eCO-friendly urban Multi-modal route PlAnning Services for mobile uSers

FP7 - Information and Communication Technologies

Grant Agreement No: 288094
Collaborative Project
Project start: 1 November 2011, Duration: 36 months

D2.3.2 - Validation and empirical assessment of algorithms for eco-friendly vehicle routing

Workpackage:	WP2 - Algorithms for Vehicle Routing
Due date of deliverable:	31 October 2013
Actual submission date:	31 October 2013
Responsible Partner:	TomTom
Contributing Partners:	CERTH, CTI, ETHZ, KIT

Nature: ☒ Report ☐ Prototype ☐ Demonstrator ☐ Other

Dissemination Level:
☒ PU: Public
☐ PP: Restricted to other programme participants (including the Commission Services)
☐ RE: Restricted to a group specified by the consortium (including the Commission Services)
☐ CO: Confidential, only for members of the consortium (including the Commission Services)

Keyword List: algorithms, shortest path, route planning, traffic prediction, time-dependent shortest path, alternative routes, robust routes, fleets of vehicles, private vehicles, heuristics

The eCOMPASS project (www.ecompass-project.eu) is funded by the European Commission, DG CONNECT (Communications Networks, Content and Technology Directorate General), Unit H5 - Smart Cities & Sustainability, under the FP7 Programme.

The eCOMPASS Consortium

Computer Technology Institute & Press 'Diophantus' (CTI) (coordinator), Greece

Centre for Research and Technology Hellas (CERTH), Greece

Eidgenössische Technische Hochschule Zürich (ETHZ), Switzerland

Karlsruher Institut fuer Technologie (KIT), Germany

TOMTOM INTERNATIONAL BV (TOMTOM), Netherlands

PTV PLANUNG TRANSPORT VERKEHR AG. (PTV), Germany

Document history			
Version	Date	Status	Modifications made by
	23.05.2013	*TOC contributions received*	
0.1	24.05.2013	TOC draft (sent to contributors)	Michael Marte, TomTom
	08.07.2013	*Draft of most sections received from contributors*	
0.9	15.07.2013	Draft of preliminary full document (sent to project coordinator)	Felix König, TomTom
1.0	23.10.2013	Sent to internal reviewers	Michael Marte, TomTom
1.1	30.10.2013	Reviewers' comments incorporated (sent to PQB)	Michael Marte, TomTom
1.2	30.10.2013	PQB's comments incorporated	Michael Marte, TomTom
1.3	31.10.2013	Final version (approved by PQB, sent to the Project Officer)	Christos Zaroliagis, CTI
1.4	21.11.2013	Updated final version (approved by PQB, sent to the Project Officer)	Michael Marte, TomTom Christos Zaroliagis, CTI

Deliverable manager

- Michael Marte, TomTom

List of Contributors

- Themistoklis Diamantopoulos, CERTH
- Julian Dibbelt, KIT
- Kalliopi Giannakopoulou, CTI
- Dimitrios Gkortsilas, CTI
- Dionisis Kehagias, CERTH
- Spyros Kontogiannis, CTI
- Florian Krietsch, PTV
- Polykarpos Meladianos, CERTH
- Matús Mihalák, ETHZ
- Sandro Montanari, ETHZ
- Eirini Papagiannopoulou, CERTH
- Andreas Paraskevopoulos, CTI
- Daniel Proskos, CTI
- Christos Zaroliagis, CTI

List of Evaluators

- Dionisis Kehagias, CERTH
- Sandro Montanari, ETHZ

- Moritz Baum, KIT

Summary
The present document reports the results of eCOMPASS Task 2.4 - Validation and empirical assessment of algorithms for vehicle routing. It comprises a description of the algorithms developed, as well as computational results to validate and assess their effectiveness on real-world test data.

The report accompanies a set of software prototypes in which the algorithms have been implemented, and which have been used to generate the results cited above.

