partial solution
2. – there are no further addable vectors ⇒ add current vector t to T^c
   – **otherwise:** recursive calls for all possible extensions of current partial solution

Algorithm 1: `determine-`T^c`()`

$T^c \leftarrow \emptyset$; $\Xi \leftarrow \emptyset$; $C \leftarrow \emptyset$; $E \leftarrow \emptyset$; $\Omega \leftarrow \{1, \ldots, |B|\}$
recursive-$T^c(C,\ 0,\ 0,\ E,\ \Omega)$
return T^c

Algorithm 2: `recursive-`T^c`(var` C`,` t`,` $\hat{t}$`, var` E`, var` Ω`)`

if $C = \emptyset$ **then**
    $N \leftarrow \emptyset$;
    $F \leftarrow \{1, \ldots, |B|\}$
else
    $N \leftarrow$ find-new-vectors-in-BB(t, $\hat{t}$,Ω)
    $C \leftarrow C \cup N$; $\Omega \leftarrow \Omega \setminus N$
    $F \leftarrow$ find-addable-vectors(t, $\hat{t}$, Ω)
end
if $F = \emptyset$ **then**
    `/* no further i (referencing vectors` b_i`) can be added to` C`; check if` C `is also maximal with respect to` E`, the actively excluded vectors */`
    if $\nexists j \in E \mid b_j^l \in \{(\hat{t}^l - \tilde{\delta}^l + 1) \bmod \tilde{v}^l, \ldots, (t + \tilde{\delta}^l - 1) \bmod \tilde{v}^l\},\ \forall l = 1, \ldots, d$ **then**
        $T^c \leftarrow T^c \cup \{t\}$
    end
else
    for $i \in F$ **do**
        `// Vectors` $B(C \cup \{i\})$ `can be represented by their` $\tau(B(C \cup \{i\}))$
        $C \leftarrow C \cup \{i\}$; $\Omega \leftarrow \Omega \setminus \{i\}$
        $(t', \hat{t}') \leftarrow$ `update-BB(`$t, \hat{t}, b_i$`)`
        `/* only perform further investigation if bounding box has not yet been considered */`
        if $\nexists j \in E \mid b_j^l \in \{t^l, \ldots, \hat{t}^l\},\ \forall l = 1, \ldots, d$ **then**
            recursive-$T^c(C,\ t',\ \hat{t}',\ E,\ \Omega)$
        end
        `/* in the next iteration of the loop, vector` b_i `is actively excluded */`
        $C \leftarrow C \setminus \{i\}$
        $E \leftarrow E \cup \{i\}$
    end
end
$C \leftarrow C \setminus N$
$E \leftarrow E \setminus F$
$\Omega \leftarrow \Omega \cup N \cup E$

Vectors t and $\hat{t}$ are assumed to represent the bounding box for the vectors referenced by C. In the first step, a further set N of references to vectors in B is determined, which are now covered by the bounding box, but which are still contained in Ω. These vectors are then directly moved from Ω to S, and no branching will occur on these vectors (actively excluding them would not make sense). Furthermore, the set F of open

variables (contained in Ω) which can be feasibly added to C (B'), hereby increasing the current bounding box up to size $\tilde{\boldsymbol{\delta}}$, is determined. Within the loop at line 15, the algorithm considers each element in F and adds it as next element to C. Procedure `update-BB(`t`, `$\hat{t}$`, `b_i`)` (Algorithm 3) updates the bounding box $(t, \hat{t})$ accordingly. The addition of further vectors is handled via recursion. Before the loop continues with the next vector from F, the current vector is moved from C to E, i.e. it is actively excluded from further consideration and must not be considered in subsequent recursive calls. At the end of the procedure, sets C, E, and Ω are restored to their initial states.

When F becomes empty, no further vectors are available for addition and the recursion terminates. We then check if a previously created template arc exists that dominates the current template arc. This is efficiently done by just considering all actively excluded vectors referred to by E. If one of them can be added to C, then C is not maximal and another template arc dominating the current one must have already previously been found.

According to the ring structure of the domain, terms of the form $b \in (l, \ldots, u)$, e.g. in line 8 or 13 in `update-BB(`t`, `$\hat{t}$`, `b_i`)` (Algorithm 3), have the following meaning: if $l \leq r$ it simply denotes $\{x \mid x \geq l, x \leq r\}$; otherwise the intervall goes across the domain border and thus the term denotes the values $\{x \mid x \geq 0, x \leq l\} \cup \{x \mid x \geq r, x \leq \tilde{v}\}$, where $\tilde{v}$ again denotes the domain border. To find new vectors, that are now covered by a just extended bounding box $(t, \hat{t})$ we use the procedure `find-new-vectors-in-BB(`t`, `$\hat{t}$`, var `Ω`)` (Algorithm 4). This procedure as well as `find-addable-vectors(`t`, `$\hat{t}$`, var `Ω`)` (Algorithm 5) run in time $O(|B| \cdot d)$.

Algorithm 3: `update-BB(`t`, `$\hat{t}$`, `b_i`)`

```
1  if |C| = 0 then
2      // b_i is the first vector in C
3      t' ← b_i, t̂' ← b_i
4  else
5      // calculate new (t', t̂') based on (t, t̂) and b_i
6      for l = 1, ..., d do
7          // we assume ∀l = 1, ..., d : δ̃^l ≤ ṽ^l/2
8          if b_i^l ∈ {(t̂^l − δ̃^l + 1) mod ṽ^l, ..., t^l} then
9              t'^l ← b_i^l
10         else
11             t'^l ← t^l
12         end
13         if b_i^l ∈ {t̂^l, ..., (t^l + δ̃^l − 1) mod ṽ^l} then
14             t̂'^l ← b_i^l
15         else
16             t̂'^l ← t̂^l
17         end
18     end
19     return (t', t̂')
20 end
```

Algorithm 4: `find-new-vectors-in-BB(`t`, `$\hat{t}$`, var `Ω`)`

```
for j ∈ Ω do
    if b_j^l ∈ {t^l, ..., t̂^l}, ∀l = 1, ..., d then
        N ← N ∪ {j}
    end
end
return N
```

Algorithm 5: `find-addable-vectors(`t`, `$\hat{t}$`, var `Ω`)`

```
F ← ∅
for j ∈ Ω do
    if b_j^l ∈ {(t̂^l − δ̃^l + 1) mod ṽ^l, ..., (t + δ̃^l − 1) mod ṽ^l}, ∀l = 1, ..., d then
        F ← F ∪ {j}
    end
    return F
end
```

Sets C, E, F, and N can efficiently be implemented by using simple arrays and corresponding variables for indicating the number of currently contained elements. In this way, the restoration of C and E at the end of `recursive-`T^c (Algorithm 2) can even be done by simply memorizing the array sizes at the beginning and finally resetting the counters.

In order to speed up the overall method, we use a k-d tree as data structure (see [1]) for maintaining Ω. In this way geometrical properties can be exploited, and not all vectors in Ω need to be explicitly considered each time.

Theorem 1 *The overall time complexity for determine-*T^c *is bounded above by* $O(d \cdot |B|^{3d})$.

Proof Let $\zeta = |B|^d$. As $t, \hat{t} \in T^c$ and $|T^c| = O(\zeta)$ there are $\frac{\zeta!}{2(\zeta-2)!}$ possible bounding boxes $(t, \hat{t})$, and therefore $O(\zeta^2)$ recursive calls in the worst case. As the worst case runtime of the first part of the algorithm is $O(d \cdot |B|)$, we get an overall worst case time complexity of $O(d \cdot |B|^{3d})$. □

Note that the running time $O(d \cdot |B|^{3d})$ to enumerate a set of maximal cardinality $O(|B|^d)$ results from the necessity to remove all dominated elements, as described above. For our application we assume $\tilde{\boldsymbol{\delta}}$ to be relatively small, which implies that $B(t)$ will be small as well, i.e. one template arc typically just represents a small number of arcs. Hence, the running times of the procedure are much lower in practice.

6 An Exact Branch-and-Cut Algorithm for Solving k-MLSA

In order to solve the k-MLSA problem to optimality, we consider a branch-and-cut algorithm for the following formulation as an integer linear program (ILP).

6.1 ILP Formulation

To be able to choose the root node of the arborescence by optimization we extend V to V^+ by adding an artificial root node 0. Further we extend A to A^+ by adding the arcs $(0, i),\ \forall i \in V$. We use the following variables for modelling the problem as an ILP:

- For each candidate template arc $t \in T^c$, we define a variable $y_t \in \{0, 1\}$, indicating whether or not the arc is part of the dictionary T.
- Further we use variables $x_{ij} \in \{0, 1\},\ \forall (i, j) \in A^+$, indicating which arcs belong to the tree.
- To express which nodes are covered by the tree, we introduce variables $z_i \in \{0, 1\},\ \forall i \in V$.

Let further $A(t) \subset A$ denote the set of tree arcs a template arc $t \in T^c$ is able to represent, and let $T(a)$ be the set of template arcs that can be used to represent an arc $a \in A$, i.e. $T(a) = \{t \in T^c \mid a \in A(t)\}$. We can now model the k-MLSA problem as follows:

$$\text{minimize} \quad m = \sum_{t \in T^c} y_t \tag{7}$$

$$\text{s.t.} \sum_{t \in T(a)} y_t \geq x_a, \qquad \forall a \in A \tag{8}$$

$$\sum_{i \in V} z_i = k \tag{9}$$

$$\sum_{a \in A} x_a = k - 1 \tag{10}$$

$$\sum_{i \in V} x_{(0,i)} = 1 \tag{11}$$

$$\sum_{(j,i) \in A^+} x_{ji} = z_i \qquad \forall i \in V \tag{12}$$

$$x_{ij} \leq z_i \qquad \forall (i, j) \in A \tag{13}$$

$$x_{ij} + x_{ji} \leq 1 \qquad \forall (i, j) \in A \tag{14}$$

$$\sum_{a \in C} x_a \leq |C| - 1 \qquad \forall \text{ cycles } C \text{ in } G,\ |C| > 2 \tag{15}$$

$$\sum_{a \in \delta^-(S)} x_a \geq z_i \qquad \forall i \in V,\ \forall S \subseteq V,\ i \in S,\ 0 \notin S \tag{16}$$

Inequalities (8) ensure that for each used tree arc $a \in A$ at least one valid template arc t is selected. Equalities (9) and (10) enforce the required number of nodes and arcs to be selected. Equation (11) requires exactly one arc from the artificial root to one of the tree nodes, which will be the actual root node of the outgoing arborescence.

Equations (12) state that selected nodes must have in-degree 1. Inequalities (13) ensure, that an arc may only be selected if its source node is selected as well. Inequalities (14) forbid cycles of length 2, and finally inequalities (15) forbid all further cycles ($|C| > 2$).

In order to strengthen the ILP we can additionaly add (directed) connectivity-constraints, given by inequalities (16), where $\delta^-(S)$ represents the ingoing cut of node

set S. These constraints ensure the existence of a path from the root 0 to any node $i \in V$ for which $z_i = 1$, i.e. which is selected for connection. In principle, equations (16) render (12), (13), (14) and (15) redundant [16], but using them jointly may be beneficial in practice.

6.2 Branch-and-Cut

As there are exponentially many cycle elimination and connectivity inequalities (15) and (16), directly solving the ILP would be only feasible for very small problem instances. Instead, we apply branch-and-cut [25], i.e. we just start with the constraints (8) to (14) and add cycle elimination constraints and connectivity constraints only on demand during the optimization process.

The cycle eliminiation cuts (15) can be easily separated by shortest path computations with Dijkstra's algorithm. Hereby we use $1 - x_{ij}^{\mathrm{LP}}$ as the arc weights with x_{ij}^{LP} denoting the current value of the LP-relaxation for (i, j) in the current node of the branch-and-bound tree. We obtain cycles by iteratively considering each edge $(i, j) \in A$ and searching for the shortest path from j to i. If the value of a shortest path plus x_{ij}^{LP} is less than 1, we have found a cycle for which inequality (15) is violated. We add this inequality to the LP and resolve it. In each node of the branch-and-bound tree we perform these cutting plane separations until no further cuts can be found.

