Karlsruhe Institute of Technology

PCA-Based Compression of Travel Time Functions

Diploma Thesis of

Marc Schmitzer

At the faculty of Computer Science
Institute for Theoretical Computer Science, Algorithmics II

Referee: Prof. Dr. Peter Sanders
Advisors: Dipl.-Inform. G. Veit Batz,
Dipl.-Inform., Dipl.-Math. Jochen Speck

February 1st – July 31st 2010

Erklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Karlsruhe, 31. 07. 2010

Marc Schmitzer

Abstract

In this work, we present and evaluate multiple approaches to compressing travel time data for time-dependent route planning in road networks. The approaches are based on principal component analysis and introduce a limited approximation error to the data. The compression achieved by the approaches is compared to that achieved by an algorithm devised by H. Imai and M. Iri which can be used for the same purpose.

The first approach uses heuristics to separate the input data into subsets of similar functions which can be compressed more efficiently than the entire data set at once. It surpasses the compression achieved by the algorithm by Imai and Iri by approximately 25%. The second approach attempts to increase the similarities exploited by the former by modifying the travel time functions, but fails to further improve the results. The third approach presented merges the clustering of the input data with the compression process. With this approach, the compression is improved by a factor of two compared to the algorithm by Imai and Iri.

Finally, we demonstrate that the approximation error introduced by the third approach is not overly detrimental to the performance of the *Time-Dependent Contraction Hierarchies* route planning algorithm.

Contents

German Summary **1**

1 Introduction **7**
1.1 Motivation 7
1.2 Problem statement 7
1.3 Related work 8

2 Foundations **10**
2.1 Piecewise Linear Functions 10
2.2 The Imai-Iri Algorithm 10
2.3 Principal Component Analysis 11
2.4 Compression Ratio 12
2.5 Overhead 13
2.6 Error Measures 14
2.7 Implementation 14

3 Building Blocks **15**
3.1 PCA-Based Compression 15
3.1.1 Run Length Coding 15
3.1.2 Comparison to the Imai-Iri Algorithm 15
3.2 Sampling 16
3.2.1 Distribution of the Data Points 16
3.2.2 Sampling Error 17
3.2.3 Memory and Runtime Requirements 17
3.3 Normalization 19
3.4 Smoothing 19
3.5 Peaks 20

4 Manual Clustering **23**
4.1 Clustering 23
4.2 Algorithm 23
4.2.1 The Simple PCA Algorithm 24
4.3 Experiments 24
4.3.1 Classification 24
4.3.2 Compression 25

5 Model-Based Transformation **28**
5.1 Algorithm 28
5.1.1 Model 28
5.1.2 Model-Fitting 29
5.1.3 Transformation 30
5.2 Experiments 31

5.2.1 Sampling . . . 31
5.2.2 Compression . . . 32

6 Automatic Clustering **35**
6.1 Algorithm . . . 35
6.1.1 Finding Clusters . . . 36
6.2 Experiments . . . 37
6.2.1 Parameters of the algorithm . . . 37
6.2.2 Overview . . . 38
6.2.3 PCA Overhead . . . 41
6.2.4 Error Bounds . . . 42
6.2.5 Smoothed Input Data . . . 42
6.2.6 Runtime . . . 44
6.2.7 Additional Data Sets . . . 45
6.3 Application . . . 46
6.3.1 Higher Error Bounds . . . 47
6.3.2 Smoothed Input Data . . . 50

7 Conclusions **51**
7.1 Future work . . . 51

References **52**

German Summary — Deutsche Zusammenfassung

Einführung

Zeitabhängige Routenplanung ist eine Erweiterung der klassischen Routenplanung, die die statischen Fahrzeitdaten durch *Fahrzeitfunktionen* ersetzt. Das heißt, die für die Fahrt zwischen zwei Knoten a und b des Straßennetzgraphen benötigte Zeit ist eine Funktion der *Abfahrtszeit*. Auf diese Weise können regelmäßige tageszeitabhängige Veränderungen der Fahrtzeit, zum Beispiel durch Berufsverkehr, bei der Routenplanung berücksichtigt werden.

Die hierfür benötigten Fahrzeitdaten sind aber unter Umständen sehr umfangreich, was vor allem auf Mobilgeräten problematisch sein kann, da die Datenübertragung vom Flash–Speicher des Geräts die Routenberechnung erheblich verlangsamen kann [SSV08, p. 739].

Ziel dieser Arbeit ist es deshalb, ein Kompressionsverfahren für Fahrtzeitdaten zu entwickeln und zu evaluieren. Das zu entwickelnde Verfahren ist *verlustbehaftet*, das heißt eine begrenzte Abweichung zwischen originalen und dekomprimierten Daten ist zulässig. Als Grundlage für das zu entwickelnde Verfahren wurde die Hauptkomponentenanalyse (engl. *Principal Component Analysis, PCA)* gewählt. Diese sollte in der Lage sein häufig wiederkehrende Strukturen in den Fahrtzeitfunktionen bei der Kompression zu nutzen. Als Messlatte für die erzielte Kompression dient der Algorithmus zur Approximation von stückweise linearen Funktionen von Imai und Iri [II87], der von S. Neubauer für die Kompression von Fahrtzeitfunktionen implementiert und evaluiert wurde [Neu09].

Die verwendeten Fahrtzeitdaten stammen aus einem von der PTV AG zur Verfügung gestellten Datensatz und wurden für die *Time-Dependent Contraction Hierarchies* Routenplanungstechnik (TCH) [BDSV09] aufbereitet. TCH erweitert die von R. Geisberger et al. entwickelte *Contraction Hierarchies (CH)* Technik um Zeitabhängigkeit [GSSD08]. *Contraction Hierarchies* ist eine hierarchische Routenplanungstechnik, die durch Vorberechnung auf dem Straßennetzgraphen schnelle und exakte Routenberechnung auf großen Straßennetzen ermöglicht.

Die vorliegenden Fahrtzeitfunktionen sind stückweise lineare Funktionen, definiert durch Stützstellen, die die Fahrtzeit der jeweiligen Funktion zu verschiedenen Abfahrtszeiten angeben.

Der für die Experimente verwendete Datensatz enthält insgesat 17.942.106 Fahrzeitfunktionen, von denen allerdings nur 2.380.285 nicht konstant sind und damit für Kompression in Frage kommen. Die nicht konstanten Funktionen haben durchschnittlich rund 103 Datenpunkte.

PCA–basierte Kompression

Um Fahrtzeitfunktionen mit Hilfe der Hauptkomponentenanalyse (PCA) zu komprimieren, müssen die Fahrzeitfunktionen durch „Sampling” in eine homogene Form

gebracht werden. Dabei werden alle Funktionen an S äquidistanten Punkten — Vielfachen der *Samplingdistanz* Δ_S — evaluiert, woraus sich für jede Funktion f ein Vektor von Sampelwerten s_f ergibt. Von den Samplingvektoren wird deren Mittelwert $\bar{s}$ abgezogen. Aus den so modifizierten Daten wird über die Hauptkomponentenanalyse eine alternative Basis, bestehend aus den Hauptkomponenten p_i, berechnet. Wird ein Sample–Vektor $s_f - \bar{s}$ in diese Basis transferiert, entsteht der Koeffizientenvektor c_f. Aus den Eigenschaften der Hauptkomponentenanalyse ergibt sich, dass die ersten Hauptkomponenten tendenziell „wichtiger" sind als die folgenden. Das heißt, dass eine Rekonstruktion s^*_f des Samplingvektors aus dem Mittelwertsvektor und den ersten m Hauptkomponenten — für ein geeignetes m — unter Umständen ausreicht, um eine akzeptable Approximation von f zu erhalten.

$$s_f = \bar{s} + \sum_{i=1}^{S} c_i \cdot p_i \qquad s^*_f = \bar{s} + \sum_{i=1}^{m} c_i \cdot p_i \ (m < S)$$

Die PCA–basierten Kompressionsansätze in dieser Arbeit finden für jede Funktion jeweils das kleinste m mit dem eine gegebene Approximationsgüte erreicht wird und reduzieren somit den Platzbedarf für die Funktion auf m Werte. Zusätzlich müssen natürlich der Mittelwertsvektor $\bar{s}$ und die benötigten Hauptkomponenten gespeichert werden.

Offensichtlich ist es nicht auszuschließen, dass die PCA–komprimierte Darstellung einer Funktion größer ist als die Originaldaten der Funktion. Ferner steht auch der Algorithmus von Imai und Iri zur Kompression zur Verfügung. Deshalb wird für jede Funktion die kleinere von PCA– und Imai–Iri–komprimierter Darstellung gespeichert.

Messgrößen

Die in den folgenden Abschnitten beschriebenen Algorithmen werden an Hand von vier Messgrößen verglichen:

- Die Kompressionsrate r_{compr} gibt das Verhältnis der komprimierten Größe der nicht konstanten Funktionen zu deren ursprünglicher Größe an.

- Der Anteil der PCA–komprimierten Funktionen r_{PCA} gibt an, wie viele der nicht konstanten Fahrzeitfunktionen PCA–basiert komprimiert wurden. Das heißt, für wie viele der Funktionen die PCA–basierte Kompression effektiver war als die vom Imai–Iri–Algorithmus erzeugte.

- Der mittlere Approximationsfehler $\bar{e}_{mean}$.

- Die Größe der zusätzlichen PCA–spezifischen Daten im Verhältnis zur Größe der Komprimierten Funktionsdaten $r_{overhead}$. Bei diesem *Overhead* handelt es sich um die Mittelwertsvektoren und die Hauptkomponenten.

Algorithmus	Min. Clustergröße	r_{compr}	r_{PCA}	$\bar{e}_{mean}$	$r_{overhead}$
Imai–Iri	–	14,26%	0%	0,12%	0%
Ohne Clustern	–	13,28%	28,9%	0,094%	0,03%
Manuelles Clustern	–	10,72%	61,46%	0,054%	0,16%
Automatisches Clustern	10%	10,05%	49,38%	0,072%	0,085%
	1%	6,3%	72,2%	0,049%	0,695%

Tabelle 1: Ergebnisse der vorgestellten Algorithmen bei maximalem Approximationsfehler von 1%.

Manuelles Clustern

Da sich früh zeigte, dass der zuvor beschriebene PCA–basierte Ansatz allein nicht zu befriedigenden Ergebnissen führt, wurden Erweiterungen für diesen entwickelt. Der *Manuelles Clustern* genannte Ansatz zerlegt den Eingabedatensatz in Teilmengen ähnlicher Funktionen und wendet die zuvor beschriebene Kompression einzeln auf die Teilmengen an. Dieses Zerlegen des Datensatzes in Teilmengen mit anschließender separater Kompression wird hier als *Clustern* bezeichnet.

Als Maß für die Ähnlichkeit von Funktionen wird hierbei das Auftreten von signifikanten Hochpunkten, sogenannten *Peaks* verwendet. Hierzu werden aus dem Definitionsbereich der Funktionen drei Intervalle — „Morgen”, „Mittag” und „Abend” — ausgewählt, und die Funktionen nach der Anzahl Peaks in den einzelnen Intervallen Klassifiziert. Die so entstehenden Teilmengen werden nach der jeweiligen Anzahl von Peaks in den einzelnen Intervallen bezeichnet, also zum Beispiel »1–0–1«.

Ergebnisse

Tabelle 1 zeigt die Resultate des manuellen Clusterings an Hand der zuvor beschriebenen Messgrößen. Als Approximationsgüte wurde 1% maximaler relativer Fehler gewählt. Zum Vergleich sind die entsprechenden Werte für PCA–basierte Kompression ohne Clustern und für den Imai–Iri–Algorithmus angegeben. Wie aus der Tabelle ersichtlich wird, verbessert der Clustering–Ansatz die Kompression gegenüber dem Imai–Iri–Algorithmus um etwa ein Viertel, während der mittlere Approximationsfehler um etwa die Hälfte sinkt. Gegenüber dem einfachen PCA–basierten Verfahren ohne Clustern konnte die Anzahl der PCA–komprimierten Funktionen etwa verdoppelt werden, allerdings stieg auch die Größe des Overheads beträchtlich.

Ferner wurden auch Tests mit zufällig ausgewählten Teilmengen des Datensatzes durchgeführt, um zu bestimmen ob beliebige Aufteilungen einen ähnlichen Effekt haben wie die oben beschriebene. Diese ergaben allerdings keine Verbesserung gegenüber der einfachen PCA–Kompression ohne Clustern.

Modell–basierte Transformation

Die Modellbasierte Transformation ist ein Versuch, die Fahrzeitfunktionen für die PCA–basierte Kompression aufzubereiten und somit deren Effektivität zu erhöhen.

Der Grundgedanke besteht darin, Funktionen mit vergleichbarer Form so zu modifizieren, dass sie sich noch ähnlicher sind, und weniger Unterschiede durch die PCA kodiert werden müssen.

Der Ansatz setzt auf den im vorigen Abschnitt entwickelten Teilmengen und Informationen über Ort und Form von Peaks in den Funktionen auf. Er wird beispielhaft an den 1–0–1 Teilmengen beschrieben, also Funktionen die ein typisches Berufsverkehrsmuster zeigen, kann aber auch auf andere Teilmengen übertragen werden.

Um eine genaueres Bild des Aussehens der Funktionen zu erhalten, wird ein Modell definiert, das ein Fahrzeitfunktion als eine Summe von Gauss–Funktionen beschreibt. Hierbei wird für jeden Peak der Funktion eine Gauss–Funktion angesetzt und ein weiterer für die grundlegende Zunahme der Fahrtzeit am Tag gegenüber der Nacht. Die verwendeten Gauss–Funktionen $g_{x,h,\sigma}(t) = h \cdot \exp\left(-\frac{1}{2} \cdot \left(\frac{t-x}{\sigma}\right)^2\right)$ werden mir ihrer Position x, der Höhe h und der Standardabweichung σ, aus der sich die Breite ergibt, parametrisiert. Um geeignete Werte für diese Parameter zu finden, wird die Differenz zwischen dem parametrisierten Modell und der Funktion mit dem Optimierungsalgorithmus von Nelder und Mead [NM65] minimiert. Die Differenz wird hierbei durch Auswerten von Funktion und Modell in regelmäßigen Abständen approximiert.

Aus den so gewonnen Informationen wird für jede Fahrzeitfunktion f eine Transformationsfunktion T_f abgeleitet. Diese verschiebt die Datenpunkte von f entlang der x–Achse, sodass die Form der Funktion an die durchschnittliche Form der Funktionen in der Teilmenge angenähert wird. Die so modifizierten Funktionsmengen werden dann mit dem einfachen PCA–basierten Algorithmus komprimiert.

Ergebnisse

Wie sich heraustellte hat die beschriebene Transformation drastische Auswirkungen auf den Samplingprozess. Die durch das Sampling erzeugte Abweichung zwischen Originaldaten und gesampelten Daten steigt durch die Transformation erheblich. Hierdurch kann für viele Funktionen die angestrebte Approximationsgüte von vornherein nicht erreicht werden, für andere steigt die Zahl der benötigten Koeffizienten und damit die Größe der komprimierten Daten. Insgesamt ist die auf den transformierten Daten erzielte Kompression nur wenig besser als die vom Imai–Iri–Algorithmus erreichte. Da mit den komprimierten Daten zusätzlich noch die zur Rücktransformation benötigten Informationen gespeichert werden müssen, ist der Transformationsansatz in dieser Form nicht verwendbar.

Automatisches Clustern

Das *Automatische Clustern* führt die Idee der Verbesserung der PCA–basierten Kompression durch Aufteilen des Datensatzes weiter. Im Gegensatz zum manuellen Clustern findet die Aufteilung hier allerdings direkt auf Basis der Hauptkomponentenanalyse statt.

Hierzu werden zunächst Hauptkomponenten und Koeffizientenvektoren für alle

Funktionen im Datensatz berechnet. Funktionen, deren Koeffizientenvektor deutlich größer als die vom Imai–Iri–Algorithmus erzeugte Darstellung ist, werden „bei Seite gelegt" und der Vorgang wird mit den verbliebenen Funktionen wiederholt. In den folgenden Iterationen wird das Kriterium zum Entfernen der Funktionen zunehmend verschärft, sodass die verbleibenden Funktionen schließlich besser komprimiert werden als durch den Imai–Iri–Algorithmus. Der Vorgang wird abgebrochen, sobald die Menge der verbleibenden Funktionen unter eine *minimale Clustergröße*, die relativ zur Gesamtzahl der Funktionen angegeben wird, fällt. Der letzte noch ausreichend große Cluster wird dann gespeichert und der gesamte Vorgang mit den zuvor bei Seite gelegten Funktionen wiederholt.

Ergebnisse

Wie aus Tabelle 1 ersichtlich wird, wurden mit dem automatischen Clustern die zuvor erzielten Ergebnisse weiter verbessert. Insbesondere durch eine relative kleine minimale Clustergröße wurde eine deutlich stärkere Kompression als mit dem manuellen Clustern erzielt. Die vom Imai–Iri–Algorithmus erreichte Kompression wurde sogar um den Faktor zwei verbessert. Auch der mittlere Approximationsfehler konnte noch etwas weiter gesenkt werden.

Auswirkungen auf die Anwendung

Experimente mit dem *Time-Dependent Contraction Hierarchies* Routenplanungsalgorithmus zeigen, dass die Auswirkung der Approximation auf die berechneten Routen relativ gering sind. Zur Ermittlung der Auswirkungen wurden 1.000.000 zufällige Routen–Anfragen auf Basis der komprimierten Daten berechnet und die berechneten Routen und deren Fahrzeit mit den Optimalwerten verglichen.

Die durchschnittliche relative Abweichung zwischen den aus den komprimierten Daten berechneten Reisezeiten und den korrekten Werten liegt über dem mittleren, aber unter dem maximalen Approximationsfehler. Eine starke Ausbreitung des Approximationsfehlers durch die Verkettung mehrer approximierter Fahrzeitfunktionen liegt also augenscheinlich nicht vor.

Die geringe Abweichung der realen Fahrzeit der auf komprimierten Daten bestimmten Route gegenüber dem Optimalwert lässt darauf schließen, dass der Routenplanungsalgorithmus trotz des Approximationsfehlers in der Regel die optimale Route findet. Dieses Verhalten setzt sich auch Kompression mit höherem Approximationsfehler von zum Beispiel 10% fort.

Fazit

Mit zwei der vorgestellten Kompressionsverfahren konnte eine beträchtliche Verbesserung der Kompressionsrate gegenüber dem zum Vergleich herangezogenen Algorithmus von Imai und Iri erreicht werden. Mit dem — allerdings recht aufwändigen

— automatischen Clustern ließ sich die Kompression sogar um den Faktor zwei verbessern.

Positiv ist weiterhin, dass der mittlere Fehler der Approximation trotz der stärkeren Kompression gesenkt werden konnte. Ferner haben Experimente gezeigt, dass die Routenberechnung mit der *Time-Dependent Contraction Hierarchies* Technik durch die Approximation nicht substantiell beeinträchtigt wird.

1 Introduction

1.1 Motivation

Time-dependency is becoming an increasingly important aspect of route planning. Classical route planning assumes a constant travel time for each edge in the road network. That is, traveling from a node a to an adjacent node b in the road network is assumed to take a static amount of time $t_{a,b}$. Time dependent route planning extends that assumption, so that the travel time $t_{a,b}$ becomes a function of the departure time, that is the point in time at which the travel from node a to node b begins. This extended model is for example able to account for increased travel times due to rush hour traffic.