The algorithms address challenges in the areas of

- Traffic Prediction
- Time-Dependent Shortest Paths
- Alternative Route Planning
- Robust Route Planning
- Route Planning for Vehicle Fleets.

The current updated version (v1.4) fixes some cross-reference problems and provides updated contents for Sections 1 and 4.2.

Contents

1 Introduction

This deliverable presents validation and assessment for the algorithmic techniques and methodologies developed in eCOMPASS WP2. WP2 aims at providing novel algorithmic methods for optimizing vehicle routes in urban spaces with respect to their environmental impact. In particular, the goal was to derive novel algorithmic solutions to be applied on: (a) intelligent on-board navigator systems for private vehicles that take into account real traffic data (as well as statistical data about traffic at different hours during the day) and seamlessly provide "green" route recommendations; (b) eco-aware logistics and fleet management systems used in conjunction with on-board systems mounted on vehicles and used by drivers, aiming at minimal environmental footprint and fuel consumption.

WP2 breaks down into five tasks:

- **Task 2.1** - New prospects in eco-friendly vehicle routing: Alternative approaches, their limitations and promising new directions.
- **Task 2.2** - Eco-friendly private vehicle routing algorithms: Optimize a private vehicle's route with respect to the environmental footprint of its movement by investigating models that explicitly account for it, and also include traffic prediction. Exact and approximate solutions will be sought, and also dynamic and robust scenarios will be considered. New methodological approaches will be pursued to trade data precision, information content, and solution robustness.
- **Task 2.3** - Eco-friendly routing algorithms for fleets of vehicles: Optimize routes of multiple vehicles in application scenarios that involve delivery/collection of goods by transportation/courier companies aiming at minimizing the environmental footprint associated with the vehicles' movement. Several applications will be considered (people transportation, parcel deliveries, etc). Exact and approximate solutions will be sought, and also dynamic and robust scenarios will be considered. New methodological approaches will be pursued to trade data precision, information content and solution robustness.
- **Task 2.4** - Validation and empirical assessment: Validate the effectiveness of the derived algorithmic solutions in Tasks 2.2 and 2.3, as well as their suitability for online mobile applications upon a variety of realistic scenarios.
- **Task 2.5** - Final assessment of eco-friendly vehicle routing algorithms: Assess the algorithmic solutions developed within WP2 in the actual implementation environments.

The present document reports the results of Task 2.4.

1.1 Structure of the Document

After providing a brief overview of the nature of the data used for testing in Section 2, the document structure reflects the different algorithmic tasks tackled in WP2, briefly describing the algorithmic advancements achieved by eCOMPASS and reporting detailed computational results on different test data sets, comparing new methods to previously existing ones.

Section 3 reports on the accuracy of methods for traffic prediction, section 4 evaluates algorithms for the computation of time-dependent shortest paths, section 5 presents a computational study of algorithms for computing alternative routes, section 6 has computational results for methods that compute robust routes, and, finally, section 7 reports test results for a variety of algorithms that plan routes for fleets of vehicles.

2 Test Data

The test data provided by TomTom includes a road map of the Berlin/Brandenburg area, Germany, including advanced attributes like maneuver restrictions and time-dependent speed profiles, as well as extensive map-matched speed probes collected in this area.

The nodes of the road map are defined in terms of latitude and longitude and its edges are annotated with length, FRC (functional road class), turn restrictions, and speed profiles (see below).

Speed probes are collected from navigation devices. Each speed probe is defined in terms of an edge, the time the vehicle entered this edge, and the average speed of the vehicle on the edge.

Speed profiles are obtained by aggregating speed probes resulting in average speeds on individual edges. Where available, these speed profiles provide speed information for every five minutes of the day for each specific day of the week.

The data set provided by TomTom contains about half a million nodes, about one million edges, about 11000 turn restrictions, about 165 million speed probes collected over half a month, and 100 speed profiles computed from speed probes collected over two years.

For a better understanding of this data, we proceed to illustrate an example. Consider a typical arterial street in downtown Berlin, depicted in Figure 1.