The directed connection inequalities (16) strengthen our formulation. Compared to the cycle elimination cuts they lead to better theoretical bounds, i.e. a tighter characterization of the spanning-tree polyhedron [16], but their separation usually is computationally more expensive. We separate them by computing the maximum flow from the root node to each of the nodes with $z_i > 0$ as target node. We separate them by computing for each node with $z_i > 0$ a minimum $(0, i)$-cut. If the value of this cut is less than z_i^{LP}, we have found an inequality that is violated by the current LP-solution. Our separation procedure utilizes Cherkassky and Goldberg's implementation of the push-relabel method for the maximum flow problem [4] to perform the required minimum cut computations.

The branch-and-cut algorithm has been implemented using C++ with CPLEX in version 11.0 [12].

7 Heuristic Methods

Practical results of the described branch-and-cut algorithm are presented in Section 9. They show that this approach is only applicable for small instances and requires relatively long running times. Therefore, we now describe a fast greedy construction heuristic and then focus on metaheuristics including a greedy randomized adaptive search procedure (GRASP) and a memetic algorithm (MA).

7.1 Greedy Construction Heuristic

Based on the so called MVCA heuristic for the classical MLST problem [3], we developed a greedy construction heuristic for our k-MLSA problem. A solution is constructed by starting with an empty codebook T and graph $G' = (V', A')$ with $A' = \emptyset, V' = \emptyset$

and iteratively adding template arcs from T^c to T in a greedy fashion. In the following we will treat T as an ordered set, and refer to its elements by $T[i], i = 1, \ldots, |T|$. Each time a template arc t is added to T, all corresponding induced arcs $A(t) \subset A$ are added to A'. For each arc (i, j) we also add the corresponding nodes i and j to V'. This is done until the resulting graph contains a feasible k-node arborescence. In constrast to the MVCA heuristic for the classical undirected MLST problem, the decision which template arc (label) to take next is significantly more difficult, as the impact of the addition of one template arc towards a final arborescence (with some specific root node) is not immediately obvious.

In the MVCA heuristic for MLST, a label that reduces the number of separated components the most is always chosen. The number of components of G' minus one corresponds to the number of edges that must be added at least to obtain a complete spanning tree, and this number of edges is an upper bound for the number of labels to be added.

In any case, a label which directly yields a spanning tree is an optimal choice and should be selected. A label which yields a G' to which only one more edge must be added is always the second best choice, since exactly one more label is necessary. Note that any such situation is equally good. In general, the assumption is that a label yielding a G' to which a lower number of further edges must at least be added will usually lead to a better solution (requiring less labels) than a label yielding a G' having a higher lower bound of edges to be necessarily added.

While the notion of simple components does not make sense in the directed k-MLSA anymore, we can still follow the idea of determining the number of edges (arcs) that must at least be added to obtain a complete arborescence in order to decide upon the next label to be added.

Let $\alpha(G')$ denote the minimum number of arcs that need to be added, so that this augmented graph contains an arborescence. In principle, $\alpha(G')$ can be calculated efficiently as follows: Determine all maximal strongly connected components (SCCs) in G' and shrink them into corresponding representative single nodes. Arcs to or from a node in a strongly connected component are replaced by corresponding arcs to/from the representative node. Multiple arcs between two nodes are replaced by corresponding single arcs, and self-loops are simply deleted. By this transformation, we obtain a directed acyclic graph G_s. The problem is reduced, but the value $\alpha(G')$ will remain the same since within each strongly connected component, any node can be reached from each other and no further edges will therefore be necessary. It further does not matter to which particular node of a strongly connected component an ingoing or outgoing arc is connected. Let $Z \subseteq V$ be the set of nodes for which no ingoing arc exists in G_s. The minimum number of required additional arcs is now $\alpha(G') = k - (|V| - |Z| + 1)$, and the label that minimizes this number the most is considered the best choice. Figure 7 shows an example of the computation and usage of $\alpha(G')$.

We do not need to explicitly shrink the SCCs each time if we keep track of all SCCs with indegree zero. Algorithm 6 details the overall procedure.

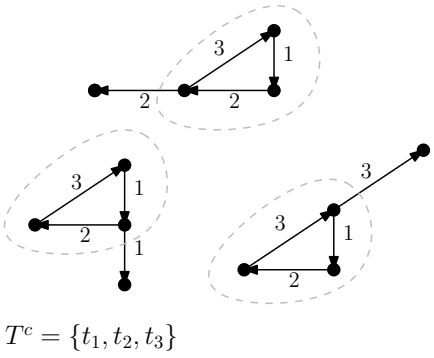

Fig. 7 Suppose we want to connect all of the 12 nodes in the given graph G, i.e. $k = 12$. After adding the template arcs t_1, t_2 and t_3 we can identify three nontrivial strongly connected components (SCCs), i.e. components consisting of more than one node. All nodes have incoming arcs, but these three SCCs do not. We need to add at least two more arcs as $\alpha(G') = 2$. Hereby G' denotes the graph where each SCC is shrinked into one single node.

Algorithm 6: k-`MLSA-greedy(V,A,` T^c`)`

```
1  G' = (V', A') with V' ← V, A ← ∅
2  T ← ∅    // currently used labels
3  while no k-arborescence exists do
4      t* ← 0    // best template arc of this iteration
5      for all t ∈ T^c do
6          z* ← ∞    //stores lowest found number of SCCs
7          A'' ← {a_ij ∈ A(t)}
8          compute SCCs of G' = (V', A' ∪ A'')
9          Z ← SCCs with indegree zero
10         if |Z| < z* then
11             z* ← |Z|
12             t* ← t
13         end
14     end
15     A' ← A' ∪ {a_ij ∈ A(t*)}
16     T ← T ∪ {t*}
17     T ← T\{t*}
18 end
19 remove redundant arcs and labels
```

Obviously, the algorithm frequently has to check if a partial solution already contains a feasible arborescence. This task can be achieved by performing depth first search (DFS) using each node as start node (time complexity $O(k^3)$). To achieve a speedup of this method we try to avoid or reduce the number of time consuming DFS calls. Let G' denote the graph containing just the edges and nodes induced by some template arc set T, i.e. if $(i, j) \in A$ is represented by template arc $t \in T$ we add the nodes i, j and the arc (i, j) to $G' = (V', A')$. Let further $\delta^-(v)$ denote the in-degree of a node v, i.e. the number of incoming arcs. Furthermore let $\delta_0^i(V')$ denote the subset of nodes from V' with $\delta^-(V') = 0$, and let us assume that the current partial solution consists of the

template arcs (labels) T. First, we check the degree of each node to see if a sufficient number of nodes v with in-degree $\delta^-(v) > 0$ is available. If $|V'| - \delta_0^i(V') + 1 < k$ then G' cannot represent a valid solution, and we do not have to perform the DFS. If a solution is possible we distinguish the following two cases. In the first case, where $k = |V|$, there can be at most one node with in-degree zero. If there is such a node it has to be the root node and we perform the DFS starting from this node. Otherwise, if all nodes $v \in V'$ have $\delta^-(v) > 0$ we have no choice but to perform DFS starting from all nodes. In the more general second case $k < |V|$, if $|V'| - \delta_0^i(V') + 1 = k$, one of the nodes with in-degree zero has to be the root of the tree, otherwise the tree would not contain the required k nodes. So it is sufficient to perform the DFS starting at just these $\delta_0^i(V')$ nodes. Otherwise we again have to perform DFS starting from all nodes.

The final step is to remove redundant tree arcs and redundant template arcs. Because of mutual dependencies of these tasks, this is a non-trivial operation itself and hence we apply a heuristic. As long as the solution remains valid we perform the following two steps: 1) try to remove redundant labels; 2) as long as $|A'| > k$ try to remove the leaves and intermediate arcs. By this procedure we finally obtain a valid k-node arborescence.

7.2 GRASP – Greedy Randomized Adaptive Search Procedure

The greedy heuristic is relatively fast but yields only moderate results. Significantly better solutions can be achieved by extending it to a *greedy randomized adaptive search procedure* (GRASP) [9]. The constructive heuristic is iterated and the template arc to be added is always selected at random from a restricted set of template arcs, the restricted candidate list (RCL). As soon as a valid solution exists, it is further improved by a local search procedure. In total, *it* iterations of these two steps are performed.

Function k-`MLSA-randomized-greedy`(V, A, T^c) (Algorithm 7) shows the randomized greedy construction of solutions in detail. One crucial part in designing an efficient GRASP is to define a meaningful RCL. The problem in our case is that there are many equally good template arcs that could be candidates to extend the current partial solution. On the other hand, finding the best template arcs, i.e. those template arcs reducing α of the current partial solution the most, can be very time consuming, as all candidate template arcs need to be considered. As GRASP also heavily relies on the subsequent local improvement, we do not necessarily need to find the best candidates to extend our partial solution. On the other hand, being too lazy with this decision might reduce the overall performance considerably. In the following we describe the parameterized procedure of building up the RCL in more detail.

Prior to each extension of the current partial solution (line 26), the RCL is built in the loop in lines 5 to 25. As soon as a further improving template arc is found (line 11) the RCL is cleared, and then successively filled with further template arcs of the same quality. This finally yields a list of equally good template arcs, which are after all the candidates for the next greedy decision. The size of this list is limited by $rcl_{\max}$ for performance reasons. There is one further, even more important parameter related to the issue of balancing the greedy solution quality versus run time efficiency of the process of building the RCL. The parameter $imp_{\max}$ limits the number of improvements according to line 11. In the special case of $imp_{\max} = 0$ the RCL finally simply consists of the first template arcs improving the current solution. In this case the major contribution to construct high quality solutions is passed to the subsequent local

Algorithm 7: k-MLSA-randomized-greedy(V,A, T^c)

```
1  G' = (V', A') with V' ← V, A ← ∅
2  T ← ∅   // currently used labels
3  while no k-arborescence exists do
4      i ← 0
5      for all t ∈ T^c do
6          z* ← ∞   //stores lowest found number of SCCs
7          A'' ← {a_ij ∈ A(t)}
8          compute SCCs of G' = (V', A' ∪ A'')
9          Z ← SCCs with indegree zero
10         if |Z| < z* then
11             z* ← |Z|
12             RCL = ∅
13             i ← i + 1
14         end
15         if |Z| = z* then RCL = RCL ∪ t
16         if |RCL| ≥ rcl_max then
17             z* ← z* − 1
18             if i ≥ imp_max then
19                 break
20             end
21         end
22     end
23     t' ← random element from RCL
24     A' ← A' ∪ {a_ij ∈ A(t')}
25     T ← T ∪ {t'}
26     T^c ← T^c\{t'}
27 end
28 remove redundant arcs and labels
```

search. Setting $imp_{\max} = \infty$ implies a situation where the loop of line 6 iterates over all candidate template arcs *each* time. Due to the relatively large number of candidate template arcs this approach may be to time consuming for practice.

In each GRASP iteration, it_{ls} local search steps are performed after the randomized construction. The local search uses a template arc insertion neighborhood, where a new template arc is added to the solution, and then redundant template arcs are removed. The goal is to find template arcs that render at least two template arcs from the current solution redundant. Figure 8 shows such a situation. Another beneficial situation arises when further nodes are connected to the existing arborescence. In each iteration the template arcs are considered in decreasing order w.r.t. the number of tree arcs they represent. The neighborhood is searched with a first improvement strategy. Furthermore only a prespecified fraction of the template arcs is used, i.e. the template arcs representing most of the tree arcs.