However, with this extension, the data needed to describe the travel time of a road segment grows from a single value to a *travel time function* describing the travel time for the road segment over a given period. Detailed travel time data for large road networks with millions of edges can result in very large amounts of data, which can be problematic, especially for mobile devices. While the bare storage space requirements are becoming less critical as the storage capacities of mobile devices grow, large data sets can still adversely affect query time. Because the larger the travel time data, the more data has to be transferred from—relatively slow—mass storage devices such as flash drives.

With *Contraction Hierarchies (CH)*, Robert Geisberger et al. [GSSD08] have presented a hierarchical route planning method that allows fast queries while having low memory requirements. This technique has been extended to *Time-Dependent Contraction Hierarchies (TCH)* by Batz et al. to support time-dependent route planning [BDSV09].

1.2 Problem statement

The goal of this work is to develop and evaluate a method that produces a space–efficient approximation of a set of travel time functions. While this method is developed using a specific data set, it should be sufficiently general to produce similar results on comparable sets of travel time functions. The primary parameter of the compression method devised is the maximum or mean relative error of the approximation.

The decision to base the algorithm to be developed on principal component analysis was founded on the supposition that principal component analysis should be able to exploit similarities between functions for an efficient compression. Generally, principal component analysis is a useful tool when little information about the structure of the data is available, as is the case with the data set at hand.

The main data set used in this work is a road network of Germany with travel time data for Tuesday to Thursday provided by PTV AG. The data set was preprocessed for the Time-Dependent Contraction Hierarchies technique (TCH) [BDSV09]. As a result of the TCH-preprocessing, the data set contains *shortcuts* in addition to

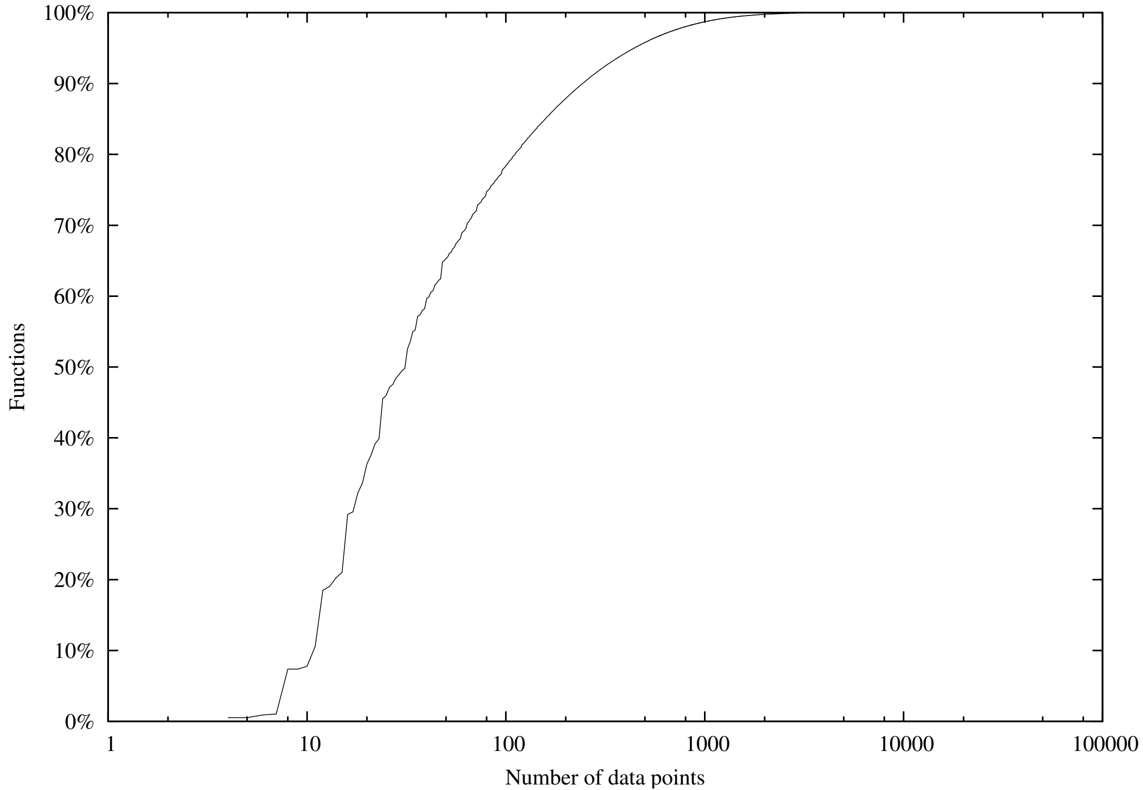

Figure 1: Cumulative distribution of the sizes of non-constant functions in the data set.

the original edges of the road network graph. Shortcuts represent paths within the original road network and have travel time functions composed of the travel time functions of the road segments represented by the shortcut. If, for example, a shortcut spans two road segments A and B (in that order) with travel time functions f_A and f_B, the travel time function of the shortcut f_S is given by $f_S(t) = f_A(t)+f_B(t+f_A(t))$.

The data set contains 17,942,106 travel time functions, of which 2,380,285 are not constant. The non-constant functions have approximately 103 data points per function on average. While some functions in the set have several thousand data points, 90% have less than 250. Figure 1 shows the cumulative distribution of the number of data points per non-constant function.

1.3 Related work

In her student thesis *Space Efficient Approximation of Piecewise Linear Functions* [Neu09], S. Neubauer discusses the algorithm designed by H. Imai and M. Iri [II87], which calculates an approximation of a piecewise linear function for a given absolute error bound. The algorithm is described in more detail in Section 2.2 of this work. Neubauer presents an implementation of the algorithm and tests this implementation on a previous version of the data set that is also used in this work. In the thesis, the algorithm is also extended to support relative as well as absolute error bounds for

the approximation.

C. Vetter et al. [SSV08] developed a “highly compressed blocked representation” of TCH graph data and a “fast yet compact” route reconstruction data structure which facilitate fast and exact route planning on mobile devices. However, the devised representation and data structure only consider the network graph data but not travel time data and do thus not solve the problem this work is aimed at.

2 Foundations

2.1 Piecewise Linear Functions

The travel time functions that form the input data for the algorithms presented in this work are continuous piecewise linear functions. A piecewise linear travel time function f is defined by a sequence of n data points (x_i, y_i) $(i = 1, \dots, n)$ in ascending order of x and the function period P. The value y_i is the *travel time* for the road segment f belongs to if travel through the segment begins at the *departure time* x_i. The travel time functions in the data set at hand have a common period of 24 hours.

A value $f(t)$ at an arbitrary time t is calculated by linear interpolation between the two data points surrounding t as shown in Equation 1. If t is outside of $[0, P)$, $f(t)$ can be calculated as through $f(t+kP) = f(t)$ $(k \in \mathbb{N})$. Values $f(t)$ for t outside of $[x_1, x_n]$ can be calculated by extending the function with the points $(x_0, y_0) = (x_n - P, y_n)$ and $(x_{n+1}, y_{n+1}) = (x_0 + P, y_0)$.

$$f(t) = \frac{y_i(x_{i+1} - t) + y_{i+1}(t - x_i)}{x_{i+1} - x_i} \qquad (x_i \le t \le x_{i+1}) \tag{1}$$

2.2 The Imai-Iri Algorithm

The algorithm presented by H. Imai and M. Iri [II87] computes an itself piecewise linear approximation of a piecewise linear function f that meets a given absolute error bound w while having a minimum number of data points. The algorithm works by calculating the polygon $P(w)$ that delineates the corridor around the input function defined by the error bounds. Within this polygon, the algorithm then constructs the approximated function by a process that can be compared to shining a light source into the corridor around the function. The border line between the illuminated part of the corridor and the part containing the end of the corridor is called a *window* and contains the first point of the approximation. From the window, the illumination step is repeated to locate the next window, until the end of the corridor is reached. Figure 2 illustrates the concept with an example function. The corridor $P(w)$ is plotted as a dashed line, and the part of the corridor that is illuminated from the entrance is shaded.

While the authors show that the approximation can be calculated by "repeatedly solving the edge–visibility problem" [II87, p. 1], they also provide a more efficient solution that exploits properties of $P(w)$ to avoid the complex construction of visibility polygons and runs in time linear in the number of data points of f.

Imai and Iri prove that the approximation produced by the algorithm has the minimum number of points for the given error bound. However, the algorithm is not designed to minimize the maximum or average difference between the original function and the approximation.

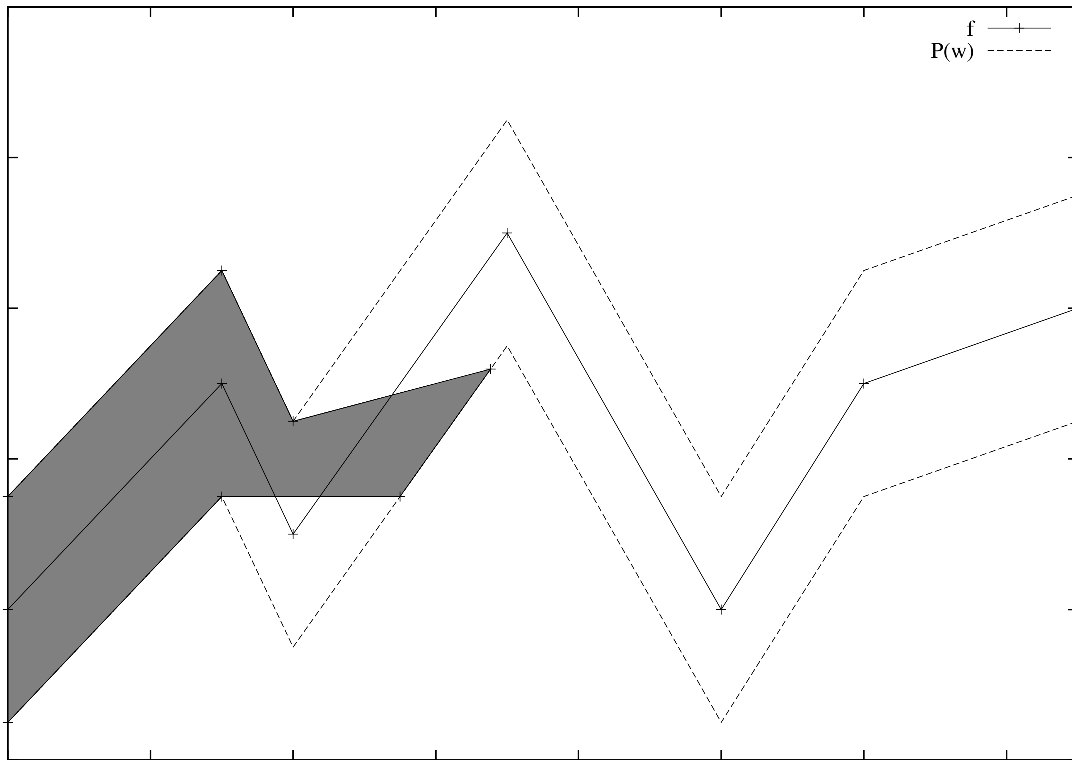

Figure 2: Imai-Iri algorithm example.

2.3 Principal Component Analysis

The idea of principal component analysis originates from K. Pearson [Pea01], however the term "principal component" was created by Hotelling [Hot33]. The following brief description is based on the work of Jolliffe [Jol86] and Falk et al. [FBM95], though. Jolliffe describes the "central idea of principal component analysis" as to "reduce the dimensionality of a data set which consists of a large number of interrelated variables, while retaining as much as possible of the variation present in the data set" [Jol86, p. 1].

The process of principal component analysis consists of multiple steps. Assume a data set D containing n observations of a set of k random variables as in Equation 2. Together, the observation vectors form the data matrix M as defined in Equation 3. First, the data is centered. For this, the mean vector $\bar{x} \in \mathbb{R}^k$ is calculated as shown in Equation 4 and subtracted from each column of the data matrix, resulting in the centered data matrix M' (Equation 5). In the next step, the covariance matrix C of the data matrix is calculated as shown in Equation 6. Finally, the eigenvectors v_i and corresponding eigenvalues λ_i of the covariance matrix are calculated. The eigenvectors of the covariance matrix are the *principal components* of the data set. The eigenvector with the largest associated eigenvalue is the direction in which the data set has the greatest variation and is thus likely to contain the most information about the data set. The following eigenvectors (in order of descending size of the

associated eigenvalue) represent successively less important directions within the data set.

$$
\begin{aligned}
D &= \{x_1, x_{2,} \ldots, x_n\} \subset \mathbb{R}^k \; (n > k) && (2)\\
M &= (x_1, x_2, \ldots, x_n) \in \mathbb{R}^{k \times n} && (3)\\
\bar{x}_i &= n^{-1} \sum_{j=1}^{n} M_{ij} \; (i = 1, \ldots, k) && (4)\\
M' &= (x_1 - \bar{x}, \ldots, x_n - \bar{x}) && (5)\\
C &= n^{-1} M' M'^T \in \mathbb{R}^{k \times k} && (6)\\
M^* &= \begin{pmatrix} p_1 & \ldots & p_m \end{pmatrix}^T M' \in \mathbb{R}^{m \times n} && (7)
\end{aligned}
$$

To use this information to reduce the number of dimensions of the data set, the number of resulting dimensions m has to be chosen. This can, for example, be done with a heuristic method like the *Scree Test* [FBM95, p. 306]. The first m principal components are selected and used to transform the data matrix M' into M^* by multiplying it with the matrix containing the first m eigenvectors of the covariance matrix (Equation 7). An approximation of the original data can be restored from M^* by reversing the above transformation and restoring the mean.

Figure 3 illustrates the effect of principal component analysis through an example. The left-hand plot shows a cloud of two-dimensional data points. The arrows originating from the center of the cloud indicate the directions of the principal components p_1 and p_2 of the data set with the longer arrow representing the first principal component p_1. The right-hand plot in Figure 3 shows the same data points transferred into the coordinate system of p_1 and p_2.

2.4 Compression Ratio

The primary criterion used to evaluate the compression algorithms presented in this work is the compression ratio r_{compr} achieved by the algorithm.

$$r_{compr} := \frac{\text{compressed size of non-constant functions}}{\text{original size of non-constant functions}}$$

This definition reflects the fact that the algorithms under discussion ignore the constant functions of the data set, as there is little to be gained by attempting to compress them. The more practical definition of a compression ratio r'_{compr} that includes the constant functions is linear in r_{compr} and can be calculated as follows.

$$r'_{compr} = \frac{r_{compr} \cdot \text{original size of non-constant functions} + \text{size of constant functions}}{\text{total size}}$$

The size of a set of functions or compressed functions is calculated by counting eight bytes for each floating point value and four bytes for each integer value,

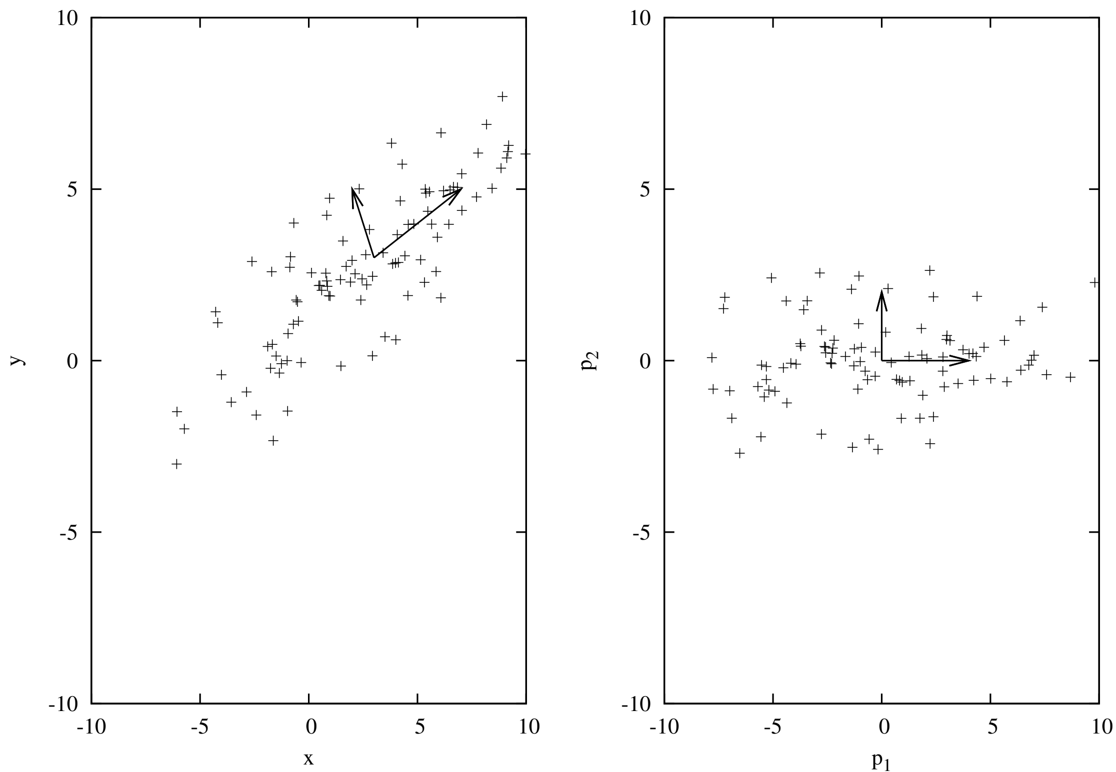

Figure 3: PCA example.

matching the sizes of the C++ data types `double` and (`unsigned`) `int` on the x86 platform.

2.5 Overhead

PCA-based compression introduces additional overhead that cannot be directly assigned to any function. This overhead is not taken into account by the compression ratio r_{compr} defined in Section 2.4 because r_{compr} is intended to reflect the compression of the data that has to be retrieved when evaluating a single function. This assumes that the function independent overhead is permanently held in memory by the application. But obviously we cannot completely ignore the overhead. To quantify the overhead introduced by a compression method, the relative overhead $r_{overhead}$ is defined as the size of the overhead data in relation to the size of the compressed function data.

$$r_{overhead} := \frac{\text{size of overhead data}}{\text{size of compressed function data}} \tag{8}$$

Again, to calculate the size of the overhead data, floating point values are counted as eight bytes and integer values as four bytes.

2.6 Error Measures

The compression approaches described in this work are *lossy* compression methods, that is, the data restored from its compressed representation will not be identical to the data originally compressed. While some deviation from the original data is deemed acceptable for the application, this deviation must certainly be within well-known bounds for the data to be useful.

The algorithms described in this work allow the specification of upper bounds for the maximum relative error of the approximation $\hat{e}$, the mean relative error $\bar{e}$, or both of them. For a travel time function f with period P and an approximation f^*, $\hat{e}$ and $\bar{e}$ are defined as follows.

$$\hat{e}(f, f^*) := \max_{t \in [0,P)} \left\{ \frac{|f(t) - f^*(t)|}{f(t)} \right\}$$

$$\bar{e}(f, f^*) := \frac{1}{P} \cdot \int_0^P \frac{|f(t) - f^*(t)|}{f(t)} dt$$

Intuitively, this means that if $\hat{e}$ is limited to, for example, 1% the value of a restored function f^* at a point t will not differ from the value of the original function f at the same point by more than one percent. On the other hand, limiting $\bar{e}$ to 0.01 may result in values deviating from the original by more than one percent, but the average value of that deviation will not exceed one percent. Note that both $\hat{e}$ and $\bar{e}$ are quite simple to calculate when both f and f^* are piecewise linear functions, which is generally the case here.