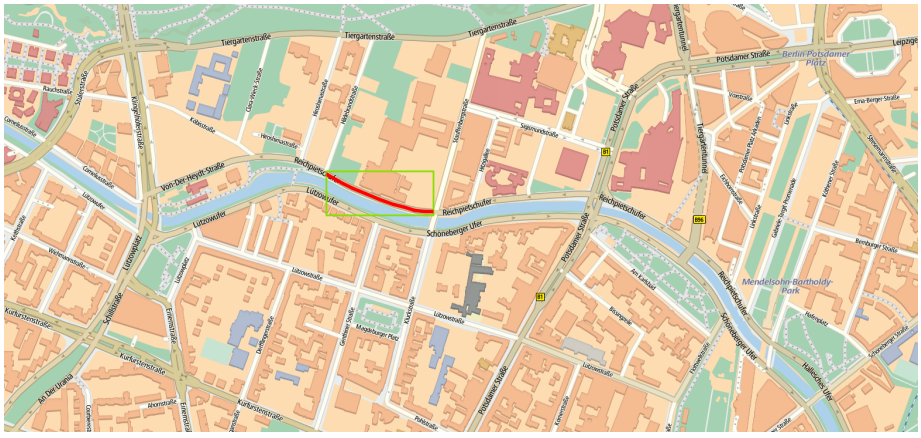


Figure 1: A section of Reichpietschufer in downtown Berlin.

Figure 2 depicts the different speeds measured over an extended period of time on this stretch of road over the course of the week. A histogram of these speed measurements is displayed in Figure 3.

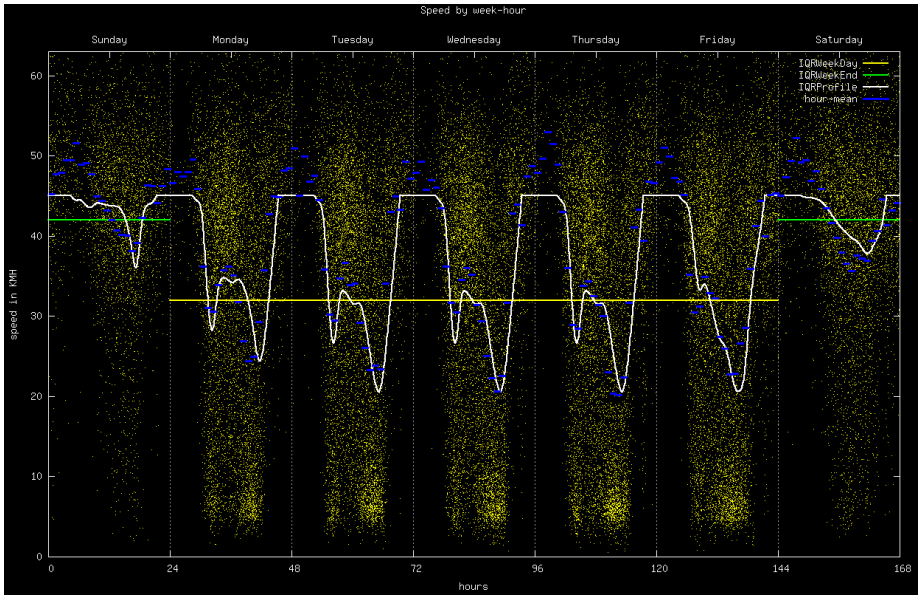


Figure 2: Each yellow dot represents one speed measurement at a certain point in time during the week. Blue bars represent the hourly average, and the white line the resulting assumed speed profile. Green and yellow lines depict average weekday and weekend speeds.