7.3 Memetic Algorithm

As an alternative to GRASP we implemented a memetic algorithm (MA). It is based on a steady-state framework, where in each iteration a single offspring solution is derived and locally improved. It replaces a randomly chosen candidate solution from the population, to retain diversity. The algorithm uses tournament selection, and local improvement steps are performed for each new candidate solution after the application

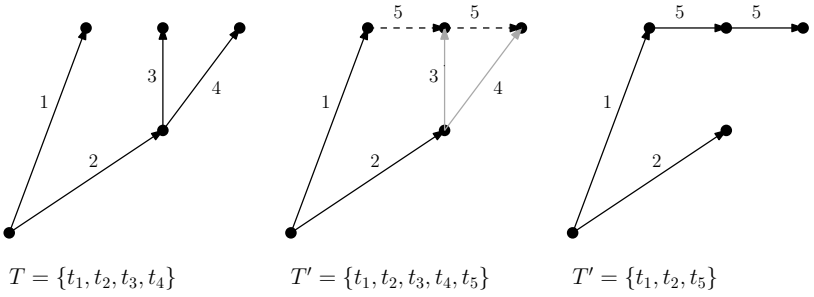

Fig. 8 Template arc insertion neighborhood: after t_5 is added to the solution t_3 and t_4 are redundant and thus can be removed from the solution

of the evolutionary operators, i.e. recombination and mutation. Algorithm 8 shows the overall framework.

Algorithm 8: k-MLSA-MA()

```
randomly create initial population
t ← 0
while t < t_max do
    select parents T' and T'' by tournament selection
    T ← crossover(T', T'')
    mutation(T)
    local improvement(T)
    t ← t + 1
end
```

Following the ideas presented in [26] we encode a candidate solution as an ordered subset of labels. In our case the template arcs correspond to these labels and the chromosome of a candidate solution is therefore denoted by T, $T[i]$ denotes the i-th template arc of candidate solution T. If these template arcs induce a k-node arborescence we have a feasible solution, otherwise further template arcs need to be added to the candidate solution in order to make the solution feasible. Note however, that a feasible solution may contain redundant template arcs, which are not necessarily part of an optimal solution induced by the other template arcs of the ordered set.

For candidate solutions of the initial population we ensure that they are feasible. To create a randomized candidate solution, all template arcs are shuffled and then added as long as the candidate solution remains infeasible.

The MA then tries to minimize the number of template arcs required for a feasible solution by iterative application of the genetic operators and local improvement. As many candidate solutions have the same number of template arcs, the total number of induced arcs is also considered in the fitness function $f(T)$, which is going to be minimized:

$$f(T) = |T| + \left(1 - \frac{|A'|}{|A|}\right). \tag{17}$$

Again, A' denotes the set of induced tree arcs. This accounts for the fact that candidate solutions whose template arcs cover many arcs are more likely to produce good offsprings and result in successful mutations.

Since the order of the template arcs does not need to be preserved, we use a crossover operator introduced in [19], which takes the template arcs for the child candidate solution alternatingly from the parents until a feasible solution is obtained. If a template arc reoccurs, it is not added to the offspring and the next template arc from the other parent is processed instead. Function `crossover(`T'`,` T''`)` (Algorithm 9) shows this procedure in detail. Again T denotes the (ordered) set of template arcs of an candidate solution, $T[i]$ denotes the i-th template arc.

Algorithm 9: `crossover(`T'`,` T''`)`

```
1  T ← ∅  // new offspring initialized with empty set
2  i ← 0, j ← 0  // counter variables
3  while T contains no k-MLSA do
4      if i mod 2 = 0 then
5          t ← T'[⌊i/2⌋]
6      else
7          t ← T''[⌊i/2⌋]
8      end
9      if t ∉ T then
10         T[j] ← t
11         j ← j + 1
12     end
13     i ← i + 1
14 end
15 return T
```

In addition to recombination we use two different types of mutation:

1. A randomly selected template arc $t \notin T$ is appended. This increases the likelyhood for the ability to remove some redundant template arc by a subsequent local improvement.
2. A randomly selected template arc $t \notin T$, replaces either a random or the worst $t' \in T$. The worst template arc is the one inducing the minimal number of arcs. If the solution is not feasible, further randomly selected template arcs are added until a feasible solution is reached.

The subsequent local improvement method `local-improvement(`T`)` (Algorithm 10), following the one presented in [19], uses the idea that a reordering of the template arcs could make some of them redundant. In contrast to the local improvement method used in the GRASP algorithm this method can only remove template arcs from a current solution if some of them are actually redundant. As the MA continuously modifies the candidate solutions from the population and also further template arcs are added to a candidate solution by mutation, there is no need to use a more expensive neighborhood search, which also considers currently unused template arcs.

Algorithm 10: `local-improvement(`T`)`

```
1  i ← 0  // counter variable
2  while i < |T| do
3      remove all arcs only labeled by T[i]
4      if T containts k-MLSA then
5          T ← T \ T[i]
6      else
7          restore respective arcs
8          i ← i + 1
9      end
10 end
```

The MA uses the same optimizations as the (randomized) greedy construction heuristics (described in Sections 7.1 and 7.2) to reduce the number of DFS calls. Furthermore it only checks for a k-node arborescence if the number of different nodes reaches the required size k, because a k-node arborescence is not theoretically possible before that.

Our computational experiments showed that the MA is able to produce optimal solutions very quickly and in almost every case. Details on the results are given in Section 9.4.

8 Encoding of the Compressed Templates

In the following we describe how the compressed templates will be encoded on a binary level. The compression results from Section 9.2 are based on the definitions given in this section.

We now need to extend the definition of the domain border given in Section 3. There, the domain border $\tilde{\mathbf{v}}$ was defined by $\tilde{v}^l = \max_{i=1,\dots,n} v_i^l,\ l = 1, \dots, d$. For the concrete specification of the original and the resulting compressed data structure we need to distinguish between the domain of one particular instance ($\tilde{\mathbf{v}}$), and a further entity, describing the size of the domain considering all possible input instances.

Definition 2 The overall domain border $\tilde{\boldsymbol{\xi}}$ is given by

$$\tilde{\xi}^l = \max_I \tilde{v}_I^l,\ l = 1, \dots, d, \tag{18}$$

where the maximum goes over all input instances I.

To see the necessity to distinguish between $\tilde{\boldsymbol{\xi}}$ and $\tilde{\mathbf{v}}$, reconsider the preprocessing, i.e. the determination of the labels. For the further optimization it is obviously beneficial when single template arcs represent many tree arcs w.r.t. $\boldsymbol{\delta}$. Figure 9 shows the representation of the arc (i, j) using the domains $\tilde{\mathbf{v}}$ and $\tilde{\boldsymbol{\xi}}$ respectively. Unlike $(i, j) \bmod \tilde{\boldsymbol{\xi}}$, $(i, j) \bmod \tilde{\mathbf{v}}$ may be covered by a template arc in the domain $\tilde{\mathbf{v}}$ together with some other tree arcs in this domain. For arcs $a_{ij} > \tilde{\boldsymbol{\xi}} - \tilde{\boldsymbol{\delta}}$ it is not even possible to represent them together with arcs from the domain $\tilde{\mathbf{v}}$. Hence, an optimal solution to the k-MLSA problem may require more template arcs when using $\tilde{\boldsymbol{\xi}}$ instead of $\tilde{\mathbf{v}}$ as parameter for the preprocessing routine. As the values $\tilde{\mathbf{v}}$ often significantly deviate from $\tilde{\boldsymbol{\xi}}$ this effect is not negligible. Since m has a high impact on the compression ratio, we use $\tilde{\mathbf{v}}$ instead of $\tilde{\boldsymbol{\xi}}$ and accept the resulting disadvantage that we have to store $\tilde{\mathbf{v}}$ for each compressed template.

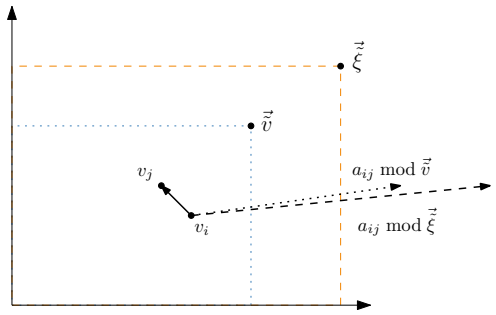

Fig. 9 Representation of the arc $a_{ij} = (v_i, v_j)$ using the domains $\tilde{\mathbf{v}}$ and $\tilde{\boldsymbol{\xi}}$ respectively.

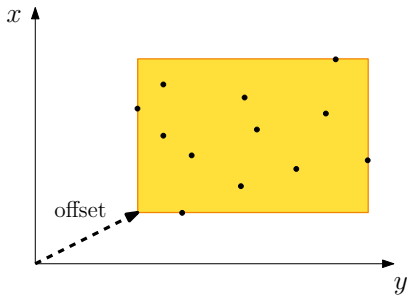

Fig. 10 The points v_i are scattered in the shaded rectangle. Commonly used encodings of such data points indicate their common offset (dashed arrow) and the respective relative coordinate values of the points themselves. A direct encoding would require unnecessary many bits.

In our experimental evaluations (Section 9) we will compare our compressed data to the following size (in bits) of the original raw data

$$\lambda_{\text{raw}} = size(\textsc{ConstData}) + n \cdot \sum_{l=1}^{d} \lceil \text{ld } \tilde{\xi}^l \rceil, \tag{19}$$

which is the size of the constant data and n times the number of bits to store one particular point. The $size()$ operator denotes the number of bits needed to encode the data given as argument.

The variable ConstData denotes additional data of constant size that is related to the data points. E.g. if we have offset values (see Fig. 10) for each dimension we need this data to achieve a lossless compression of the data points themselves.

As the algorithms may be applied to a subset of the dimensions, we need the following function to define the total encoding length:

$$\chi^l = \begin{cases} 1 & \text{dimension } l \text{ is considered by the compression method} \\ 0 & \text{otherwise.} \end{cases} \tag{20}$$

The total encoding length of the arborescence (achieved by the compression procedure) is given by the following formula:

$$\lambda(m,\boldsymbol{\delta},k,\tilde{\mathbf{v}},\boldsymbol{\xi},\boldsymbol{\chi}) = {}_{size}\big(\textsc{ConstData}'\big) + \underbrace{2\cdot 7}_{\text{values } k,m} + \underbrace{2\cdot\sum_{l=1}^{d}\lceil \chi^l \text{ ld } \tilde{\xi}^l\rceil}_{\text{root node, domain } \tilde{\mathbf{v}}}$$
$$+ \underbrace{2\cdot(k-1)}_{\text{encoding of tree structure}} + \underbrace{\lceil m\cdot\sum_{l=1}^{d}\chi^l \text{ ld } \tilde{v}^l\rceil}_{\text{template arcs}}$$
$$+(k-1)\cdot\Big\lceil \underbrace{\text{ld } m}_{\text{index to template arc}} + \underbrace{\sum_{l=1}^{d}\chi^l \text{ ld } \tilde{\delta}^l}_{\tilde{\delta}-\text{values}} + \underbrace{\sum_{l=1}^{\tilde{d}}(1-\chi^l) \text{ ld } \tilde{v}^l}_{\text{remaining dimensions}}\Big\rceil \tag{21}$$

Note that ConstData of the compressed data is not necessarily the same as the corresponding entity of the raw data. For instance this can be the case if the raw data does not contain explicit offset values, but however $\exists l \in \{1,\ldots,d\} \mid \min_{i=1,\ldots,n} v_i^l > 0$. We account for this by drawing a distinction between the constant data of the raw data (ConstData) and that of the compressed data, namely ConstData$'$.

The second term in equation (21) constitutes 14 bits for the storage of k and m; the third term denotes the number of bits that are necessary to store the root node and the size of the domain $\tilde{\mathbf{v}}$. The next term represents the encoding of the tree structure. The encoding is based on the parenthesis theorem of the depth first search (DFS) algorithm [7]. If we travers the resulting arborescence by DFS and write a "(" each time we traverse an arc forward, and write ")" each time we traverse an arc backward the resulting string is an representation of the structure of the DFS-tree. In our case we simply write "0"s and "1"s instead of "("s and ")"s, thus requiring $(k-1)\cdot 2$ bit in total. The next term in equation (21) constitutes the size of the m template arcs. We only need to store the components that are indicated by the characteristic function χ^l, as we do not consider the other dimensions for compression, but directly store their values instead. The term ld $\tilde{v}^l$ denotes the number of bits that are necessary to store the respective component of the template arc. The last term in equation (21) describes the size of the encoding of the $(k-1)$ tree arcs. Their representation consists of an index to a template arc, the appropriate correction vectors and finally the components of the remaining dimensions. Note that it is sufficient to round up the whole last term in equation (21) (and not each individual logarithm in the respective sums), because it is always possible to find such an appropriate encoding. For this purpose consider the following example, where we want to encode values of the following domain: $\{0\ldots4\}\times\{0\ldots8\}\times\{0\ldots17\}$. The number of bits necessary to encode each individual dimension are 3, 4 and 5 respectively, which yields 12 in total. Contrary a more tight encoding would use the representation $n = c_1 + c_2\cdot 18 + c_3\cdot(18\cdot 9)$, which just requires 10 bit in total.