While $\hat{e}$ and $\bar{e}$ apply only to one function at a time, the concepts can be transferred to a set S of functions and their approximations as defined below.

$$\begin{aligned}
\bar{e}_{mean}(S) &:= |S|^{-1} \sum_{(f,f^*) \in S} \bar{e}(f, f^*) \\
\bar{e}_{max}(S) &:= \max_{(f,f^*) \in S} \{\bar{e}(f, f^*)\} \\
\hat{e}_{mean}(S) &:= |S|^{-1} \sum_{(f,f^*) \in S} \hat{e}(f, f^*) \\
\hat{e}_{max}(S) &:= \max_{(f,f^*) \in S} \{\hat{e}(f, f^*)\}
\end{aligned}$$

2.7 Implementation

The algorithms and techniques described in Sections 3 through 6 were implemented in C++ and are based on the Standard Template Library (STL). Parallelization was implemented using OpenMP and the GNU Scientific library (GSL) [Gal] was used for mathematical calculations such as linear algebra operations, eigensystem calculation and optimization.

3 Building Blocks

The approaches to solving the problem at hand described in Sections 4 through 6 share a number of components. These basic concepts are described in this section.

3.1 PCA-Based Compression

The choice to use principal component analysis for the compression of travel time functions was based on the assumption that travel time functions share a number of patterns. For example, we can expect a considerable number of functions to feature similar rush hour peaks in the morning or afternoon. The Imai-Iri algorithm described in Section 2.2 which has already been evaluated for the compression of travel time functions [Neu09] makes no use of such similarity, as it compresses each function independently. In contrast to this, principal component analysis is likely to find predominant shared patterns in its input data.

Ideally, we hope to be able to describe a large number of functions using only a few patterns found by the principal component analysis. Using this technique, the compressed representation of a set of travel time functions consists of two parts: the set of patterns $p_1, \ldots, p_k$, which is shared among all functions, and a coefficient vector $c = (c_1, \ldots, c_k)$ for each function. As described in Section 2.3, the mean of the data set is first subtracted from each function during principal component analysis. This mean vector can be interpreted as an additional pattern vector p_0 with an implicit coefficient of 1. A compressed function can be restored from the pattern vectors and its coefficient vector as a linear combination of the patterns weighted with the elements of the coefficient vector.

$$f^* = p_0 + p_1 \cdot c_1 + \ldots + p_k \cdot c_k = \begin{pmatrix} p_0 & p_1 & \ldots & p_k \end{pmatrix} \cdot \begin{pmatrix} 1 \\ c_1 \\ \vdots \\ c_k \end{pmatrix}$$

3.1.1 Run Length Coding

In practice, the number of pattern vectors needed to meet the error bounds varies from function to function. Storing the same number of coefficients for each function would thus result in largely inefficient compression. Instead, the number of coefficients necessary to meet the error bounds is determined individually for each function using a bisection method, and only the required coefficients are stored. This method obviously adds the overhead of storing an integer value indicating the number of coefficients for each function.

3.1.2 Comparison to the Imai-Iri Algorithm

Due to its availability and known properties, the Imai-Iri algorithm is used as the main comparison standard for the compression methods developed in this work. Gen-

erally speaking, it is our goal to surpass the compression achieved by the Imai-Iri algorithm for as many functions of the input data set as possible. To be able to conveniently refer to this comparison, the ratio $r_{ImaiIri}$ is defined for each function f as shown in equation 9.

$$r_{ImaiIri}(f) := \frac{values_{PCA}(f)}{values_{ImaiIri}(f) - 1} \tag{9}$$

In this definition, $values_{PCA}(f)$ denotes the number of floating point values required to represent f using the PCA-based compression discussed in that context, counting only the per function data (the coefficient vector), not the global overhead induced by the pattern vectors. By the same token, $values_{ImaiIri}(f)$ denotes the number of values required by the Imai-Iri approximation of f to the current error bounds. The subtraction of 1 in the denominator of the definition is owed to the implementation of the automatic clustering algorithm discussed in Section 6. Because $values_{PCA}$ and $values_{ImaiIri}$ are both integer, the condition $r_{ImaiIri}(f) \leq 1$ is equivalent to $values_{PCA}(f)/values_{ImaiIri}(f) < 1$.

3.2 Sampling

As described in subsection 2.3, principal component analysis works on a set of random variables with each item in the input data containing a value for each of these variables. In the original data, each function is defined as a sequence of data points $(t, f(t))$. The values t for which $f(t)$ is specified vary from function to function, as does the number of data points. To obtain a uniform set of data points for all functions, the functions are sampled at S equidistant sampling points using linear interpolation. From this follows directly the sampling distance $\Delta_S := \frac{P}{S}$. The sample vector s_f of a function f is defined as follows.

$$s_f := \left(f(0) \quad f(\Delta_S) \quad \ldots \quad f((S-2)\Delta_S) \quad f((S-1)\Delta_S) \right)$$

The choice of the sampling distance has an effect on multiple aspects of the compression algorithm. Naturally, choosing a large sampling distance will increase the sampling error. As the principle component analysis can only approximate the sampled data, a high sampling error will lead to a greater approximation error and poor compression ratio. Choosing a low sampling distance will decrease the sampling error, but will also significantly increase the memory requirements and runtime of the compression algorithm. Because the PCA-based compression actually compresses the sampled data, increasing the number of sampling points could also lead to worse overall compression, because more input data is supplied to the algorithm.

3.2.1 Distribution of the Data Points

To minimize the sampling error, it is obviously desirable to sample the functions at points close to the positions of the data points the function is made up of. Ideally,

we would sample each function at each of its definition points and thus achieve error-free sampling. Due to the high number of distinct definition points—and because the same set of sampling points must be used for all functions—doing this would result in a prohibitively high number of sampling points. Instead, we attempt to find a set of equidistant sampling points that is reasonably close to most of the definition points of the functions.

Figure 4 illustrates the influence of the sampling distance on the mean distance between data points and the closest sampling points. For each sampling distance, the left column and the values on the left axis show the absolute average distance to the next sampling point. Note that the average absolute distance for a sampling distance of 10 minutes is actually slightly higher than that for 15 minutes. This indicates that 10 minutes is a particularly ill–fitting sampling distance for this data set. Apart from this one case, the figure shows that the average absolute distance decreases with the sampling distance, as one would expect.

The right columns together with the values on the right axis in Figure 4 show the relative average distance to the next sampling point, that is the absolute distance divided by half the sampling distance, which is the maximum possible distance. The horizontal line at 50% indicates the value that would result if the data points were uniformly distributed. The figure shows that several sampling distances achieve relative distances lower than 50%, with 15 minutes reaching as low as approximately one third. The fact that sampling distances of 30, 10 and 2 minutes result in average distances of roughly 50% while 15, 5, 1.5, 1 and 0.5 minutes achieve significantly better results indicates that choosing a sampling rate that is a factor of 15 minutes is advisable.

3.2.2 Sampling Error

Figure 5 shows the relation of the relative sampling error to the sampling distance. For each sampling distance, the left column shows the average mean error $\bar{e}_{mean}$ as defined in Section 2.6, while the right column indicates the global maximum error $\hat{e}_{max}$.

The figure shows a pattern similar to the observations from the previous experiment: Again, a sampling rate of ten minutes leads to significantly worse results than one of 15 minutes. The other values behave like expected, with the sampling error decreasing with the sampling distance.

3.2.3 Memory and Runtime Requirements

The dominant factor in memory consumption in the compression algorithm is storing the sampled representations of the travel time functions that are to be compressed. The data set contains 2,380,285 non–constant functions. When storing each sample as an eight byte floating point value, the memory requirements range from approximately 436 megabytes to 51 gigabytes of data for the sampling rates examined above.

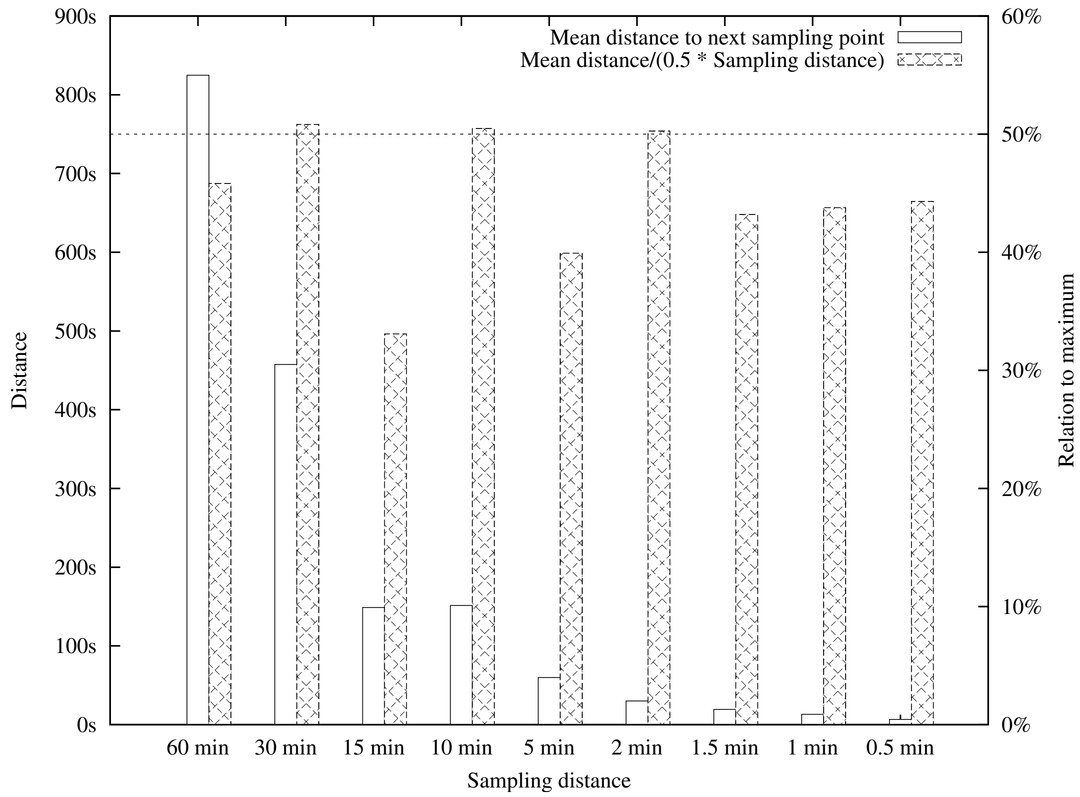

Figure 4: Average distance between data points and closest sampling point.

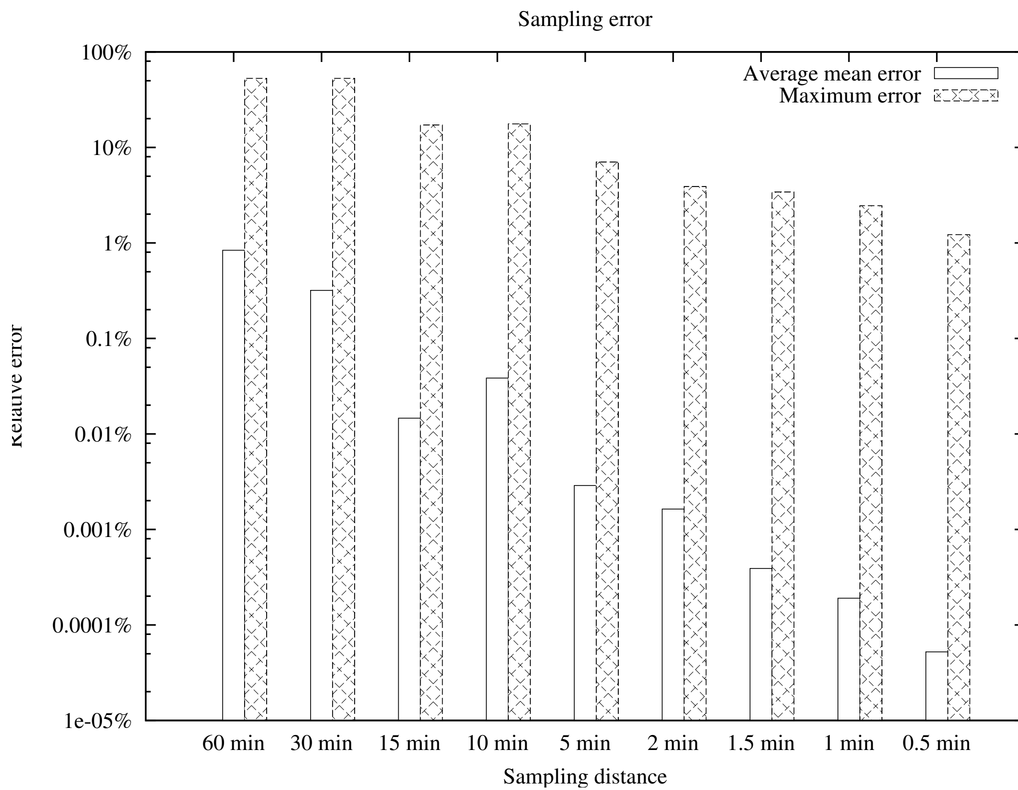

Figure 5: Average and maximum sampling error for different sampling distances.

In addition to the memory consumption, the sample rate also significantly impacts the runtime of the program. For example, the time needed for the calculation of the covariance matrix for the principal component analysis increases with the square of the number of samples used.

3.3 Normalization

During principal component analysis, the covariance matrix C of the sampled data is calculated as shown in Equation 10, where s_f denotes the sample vector for function f and $\bar{s}$ the mean of all sample vectors.

$$C = \sum_f (s_f - \bar{s}) \cdot (s_f - \bar{s})^T \tag{10}$$

Because the dataset contains not only short inner-city street segments and multiple kilometers long highway segments, but also the *shortcuts* used by the TCH algorithm, the magnitude of the functions varies greatly. The average travel times found in the data set range from a few seconds to up to multiple hours. When calculating the covariance matrix, functions with large magnitudes dominate the result, diminishing the influence of many low to medium magnitude functions.

This effect is detrimental to the compression ratio achieved by the algorithm because the algorithm aims to meet *relative* error bounds. This means that small absolute errors on low-magnitude functions are as critical as large absolute errors on high-magnitude functions. When high-magnitude functions virtually eliminate the impact of low-magnitude functions on the covariance matrix, and subsequently the principal components, it is difficult to efficiently compress those functions using the resulting principal components.

To obviate this effect, the sampled function data can be *normalized* before calculating the covariance matrix. Through normalization the sampled representation of a function is effectively split up into a magnitude value and a vector describing the shape of the function. We use the average of the sample values of a function as its magnitude. The sample vector is divided by the magnitude to obtain the shape vector. In the remaining process, only the shape vectors of the functions are considered. At the end of the compression process, the magnitude value is stored with the compressed representation of the function. A function is restored by decompressing the shape vector and multiplying the resulting approximation of the shape vector with the magnitude value.

3.4 Smoothing

As the data is largely derived from measurements, it is possible—if not likely—that it contains some sort of measurement noise. The presence of such noise would be detrimental to the compression performance because it adds superfluous information and blurs the patterns the principal component analysis is supposed to find.

To remove or at least dampen such measurement noise, the function data can be smoothed using a *Gaussian filter*. In theory, smoothing with a Gaussian filter is calculated by folding the data to be smoothed with a Gaussian function g_σ with standard deviation σ as defined in equation 11. To simplify the implementation, we use a discrete approximation of this calculation.

The smoothing process works by replacing the travel time value y_i at each data point (x_i, y_i) of a travel time function $f = ((x_1, y_1), \dots, (x_n, y_n))$ with a weighted average of the values of f within a radius r around x_i. The weights used for the average are the values $g_\sigma(k)$ of a Gaussian function with standard deviation σ for $k = -r, \dots, r$. The radius r is calculated from the standard deviation σ in such a way that $g_\sigma(r)$ is greater than or equal to a threshold value w_{min}, while $g_\sigma(r+1)$ is not. For w_{min} a small value (e.g., 0.001) is chosen to ensure that only insignificant weights are discarded. The values of f used for the average are sampled at multiples of a sampling distance Δx that is a parameter of the process.

Formal definitions for these calculations are given in equations 11 through 14.

$$g_\sigma(x) := \exp\left(-\frac{1}{2}\left(\frac{x}{\sigma}\right)^2\right) \tag{11}$$

$$r := \max\{m \in \mathbb{N} \mid g_\sigma(m) \geq w_{min}\} \tag{12}$$

$$smooth_{\sigma,\Delta x,f}(x) = \frac{\sum_{i=-r}^{r} g_\sigma(i) f(x + i\Delta x)}{\sum_{i=-r}^{r} g_\sigma(i)} \tag{13}$$

$$f^* = \Big((x_1, smooth_{\sigma,\Delta x,f}(x_1)), \dots, (x_n, smooth_{\sigma,\Delta x,f}(x_n))\Big) \tag{14}$$

The impact of smoothing on the performance of the compression algorithm is, however, of limited value. It is not possible to distinguish between measurement noise and genuine data being smoothed out, and either case is likely to lead to lower compression ratio.

Figure 6 shows an example of a travel time function smoothed with σ=15 minutes and Δx=1 minute.

3.5 Peaks

Some of the approaches described in the following sections make use of *peaks* of travel time functions. We define a peak of a travel time function as a significant, but not necessarily global, maximum of the function. Travel time functions generally contain multiple local maxima, and while the peaks that are of interest are among these local maxima, most of the local maxima are insignificant fluctuations. The global maximum is obviously a far too restrictive definition, as there is usually only one in each function. Consequently, a *peak* is defined as a *regional maximum* within a radius W. Equation 15 gives the formal definition of the set of peaks of a travel time function f for a radius W.

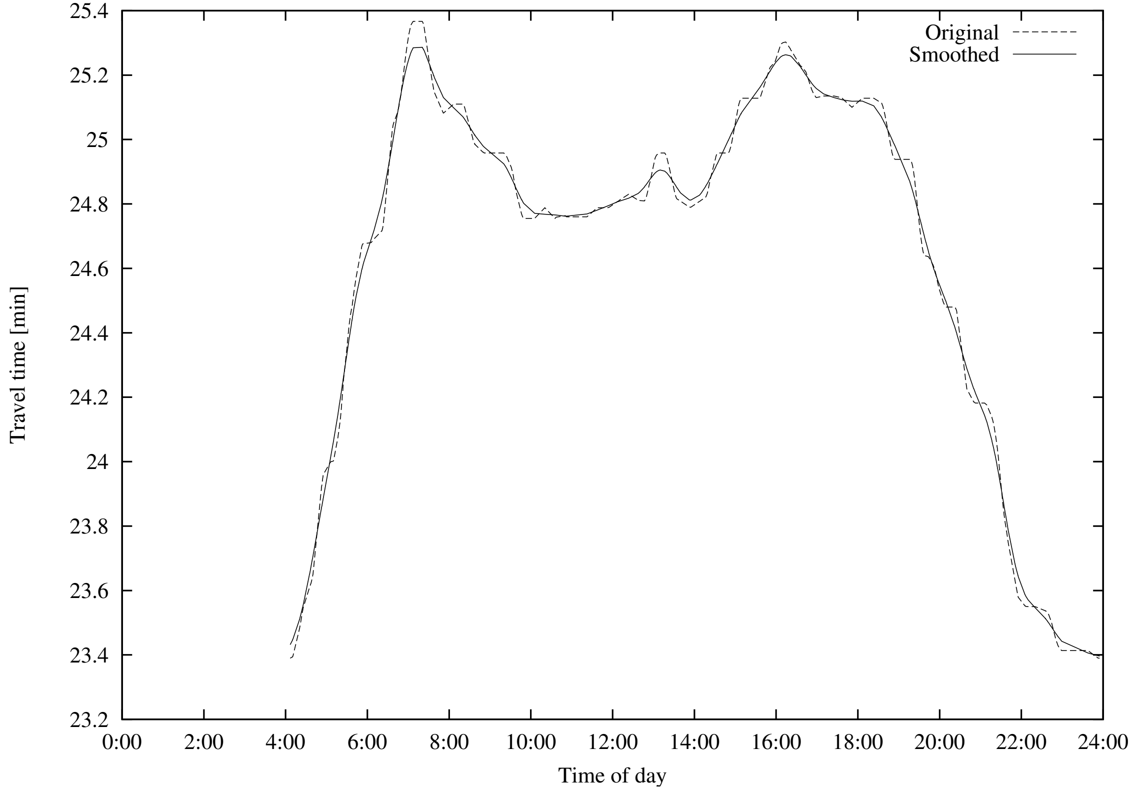

Figure 6: Example function with smoothing for σ=15 minutes and Δx=1 minute.

$$peaks_W(f) := \Big\{(x_i, y_i) \in f \;\Big|\; y_i \geq y_j \vee |x_i - x_j| > W \;\forall (x_j, y_j) \in f\Big\} \tag{15}$$

In other words, a peak is a point that is at least as high as all other points of the function that are not further away from it than a distance W.