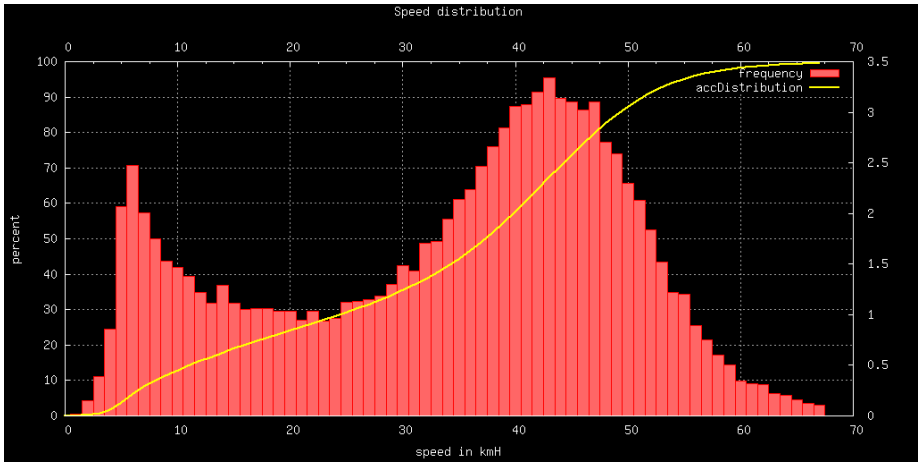


Figure 3: A histogram of the speed measurements displayed in Figure 2, along with the acceleration distribution (yellow).

3 Traffic Prediction

3.1 Introduction

Forecasting travel times to improve multi-modal routing for individuals or fleets is a challenging task. In terms of the eCOMPASS project, the traffic prediction problem was stated in deliverable D2.2 and several state-of-the-art approaches were discussed. Furthermore, as part of the task, four different forecasting techniques were implemented in order to analyze their potential in effectively forecasting travel times. Those techniques were the following:

- A *k-Nearest Neighbors (kNN)* approach, which utilizes data from each road and its neighbors and when a prediction is requested it compares data vectors to identify the nearest ones to the one to be predicted. The significance of every neighbor, and hence its participation in the comparison, is determined using the *Coefficient of Determination (CoD)* between its time series and the series of the road to be predicted.

- A *Random Forest (RF)* approach, adopted from [16], which used an RF regression algorithm to forecast the next speed value of each road given three different kinds of input. Global inputs are constructed by certain statistics for all roads, such as number of samples or mean speed and a *Principal Component Analysis PCA* is performed to isolate the most important features for all statistic metrics (i.e. dimensions). Neighborhood inputs were constructed similarly, however taking into account only neighboring roads, while the last type of input concerned the harmonic speed of the road to be predicted.

- A *Space-Time Auto-Regressive Integrated Moving Average (STARIMA)* approach using a simple model derived from [17]. The algorithm constructs a parametric equation for each group of past speed values corresponding to the road to be predicted and certain neighbors that are determined using CoD, as in kNN. The parameters of the model are determined using a least square estimate, thus each time a prediction is requested the next value is calculated as a linear combination of past values according to the determined parameters.

- A *lag-based STARIMA* approach, derived from the plain STARIMA case, yet the parameters correspond to the lag, instead of the time series order with respect to the series of the road to be predicted. In addition, the roads that are used for creating the model and determining its parameters are not necessarily neighboring. Instead, the CoD is performed globally to reveal any hidden dependencies among roads which are located in long distance to the road in question. This comprises the novel element of the traffic prediction algorithm that we developed in the context of the project that achieved the best performance among all aforementioned benchmarking algorithms.

The aforementioned approaches are presented in detail in the previous deliverable D2.2. In this deliverable, these approaches are evaluated against two different datasets.