8.1 Encoding Example

Let $\tilde{\boldsymbol{\xi}} = (512,512,512)^T$, $\tilde{\boldsymbol{\delta}} = (5,5)^T$ and $k = 9$. As $\tilde{\boldsymbol{\delta}}$ is only two-dimensional we do not consider the third dimension of the input data for compression. Instead we simply

store the respective values for each tree arc. The input data is given by the following set:

$$\left\{ \begin{pmatrix} 200 \\ 200 \\ 21 \end{pmatrix}, \begin{pmatrix} 208 \\ 304 \\ 30 \end{pmatrix}, \begin{pmatrix} 211 \\ 386 \\ 97 \end{pmatrix}, \begin{pmatrix} 261 \\ 356 \\ 210 \end{pmatrix}, \begin{pmatrix} 313 \\ 330 \\ 293 \end{pmatrix}, \begin{pmatrix} 314 \\ 409 \\ 22 \end{pmatrix}, \begin{pmatrix} 503 \\ 252 \\ 268 \end{pmatrix}, \begin{pmatrix} 608 \\ 280 \\ 157 \end{pmatrix}, \begin{pmatrix} 414 \\ 356 \\ 77 \end{pmatrix}, \begin{pmatrix} 662 \\ 332 \\ 104 \end{pmatrix}, \begin{pmatrix} 702 \\ 676 \\ 78 \end{pmatrix} \right\}.$$

Thus the offsets are 200 for the first and second coordinate and the domain borders are $\tilde{\mathbf{v}} = (503, 477, 294)^T$. Figure 11 shows a solution to the given problem. The trivial encoding requires 243 bit, whereas the encoded template has a size of 232 bit; ConstData' $= 14 + 27 + 27 + 16 = 84$. The resulting compression ratio is not very impressive, but in the example only two dimensions have been considered for compression and $\tilde{\boldsymbol{\delta}}$ is extremely small. Beeing able to reconstruct the original datapoints losslessly would require additional $\mathrm{ld}(\xi_1) + \mathrm{ld}(\xi_2) = 18$ bits for the offsets.

9 Experimental Results

This section starts by a description of the input data used for our computational experiments. Then we present the results of the exact method in order to analyse the compression ratios achievable by our methods. Furthermore we shortly evaluate the impact of the compression to the false non-match rate (FNMR). Finally we present the results of the metaheuristic techniques (greedy algorithm, GRASP, and the memetic algorithm) in comparison to the optimal results of branch-and-cut. All running times in this section refer to a 2.4 GHz Opteron processor with 4 GB RAM.

9.1 Test Instances

For our tests we use two different data sets. The first set of 20 templates was provided by the Fraunhofer Institute Berlin and is in the following referred to as *Fraunhofer Templates*. In addition we use a minutia data set from the U.S. National Institute of Standards and Technology [10]. For the instances we use the prefix *ft* and *nist*, respectively.

The Fraunhofer data set contains 20 templates which are multiple scans of four different fingers from two persons. The encoding of the points is $\tilde{\xi} = (\xi_x, \xi_y, \xi_\theta, \xi_{\mathrm{type}})^T = (2^9, 2^9, 2^9, 2^1)^T$. The size of ConstData is 14 bit, i.e. 7 bit for the offset value for each spatial dimension respectively. The templates consist of 15 to 40 minutia which corresponds to a typical amount of minutia detected by an electronic fingerprint scanner. The templates are listed in the first part of Table 3, i.e. ft-01 to ft-20. The full name matches the pattern P[0-9999]_F[0-99]_R[0-99] where P abbreviates "Person", F "Finger", and R "Release". One can see, that various releases of the same finger involve significant differences to the number of minutia that are detected. The second part of the table lists the templates from the NIST data set. From the large set of NIST Templates we selected a subset for our experiments, see Table 3. We chose five templates from each of the categories *ugly*, *bad* and *good*. The instance names reflect this classification. Furthermore for each fingerprint, there exists minutia data to a *latent* and a corresponding *tenprint* image. Latent refers to fingerprints on e.g. crime scenes that are invisible to the eye and require some type of chemical processing or dusting to make them visible. Fingerprint images that are created by inking and

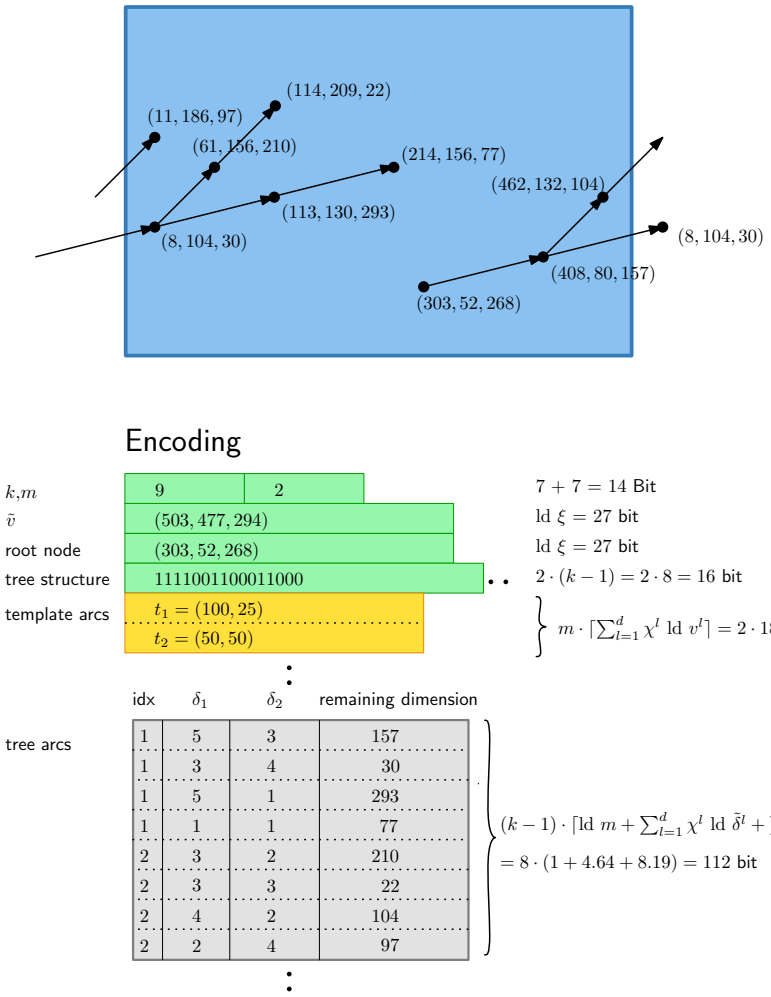

Fig. 11 This figure shows a concrete encoding example. The first block basically contains information to be able to process the following blocks. It is followed by the list of the templace arcs. This can be compared to a dictionary or codebook of traditional compression methods. The block on the bottom contains the actual tree information, i.e. a list of arcs encoded by an index to one of the template arcs, the respective correction vectors, and finally the values of the dimensions which are not considered for compression. The black dots indicate that the size of the respective (sub)blocks is not known in advance, because it depends on output values of the compression algorithm (like the number of template arcs m).

rolling fingertips onto a paper or some scanning device and traditionally have been captured of all ten fingers are usually referred to as tenprints. Obviously the quality of tenprints is superior to the latents, which typically just consist of a few dozen minutiae (min=5, max=82, avg=20.55, std-dev=13.25). The tenprints have between 48 and 193 minutiae (avg = 106.3, std-dev=25), which is a significantly higher number than we can expect to get from an electronic fingerprint scanning device. None the less, in our experiments we only use the tenprint data, as the latents do not contain enough

Table 2 Characterization of the two template data sets

	Fraunhofer	NIST
avg($\|V\|$)	30.75	96.47
min($\|V\|$)	15	72
max($\|V\|$)	40	120
$\tilde{v}_{\text{avg}}$	$(286, 383, 358, 2)^T$	$(3993, 3368, 359, 2)^T$
$\tilde{v}_{\text{min}}$	$(129, 191, 252, 2)^T$	$(2936, 2281, 359, 2)^T$
$\tilde{v}_{\text{max}}$	$(224, 287, 312, 2)^T$	$(3293, 2788, 353, 2)^T$

minutiae and the larger size of the tenprints enables us to test the performance of our method concerning higher numbers of data points. The encoding of the NIST points is $\tilde{\xi} = (\xi_x, \xi_y, \xi_\theta, \xi_{\text{type}})^T = (2^{12}, 2^{12}, 2^9, 2^1)^T$. The size of ConstData is 33, which are the offsets for the dimensions x, y and θ.

Column ρ^* shows the best compression ratios that could be achieved by our computational experiments, the column right to it show the respective parameter values that yielded the result. The results for the Fraunhofer templates are exact ones, whereas the results for the NIST templates are results of metaheuristics. Table 2 gives an overview of the characteristics of the two data sets.

9.2 Compression Results

Table 3 gives an overview of the data instances used for our experimental evaluations, and the corresponding best results. Besides the number of nodes and resulting tree arcs we list the best compression ratios we could achieve by our method together with the respective parameter values. The compression ratios ρ are defined by

$$\rho\ [\%] = 100 - \frac{100 \cdot \lambda_{\text{raw}}}{\lambda_{\text{enc}}}, \tag{22}$$

where λ_{raw} refers to equation (19) of the k selected points and λ_{enc} to equation (21). As ρ does not reflect the size of the compressed template compared to the full template, but only to the trivial encoding of the k selected points, we will also refer to ρ as *relative compression ratio*. We denote the best found compression ratio by ρ^*. Though adequate for our application background, we do not set $\text{ConstData}' = 0$ for the subsequent experiments, in order to evaluate the compression ratios more objectively.

In Table 4 we present some data regarding the preprocessing – the determination of the candidate template arcs T^c. For larger values of $\boldsymbol{\delta}$ the running times become quite large, which is clearly not satisfactory. Obviously, the running times for $\boldsymbol{\delta}$ yielding the best compression ratios (e.g. $\boldsymbol{\delta} = (25, 25)^T$ or $\boldsymbol{\delta} = (30, 30, 30)^T$, see Tables 6 and 8) are of high importance. Fortunately the running times for these parameter values seem to be still reasonable for practical applications.