While the definition above is a useful definition for the significant peaks of a travel time function, it gives little information about the shape of the peak. For example, it does not distinguish between a narrow peak and the highest point of a flat plateau within a function. To compensate this, we define *extended peaks*. An extended peak (a, b, h) consists of the interval $[a, b]$ the peak spans and its height h. The calculation of an extended peak from a normal peak (x_i, y_i) of a travel time function $f = ((x_1, y_1), \ldots, (x_n, y_n))$ is described in the following equations.

$$var(f) := \max(f) - \min(f) \tag{16}$$

$$left(x_i, y_i) := \min\Big\{k \in [1, i] \;\Big|\; y_i - f(x_j) \leq var(f) \cdot r \;\forall j \in [k, i)\Big\} \tag{17}$$

$$right(x_i, y_i) := \max\Big\{k \in [i, n] \;\Big|\; y_i - f(x_j) \leq var(f) \cdot r \;\forall j \in (i, k]\Big\} \tag{18}$$

$$extended(x_i, y_i) := \big(x_{left(x_i,y_i)}, x_{right(x_i,y_i)}, y_i\big) \tag{19}$$

Equation 16 defines $var(f)$ as the difference between the maximum and the min-

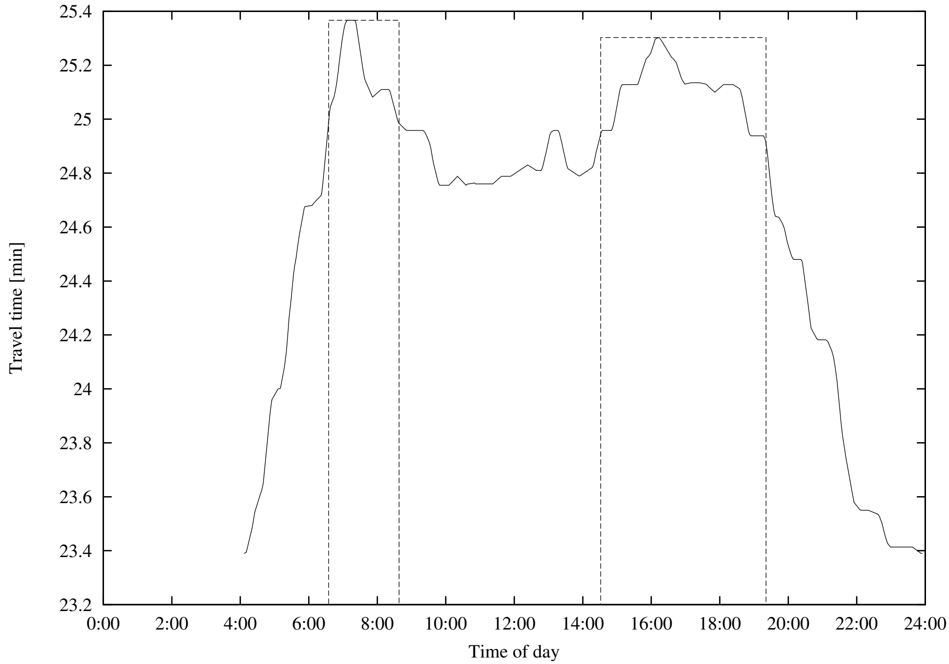

Figure 7: Example function with peaks for W=1 h and r=20%.

imum of f. This is used in the two following definitions to compare the difference between values of the function to the total range of its values. Equations 17 and 18 define the indices of the points in f that form the borders of the interval of the extended peak. Given a normal peak (x_i, y_i), $a = x_{left(x_i,y_i)}$ is the location of the data point in f that is furthest left of x_i but still fulfills the condition that all values y_j of f between a and x_i are close to y_i. Closeness is defined by limiting the difference between y_j and y_i to a fraction r of $var(f)$. In Equation 19 the pieces are put together, constructing the extension of the peak (x_i, y_i) from the preceding definitions. The right border of the interval b is calculated in the same way.

Extended peaks can of course also be identified by their center $\frac{a+b}{2}$, width $r - l$, and height h.

Figure 7 shows an example travel time function with the extended peaks found for W=1 h and r=20% drawn as dashed boxes in the plot.

4 Manual Clustering

Manual clustering is a simple, heuristics-based attempt at preprocessing the input data before applying PCA-compression. It is grounded in the assumption that the dataset contains subsets of similar functions and that PCA-compression of such subsets, if they can be identified, should be more efficient than compressing the entire data set.

4.1 Clustering

As described in Section 3.1.1, a compressed function is stored as the shortest prefix of its coefficient vector that is sufficient to meet the given error bounds. Consequentially, the size of the compressed representation of a function is determined by the number of pattern vectors needed, and their position within the pattern set. This has some notable implications on the behaviour of the compression algorithm. Even if a function can be adequately described by a single vector, we still require k values for its compressed representation if that vector is at position k in the set. This problem could of course be solved by storing only coefficients for subset of pattern vectors actually needed to describe the function. This approach is not pursued here because it drastically increases the complexity of finding the compressed representation of a function: While there are only S possible prefixes of the coefficient vector, there are 2^S subsets of vectors that might be used.

Assuming that the problem described above adversely affects PCA-based compression, it is appropriate to find and use multiple sets of pattern vectors, and to represent each function through only one of them. This approach makes it more likely that the pattern vectors needed to describe a given function are among the first of the respective set, which leads to more efficient compression.

The pair formed by a set of functions and the pattern vectors used to describe the functions is called a *cluster*. A determining characteristic of a cluster is the number of functions it contains, the cluster's *size*. While functions in a very small cluster may be efficiently compressed, the overhead induced by the pattern vectors will likely be too large. It is only sensible to speak of »compression« at all, if the functions in a cluster clearly outnumber the pattern vectors.

4.2 Algorithm

In this approach, clusters of similar functions are chosen based on peak patterns. To determine criteria for classifying the functions into subsets, we examine the general distribution of peaks in the data set. Based on these observations, several intervals from the definition domain of the functions are selected and the functions are classified based on how many peaks they have in each interval. This results in a partitioning of the data set into multiple subsets, which are then compressed separately using a simple PCA-based algorithm. To limit the number of subsets and to avoid overly

```
1  prog SimplePCA(D, # Data set.
2                 e, # Error bounds.
3                 s) # Sampling rate.
4  do
5    sample(D, s);
6    normalize(D);
7    PC = principal_components(D);
8    for each f in D
9    do
10     if pca_compress(f, PC, e) < imai_iri_compress(f, e)
11     then
12       # Use PCA compression for f.
13     else
14       # Use Imai–Iri compression for f.
15     endif
16   done
17 done
```

Listing 1: The simple PCA algorithm.

small subsets, subsets containing less than 5% of the non-constant functions of the data set are not treated separately, but are instead merged into a single set.

4.2.1 The Simple PCA Algorithm

The simple PCA algorithm uses normalization (Subsection 3.3) and calculates the compressed representation of the functions as described in Subsection 3.1. For each function, either the coefficient vector resulting from the PCA process, or the output of the Imai–Iri algorithm is used, depending on which one is smaller. A pseudocode representation of the simple PCA algorithm is shown in Listing 1.

4.3 Experiments

4.3.1 Classification

As described in Subsection 3.5, the peak search algorithm requires two parameters, the window size W and the relative difference r. For the experiments in this section, the peak search was conducted with values of 0.5, 1, 2 and 3 hours for W and 10% and 20% for r.

The peak search finds approximately 2.2 to 2.6 peaks per non-constant function on average. As an example, Figure 8 shows the distribution of the peaks for W=2 h and r=10%. The other parameter configurations produce similar results. In the figure, three regions with a higher concentration of peaks are visible. The three intervals $I_1 = [6\,h, 10\,h)$, $I_2 = [10\,h, 15\,h)$, and $I_3 = [15\,h, 18\,h)$ containing these regions are selected as marked by the dashed lines in Figure 8 and used to classify the non-constant functions of the data set. Figure 9 shows the frequency of the different peak

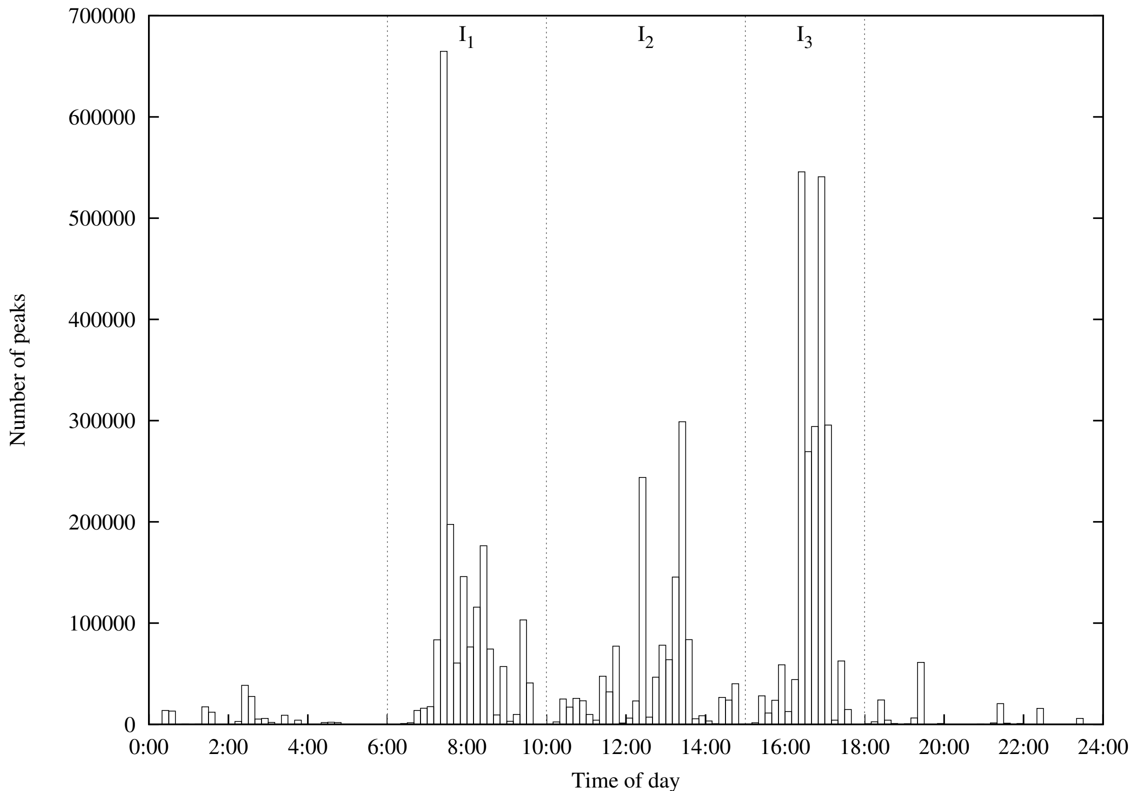

Figure 8: Example peaks distribution.

patterns found in the data set. The pattern strings in the key of the figure indicate how many peaks were found in each of the three intervals.

4.3.2 Compression

The subsets obtained from the classification were compressed using the simple PCA algorithm described above with a sampling distance Δ_S of five minutes and a maximum relative error $\hat{e}_{max}$ of 1%. The results of these experiments are shown in Table 2. In addition to the compression ratio r_{compr} (see Section 2.4) the table shows the fraction of non-constant functions that were PCA-compressed r_{PCA} and the relative overhead $r_{overhead}$ (Section 2.5) introduced by the compression. The bottom two rows of the table show the results produced by the Imai-Iri algorithm and the simple PCA algorithm without clustering are listed for comparison. As the table shows, the clustering results in approximately twice as many PCA-compressed functions while improving the compression ratio r_{compr} by approximately one quarter. While the overhead increases considerably, it is still relatively small compared to the compressed function data.

While these results suggest that manual clustering can be used to improve the compression ratio achieved by the simple PCA algorithm (see Listing 1), it is unclear whether this is due to the specific clustering used here or if the algorithm simply performs better on smaller data sets. To answer this question, we applied the simple

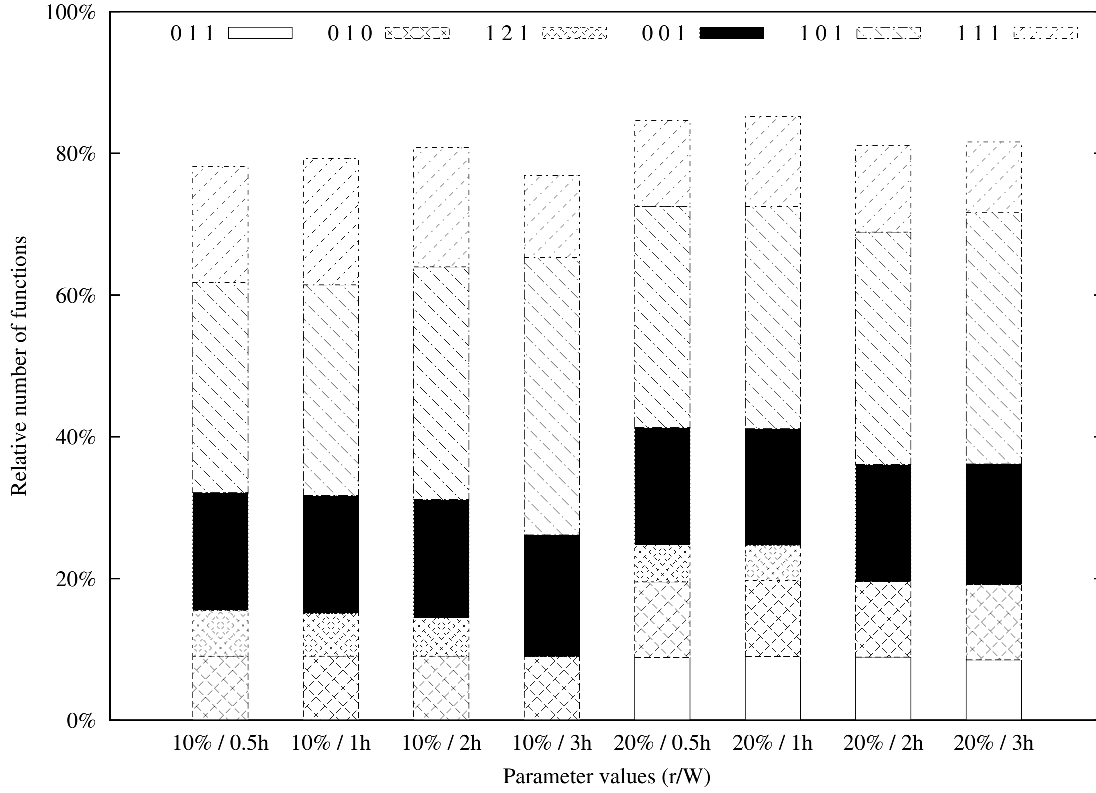

Figure 9: Frequencies of the peak patterns.

r	W	r_{compr}	r_{PCA}	$r_{overhead}$
10%	0.5 h	10.658%	63.5%	0.15%
	1 h	10.729%	63.4%	0.16%
	2 h	10.773%	62.2%	0.14%
	3 h	11.199%	56.7%	0.12%
20%	0.5 h	10.369%	62.9%	0.20%
	1 h	10.427%	63.2%	0.19%
	2 h	10.785%	60.8%	0.16%
	3 h	10.856%	59.0%	0.16%
Imai-Iri		14.256%	0%	0%
Not clustered		13.283%	28.9%	0.03%

Table 2: Compression results for different peak search parameters.

PCA algorithm to ten random sample data sets containing 5% of the non-constant functions of the original data set each. For each of these samples, the algorithm produced results similar to those produced for the entire data set (listed in the last row of Table 2), albeit with much higher overhead. Furthermore, random partitions of the data set with varying subset sizes also produced the same results.

5 Model-Based Transformation

In this section, we discuss an attempt improve the compression achieved by the simple PCA algorithm presented in Section 4 through another preprocessing step. The basic idea is to modify the travel time functions in a way that increases the similarities between them, hopefully improving the efficiency of the PCA-based compression.

5.1 Algorithm

We define and test the model-based transformation approach with the 1–0–1 subset produced in Section 4, that is the subset of functions containing one peak between six and ten o'clock in the morning and one between three and six o'clock in the afternoon. This is done because these functions match our intuition of a typical commuter traffic pattern with rush hour peaks in the morning and afternoon. However, the approach can be analogously applied to the other subsets found in Section 4 that have at most one peak per interval.

Figure 10 shows the distribution of the peaks found in the 1–0–1 set for r=20% and W=1 h. Peaks outside of I_1, I_2, and I_3 were omitted for clarity. The results of Section 4 showed that a partitioning of the data set based on the number and location of the peaks of the functions can improve the compression ratio achieved by the simple PCA algorithm. But as can be seen in Figure 10, the peaks are still distributed over the respective interval. The approach presented in this section is intended to modify the functions in a way that makes their peaks more similar both in shape and location, hopefully leading to improved compression by the simple PCA algorithm.

First, we attempt to obtain a better understanding of the shapes and locations of the peaks. To refine the information returned by the peak search algorithm (Section 3.5), a model based on Gaussian functions is fitted to each function.

5.1.1 Model

The model used in this approach approximates the peaks of the functions with Gaussian functions $g_{x,h,\sigma}(t)$ as defined in Equation 20. Gauss functions are a natural choice for this purpose because they have a peak-like shape and converge towards zero on both sides of the peak. The location, height and width of the peak can be adjusted through the x, h, and σ parameters respectively.

For a function with two peaks, like those in the 1–0–1 set, the approximation in the model consists of a constant term c and three Gaussian functions—one *base* function and one for each peak—that are added up (see Equation 21). The constant term c models the minimum travel time of the road segment that is independent of the traffic situation. The base Gaussian function models a general increase in travel time during the day time. The two remaining Gaussian functions model the peaks.