Since no dataset was provided at the time that our research on traffic prediction had been initiated, the first dataset that we used was taken from the third task of the *IEEE ICDM Contest: TomTom Traffic Prediction for Intelligent GPS Navigation* [25]. It represents speed measurements from the city of Warsaw, thus it is thereafter named as the *Warsaw dataset*. Despite being simulated, the task of the contest is quite realistic. Concerning our evaluation purpose, the dataset used is drawn from the third task of the contest which is defined as a realistic problem of acquiring sparse data from GPS navigators all over a road network. The problem required forecasting travel time for six and 30 minutes ahead of present time. Since data was given in raw form, as instantaneous vehicle speeds at given coordinates, it was necessary to match the GPS-observed data to edges on the map. This procedure, known as map-matching, is performed according to a method proposed in [15] by J. Greenfeld. The method includes creating trajectories using consecutive instantaneous

(a) (b)

Figure 4: Road networks of (a) Warsaw and (b) Berlin with speed probes. The roads which are colored in red (■) have average speed below 20 km/h, the speed of the yellow-colored ones (■) is above 20 and below 50 km/h, and the speed of the green-colored roads (■) is above 50 km/h.

speed records for every car and matching the trajectories to edges using geometric distance metrics. An illustration of the speed data matched on the map of Warsaw is shown in Figure 4(a).

Figure 4(b) illustrates the second dataset, which concerns the city of Berlin that was presented in Section 2, and shall be named as the *Berlin dataset*. Although it was drawn by GPS locators, when given to us, it was already map-matched. Of course, the Berlin dataset is far more interesting since it is based on actual measurements. After preprocessing the *Warsaw dataset* as mentioned above, both datasets are given in the form of instantaneous speeds that correspond to edges, including also the direction of the vehicle in the edge.

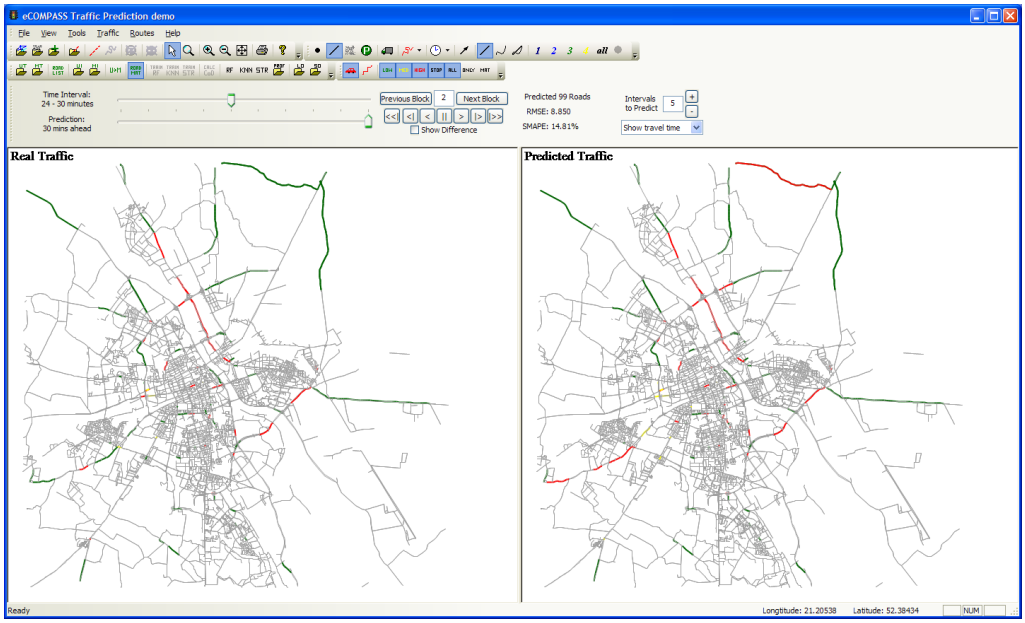
In order to efficiently deal with the fact that for some edges in both datasets no data were provided, we decided to combine edges in order to form road segments. In particular, each road segment (or simply a road) is defined as any segment between two intersections, whereas edges are defined as straight lines. Thus each road segment contains an arbitrary number of edges. Furthermore, instead of individual records, we store only the arithmetic and harmonic mean of recorded speeds, on the specified intervals, different for each dataset, i.e., five-minute and six-minute intervals for Berlin and Warsaw, respectively. This coarse-grained approach ensures not only data tolerance but also algorithm scalability.