Tables 5, 6, 7, and 8 give an overview of the compression ratios of this approach. For the Fraunhofer data we used the parameter values $k \in \{10, 15, \ldots, 40\}$ and $\boldsymbol{\delta} \in \left\{ \binom{10}{10}, \binom{15}{15}, \ldots, \binom{45}{45}, \binom{50}{50} \right\}$ (i.e. applying the algorithm to two dimensions) as well as $\boldsymbol{\delta} \in \left\{ \binom{10}{10}, \binom{15}{15}, \binom{20}{20}, \ldots, \binom{45}{45}, \binom{50}{50} \right\}$ (applying the algorithm

Table 3 Overview about the test instances used for our experiments and their best results

short name	**full name**	$\|V\|$	$\|B\|$	ρ^*[%]	**parameters**
ft-01	P0001_F00_R00	31	930	9.9	$k = 25, \delta = (20, 20)^T$
ft-02	P0001_F00_R01	38	756	8.7	$k = 25, \delta = (40, 40)^T$
ft-03	P0001_F00_R02	35	1190	9.6	$k = 35, \delta = (35, 35, 35)^T$
ft-04	P0001_F00_R03	20	380	5.1	$k = 15, \delta = (35, 35, 35)^T$
ft-05	P0001_F00_R04	39	1482	11.5	$k = 30, \delta = (20, 20)^T$
ft-06	P0001_F01_R00	15	210	2.8	$k = 15, \delta = (40, 40)^T$
ft-07	P0001_F01_R01	28	756	7.2	$k = 25, \delta = (20, 20)^T$
ft-08	P0001_F01_R02	27	702	9.8	$k = 20, \delta = (35, 35)^T$
ft-09	P0001_F01_R03	27	702	8.6	$k = 25, \delta = (50, 50)^T$
ft-10	P0001_F01_R04	31	930	8.5	$k = 25, \delta = (45, 45)^T$
ft-11	P0001_F03_R00	38	1406	11.7	$k = 39, \delta = (45, 45)^T$
ft-12	P0001_F03_R01	28	756	12.0	$k = 25, \delta = (35, 35)^T$
ft-13	P0001_F03_R02	25	600	8.7	$k = 25, \delta = (50, 50)^T$
ft-14	P0001_F03_R03	33	1056	10.2	$k = 30, \delta = (45, 45)^T$
ft-15	P0001_F03_R04	29	812	9.9	$k = 29, \delta = (45, 45)^T$
ft-16	P0014_F00_R00	37	1332	10.6	$k = 25, \delta = (40, 40, 40)^T$
ft-17	P0014_F00_R01	31	930	8.6	$k = 25, \delta = (45, 45)^T$
ft-18	P0014_F00_R02	40	1560	13.5	$k = 30, \delta = (30, 30, 30)^T$
ft-19	P0014_F00_R03	35	1190	10.1	$k = 30, \delta = (45, 45)^T$
ft-20	P0014_F00_R04	28	756	7.1	$k = 20, \delta = (45, 45)^T$
nist-b-01-t	u201t6i	99	9702	18.9	$k = 80, \delta = (120, 120)^T$
nist-b-02-t	u202t8i	93	8556	18.9	$k = 80, \delta = (120, 120)^T$
nist-b-03-t	u204t2i	100	9900	18.9	$k = 80, \delta = (120, 120)^T$
nist-b-04-t	u205t4i	84	6972	13.8	$k = 80, \delta = (120, 120)^T$
nist-b-05-t	u206t3i	72	5256	18.9	$k = 80, \delta = (80, 80)^T$
nist-g-01-t	b101t9i	106	11130	18.9	$k = 80, \delta = (80, 80)^T$
nist-g-02-t	b102t0i	94	8742	13.4	$k = 80, \delta = (80, 80)^T$
nist-g-03-t	b104t8i	107	11342	18.9	$k = 80, \delta = (120, 120)^T$
nist-g-04-t	b105t2i	81	6480	18.6	$k = 80, \delta = (80, 80)^T$
nist-g-05-t	b106t8i	93	8556	13.8	$k = 80, \delta = (80, 80)^T$
nist-u-01-t	g001t2i	99	9702	13.4	$k = 80, \delta = (80, 80)^T$
nist-u-02-t	g002t3i	99	9702	13.8	$k = 80, \delta = (80, 80)^T$
nist-u-03-t	g003t8i	101	10100	18.9	$k = 80, \delta = (120, 120)^T$
nist-u-04-t	g004t8i	102	10302	13.8	$k = 80, \delta = (80, 80)^T$
nist-u-05-t	g005t8i	120	14280	13.8	$k = 73, \delta = (80, 80)^T$

to three dimensions) and all their combinations for our experiments. The results for the Fraunhofer data have been computed with the exact branch-and-cut method, and are therefore optimal values. Due to the long running time and memory requirements the branch-and-cut algorithm is not practicable for the NIST templates. These results have therefore been computed with the memetic algorithm. Tables 5, 6, 7, and 8 list average compression values, where the average goes over all instances and parameter settings listed in the table caption.

At a first glance the compression ratios are not too high. However, when compared to other well established compression techniques, it turns out that all these methods consistently enlarge the templates. Table 9 shows the results of our application of other well known compression tools to our test data. Colums 2 to 7 list the results of some compression algorithms implemented in the LEDA C++ library [15] for var-

Table 4 Number of candidate template arcs $|T^c|$ and running times of the preprocessing for the Fraunhofer data with some typical values of $\boldsymbol{\delta}$

$\boldsymbol{\delta}$	$(10,10)^T$		$(20,20)^T$		$(30,30)^T$		$(40,40)^T$		$(10,10,10)^T$		$(20,20,20)^T$		$(30,30,30)^T$		$(40,40,40)^T$	
inst.	$\lvert T^c\rvert$	$t[s]$	$\lvert T^c\rvert$	$t[s]$	$\lvert T^c\rvert$	$t[s]$	$\lvert T^c\rvert$	$t[s]$	$\lvert T^c\rvert$	$t[s]$	$\lvert T^c\rvert$	$t[s]$	$\lvert T^c\rvert$	$t[s]$	$\lvert T^c\rvert$	$t[s]$
ft-01	797	7	1863	74	3747	526	5802	2810	826	3	860	6	1693	34	3897	272
ft-02	610	4	1443	30	2666	185	4374	911	664	2	715	4	1560	23	3353	241
ft-03	1002	12	2684	146	4644	902	7786	4799	1042	5	1152	11	2620	82	6464	1019
ft-04	296	1	510	4	1035	23	31582	101	356	1	352	1	695	8	1223	55
ft-05	1401	23	3398	297	7044	2052	11276	12144	1246	8	1486	20	3546	237	8669	2578
ft-06	145	1	248	1	354	3	574	11	202	1	156	1	231	1	439	2
ft-07	517	3	1019	19	2035	102	3280	426	680	2	614	3	1129	11	2237	63
ft-08	533	3	1030	19	1898	92	3402	463	630	2	570	3	1064	10	2164	57
ft-09	506	3	1022	17	1828	94	3429	382	626	2	578	3	1061	11	2091	58
ft-10	767	6	1721	46	3132	283	5365	1334	812	3	812	5	1548	28	2930	165
ft-11	1565	28	3751	377	7561	3261	11542	15580	1168	7	1677	25	4446	305	11497	3748
ft-12	699	5	1548	51	3268	356	5064	1836	632	2	752	5	1718	40	4224	341
ft-13	498	3	1010	19	2201	122	3980	617	517	1	531	3	1143	15	2573	143
ft-14	1002	10	2292	109	4458	991	7130	3468	842	4	1024	8	2286	58	5150	522
ft-15	742	6	1768	63	3487	442	5720	2549	656	2	786	5	1676	30	3980	333
ft-16	1144	14	2501	145	4984	919	7330	4431	1142	6	1457	21	3125	190	6095	2858
ft-17	800	5	1633	42	3317	301	5585	1365	814	3	878	6	1878	31	3847	273
ft-18	1288	22	2858	238	5632	1577	8950	6106	1298	10	1710	31	3963	270	8889	3435
ft-19	1017	10	2161	87	3857	531	6247	2515	1000	5	992	10	1944	49	4538	399
ft-20	622	3	1086	20	1649	107	3022	395	658	2	672	4	990	20	2021	104

Table 5 Average compression ratios for the Fraunhofer templates for $\boldsymbol{\delta} = (30, 30, 30)^T$

k	$\rho_{\max}$	$\rho_{\min}$	ρ_{avg}	σ_ρ
10	-6.12	-6.80	-6.63	0.29
15	5.07	-9.22	1.15	3.67
20	6.62	-1.57	4.60	3.11
25	10.92	-0.14	5.05	3.68
30	13.47	1.29	6.31	3.05
35	9.66	4.63	7.56	2.26

Table 6 Average compression ratios for the Fraunhofer templates with $k \in \{20, 25, 30\}$

$\boldsymbol{\delta}$	$\rho_{\max}$	$\rho_{\min}$	ρ_{avg}	σ_ρ
$(10, 10)^T$	9.48	-0.87	4.98	2.11
$(15, 15)^T$	9.93	1.39	5.44	2.06
$(20, 20)^T$	11.71	0.35	5.95	2.31
$(25, 25)^T$	9.80	0.87	5.43	1.89
$(30, 30)^T$	9.76	2.69	5.79	1.97
$(35, 35)^T$	12.04	0.00	4.52	2.87
$(40, 40)^T$	10.19	0.00	5.87	2.58
$(45, 45)^T$	10.19	0.35	6.03	2.56
$(50, 50)^T$	8.68	-2.96	4.54	2.31
$(10, 10, 10)^T$	0.84	-20.21	-5.82	4.95
$(15, 15, 15)^T$	6.72	-8.01	-0.30	3.73
$(20, 20, 20)^T$	11.01	-5.92	2.61	3.57
$(25, 25, 25)^T$	10.54	-3.36	3.90	2.83
$(30, 30, 30)^T$	13.47	-1.57	5.14	3.39
$(35, 35, 35)^T$	10.45	0.82	5.70	2.81
$(40, 40, 40)^T$	10.64	-0.52	4.29	2.93
$(45, 45, 45)^T$	9.60	0.35	4.99	1.92
$(50, 50, 50)^T$	9.60	0.42	4.84	2.51

Table 7 Average compression ratios for the NIST templates with $\boldsymbol{\delta} = (80, 80, 80)^T$

k	$\rho_{\max}$	$\rho_{\min}$	ρ_{avg}	σ_ρ
20	17.35	4.85	9.47	2.87
40	18.90	11.03	15.42	2.97
60	18.97	10.00	16.08	2.20
80	19.30	10.70	15.49	3.26

Table 8 Average compression ratios for the NIST templates with $k \in \{40, 60, 80\}$

$\boldsymbol{\delta}$	$\rho_{\max}$	$\rho_{\min}$	ρ_{avg}	σ_ρ
$(40, 40)^T$	19.02	12.40	15.09	1.77
$(80, 80)^T$	15.51	3.63	10.01	3.42
$(40, 40, 40)^T$	18.24	11.80	15.85	1.80
$(80, 80, 80)^T$	19.30	10.00	15.67	2.85

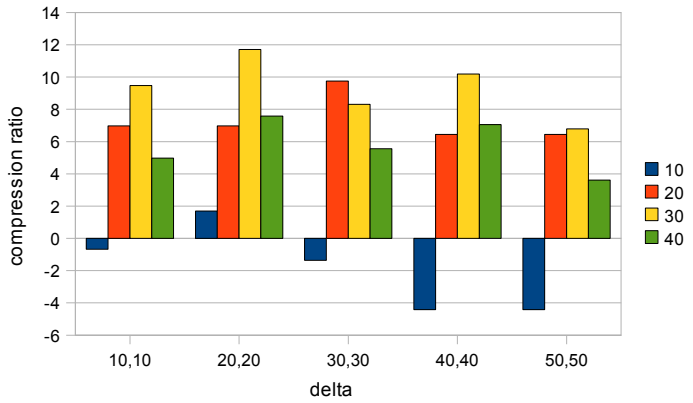

Fig. 12 Average compression ratios for the Fraunhofer data for $k \in \{10, 20, 30, 40\}$ and 2-dimensional δ

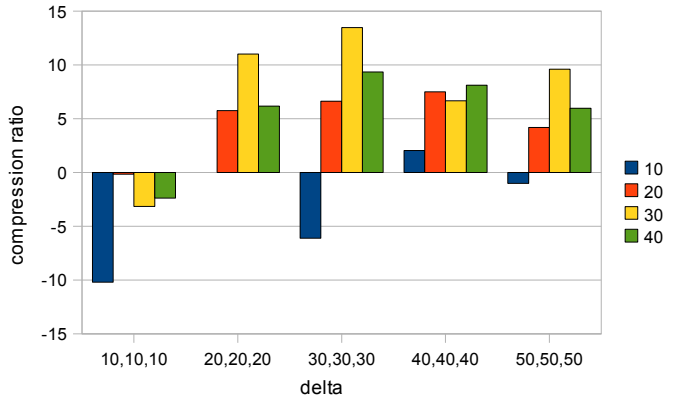

Fig. 13 Average compression ratios for the Fraunhofer data for $k \in \{10, 20, 30, 40\}$ and 3-dimensional δ

ious values of k. For this purpose we selected k points at random. The abbrevations in the first header row denote: Huff $\hat{=}$ Huffman Coding; BMA $\hat{=}$ Burrows-Wheeler Transform in combination with a Move-To-Front coder and an Adaptive Arithmetic Coder [24, 2, 18]; Columns 8 to 19 show the results for the application of some commonly used (Unix/Linux) compression tools, namely `zip`, `gzip`, `bzip2` and `compress` under Kubuntu Linux in version 8.04. Again, we selected k points at random and applied the compression tools to their binary encoding. Finally we subtracted the size of constant data of known size from the compressed file size, in particular 22 byte for `zip`, 18 byte for `gzip` and 8 byte for `bzip2`.