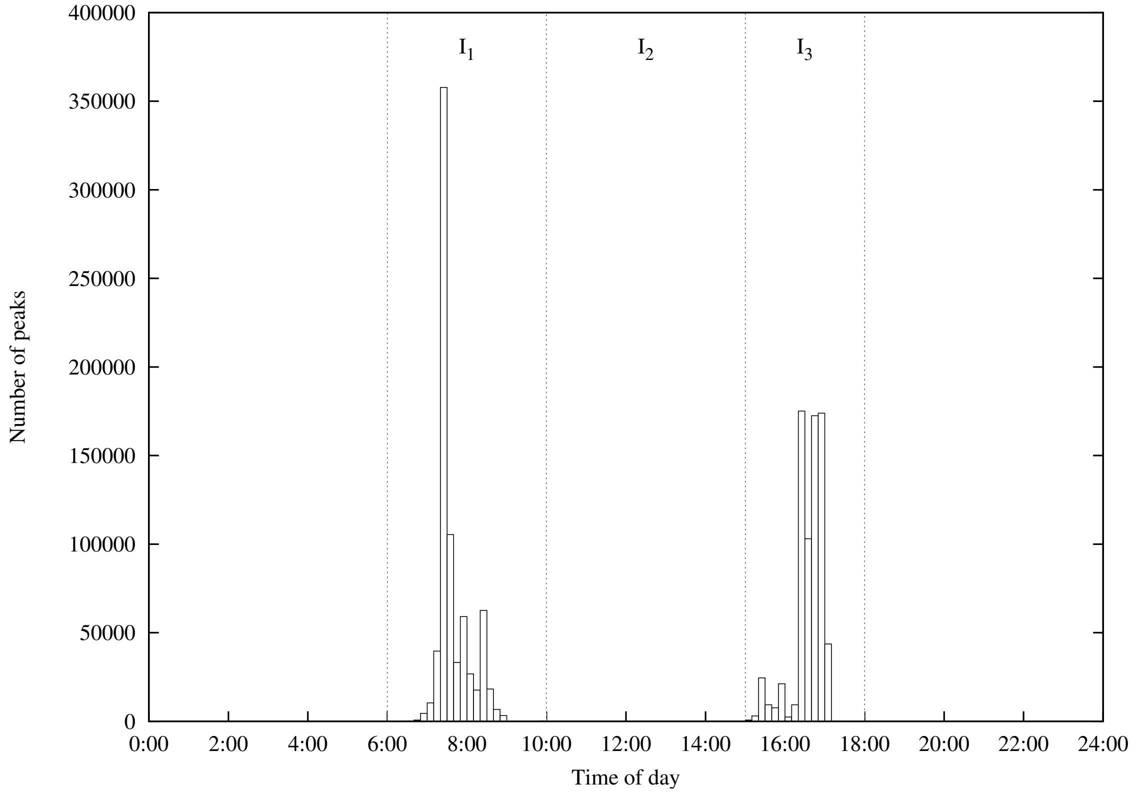

Figure 10: Distribution of peaks in the 1–0–1 set for r=20%, W=1 h.

$$g_{x,h,\sigma}(t) := h \cdot \exp\left(-\frac{1}{2} \cdot \left(\frac{t-x}{\sigma}\right)^2\right) \tag{20}$$

$$f^*(t) = c + \sum_{i=0}^{2} g_{x_i,h_i,\sigma_i}(t) \tag{21}$$

5.1.2 Model-Fitting

To fit the model defined above to a given travel time function f, we must determine values for the parameters c and x_i, h_i, σ_i for $i = 0, 1, 2$ that minimize the difference between f and its approximation f^*. To facilitate this, we define the difference function $\Delta_d(f, f^*)$ in Equation 22. The difference function approximates the difference between f and f^* by evaluating both functions at multiples of the sampling distance d and calculating the sum of the squares of the differences. For each travel time function f the sought after parameters are first estimated from the results of the peak search. These estimates are then refined by using a generic optimization algorithm to minimize $\Delta_d(f, f^*)$. We use the Simplex algorithm of Nelder and Mead [NM65], implemented in the *GNU Scientific Library*, version 1.14 [Gal, p. 394f] for this minimization. The algorithm finds a local minimum of the supplied function.

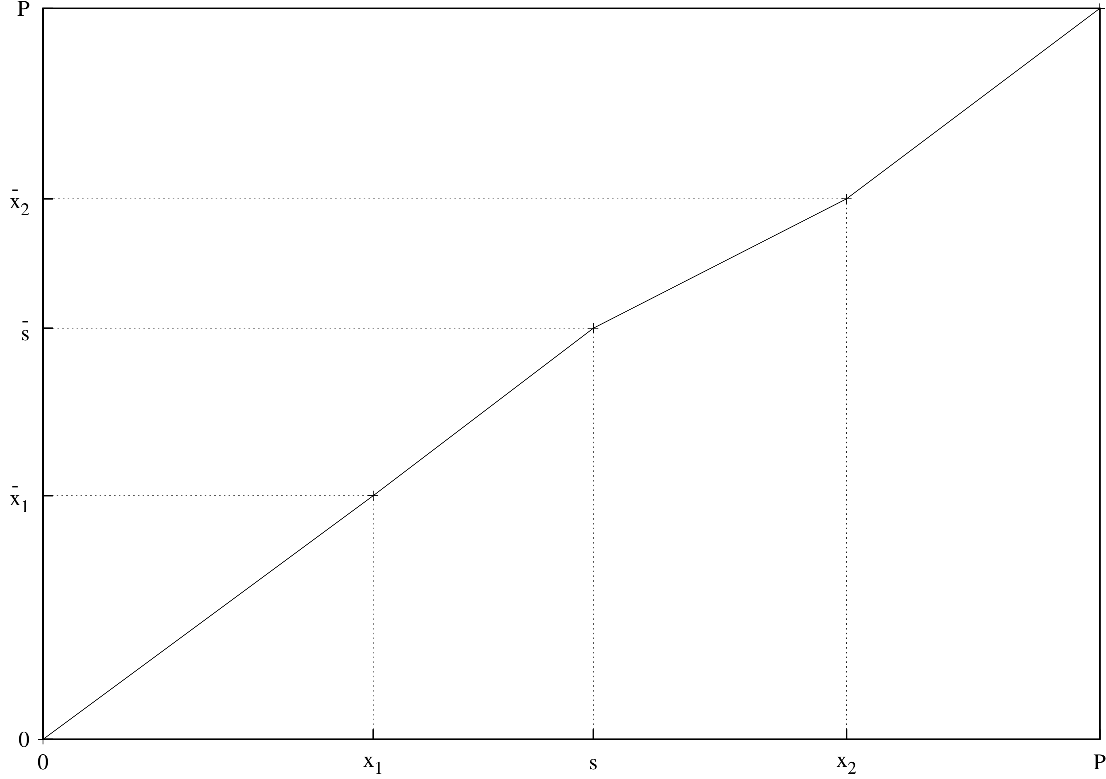

Figure 11: Example transformation function.

$$\Delta_d(f, f^*) = \sum_{i=0}^{\lfloor \frac{P}{d} \rfloor} (f(i \cdot d) - f^*(i \cdot d))^2 \tag{22}$$

5.1.3 Transformation

The location and width information about the peaks of the functions produced by the fitting step (i.e., the values of x_1, σ_1, x_2, and σ_2) are used to unify the shapes of the functions. For each travel time function f, the piecewise linear transformation function $T_f : [0, P) \rightarrow [0, P)$ is defined by five data points: $P_0 = (0, 0)$, $X_1 = (x_1, \overline{x_1})$, $S = (s, \bar{s})$, $X_2 = (x_2, \overline{x_2})$, $P_P = (P, P)$, where P is the period of the travel time functions. Figure 11 shows an example transformation function. While the first and last data points P_0 and P_P are the same for all functions, the remaining three depend on the x_i and σ_i values produced by the fitting step. X_1 and X_2 consist of the locations of the peaks of the function x_1 and x_2 and their respective mean values $\overline{x_1}$ and $\overline{x_2}$ for the entire data set; S consists of the value s as defined in Equation 23 and its mean value $\bar{s}$. The formula for s is chosen as a weighted average of the locations of the peaks, putting s closer to the peak with the smaller width σ.

$$s = \frac{x_1\sigma_2 + x_2\sigma_1}{\sigma_1 + \sigma_2} \tag{23}$$

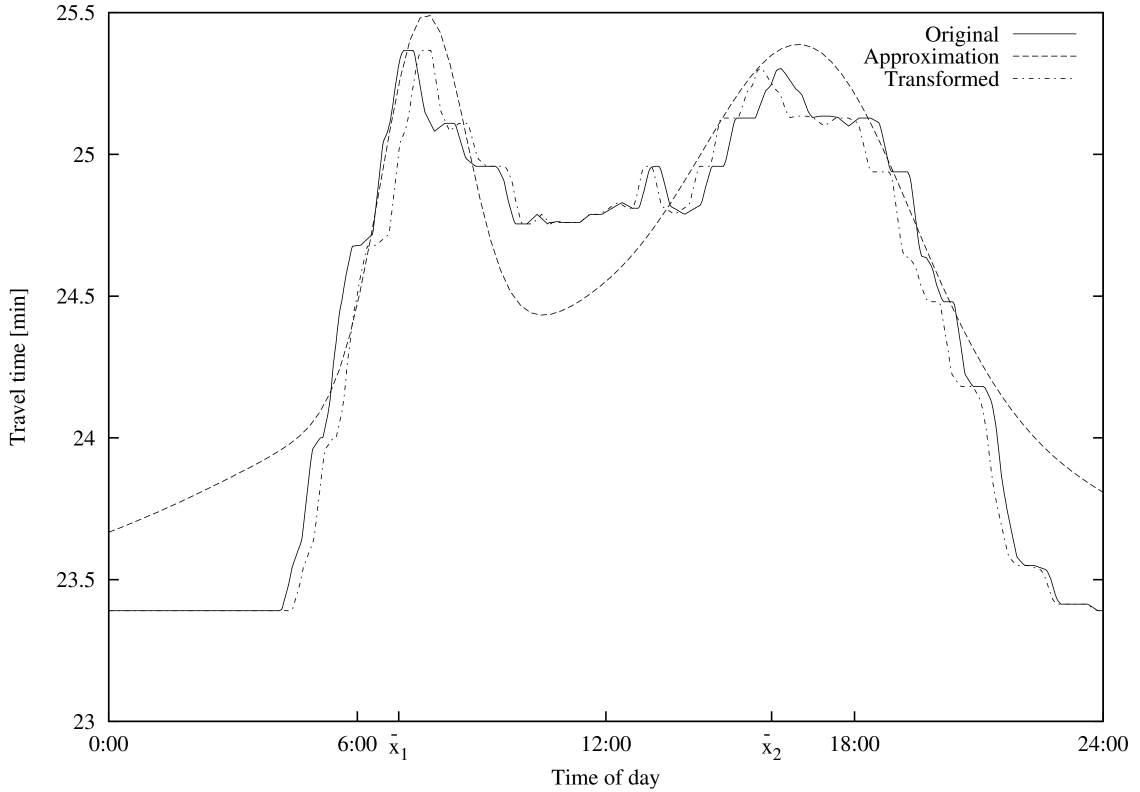

Figure 12: Example function with approximation and transformation.

The transformation f^T of a travel time function f is calculated by replacing each data point (x, y) of f with $(T_f(x), y)$, thus moving the peaks of f to the average position within the data set. Figure 12 shows an example of a travel time function, its approximation through the Gaussian-based model and the result of the transformation. The average peak locations $\overline{x_1}$ and $\overline{x_2}$ are marked on the x-axis.

$$f^T = \Big((T_f(x_1), y_1), (T_f(x_2), y_2), \ldots, (T_f(x_n), y_n)\Big)$$

5.2 Experiments

The algorithm described in Section 5.1 was executed on the eight 1–0–1 subsets resulting from the different peak search parameters used in Section 4. The functions in the 1–0–1 sets have one peak in each of I_1 and I_3. Before the compression results are presented in Section 5.2.2, the sampling behavior of the transformed data is discussed.

5.2.1 Sampling

In Figure 13, the impact of the transformation on the sampling process is illustrated by two plots. The plot on the left shows the average mean sampling error $\bar{e}_{mean}$ before and after the transformation for different sampling distances; the plot on the

right shows the percentage of functions with maximum sampling error $\hat{e}$ greater than 1%, again before and after the transformation and for different sampling distances. The values shown in the plots are the averages of the eight 1–0–1 sets used for the experiments. While the plots show an expected decrease in sampling error for higher sampling rates (cf. Section 3.2), they also show a considerably higher sampling error for the transformed data. The increase in functions with a sampling error $\hat{e}$ greater than 1% is also considerable.

This considerable deterioration in the sampling behavior of the data can be explained as a direct result of the transformation. As discussed in Section 3.2, the original data can be sampled relatively effectively because many data points of the travel time functions are located at multiples of 15 minutes. As the transformation defined in Section 5.1 works by moving the data points of the functions, this useful structure is likely destroyed by the transformation. This effect is visible in Figure 14 which shows the average distance of the data points to the closest sampling point in relation to the sampling distance Δ_S. The figure shows an average relative distance as low as approximately 18% of the sampling distance for Δ_S=15 minutes before the transformation. In contrast, the same measure for the transformed data is always approximately 25% of Δ_S, which indicates a more uniform distribution of the data points in the transformed data.

Figure 15 shows an example for the distribution of the relative distance to the closest sampling point for Δ_S=15 minutes which supports this indication. While approximately 15% of the data points of the original data are located at or very close to sampling points, the data points in the transformed data are distributed uniformly.

5.2.2 Compression

The simple PCA algorithm (see Section 4.2.1) was executed with sampling distance Δ_S=5 minutes and maximum relative error $\hat{e} = 1\%$. Table 3 gives an overview of the results. The first four columns are similar to Table 2 in Section 4, showing the values of the peak search parameters r and W (see Section 3.5) and the compression ratio r_{compr} and percentage of PCA-compressed functions r_{PCA} achieved by the simple PCA algorithm on the original data. The fifth and sixth column show r_{compr} and r_{PCA} achieved on the transformed data, while the last column shows the compression ratio achieved by the Imai-Iri algorithm.

The table shows that the transformation failed to achieve the desired improvements. The most striking aspect is the reduction in r_{PCA} to approximately a third to a quarter of the values achieved without the transformation. The low number of PCA-compressed function is likely a direct cause of the increase in compression ratio, which lies for the transformed data only marginally below that achieved by the Imai-Iri algorithm.

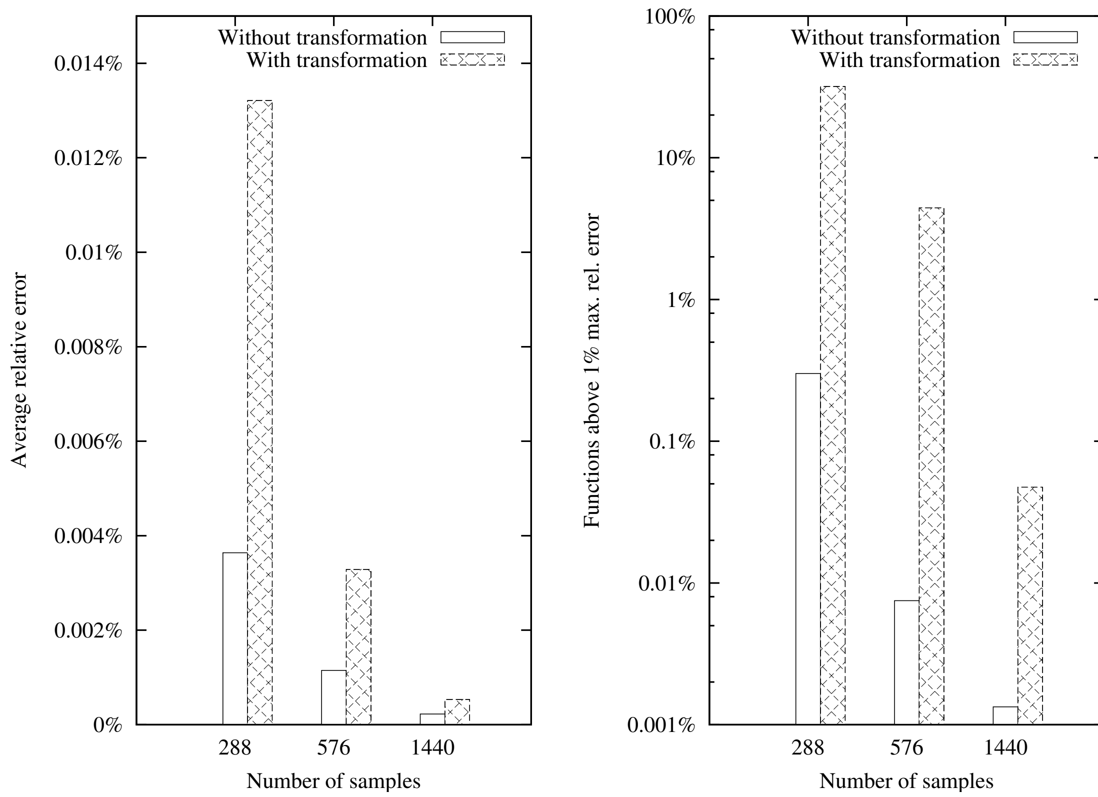

Figure 13: Sampling of transformed function data.

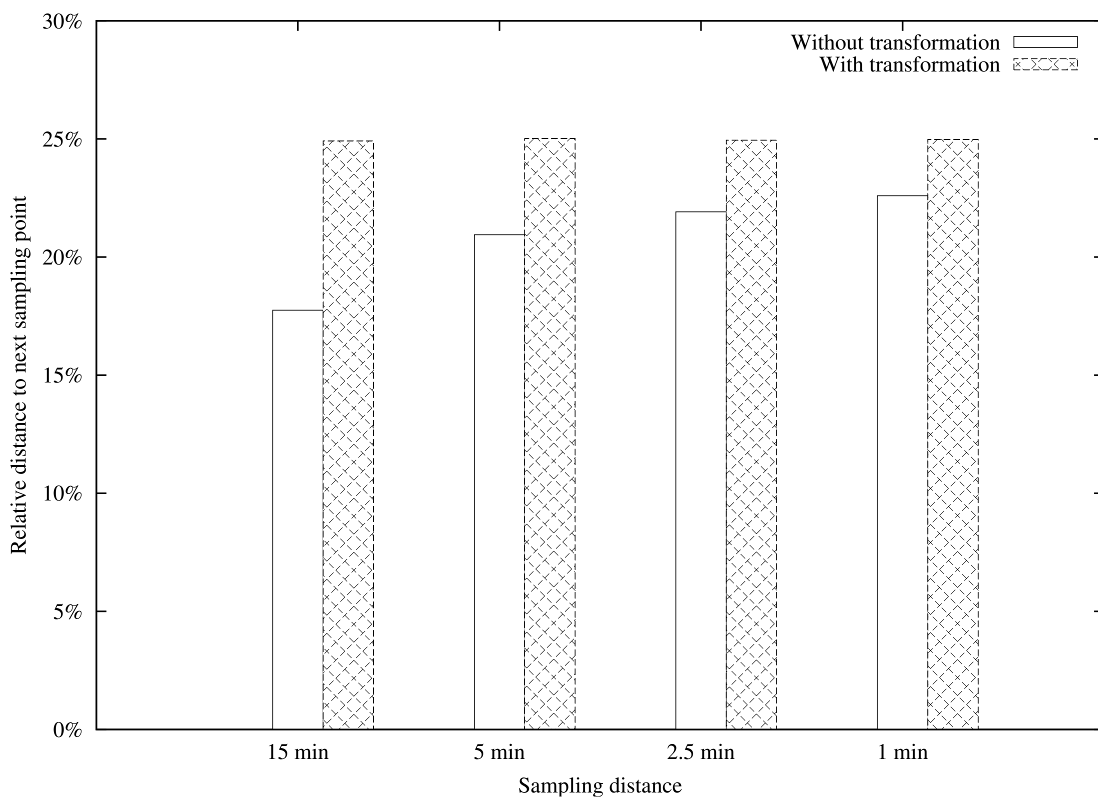

Figure 14: Comparison of average relative distance to the closest sampling point.