The purpose of this section is to provide efficient means for evaluating the traffic prediction techniques, which are briefly outlined here and presented in detail in deliverable D2.2, based on the two datasets at hand. Thus, subsection 3.2 provides the results of the evaluation, illustrating the major achievements and drawbacks of the aforementioned techniques. Finally, subsection 3.3 summarizes the useful conclusions drawn from the application of the various benchmarking techniques, while providing useful insight for further research in the field.

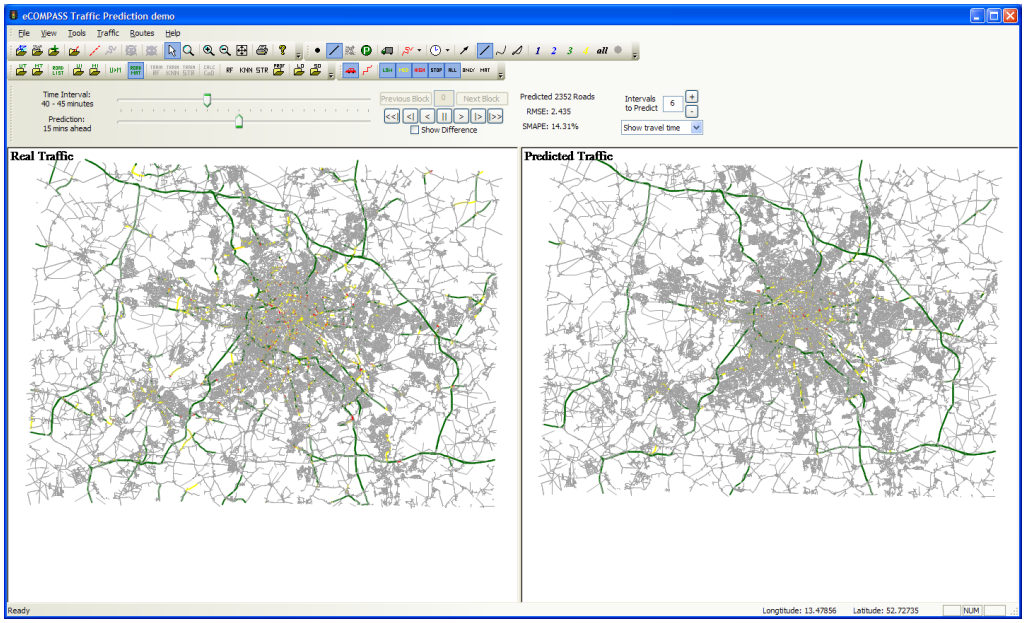
3.2 Experimental Results

This subsection describes the evaluation procedure that was conducted in order to benchmark the performance of the aforementioned algorithms based on the two previously described datasets.

The aforementioned algorithms were implemented as part of a fully functional demo application that we use in order to visualize results. The eCOMPASS traffic prediction demo application, is compatible with numerous formats, including the shape-file format in which the maps of Figure 4



(a)



(b)

Figure 5: Screenshots of the eCOMPASS traffic prediction demo application for (a) Warsaw and (b) Berlin. Screenshot (a) depicts the predicted travel time of the RF algorithm with PCA 30 minutes ahead of present time (i.e. interval from 24 to 30 min.), while screenshot (b) depicts the travel time prediction of lag-based STARIMA 15 minutes ahead of present time (i.e. interval from 40 to 45 min.).

were provided, as well as the speed probe datasets for both Warsaw and Berlin. As shown in Figure 5, the graphical user interface is split in two parts; the left view depicts real traffic data

and the right view depicts the predicted data. This presentation, along with the different colors for different speed areas and the numerous metrics depicted, is certainly helpful in understanding the flow of the traffic and illustrating the effectiveness of the various algorithms.