The bad performance of these tools on our data set may be explained on the one hand by the fact that they are not specifically designed to compress very small data sets, but on the other hand also by the observation that the input data does not contain a significant amount of redundant information. Furthermore the underlying

Table 9 Compression ratios in % of various well known compression techniques applied to our data sample

instance	Huff	BMA	Huff	BMA	Huff	BMA	zip	gzip	bzip2	compr	zip	gzip	bzip2	compr	zip	gzip	bzip2	compr
	k=10		k=20		k=30		k=10				k=20				k=30			
ft-01	-91.7	-47.5	-90.9	-29.9	-89.9	-21.8	-331.2	-9.3	-165.6	-21.8	-155.8	-7.3	-79.4	-17.6	-101.9	-4.8	-66.3	-15.3
ft-02	-92.2	-47.5	-91.6	-30.6	-91.1	-23.8	-331.2	-9.3	-168.7	-21.8	-155.8	-7.3	-97.0	-17.6	n/a	n/a	n/a	n/a
ft-03	-91.9	-47.5	-91.1	-29.9	-90.7	-22.4	-331.2	-9.3	-159.3	-21.8	-155.8	-7.3	-86.7	-17.6	-101.9	-4.8	-72.1	-15.3
ft-04	-91.9	-47.5	-91.2	-29.9	-91.2	-29.9	-331.2	-9.3	-156.2	-21.8	-155.8	-7.3	-88.2	-17.6	n/a	n/a	n/a	n/a
ft-05	-92.2	-47.5	-91.5	-29.9	-90.8	-22.4	-331.2	-9.3	-175.0	-21.8	-155.8	-7.3	-100.0	-17.6	-101.9	-4.8	-68.2	-15.3
ft-06	-91.4	-47.5	-91.2	-35.8	-91.2	-35.8	-331.2	-9.3		-21.8	n./a	n./a	n./a	n./a	n/a	n/a	n/a	n/a
ft-07	-91.9	-47.5	-91.5	-30.6	-91.0	-23.8	-331.2	-9.3	-171.8	-21.8	-155.8	-7.3	-98.5	-17.6	n/a	n/a	n/a	n/a
ft-08	-91.7	-47.5	-90.9	-30.6	-90.6	-24.0	-331.2	-9.3	-153.1	-21.8	-155.8	-7.3	-83.8	-17.6	n/a	n/a	n/a	n/a
ft-09	-91.4	-47.5	-90.6	-30.6	-90.2	-24.6	-331.2	-9.3	-159.3	-21.8	-155.8	-7.3	-82.3	-17.6	n/a	n/a	n/a	n/a
ft-10	-91.7	-46.7	-91.1	-30.6	-90.5	-22.4	-331.2	-9.3	-175.0	-21.8	-155.8	-7.3	-92.6	-17.6	-101.9	-4.8	-75.0	-15.3
ft-11	-91.4	-47.5	-91.2	-29.9	-90.3	-22.4	-331.2	-9.3	-162.5	-21.8	-155.8	-7.3	-85.2	-17.6	-101.9	-4.8	-71.1	-15.3
ft-12	-91.4	-47.5	-91.4	-30.6	-90.7	-23.8	-331.2	-9.3	-153.1	-21.8	-155.8	-7.3	-89.7	-17.6	n/a	n/a	n/a	n/a
ft-13	-91.9	-47.5	-91.5	-29.9	-91.0	-25.7	-331.2	-9.3	-165.6	-21.8	-155.8	-7.3	-92.6	-17.6	n/a	n/a	n/a	n/a
ft-14	-91.9	-47.5	-90.9	-29.2	-90.2	-21.8	-331.2	-9.3	-162.5	-21.8	-155.8	-7.3	-89.7	-17.6	-101.9	-4.8	-69.2	-15.3
ft-15	-91.9	-47.5	-91.1	-30.6	-90.3	-22.5	-331.2	-9.3	-156.2	-21.8	-155.8	-7.3	-80.8	-17.6	n/a	n/a	n/a	n/a
ft-16	-91.1	-47.5	-90.5	-29.9	-89.8	-21.8	-331.2	-9.3	-156.2	-21.8	-155.8	-7.3	-89.7	-17.6	-101.9	-4.8	-71.1	-15.3
ft-17	-91.4	-47.5	-91.1	-29.2	-90.6	-23.0	-331.2	-9.3	-153.1	-21.8	-155.8	-7.3	-88.2	-17.6	-101.9	-4.8	-74.0	-15.3
ft-18	-91.9	-47.5	-91.2	-30.6	-90.8	-23.0	-331.2	-9.3	-181.2	-21.8	-155.8	-7.3	-94.1	-17.6	-101.9	-4.8	-67.3	-15.3
ft-19	-91.9	-47.5	-91.2	-29.9	-90.5	-22.4	-331.2	-9.3	-168.7	-21.8	-155.8	-7.3	-89.7	-17.6	-101.9	-4.8	-70.1	-15.3
ft-20	-91.7	-47.5	-91.4	-30.6	-90.7	-23.8	-331.2	-9.3	-156.2	-21.8	-155.8	-7.3	-89.7	-17.6	n/a	n/a	n/a	n/a
	k=20		k=40		k=80		k=20				k=40				k=80			
nist-u-01	-90.7	-24.6	-88.7	-14.5	-84.6	-8.2	-128.2	-5.4	-78.2	-16.3	-62.7	-2.6	-45.7	-14.3	-31.0	-1.3	-40.2	-16.0
nist-u-02	-90.2	-24.6	-87.8	-13.8	-84.2	-8.2	-128.2	-5.4	-76.0	-16.3	-62.7	-2.6	-34.5	-13.8	-31.0	-1.3	-39.9	-16.5
nist-u-03	-90.3	-24.6	-88.1	-14.2	-83.3	-7.5	-128.2	-5.4	-79.3	-16.3	-62.7	-2.6	-41.4	-14.3	-31.0	-1.3	-38.4	-16.0
nist-u-04	-90.3	-24.6	-88.3	-14.5	-84.6	-8.2	-128.2	-5.4	-72.8	-16.3	-62.7	-2.6	-39.3	-13.8	-31.0	-1.3	-39.9	-16.0
nist-u-05	-90.4	-24.6	-88.4	-14.5	-84.8	-8.5	-128.2	-5.4	-80.4	-16.3	-62.7	-2.6	-48.4	-14.3	-31.0	-1.3	-42.4	-16.8
nist-b-01	-90.5	-24.0	-88.2	-13.8	-84.8	-8.7	-128.2	-5.4	-79.3	-16.3	-62.7	-2.6	-42.0	-14.3	-31.0	-1.3	-39.9	-17.1
nist-b-02	-89.8	-24.6	-87.7	-14.5	-83.5	-8.0	-128.2	-5.4	-78.2	-16.3	-62.7	-2.6	-34.5	-13.8	-31.0	-1.3	-37.1	-16.8
nist-b-03	-89.9	-24.0	-88.5	-13.8	-84.5	-8.2	-128.2	-5.4	-75.0	-16.3	-62.7	-2.6	-44.6	-13.8	-31.0	-1.3	-38.1	-16.8
nist-b-04	-90.6	-24.6	-88.6	-14.5	-84.5	-8.0	-128.2	-5.4	-77.1	-16.3	-62.7	-2.6	-39.8	-13.8	-31.0	-1.3	-39.9	-16.5
nist-b-05	-90.3	-24.6	-88.3	-14.5	-84.4	-8.0	-128.2	-5.4	-75.0	-16.3	-62.7	-2.6	-44.6	-14.3	-31.0	-1.3	-38.4	-16.8
nist-g-01	-90.1	-24.0	-88.5	-13.8	-84.6	-8.4	-128.2	-5.4	-72.8	-16.3	-62.7	-2.6	-43.6	-13.2	-31.0	-1.3	-38.6	-15.5
nist-g-02	-90.9	-24.6	-88.8	-13.8	-84.2	-8.0	-128.2	-5.4	-80.4	-16.3	-62.7	-2.6	-46.2	-13.2	-31.0	-1.3	-39.4	-16.0
nist-g-03	-90.7	-24.6	-88.1	-13.8	-84.5	-8.4	-128.2	-5.4	-80.4	-16.3	-62.7	-2.6	-40.4	-14.3	-31.0	-1.3	-40.5	-17.1
nist-g-04	-90.6	-24.6	-88.3	-14.2	-84.9	-8.4	-128.2	-5.4	-78.2	-16.3	-62.7	-2.6	-47.3	-13.8	-31.0	-1.3	-78.2	-16.5
nist-g-05	-90.1	-24.0	-87.5	-14.2	-83.8	-7.1	-128.2	-5.4	-78.2	-16.3	-62.7	-2.6	-38.8	-14.3	-31.0	-1.3	-38.6	-16.8

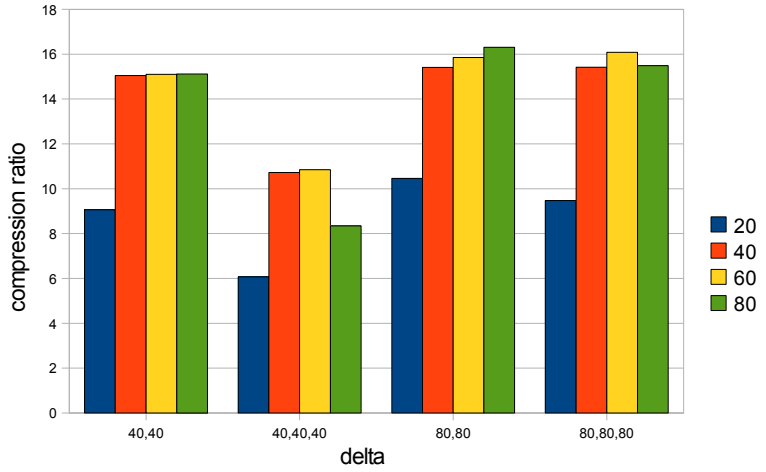

Fig. 14 Average compression ratios for the NIST data for $k \in \{20, 40, 60, 80\}$

algorithms are not able to benefit from the special structure of the input data, i.e. spatial coordinates, but also cannot benefit from the fact that the relative order of the points need not be preserved.

9.3 Matching Results

The impact of the reduction of the number of minutia, i.e. the parameter k to the reliability of the matching of different minutiae templates has been evaluated in [8]. Obviously k is the only parameter of our algorithm having an impact to matching quality. Other influences result from the concrete hardware (fingerprint scanner), the minutiae detection and extraction algorithm and also the concrete matching algorithm.

For this purpose, computational experiments have been performed with the Fraunhofer data together with a fingerprint minutiae matching algorithm, also provided by the Fraunhofer institute [8]. The false match rate (FMR) turned out not to be critical for our application scenario. For nearly all reasonable values of k, i.e. $k \geq 5$, it remained zero. For a verification application the false non-match rate (FNMR) is of higher importance. When mating different templates from the same finger the FNMR is about 5% on average. This non vanishing FNMR is due to some randomness in the template aquisition process (scanning, detection, extraction). Hence it is unlikely to achieve exactly the same minutiae sets by performing the aquisition process several times. Values of $k \leq 20$ yielded FNMRs of more than 30% on average, which definitely seems to be too high for our application scenario. Nevertheless, larger values, in particular $k \geq 25$ yielded FNMR of less than 20% on average. This seems to be reasonable as the aquisition process can be repeated several times in the case of failure of verification. Demanding $k \geq 25$ might indeed be very pessimistic, as 12-20 minutiae are often thought to be sufficient for a trusty matching [21].