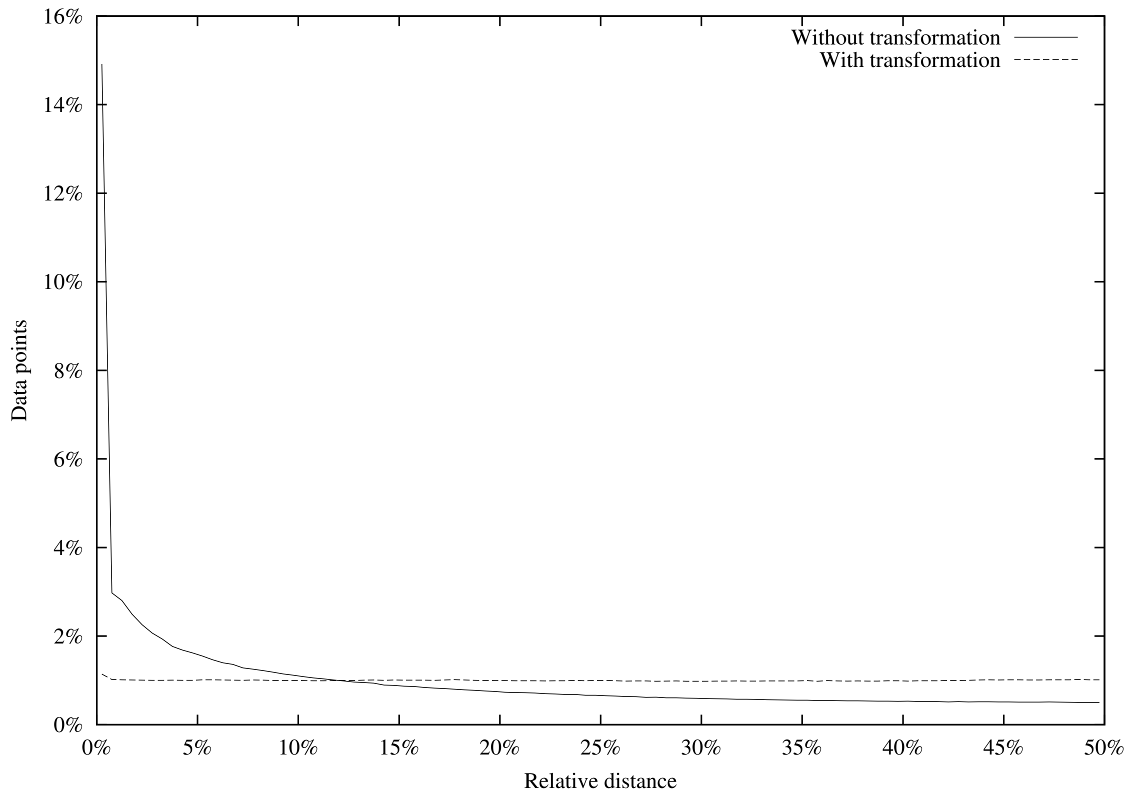

Figure 15: Example distribution of the relative distance to closest sampling point for Δ_S=15 min.

r	W	Before transformation r_{compr}	Before transformation r_{PCA}	After transformation r_{compr}	After transformation r_{PCA}	Imai-Iri
10%	0.5 h	11.96%	42.67%	13.11%	10.40%	13.23%
	1 h	13.02%	42.18%	14.24%	10.09%	14.36%
	2 h	9.70%	41.22%	10.52%	12.43%	10.66%
	3 h	7.99%	37.12%	8.55%	13.62%	8.68%
20%	0.5 h	9.73%	43.89%	10.66%	11.75%	10.79%
	1 h	9.76%	42.77%	10.67%	11.70%	10.79%
	2 h	9.27%	41.71%	10.11%	12.41%	10.24%
	3 h	9.05%	39.69%	8.82%	12.65%	9.95%

Table 3: Results of the model-based transformation approach.

```
1  prog AutomaticClusteringPCA(D) # Data set.
2  do
3    while |D| ≥ min_size
4    do
5      C = findCluster(D)
6      if C ≠ ∅
7      then
8        # Store C.
9        D = D \ C
10     else
11       break
12     endif
13   ImaiIriCompress(D)
14 done
```

Listing 2: Basic structure of the automatic clustering algorithm.

6 Automatic Clustering

In this section, the *automatic clustering algorithm* is described and evaluated. It is based on the concepts described in Section 3 and employs some additional techniques devised for this algorithm. The algorithm is similar to the manual clustering approach presented in Section 4 in so far as it also separates the data set into clusters that are compressed separately. But while the manual clustering approach chooses the clusters beforehand, the automatic clustering approach uses the PCA-compression itself to find clusters. After calculating the principal components and PCA-based compression for a set of functions, some functions are compressed relatively good, that is they have $r_{ImaiIri}$ close to or below one, while others are poorly compressed with $r_{ImaiIri}$ clearly above one. The idea of the algorithm is, that after removing the poorly compressed functions from the set, the principal component of the resulting set should be even better suited to the already well-compressed functions, further improving their compression and thus decreasing their $r_{ImaiIri}$ value.

The algorithm is explained by means of pseudocode fragments that omit some details and use a rather liberal syntax but convey the basic function of the algorithm.

6.1 Algorithm

A pseudocode representation of the basic structure of the automatic clustering algorithm is given in Listing 2. The algorithm uses the `findCluster` subroutine to find valid clusters in the input data set `D`. Here, a cluster is considered valid if it is at least as large as a minimum size `min_size` that is a parameter of the algorithm. Other requirements, for example taking into account the PCA-overhead (cf. Section 2.5) would also be possible. When no more valid clusters can be found, the functions remaining in `D` are compressed using the Imai-Iri algorithm and the algorithm terminates.

```
1  prog findCluster(D) # Input data set.
2  do
3    ΔT = T_decrement
4    T = T_start − T_decrement
5    C = D
6   while T − T_{\mathit{decrement}} ≥ 0 and ΔT ≥ ΔT_min
7    do
8      C' = stabilize(C, T − ΔT)
9      if |C'| ≥ min_size
10     then
11       C = C'
12       T = T − ΔT
13     else
14       ΔT = ΔT/2
15     endif
16   done
17
18   if T ≤ 1
19   then
20     return C
21   else
22     return ∅
23   endif
24 done
```

Listing 3: The `findCluster` subroutine.

6.1.1 Finding Clusters

The `findCluster` subroutine outlined in Listing 3 uses a *tolerance value* `T` to find optimally compressed clusters that are still valid. The tolerance value is used as an upper bound for $r_{ImaiIri}$ (see Section 3.1.2) of the functions in the cluster and gradually decreased in the course of the subroutine.

The new cluster `C` is initialized with all functions remaining in the input data set. The tolerance value T and the tolerance decrement ΔT are initialized with the T_{start} and $T_{decrement}$ parameters, respectively. For the `while` loop beginning in Line 6, the invariant »`|C|` ≥ `min_size`« holds at all times. The algorithm *stabilizes* the cluster at the next tolerance value `T`-Δ`T` using the `stabilize` subroutine. If the returned cluster `C'` is valid the used tolerance value is stored in `T` and `C'` replaces `C`. Otherwise, Δ`T` is reduced by half and the loop continues with the cluster `C`.

The loop continues while `T` is positive and ΔT is greater than the minimum ΔT_{min}. Once the loop has terminated, the last valid cluster `C` is returned if it was produced with a tolerance value not greater than one. Otherwise, ∅ is returned to indicate that the subroutine failed to find a valid cluster. This is necessary because a cluster that was produced with a tolerance value greater than one might contain functions that are compressed worse than they would be by the Imai-Iri algorithm.

The `stabilize` subroutine outlined in Listing 4 repeatedly compresses and *prunes*

```
1 prog stabilize(C, # Cluster.
2                T) # Tolerance value.
3 do
4   do
5     compress(C)
6     pruned_out = prune(C, T)
7   while pruned_out > 0
8   return C
9 done
```

Listing 4: The `stabilize` subroutine.

the cluster. In the compression step the coefficient vector of each function in the cluster is calculated (cf. Section 3.1). In the pruning step, all functions for which $r_{ImaiIri}(f) > \mathtt{T}$ holds are removed from the cluster. If the number of such functions is not zero, the cycle is repeated.

When no more functions were removed by the prune step, the cluster is considered *stable*. In this state, the functions in the cluster are compressed with pattern vectors that were calculated only from those functions.

6.2 Experiments

In this subsection, we attempt to determine the influence of the parameters of the algorithm on its performance. Due to the large number of parameters and possible values and the considerable run time of the algorithm, the parameter space is not completely covered. Instead, a basic setup with typical values for all parameters is chosen, and the values for each parameter are varied separately.

6.2.1 Parameters of the algorithm

The output and behavior of the algorithm is controlled by the following parameters.

Error bounds. The error bounds limit the relative error of the approximation produced by the algorithm. The limit for the maximum relative error $\hat{e}_{max}$ is also used as the maximum error parameter for the Imai-Iri algorithm when a function cannot PCA-compressed.
The following experiments were conducted with the maximum error $\hat{e}_{max}$ of the approximation limited to 1%. The influence of different error bounds on the results of the algorithm is discussed separately in Section 6.2.4.
The error bounds parameter is the only parameter that directly influences the quality of the approximation. The remaining parameters only influence the achieved compression ratio, the run time of the algorithm and, indirectly, other quality attributes such as the mean relative error of the approximation.

Sampling rate. The sampling rate as described in Section 3.2 is used to derive a data matrix suitable for principal component analysis from the set of travel

Parameter	Abbr.	Default	Alternative values
Maximum relative error	err	1%	0.25% 0.5% 2.5% 5% 10%
Number of samples	smp	288	96 144 192 576 960 1440
Normalization	norm	yes	no
Minimum cluster size	size	10%	1% 5% 20%
Initial tolerance	ts	1.6	1.2 1.4 1.8
Tolerance decrement	td	0.2	0.1 0.3 0.5

Table 4: Parameter values

time functions. In addition to influencing the compression performance by introducing a sampling error, the sampling rate has a major influence on the run time of the algorithm.

Normalization. As described in Section 3.3, normalization is an optional operation that is expected to give low-magnitude functions the same influence on the calculated principal components as high-magnitude ones.

Minimum cluster size. The minimum cluster size is specified relative to the number of non-constant functions in the data set and determines the minimum number of functions in a cluster produced by `findCluster` subroutine described in Section 6.1.1.

Tolerance setup. The tolerance and adaption mechanism described above is the main feature of the algorithm. The setup consists of the initial tolerance T_{start} value and the decrement value $T_{decrement}$.

The following experiments are based on a default setup from which variations were derived for each parameter. The default and alternative values for each parameter are listed in Table 4. Additionally, the second column of the table lists an abbreviation for each value that is used in plots and tables.

6.2.2 Overview

To give an overview of the results of the experiments, the more interesting parameter sets are plotted in Figure 16, showing the two primary measures compression ratio r_{compr} and mean error $\bar{e}_{mean}$. Most of the parameter combinations produced results similar to those of the default values and were omitted from the plot. The point labeled »imai-iri« marks the compression ratio and mean error achieved by the Imai-Iri algorithm for the same error bounds. Table 5 lists the same values for all experiments that were conducted. Note that default parameter set is listed in each block of the table to facilitate comparison to the non-default values. The default values are printed in italics in the »Value« column of Table 5.

As could be expected, all parameter variations achieve better compression than the benchmark Imai-Iri result. This follows directly from the way the algorithm works, as it will not use more values than the Imai-Iri algorithm for any function. In

Parameter	Value	$\bar{e}_{mean}$	r_{compr}	r_{PCA}
Defaults		0.0722136%	10.0487%	49.3756%
Imai-Iri		0.117485%	14.2559%	0.0%
norm	No	0.103175%	13.4081%	20.6027%
	Yes	0.0722136%	10.0487%	49.3756%
size	1%	0.0492049%	6.29995%	72.1952%
	5%	0.0648045%	8.9417%	55.7808%
	10%	0.0722136%	10.0487%	49.3756%
	20%	0.0731979%	11.2144%	47.3614%
smp	48	0.112611%	14.0418%	10.0927%
	96	0.076667%	10.1056%	43.8132%
	144	0.10083%	13.2704%	22.8581%
	192	0.0884565%	11.0551%	31.0062%
	288	0.0722136%	10.0487%	49.3756%
	576	0.0771094%	10.2677%	42.3242%
	960	0.0770273%	10.4436%	43.0677%
	1440	0.0764876%	10.1549%	42.1606%
td	0.1	0.078807%	10.5249%	41.7945%
	0.2	0.0722136%	10.0487%	49.3756%
	0.3	0.0794997%	10.4486%	41.2767%
	0.5	0.0816904%	10.5346%	38.7605%
ts	1.0	0.0810634%	10.7033%	41.7174%
	1.2	0.0794165%	10.3755%	42.7345%
	1.4	0.0803527%	10.5351%	39.7601%
	1.6	0.0722136%	10.0487%	49.3756%
	1.8	0.0755407%	10.349%	43.6036%

Table 5: Results of the automatic clustering algorithm.

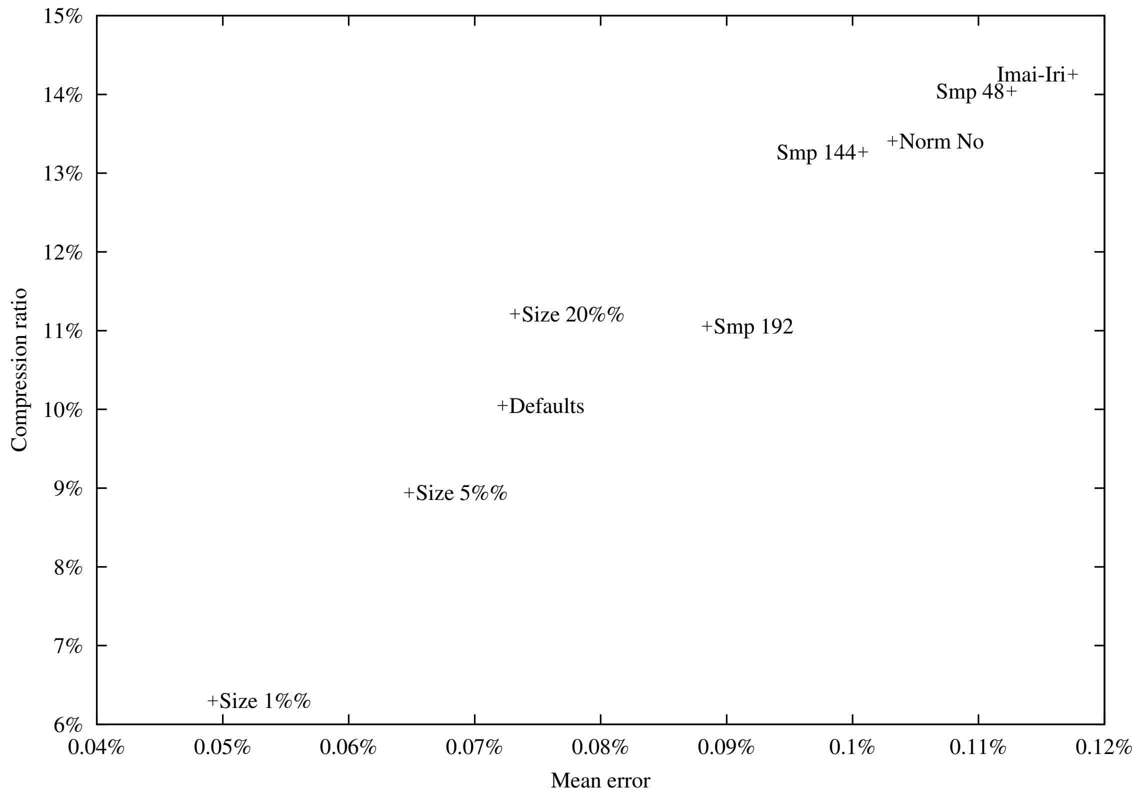

Figure 16: Overview of the results of the automatic clustering algorithm.

addition to the superior compression performance, the automatic clustering algorithm also achieves a lower mean error.

The influence of the parameters on compression ratio and mean error is rather mixed. The tolerance-related parameters appear to have little influence on the compression result and show no clear correlation between the parameter values and the the results.

The minimum cluster size parameter shows a clearer behavior with smaller cluster sizes achieving better compression and a smaller mean error. A smaller minimum cluster is however likely result in more clusters and possibly increased function-independent overhead that is not reflected in the compression ratio listed here. The overhead is discussed separately in Section 6.2.3.

The numbers resulting from the different sampling rates appear to confirm our assumptions from Section 3.2 at least in part. As expected, a sampling distance of 10 minutes (144 samples per day) yields the worst results, both in terms of compression ratio and mean error. Surprisingly, the remaining sampling rates except 192 (7.5 minutes distance) yield very similar results. It is unclear whether the lower sampling error achieved by a high sampling rate fails to improve the performance of the algorithm or whether this effect is counteracted by another effect of the sampling rate on the process.

Finally, the normalization proves to be useful, as demonstrated by the significantly worse results achieved without it. Note that the overhead induced by the

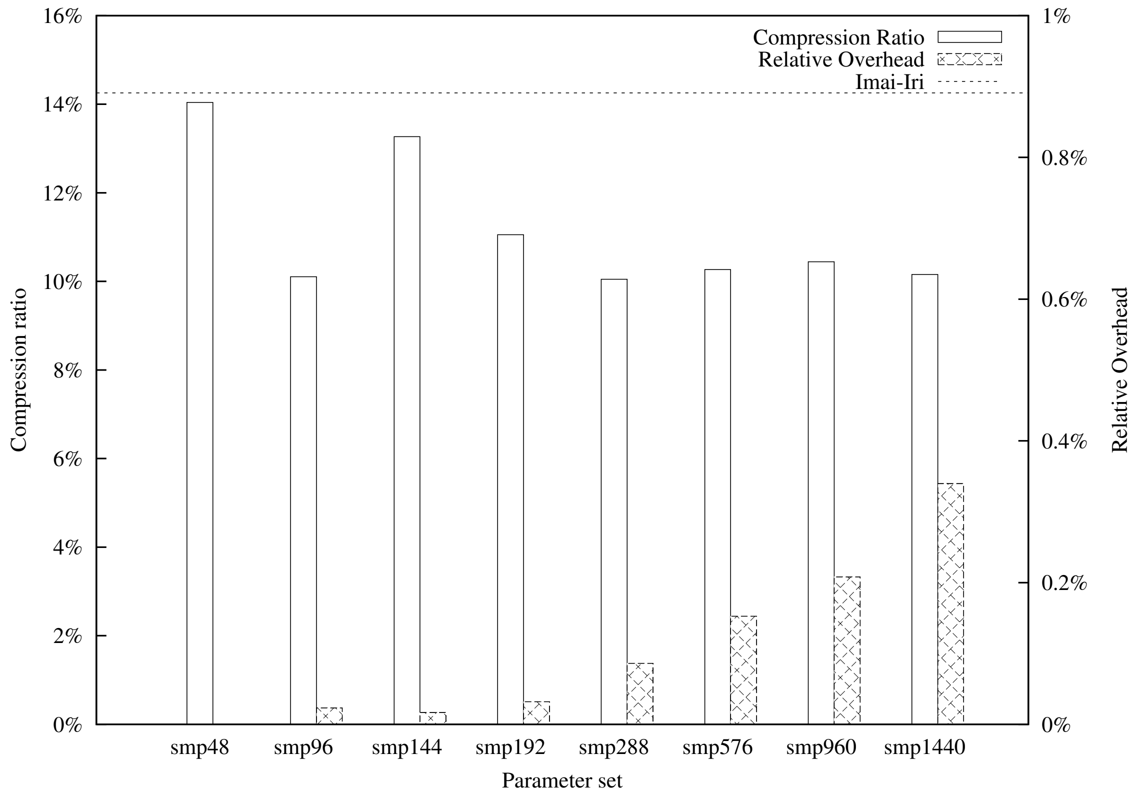

Figure 17: PCA–overhead at different sampling rates.

normalization is taken in account when calculating the compression ratio as it is plotted here.