Nevertheless the optimal value of k is highly dependend on the concrete implementation of this application, in particular the fingerprint scanner and the subsequent

image processing algorithms. The exact requirements regarding the FNMR need to be specified for a concrete implementation; also the acceptable computation times have influence.

9.4 Algorithmic results

In this section we compare the presented k-MLSA algorithms regarding the running times and solution quality.

Altough the running times of the branch-and-cut algorithm are sometimes very low, there are also many cases where the running times are clearly too high for practical purposes. Due to the larger size of the NIST templates the branch-and-cut approach is not practicable for them anymore. Computational experiments showed that using the cycle elimination cuts exclusively, i.e. without the directed connection cuts, yields lower runnung times than using both kinds of cuts. The exclusive use of the directed connection cuts turned out to be very slow, due to the computationally more expensive separation problem, i.e. the maximum flow problem. Most of the instances can be solved to provably optimality within a couple of seconds, but there are also many instances where the total running time lies between 10 and 30 minutes, or even more in some rare cases. These running times are clearly too high for practical applications, where a total running time of a few minutes will be acceptable. The metaheuristics are able to fulfill this goal.

The results regarding the running times and solution quality, i.e. the number of template arcs required for the resulting arborescence, are presented in Tables 10 and 11. The first three columns show the instance names and parameters k and $\boldsymbol{\delta}$. Then, in the first part of the table, we list the results from the exact branch-and-cut method, the second part contains the objective value of the currently best known solution. The column m_{best} shows the best result of the multiple runs of the algorithm, column m_{avg} shows the average value. By σ_x we denote the standard deviation of the entity x. The column $\#b.s.$ shows the percentage of runs, where the solution listed in m_{best} has been found. Average running times are listed in column t_{avg}. For each algorithm we compare two parameter settings yielding good compression ratios for the Fraunhofer and the NIST data respectively. Instead of using $k = 30$ which for the Fraunhofer data yielded the best compression results with $\boldsymbol{\delta} = (30, 30, 30)^T$, we used $k = 20$ instead, as many Fraunhofer templates have $|V| < 30$.

Table 10 shows the results of the GRASP algorithm for some parameters k and $\boldsymbol{\delta}$ in comparison to the best known solution. The presented results do not substantially differ from the ones for any other parameter settings of k and $\boldsymbol{\delta}$. The average running time to find reasonably good solutions w.r.t. our application background (i.e. to find the optimal solution in most of the cases) is roughly less than ten seconds for the Fraunhofer templates. Due to their larger size it is much more expensive to solve the NIST data with relatively high reliability. In this case the running times range from less than one minute to slightly more than three minutes. Preceding experiments for finding a good parameter setup indicated that $imp_{\max}$ has little impact on the solution quality. Nevertheless setting $imp_{\max} = 0$ corresponds to an completely arbitrary decision in the randomized construction process. In such a case the quality of the GRASP solely relies on the subsequent local search. On the other hand, too high values of $imp_{\max}$ lead to unnecessary high running times. When comparing setups of $imp_{\max}$ and it_{ls} which yield approximately the same running times, it turned out that higher values of

it_{ls} yield better average solutions. Setting $rcl_{\max} \approx 10$ turned out to be a good choice, but the exact value is uncritical. Of course, higher values imply more diversity, which may enable to find the global optimum of difficult instances more quickly.

The results of the memetic algorithm are presented in Table 11. We used a population size $size_{\mathrm{pop}} \in \{100, 200\}$, and a group size of four for the tournament selection. The crossover and mutation probability is set to one, i.e. each offspring is created by crossover and subsequent mutation. In each iteration a randomly selected candidate solution from the population was replaced by the newly generated one. Local improvement is performed for each newly created candidate solution. As mutation type 2 produced better overall results than mutation type 1, the former was used to create the results listed in Table 11. Replacing a randomly selected $t \in T$ turned out to be advantageous over replacing the worst one.

Table 11 shows the results of 30 runs with 10000 and 30000 iterations for the Fraunhofer templates (population size 100); for the NIST templates we list the results for 10000 and 60000 iterations with a population size of 100 and 200, respectively. Again, the presented results are not essentially different to the ones for any other parameter settings of k and $\boldsymbol{\delta}$. The Fraunhofer data can be compressed reliably within 10000 iterations, which takes an average running time of roughly 2 seconds. Due to the larger number of points, the compression of the NIST data is computationally more expensive. At least 60000 iterations must be used in order to be able to produce reliable results. The respective running times are roughly 100 seconds.

Due to the relatively high running time of the randomized greedy construction heuristic (Algorithm 7) it is impossible to find a parameter setup which does not degenerate the GRASP to some extend but still keeps the overall running times small. Such a setup would either start local search on nearly arbitrary random solutions (i.e. $imp_{\max} = 0$) or extremely limit the number of local search iterations, i.e. $it_{\mathrm{ls}} < 5$. Allowing slightly higher running times enables to find the optimal solution in almost every case (Fraunhofer), except two difficult instances. For the NIST data, sufficiently good solutions can be produced with adequate reliability in less than four minutes. In the case of the Fraunhofer data the MA is clearly superior to the GRASP, as it produces better solutions in less time. In contrast to GRASP it is also possible to create reasonable solutions (though with moderate quality) in less than 20 seconds. However, when allowing higher running times of up to five minutes, GRASP clearly outperformes the MA.

9.5 Absolute Compression Ratios

In Section 9.2 we presented the compression ratios achievable on our test data by our compression model in comparison to the trivial representation of the k selected points, for which the size is given by equation (19). With respect to our particular application background of fingerprint template verification not all entities considered in equation (21) need to be encoded. In the following we describe which information can be neglected.

It is obviously not possible that two scans of the same finger yield exactly the same minutia. Distortions like rotations, scalings and shifts are always involved. Matching algorithms thus have to account for such distortions in order to be able to reliably match two minutiae sets from the same finger with e.g. different coordinate offset values. Consequently we do not need to store such offset values in our compressed template

Table 10 Results and running times of the GRASP

instance	k	$\boldsymbol{\delta}$	m	$t[s]$	m_{best}	m_{avg}	σ_m	#b.s. [%]	$t_{\text{avg}}[s]$	m_{best}	m_{avg}	σ_m	#b.s. [%]	$t_{\text{avg}}[s]$	$\rho_{\text{b.s.}}$[%]
			B&C results		GRASP $it = 10, it_{ls} = 5, rcl_{\max} = 5, imp_{\max} = 0$					GRASP $it = 10, it_{ls} = 20, rcl_{\max} = 10, imp_{\max} = 10$					
ft-01			3	79	**4**	4.00	0.00	100	1.57	3	3.87	0.34	13	7.97	6.10
ft-02			3	0	**4**	4.00	0.00	100	1.00	3	3.67	0.47	33	6.20	6.10
ft-03			3	85	3	3.00	0.00	100	2.03	3	3.00	0.00	100	10.03	6.10
ft-04			4	10	4	4.70	0.46	30	0.15	4	4.30	0.46	70	2.03	-0.87
ft-05			3	327	3	3.00	0.00	100	4.82	3	3.00	0.00	100	16.90	6.10
ft-07			4	14	4	4.00	0.00	100	0.33	4	4.00	0.00	100	4.00	-1.57
ft-08			4	13	4	4.00	0.00	100	0.93	4	4.00	0.00	100	3.93	-1.57
ft-09			4	24	4	4.00	0.00	100	0.33	4	4.00	0.00	100	4.00	-1.57
ft-10			3	36	3	3.13	0.34	87	1.00	3	3.00	0.00	100	6.30	6.10
ft-11	20	$\begin{pmatrix}30\\30\\30\end{pmatrix}$	3	853	3	3.00	0.00	100	4.27	3	3.00	0.00	100	23.07	6.10
ft-12			3	52	3	3.00	0.00	100	1.00	3	3.00	0.00	100	5.93	6.62
ft-13			3	21	3	3.00	0.00	100	0.20	3	3.00	0.00	100	2.97	6.62
ft-14			3	275	3	3.00	0.00	100	2.00	3	3.00	0.00	100	8.73	6.10
ft-15			3	15	3	3.00	0.00	100	1.00	3	3.00	0.00	100	5.93	6.62
ft-16			3	282	3	3.00	0.00	100	2.97	3	3.00	0.00	100	16.53	6.10
ft-17			3	235	3	3.00	0.00	100	1.03	3	3.00	0.00	100	5.97	6.10
ft-18			3	823	3	3.00	0.00	100	5.57	3	3.00	0.00	100	21.93	6.10
ft-19			3	97	3	3.00	0.00	100	1.90	3	3.00	0.00	100	8.10	6.10
ft-20			3	0	3	3.00	0.00	100	0.30	3	3.00	0.00	100	3.00	6.10
			best known solution		$it = 5, it_{ls} = 5, rcl_{\max} = 10, imp_{\max} = 0$					$it = 5, it_{ls} = 10, rcl_{\max} = 10, imp_{\max} = 5$					
nist-b-01			4	n/a	4	4.83	0.37	17	91.60	4	4.50	0.50	50	189.97	18.90
nist-b-02			4	n/a	4	4.37	0.48	63	78.47	4	4.00	0.00	100	180.17	18.90
nist-b-03			4	n/a	4	4.43	0.50	57	89.57	4	4.03	0.18	97	190.67	18.90
nist-b-04			5	n/a	5	5.27	0.44	73	24.70	5	5.00	0.00	100	53.97	13.75
nist-b-05			4	n/a	4	4.93	0.25	7	73.37	4	4.77	0.42	23	157.47	18.90
nist-g-01			4	n/a	4	4.83	0.37	17	71.23	4	4.37	0.48	63	159.57	18.90
nist-g-02			5	n/a	5	5.73	0.44	27	67.50	5	5.40	0.49	60	151.13	13.38
nist-g-03	40	$\begin{pmatrix}80\\80\\80\end{pmatrix}$	4	n/a	4	4.17	0.37	83	94.50	4	4.00	0.00	100	189.57	18.90
nist-g-04			4	n/a	**5**	5.00	0.00	0	85.77	4	4.97	0.18	3	201.70	18.60
nist-g-05			5	n/a	5	5.13	0.34	87	40.13	5	5.03	0.18	97	64.57	13.75
nist-u-01			5	n/a	**6**	6.03	0.18	0	68.90	5	5.90	0.30	10	156.70	18.38
nist-u-02			5	n/a	5	5.00	0.00	100	69.87	5	5.00	0.00	100	152.37	13.75
nist-u-03			4	n/a	4	4.47	0.50	53	82.20	4	4.13	0.34	87	193.13	18.90
nist-u-04			5	n/a	5	5.17	0.37	86	44.53	5	5.00	0.00	100	74.83	13.75
nist-u-05			5	n/a	5	5.93	0.25	6	20.90	5	5.70	0.46	30	42.77	13.75

Table 11 Results and running times of the memetic algorithm

instance	k	$\boldsymbol{\delta}$	m	$t[s]$	m_{best}	m_{avg}	σ_m	#b.s. [%]	$t_{\text{avg}}[s]$	m_{best}	m_{avg}	σ_m	#b.s. [%]	$t_{\text{avg}}[s]$	$\rho_{\text{b.s.}}[\%]$
			B&C results		MA ($it = 10000, size_{pop} = 100$)					MA ($it = 30000, size_{pop} = 100$)					
ft-01	20	$\binom{30}{30}{30}$	3	79	3	3.70	0.47	30	1.84	3	3.23	0.42	77	5.58	6.10
ft-02			3	0	3	3.04	0.18	97	1.38	3	3.00	0.00	100	4.29	6.10
ft-03			3	85	3	3.00	0.00	100	2.45	3	3.00	0.00	100	7.51	6.10
ft-04			4	10	4	4.00	0.18	100	1.01	4	4.00	0.00	100	3.08	-0.87
ft-05			3	327	3	3.00	0.00	100	2.23	3	3.00	0.00	100	6.48	6.10
ft-07			4	14	4	4.00	0.00	100	1.49	4	4.00	0.00	100	4.62	-1.57
ft-08			4	13	4	4.00	0.00	100	1.50	4	4.00	0.00	100	4.08	-1.57
ft-09			4	24	4	4.00	0.00	100	1.35	4	4.00	0.00	100	4.54	-1.57
ft-10			3	36	3	3.43	0.50	57	1.53	3	3.17	0.38	84	5.79	6.10
ft-11			3	853	3	3.00	0.00	100	2.01	3	3.00	0.00	100	6.50	6.10
ft-12			3	52	3	3.00	0.00	100	2.26	3	3.00	0.00	100	4.86	6.10
ft-13			3	21	3	3.03	0.18	97	1.59	3	3.00	0.00	100	3.60	6.10
ft-14			3	275	3	3.16	0.37	83	1.15	3	3.00	0.00	100	7.04	6.10
ft-15			3	15	3	3.00	0.00	100	2.33	3	3.00	0.00	100	4.88	6.10
ft-16			3	282	3	3.00	0.00	100	1.69	3	3.00	0.00	100	7.50	6.10
ft-17			3	235	3	3.46	0.48	67	1.82	3	3.17	0.38	84	5.36	6.10
ft-18			3	823	3	3.00	0.00	100	2.75	3	3.00	0.00	100	8.50	6.10
ft-19			3	97	3	3.06	0.18	97	2.69	3	3.00	0.00	100	8.05	6.10
ft-20			3	0	3	3.00	0.00	100	1.58	3	3.00	0.00	100	4.79	6.10
			best known solution		MA ($it = 10000, size_{pop} = 100$)					MA ($it = 60000, size_{pop} = 200$)					
nist-b-01	40	$\binom{80}{80}{80}$	4	n/a	**5**	5.10	0.31	0	24.66	**5**	5.00	0.00	0	98.90	13.75
nist-b-02			4	n/a	4	4.74	0.44	27	15.16	4	4.20	0.40	80	89.95	18.90
nist-b-03			4	n/a	4	4.60	0.50	40	17.94	4	4.10	0.31	90	104.43	18.90
nist-b-04			5	n/a	5	4.94	0.37	10	16.32	5	5.60	0.50	40	91.75	13.75
nist-b-05			4	n/a	**5**	5.94	0.51	0	16.86	4	5.00	0.26	3	93.73	18.90
nist-g-01			4	n/a	4	4.87	0.35	14	17.57	4	4.64	0.49	33	101.82	18.90
nist-g-02			5	n/a	**6**	6.17	0.38	0	21.03	5	5.97	0.18	3	117.78	13.38
nist-g-03			4	n/a	4	4.80	0.41	46	16.36	4	4.30	0.47	70	94.29	18.90
nist-g-04			4	n/a	**5**	5.38	0.49	0	21.38	**5**	5.00	0.00	0	115.99	13.38
nist-g-05			5	n/a	5	6.57	0.57	3	16.97	5	5.74	0.45	26	92.97	13.75
nist-u-01			5	n/a	**6**	6.90	0.31	0	21.17	**6**	6.10	0.31	0	117.19	11.03
nist-u-02			5	n/a	5	5.60	0.49	40	17.54	5	5.00	0.00	100	100.68	13.75
nist-u-03			4	n/a	**5**	5.04	0.18	0	17.41	4	4.50	0.50	50	98.18	18.90
nist-u-04			5	n/a	5	5.84	0.38	17	17.75	5	5.24	0.43	76	98.87	13.75
nist-u-05			5	n/a	**6**	6.37	0.49	0	15.30	**6**	6.04	0.18	0	83.56	11.47

Table 12 Average absolute compression ratios

instances	$\boldsymbol{\delta}$	k	ρ_{avg}	σ_ρ	$\tilde{\rho}_{\text{avg}}$	$\sigma_{\tilde{\rho}}$
Fraunhofer	$(30,30)^T$	20	4.81	2.13	34.50	12.08
	$(30,30,30)^T$	20	4.41	3.27	36.08	14.34
	$(30,30)^T$	30	7.94	1.73	18.20	7.68
	$(30,30,30)^T$	30	5.98	3.23	17.68	10.11
NIST	$(80,80)^T$	20	10.46	1.52	82.45	2.18
	$(80,80,80)^T$	20	9.47	2.87	82.25	2.21
	$(80,80)^T$	40	15.41	2.26	65.92	4.15
	$(80,80,80)^T$	40	15.26	2.97	65.84	4.38
	$(80,80)^T$	60	15.85	0.58	48.79	6.29
	$(80,80,80)^T$	60	16.08	2.20	48.79	6.27
	$(80,80)^T$	80	16.31	1.99	30.49	8.25
	$(80,80,80)^T$	80	15.49	3.26	29.76	9.03

as they are of no importance for the matching algorithm. Therefore $\textsc{ConstData}' = 0$. Moreover, as we do not necessarily need all minutiae in order to perform a reliable matching (see Section 9.3), the *absolute compression ratios* are much better than the values given in Section 9.2, where the ratios are always related to the simple storage of k minutiae. By absolute compression ratio $\tilde{\rho}$ we mean the ratio of the simple encoding size of the full template, which is given by equation (19), to the compressed template with k points given by equation (21) with $\textsc{ConstData}' = 0$.

Whereas on the one hand the relative compression ratios are of more importance for evaluating the magnitude of our compression model, the absolute compression ratios are of higher importance for practical purposes on the other hand. Table 12 summarizes the absolute compression ratios for the Fraunhofer and the NIST data.

10 Conclusions and Further Work

We presented a new approach for compressing fingerprint templates, or more generally d-dimensional data points. A subset of k data points is encoded via a directed spanning tree, for which the arcs are represented by indizes to a set of template arcs plus correction vectors from a small domain. The selection of stored data points (nodes), the tree structure, and the template arc dictionary are optimized at the same time with the objective to find a feasible encoding requiring the least number of template arcs.

The general idea of compressing data by solving a graph-based combinatorial optimization problem is completely novel to our knowledge. In our approach, we determine a (large) set of candidate template arcs during preprocessing and then solve an extended variant of the minimum label spanning tree problem. An exact branch-and-cut based algorithm as well as heuristic approaches are investigated for the solution of the latter.

The compression ratios presented in Section 9.2 are not really compelling, as the data instances do not contain many structural or redundant information, which would enable higher compression. Nevertheless our experiments showed that our approach outperforms several other well known compression techniques on these data sets. When considering reasonable large values of k (i.e. $k \geq 20$) that keep the false non-match

rates reasonable small, average absolute compression ratios of more than 30% can be achieved. Hence the presented method can be suitable for compressing minutiae templates for embedding them into images by watermarking techiques.

The presented branch-and-cut algorithm finds provably optimal solutions in a couple of seconds in many cases. Unfortunately there are also instances for which the running times are much too high for practical applications. For this reason we developed faster metaheuristics and compared their running times and solution qualities. The MA turned out to be very fast, in particular the Fraunhofer instances can be solved to optimality in less than 10 seconds in almost every case. In contrast to the MA, GRASP is able to find the best known solutions with very high probability also for the larger NIST instances. Nevertheless, the running times are slightly higher and range from less than one minute to more than three minutes in this case.

As the preprocessing consumes a significant amount of time, it seems promising to incorporate the preprocessing into the optimization itself. In this case the optimization would start with a small set of template arcs and then iteratively create new promising template arcs on demand. Following this idea, the branch-and-cut algorithm can be extended to a branch-and-cut-and-price algorithm. Thereby, the pricing problem addresses the question which template arc to create next. We designed an effective method to solve the pricing problem by means of a k-d tree. Nevertheless, the branch-and-cut-and-price is still work in progress, but preliminary results indicate that this, as well as the hybridization of the exact methods with the metaheuristics, may significantly improve the overall performance.

References

1. J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975.
2. M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital SRC Research Report, 1994.
3. R.-S. Chang and S.-J. Leu. The minimum labeling spanning trees. *Information Processing Letters*, 63(5):277–282, 1997.
4. B. V. Cherkassky and A. V. Goldberg. On implementing the push-relabel method for the maximum flow problem. *Algorithmica*, 19(4):390–410, 1997.
5. A. M. Chwatal and G. R. Raidl. Applying branch-and-cut for compressing fingerprint templates (short abstract). In *Proceedings of the European Conference on Operational Research (EURO) XXII*, Prague, Czech Republic, 2007.
6. A. M. Chwatal, G. R. Raidl, and O. Dietzel. Compressing fingerprint templates by solving an extended minimum label spanning tree problem. In *Proceedings of the Seventh Metaheuristics International Conference (MIC)*, Montreal, Canada, 2007.
7. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 2nd edition, 2001.
8. O. Dietzel. Combinatorial Optimization for the Compression of Biometric Templates. Master's thesis, Vienna University of Technology, Institute of Computer Graphics and Algorithms, May 2008.
9. T. Feo and M. Resende. Greedy randomized adaptive search procedures. *Journal of Global Optimization*, 6:109–133, 1995.
10. Garris M. D. and McCabe R. M. NIST special database 27: Fingerprint minutiae from latent and matching tenprint images. Technical report, National Institute of Standards and Technology, 2000.
11. D. S. Hochbaum and W. Maass. Approximation schemes for covering and packing problems in image processing and vlsi. *Journal of the ACM*, 32(1):130–136, 1985.
12. ILOG Concert Technology, CPLEX. ILOG. http://www.ilog.com. Version 11.0.
13. A. Jain and U. Uludag. Hiding fingerprint minutiae in images. In *Proceedings of Third Workshop on Automatic Identification Advanced Technologies*, pages 97–102, 2002.
14. S. O. Krumke and H.-C. Wirth. On the minimum label spanning tree problem. *Information Processing Letters*, 66(2):81–85, 1998.
15. Library for Efficient Datastructures and Algorithms (LEDA). Algorithmics Solutions Software GmbH. http://www.algorithmic-solutions.com/. Version 5.1.
16. T. Magnanti and L. Wolsey. Optimal trees. *Network Models, Handbook in Operations Research and Management Science*, pages 503–615, 1995.
17. D. Maltoni, D. Maio, A. K. Jain, and S. Prabhakar. *Handbook of fingerprint recognition*. Springer, 2003.
18. A. Moffat, R. M. Neal, and I. H. Witten. Arithmetic coding revisited. *ACM Transactions on Information Systems*, 16(3):256–294, 1998.
19. J. Nummela and B. A. Julstrom. An effective genetic algorithm for the minimum-label spanning tree problem. In *GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 553–558, New York, NY, USA, 2006. ACM.
20. G. R. Raidl and A. Chwatal. Fingerprint template compression by solving a minimum label k-node subtree problem. In E. Simos, editor, *Numerical Analysis and Applied Mathematics*, volume 936 of *AIP Conference Proceedings*, pages 444–447.

American Institute of Physics, 2007.
21. A. Saleh and R. Adhami. Curvature-based matching approach for automatic fingerprint identification. In *Proceedings of the Southeastern Symposium on System Theory*, pages 171–175, 2001.
22. K. Sayood. *Introduction to Data Compression*. Morgan Kaufmann Publishers Inc., USA, third edition, 2006.
23. L. R. Varshney and V. K. Goyal. Benefiting from disorder: Source coding for unordered data. *arXiv*, abs/0708.2310, 2007.
24. J. S. Vitter. Design and analysis of dynamic huffman codes. *Journal of the ACM*, 34(4):825–845, 1987.
25. L. A. Wolsey and G. L. Nemhauser. *Integer and Combinatorial Optimization*. Wiley-Interscience, November 1999.
26. Y. Xiong, B. Golden, and E. Wasil. A one-parameter genetic algorithm for the minimum labeling spanning tree problem. *IEEE Transactions on Evolutionary Computation*, 9(1):55–60, 2 2005.
27. J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.
28. J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.