6.2.3 PCA Overhead

Unlike the Imai-Iri algorithm, PCA-based compression introduces additional overhead that cannot be directly assigned to a single function. While this overhead is not taken into account when calculating the compression ratio because we are primarily concerned with the amount of data that has to be retrieved to evaluate a single function, it shall not be omitted entirely from this work.

For the most part, the overhead consists of the mean and pattern vectors produced during the PCA-compression. While there is one mean vector for each cluster produced, the number of pattern vectors per cluster varies. In addition to the number of vectors, the extent of the overhead obviously depends on the number of sampling points used, which is the number of elements of these vectors.

Figure 17 shows the compression ratio achieved by different sampling rates and the induced overhead per PCA-compressed function; the dashed horizontal line indicates the compression achieved by the Imai-Iri algorithm. As mentioned before, the two major characteristics of the output, compression ratio and mean error, are only in part influenced by the sampling rate. In contrast, the overhead per compressed function increases steadily with the sampling rate.

The number of base vectors produced by the compression appears to be influenced by multiple factors. The minimum cluster size parameter has the most obvious influence on the number of vectors as a smaller cluster size leads to more clusters. This is reflected by the results of the experiments, which show that the smallest used cluster size of 1% produces one base vector for approximately 1000 compressed functions, while a minimum cluster size of 20% results in one base vector per approximately 6500 functions. The lowest number of vectors was produced by the experiment using a sampling rate of 48 samples per day, which resulted in one vector per roughly 24,000 compressed functions.

6.2.4 Error Bounds

Figure 18 illustrates the behavior of the algorithm for different values for the maximum relative error $\hat{e}_{max}$ compared to that of the Imai-Iri algorithm. In contrast to most other results presented in this section, these experiments were conducted with a minimum cluster size of 5%. At the default size of 10%, the results of the algorithm for low error bounds are of little use because it is unable to find a sufficient number of clusters. This is still the case for the 0.1% error bound, which is why the corresponding data points in the plot coincide. At higher error bounds, the PCA-based algorithm produces results superior to those of the Imai-Iri algorithm, with lower compression ratio and mean error.

6.2.5 Smoothed Input Data

As discussed in 3.4, smoothing of the input data before compression is based on the assumption that the data may contain measurement noise that adversely effects the compression process. While it seems likely that the algorithm will produce better compression on smoothed data, it is necessary to compare this to the behavior of the Imai-Iri algorithm. Experiments were conducted with the »default« parameter setup used above and $\hat{e}_{max}$ limited to 1%. The resulting compression ratio is shown in Figure 19.

The experiments show that both algorithms profit from the smoothing as expected, although the PCA-based algorithm slightly more so. For all σ values tested, the output produced by the PCA-based algorithm is approximately 30% smaller than that produced by the Imai-Iri algorithm. Figure 19 shows the relationship between the smoothing standard deviation and the resulting compression ratio r_{compr} for both algorithms.

Smoothing of the input data does also cause a reduction of overhead. The ratio of base vectors to compressed functions ranges from approximately 1:4,400 without smoothing to 1:10,000 with $\sigma = 15$.

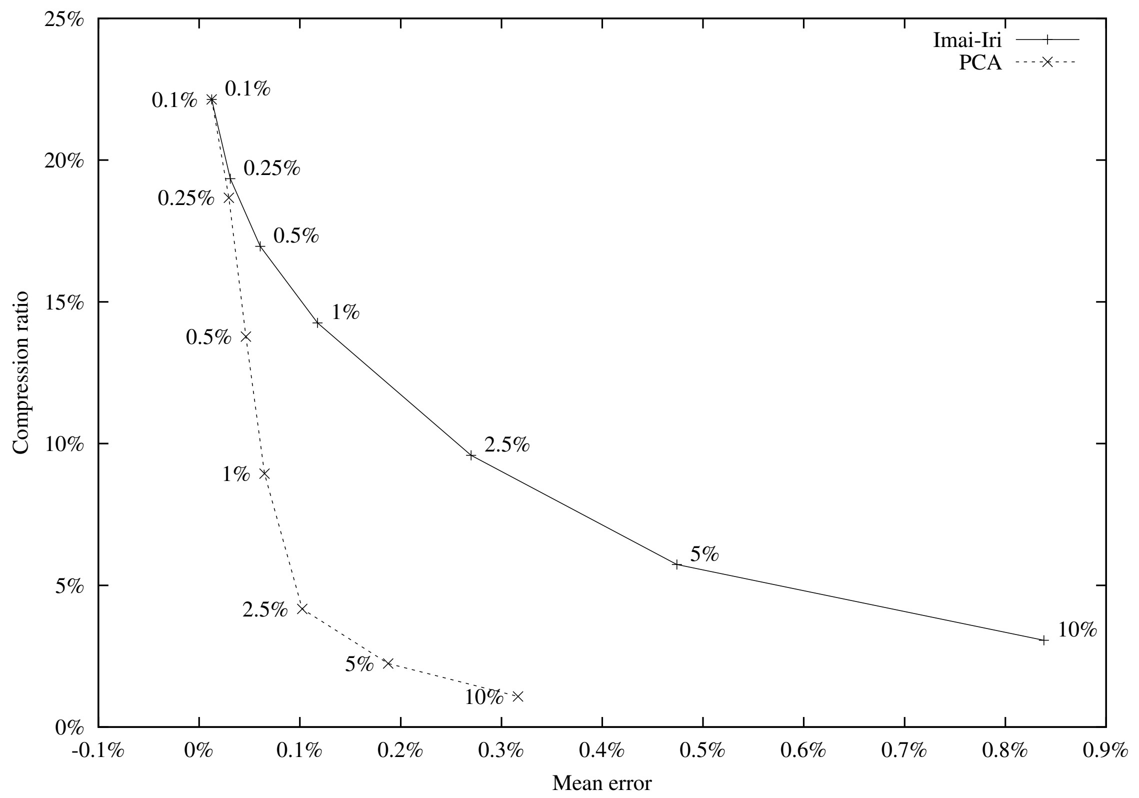

Figure 18: Compression ratio and mean approximation error for different maximum approximation errors.

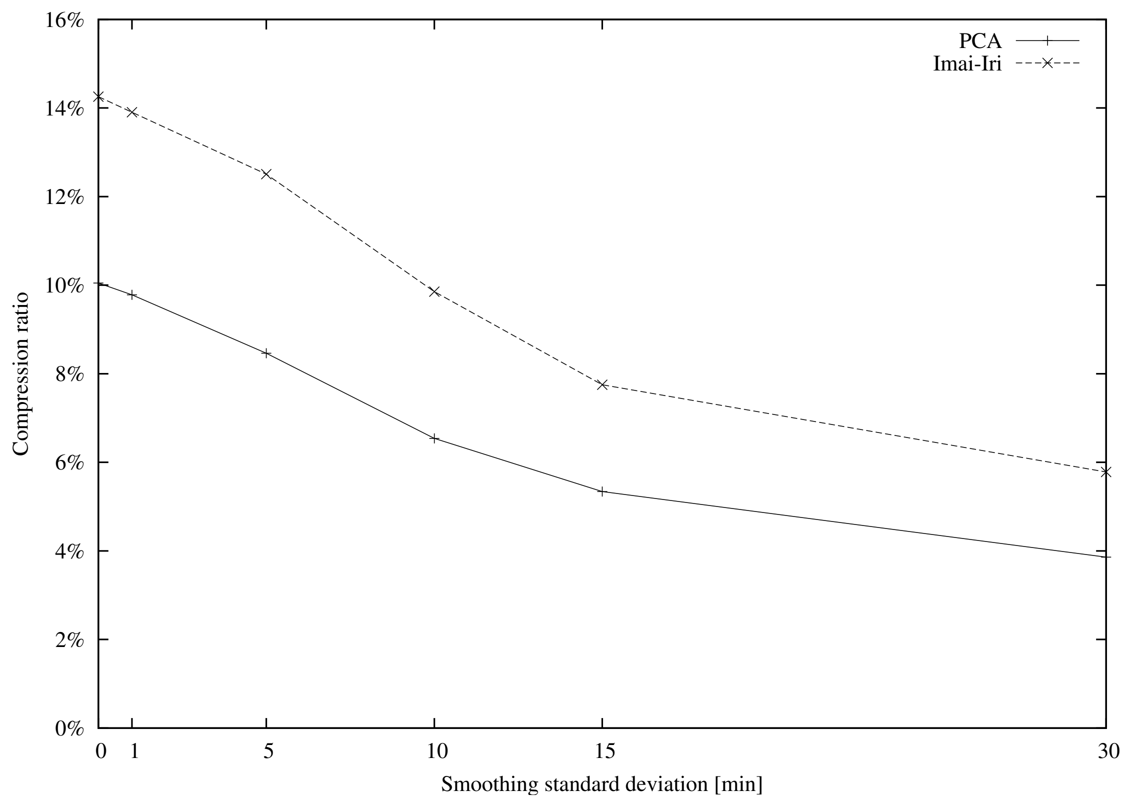

Figure 19: Compression ratio on smoothed input data.

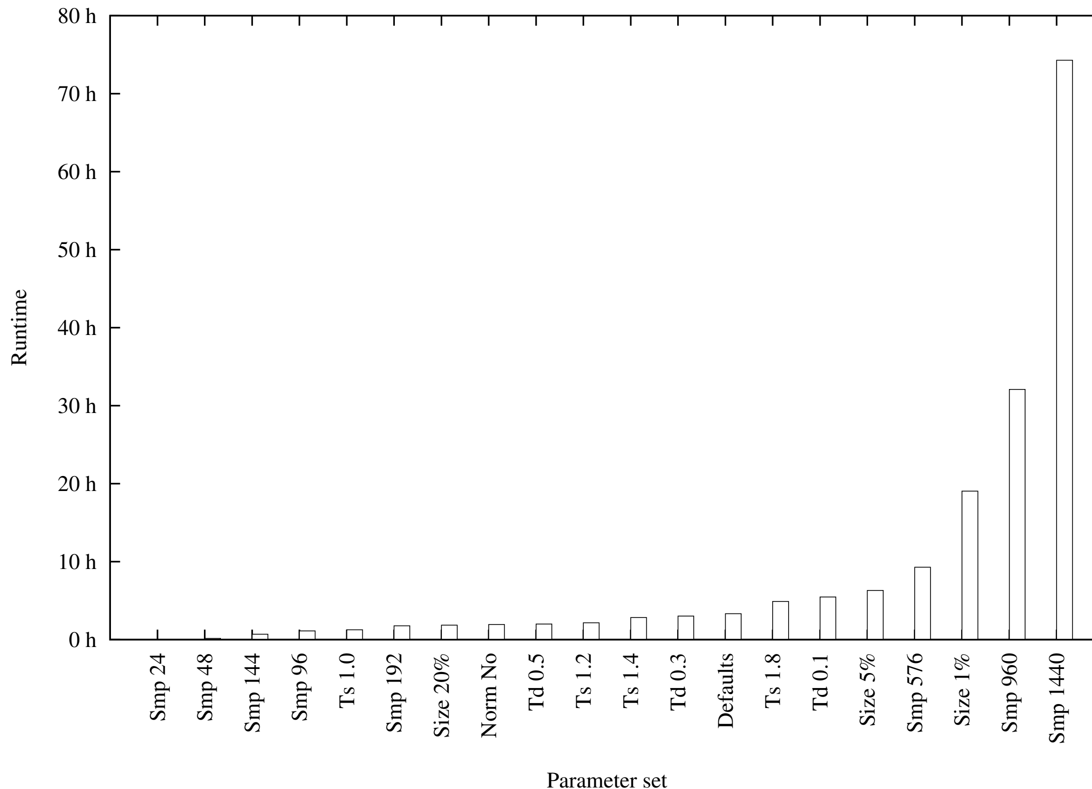

Figure 20: Runtimes for the different parameter configurations.

6.2.6 Runtime

The runtime of the automatic clustering algorithm was not measured in a strict fashion, but enough data was collected to give a qualitative impression of the runtime and how it is influenced by the parameters of the algorithm. Figure 20 shows the runtime measured for the different parameter settings. Each experiment was conducted with eight Intel Xeon processors with 2.67 GHz each in parallel on a machine with 64 GB of memory.

Evidently, the minimum cluster size and sampling rate parameters have the strongest influence on the runtime of the the algorithm. The sampling rate influences the runtime for a single compression step because it directly determines the size of the data vectors, and thereby the size of the covariance matrix that has to be calculated for each compression step. The calculation of the covariance matrix and the calculation of the number of coefficients needed for each function, which is influenced by the sampling rate in the same way, have proved to be the most time-consuming parts of the algorithm.

The minimum cluster size parameter does not influence the calculation time of a single compression step, but the total number of compression steps that are performed. This is the case because the while loops in Listing 2 and Listing 3 are executed more often with a smaller minimum cluster size. The former because more clusters are produced and the latter because smaller clusters tend to reach a lower

Data set	Non-constant functions	Avg. points per function
Tue – Thu	2,380,285	10.08
Fri	2,153,252	9.28
Sat	1,411,731	6.51
Sun	966,599	5.02
Mon	2,315,677	9.91

Table 6: Basic information about the input data sets.

tolerance value.

6.2.7 Additional Data Sets

In addition to the data set used for the detailed experiments described so far which contains travel time data for the days from Tuesday to Thursday, the algorithm was also tested on four further data sets. These data sets are based on the same road network graph, but contain travel time data for Friday, Saturday, Sunday, and Monday, respectively. The additional sets contain less non-constant travel time functions than the Tuesday to Thursday set, and those functions have, on average, less data points. Table 6 shows some more information about the data sets.

The additional data sets were compressed with the automatic clustering algorithm to a maximum relative error $\hat{e}_{max}$ of 1%, using a sampling distance Δ_S of five minutes and normalization (cf. Section 6.2.1). For all sets, two compressions were calculated, with a minimum cluster size of 1% and 10%, respectively.

Table 7 shows the results of the compression. For both PCA-compression variants, the compression ratio r_{compr}, the fraction of PCA-compressed functions r_{PCA}, the mean approximation error $\bar{e}_{mean}$, and the relative overhead $r_{overhead}$ are listed. For comparison, the compression ratio and approximation error achieved by the Imai-Iri algorithm are listed as well.

The results for the additional data sets are generally similar to those for the Tuesday to Thursday data set. There is, although, a divergence between the performance of the automatic clustering algorithm and the Imai-Iri algorithm, especially on the Sunday set. For this set, the Imai-Iri algorithm produces the worst compression ratio (17.22%), while the compression ratio of the PCA-algorithm is among the best (9.59%/5.42%). This is likely caused by two factors: As shown in Table 6, the Sunday set has a relatively small average function size, which leaves little room for improvement for the Imai-Iri algorithm. For the PCA-algorithm, the function size is irrelevant because the sampled data the algorithm operates on has always the same size. On the other hand, The PCA-algorithm does probably benefit from the more uniform shape of the travel time functions due to the generally low traffic on Sundays.

		Tue – Thu	Fri	Sat	Sun	Mon
Imai-Iri	r_{compr}	14.26%	14.8%	17.08%	17.22%	14.07%
	$\bar{e}_{mean}$	0.117%	0.107%	0.069%	0.0473%	0.114%
Min. cluster size 10%	r_{compr}	10.05%	10.16%	10.82%	9.59%	9.54%
	r_{PCA}	49.38%	47.64%	64.61%	74.67%	53.86%
	$\bar{e}_{mean}$	0.0722%	0.0662%	0.0348%	0.0198%	0.0658%
	$r_{overhead}$	0.086%	0.0702%	0.126%	0.128%	0.1%
Min. cluster size 1%	r_{compr}	6.3%	6.67%	6.53%	5.42%	5.81%
	r_{PCA}	72.2%	69.65%	83.34%	90.34%	76.69%
	$\bar{e}_{mean}$	0.0492%	0.0485%	0.026%	0.0143%	0.0439%
	$r_{overhead}$	0.695%	0.613%	0.729%	0.664%	0.769%

Table 7: Results of the automatic clustering algorithm.

6.3 Application

The experiments discussed in the previous section examine the error introduced by the compression on a per-function basis. However, when the travel time data is used for route planning in an application based on the Time-Dependent Contraction Hierarchies technique, paths in the road network are constructed from multiple road segments with separate travel time functions.

When the travel time of a path is calculated from the travel time functions of the links of the path, errors in the travel time functions caused by approximation can influence the resulting total travel time beyond the relative error of each function. Consider for example a path consisting of two segments A and B with travel time functions f_A and f_B. The travel time function for the path $f_{A,B}$ is then given by Equation 24. Now consider approximations of f_A and f_B that introduce a relative error ϵ as defined in Equation 25. If we combine these approximations to calculate the total travel time for the path, we obtain Equation 26. This means that f_B^* is evaluated with the potentially incorrect result of f_A^* which might lead to an error that is not bounded by ϵ.

$$f_{A,B}(t) = f_A(t) + f_B(t + f_A(t)) \quad (24)$$

$$f_{A/B}^*(t) = f_{A/B}(t) \cdot (1+\epsilon) \quad (25)$$

$$f_{A,B}^*(t) = f_A^*(t) + f_B^*(t + f_A^*(t)) = (1+\epsilon)\Big(f_A(t) + f_B(t + f_A(t)(1+\epsilon))\Big) \quad (26)$$

To determine how the error introduced by the compression impacts the results of the route planning algorithm, 1,000,000 queries were performed using compressed travel time data. The travel times and routes calculated by the algorithm were compared to the optimal values calculated based on uncompressed travel time data by three measurements. The relative error of the packed routes e_{packed} is the relative difference between the optimal travel time and that calculated based on the compressed data. This travel time is calculated from the travel time functions of the

segments of the route including shortcuts. The second measure, $e_{unpacked}$ it calculated after *unpacking* the rout returned by the algorithm. That is, the shortcuts within the route are recursively replaced with the shortcuts and road segments they stand for, eventually producing a route consisting only of real world road segments. The travel time of the unpacked route is calculated from the approximated travel time functions of the road segments and compared to the optimal value, yielding $e_{unpacked}$. The last measure, $e_{transferred}$, is the relative error between the real travel time of the route returned by the algorithm and the optimal value. This is also referred to as *transferring* the route into the original road network graph.

The experiments were conducted with two compressed data sets from Section 6.2, the »defaults« set and the one using 1% relative cluster size because it achieved the best compression ratio of all sets. For comparison, the same experiments were conducted with a data set compressed only with the Imai-Iri algorithm. All three were compressed to a maximum relative error $\hat{e}_{max}$ of 1%.

The following figures show the density of e_{packed}, $e_{unpacked}$, and $e_{transferred}$ for the three data sets within the interval from 0 to 1%, respectively. For all of them, more than 99,99% of the queries produced an error within that interval. To calculate the density, the interval was divided into 100 sub-intervals and the number of queries was counted for each sub-interval.

Figure 21 shows the density of the relative error of in the packed routes, e_{packed}. Contrary to our apprehension, the error for the travel times of the paths is not higher than the maximum approximation error of the path segments. Similar to the results in Section 6.2, the Imai-Iri compressed data set produces the highest average error, followed by the »defaults« set and »size0.01« sets.

Figure 22 shows the density of the relative error after unpacking the routes, $e_{unpacked}$. As the figure shows, the error for unpacked routes is actually lower on average than for the still packed routes shown in Figure 21.

Finally, Figure 23 shows the the distribution of the relative difference between the real travel time of the route calculated from the compressed data and the travel time of the optimal route, $e_{transferred}$. For this measure, the difference between the three compression variants is only marginal. It is noteworthy, that the real travel time is virtually optimal for the vast majority of routes calculated based on approximated data. This indicates that the algorithm finds the optimal route in most cases, in spite of using approximated data.

6.3.1 Higher Error Bounds

The previous experiment raises the question how much approximation error we can allow before the real travel time of the calculated routes differs severely from the optimal time. Figure 24 shows the maximum and average relative travel time error $e_{transferred}$ for different approximation errors. Evidently, even relatively high approximation errors produce only a relatively low average error in the real travel time of the calculated route.

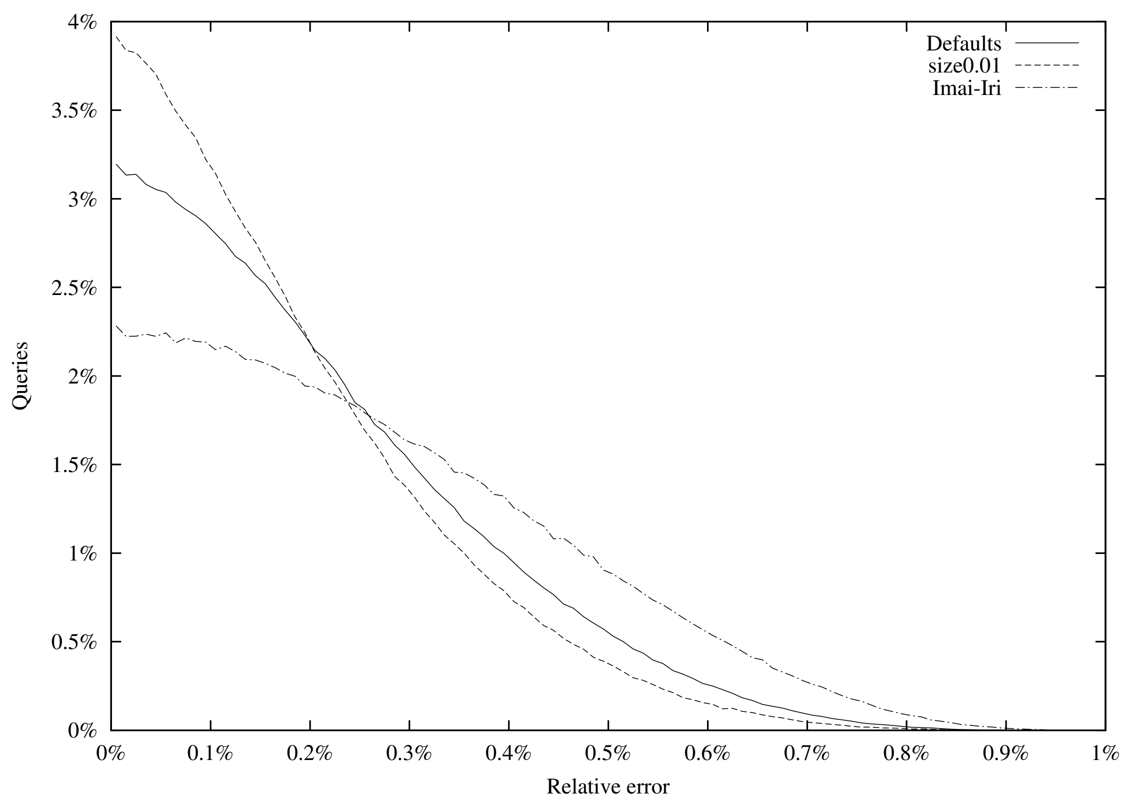

Figure 21: Density of the relative travel time error of packed paths e_{packed}.

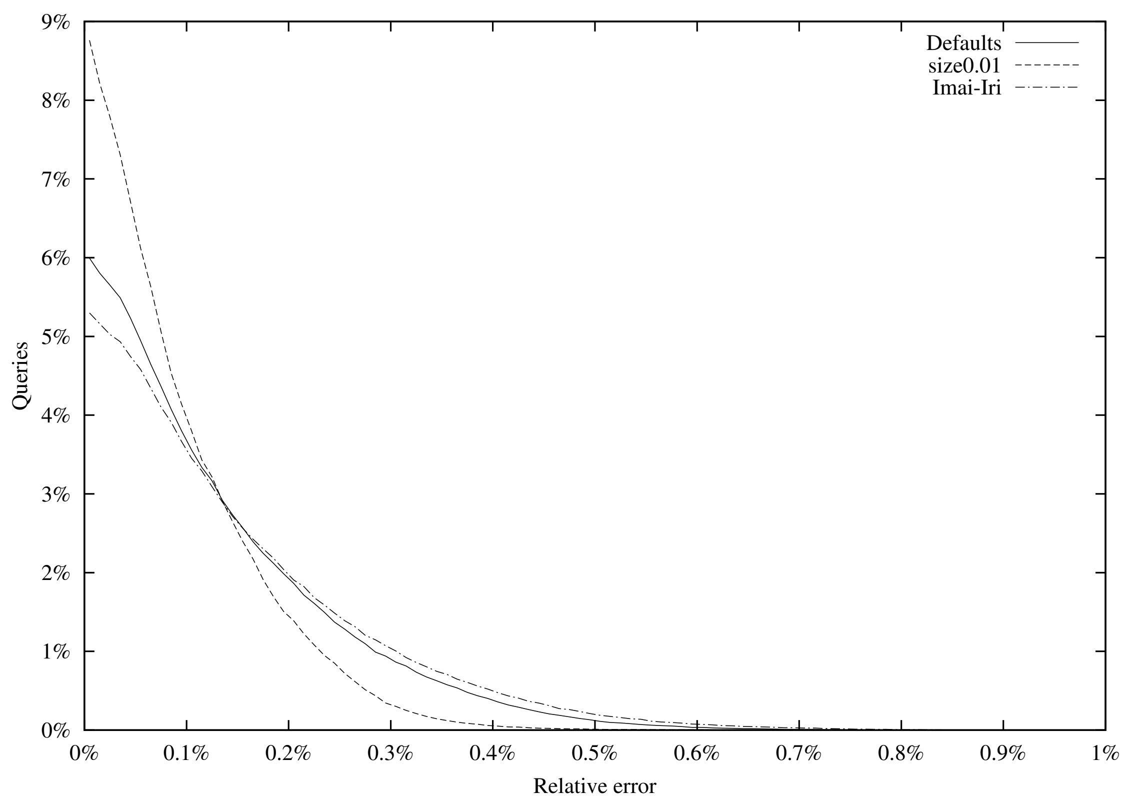

Figure 22: Density of the relative travel time error of unpacked paths $e_{unpacked}$.

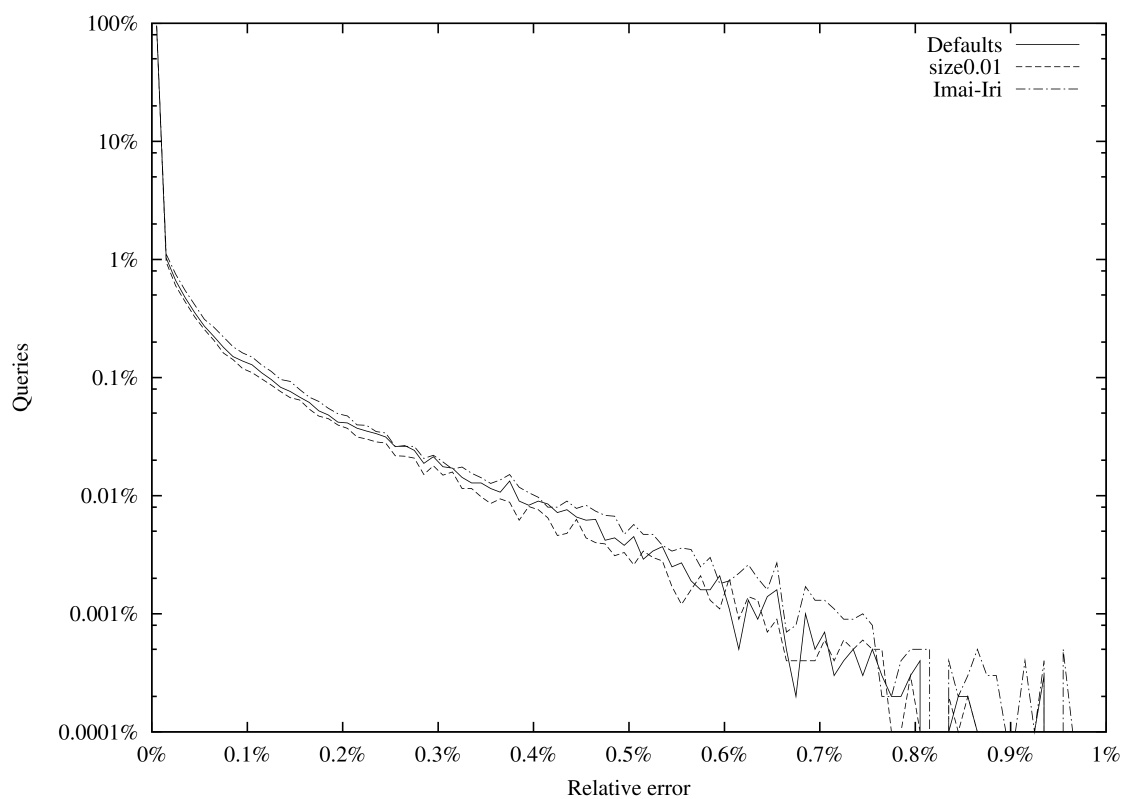

Figure 23: Density of the relative error of real travel times $e_{transferred}$.

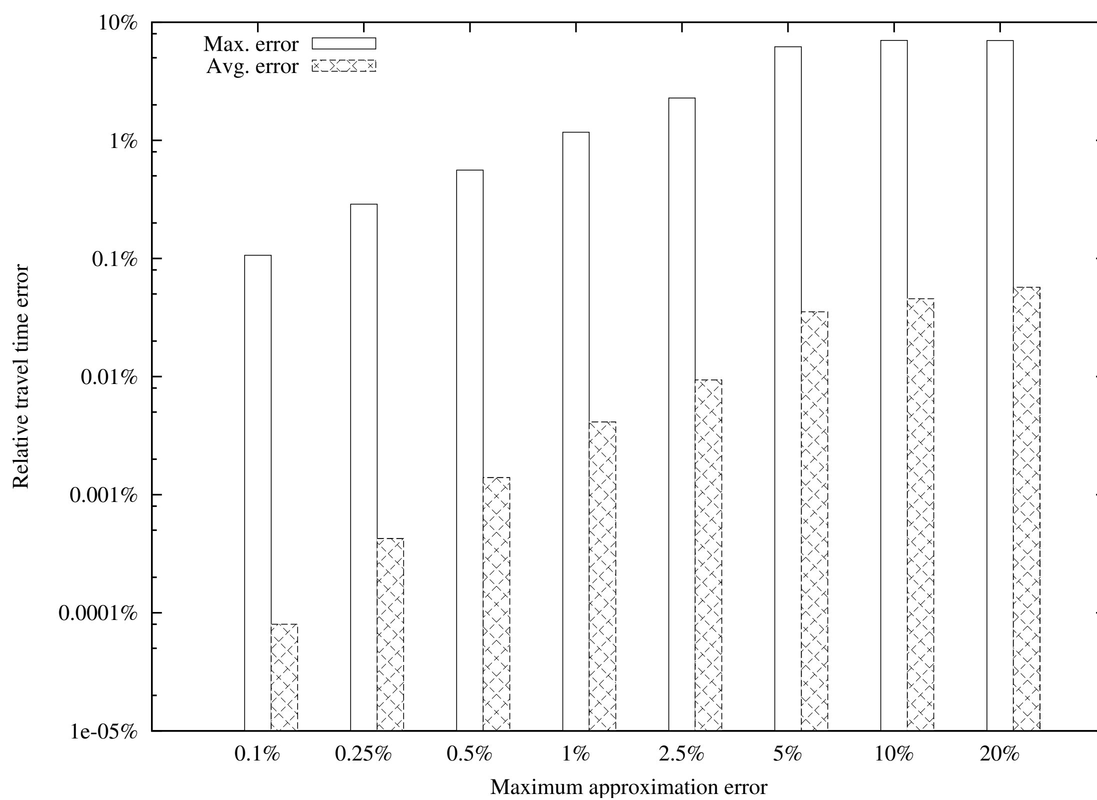

Figure 24: Average and maximum travel time error $e_{transferred}$ for different approximation errors.

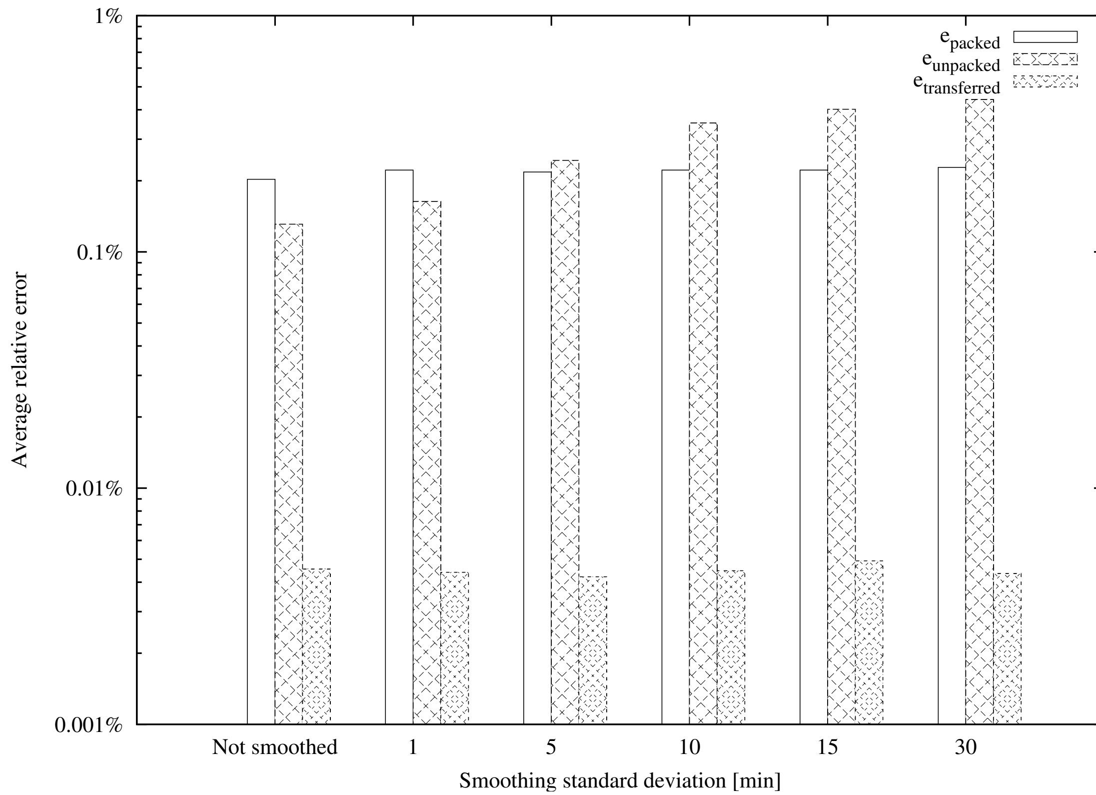

Figure 25: Average relative travel time errors on smoothed input data.

6.3.2 Smoothed Input Data

As the route planning algorithm appears to cope relatively well with approximated data, it is interesting to examine the effect of smoothed input data on the route planning. As discussed in Section 6.2.5, smoothing of the input data can considerably improve the compression achieved by the automatic clustering algorithm. Figure 25 shows the average relative travel time error for smoothed input data that was compressed with a maximum relative error $\hat{e}_{max}$ of 1%. Similar to the previous experiments, the first column shows the error for the packed routes e_{packed}, the second column shows the relative travel time error after unpacking the shortcuts $e_{unpacked}$, and the third column shows the relative difference between the real travel time of the route found by the algorithm and the travel time of the optimal route, $e_{transferred}$.

It appears that only $e_{unpacked}$ is affected by the smoothed input data as it increases considerably. The other two measures change only marginally.

7 Conclusions

Of the three approaches presented in this work, two manage to significantly improve the compression of the travel time data compared to the algorithm by Imai and Iri. The—albeit quite complex—automatic clustering algorithm even surpasses it by as much as a factor two. In spite of the stronger compression, both approaches produce a lower average approximation error than the Imai-Iri algorithm.

Furthermore, experiments show that the introduced approximation error does not substantially impair route planning based on the compressed data.

7.1 Future work

The compression of the travel time data could be further improved by an optimized low-level coding of the data similar to the work of C. Vetter et al. on the *Contraction Hierarchies* graph structure [SSV08].

It would also be interesting to further investigate the transformation approach described in Section 5. To improve this approach, it would be necessary to devise a transformation that avoids the sampling-related problems introduced by the approach presented in this work.

The original motivation for this work was the use of time dependent route planning on mobile devices. To complete this, the next step would be to implement the decompression procedure for the devised compression methods and to integrate it into existing route planning implementations for mobile devices.

References

[BDSV09] Batz, G. V., Delling, D., Sanders, P., Vetter, V.: *Time-Dependent Contraction Hierarchies.* In: Proceedings of ALENEX 2009, SIAM, 2009.

[FBM95] Falk, M., Becker, R., Marohn, F.: *Angewandte Statistik.* Springer, Heidelberg a. o., 1995, p. 297–315.

[Gal] Galassi, M. et al.: *GNU Scientific Library Reference Manual* (3rd Ed.), Network Theory Ltd., 2009.

[GSSD08] Geisberger, R., Sanders, P., Schultes, D., Delling, D.: *Contraction hierarchies: Faster and simpler hierarchical routing in road networks.* In: McGeoch, C.C. (ed.) WEA 2008. LNCS, vol. 5038, p. 303–318. Springer, Heidelberg, 2008.

[Hot33] Hotelling, H.: *Analysis of a complex of statistical variables into principal components.* In: J. Educ. Psychol., 24, p. 417-441, 498-520, 1933.

[II87] Imai, H., Iri. M.: *An optimal algorithm for approximating a piecewise linear function.* In: Journal of information processing, 9(3):159–162, 1987.

[Jol86] Jolliffe, I. T.: *Principal Component Analysis*, Springer Series in Statistics, Springer–Verlag New York, 1986

[Neu09] Neubauer, S.: *Space Efficient Approximation of Piecewise Linear Functions*, Institute for Theoretical Computer Science, Algorithmics II, 2009.

[NM65] Nelder, J. A., Mead, R.: *A simplex method for function minimization.* Computer Journal, 1965, vol 7, p. 308–313.

[Pea01] Pearson, K.: *On lines and planes of closest fit to systems of points in space.* In: Phil. Mag. (6), 2, p. 559–572. 1901.

[SSV08] Sanders, P., Schultes, D., Vetter, C.: *Mobile Route Planning.* In: Halperin, D., Mehlhorn, K. (eds.): ESA 2008, LNCS 5193, p. 732–743, Springer, Heidelberg 2008.