Although, as shown in Figure 5, both the *Root Mean Square Error (RMSE)* and the *Mean Average Precision Error (MAPE)* metrics were used, RMSE was selected as the major metric, since it is robust with respect to distortion introduced by near to zero values. The RMSE for a specific interval is calculated as:

$$RMSE(i) = \sqrt{\frac{1}{n}\sum_{r=1}^{n}(\hat{V}_{ri} - V_{ri})^2} \tag{1}$$

where V_{ri} and $\hat{V}_{ri}$ are the real and the predicted speed values respectively of road r for interval i. The total RMSE of all roads for all intervals is found by calculating the mean value of the RMSEs of all intervals:

$$TotalRMSE = \frac{1}{N}\sum_{i=1}^{I} RMSE(i) \tag{2}$$

where N is the total number of intervals. MAPE is provided by eq. (3)

$$MAPE(i) = \frac{1}{n}\sum_{r=1}^{n} |\frac{\hat{V}_{ri} - V_{ri}}{V_{ri}}| \tag{3}$$

Tables 1 and 2 present the average RMSE for each time interval ahead for all algorithms for the Berlin and Warsaw datasets respectively.

Table 1: Experimental results of traffic prediction algorithms for the Berlin dataset. The results for each number of intervals ahead are obtained by averaging over all data that corresponds to the respective number of intervals ahead of present time.

	Intervals Ahead					
	1	2	3	4	5	6
kNN	$3.079^{\pm 0.29}$	$3.082^{\pm 0.29}$	$3.096^{\pm 0.29}$	$3.116^{\pm 0.28}$	$3.135^{\pm 0.24}$	$3.169^{\pm 0.22}$
RF with PCA	$2.800^{\pm 0.26}$	$2.789^{\pm 0.24}$	$2.797^{\pm 0.24}$	$2.805^{\pm 0.24}$	$2.805^{\pm 0.25}$	$2.822^{\pm 0.22}$
RF	$2.796^{\pm 0.25}$	$2.797^{\pm 0.24}$	$2.807^{\pm 0.24}$	$2.823^{\pm 0.24}$	$2.805^{\pm 0.24}$	$2.826^{\pm 0.22}$
Lag-based STARIMA	$2.501^{\pm 0.25}$	$2.476^{\pm 0.32}$	$2.460^{\pm 0.28}$	$2.500^{\pm 0.32}$	$2.498^{\pm 0.35}$	$2.565^{\pm 0.28}$
STARIMA	$2.631^{\pm 0.24}$	$2.646^{\pm 0.19}$	$2.644^{\pm 0.21}$	$2.673^{\pm 0.22}$	$2.653^{\pm 0.24}$	$2.688^{\pm 0.21}$

Most algorithms seem to perform in a reasonable manner with smaller error for less intervals and larger for more intervals ahead of present time. Interestingly enough, kNN performs better for five rather than one intervals ahead in the dataset of Warsaw (see Table 2). This could be interpreted as an intuition about the existing noise of road time series of Warsaw and the weak correlation among them. As far as the Berlin dataset is concerned, lag-based STARIMA outperforms all algorithms, regardless of the number of intervals ahead given to predict. The performance of the "plain" STARIMA is also satisfactory, thus implying that the dataset may favor Time Series Analysis methods. RF methods exhibit larger error, while the PCA has no significant effect to the performance of the algorithm. By contrast, it appears to strongly affect the performance in the Warsaw dataset (see Table 2), where the RF with PCA method outperforms all others. However, the effectiveness of lag-based STARIMA is rather satisfactory, since it has less error than "plain"

Table 2: Experimental results of traffic prediction algorithms for the Warsaw dataset. Each value is obtained by averaging over all data that corresponds to the number of intervals ahead.

	Intervals Ahead	
	1	5
kNN	$12.214^{\pm 1.87}$	$11.421^{\pm 2.24}$
RF with PCA	$6.087^{\pm 1.28}$	$6.833^{\pm 1.31}$
RF	$6.679^{\pm 1.35}$	$7.663^{\pm 1.65}$
Lag-based STARIMA	$7.331^{\pm 1.45}$	$11.747^{\pm 2.35}$
STARIMA	$8.859^{\pm 2.19}$	$12.694^{\pm 2.66}$

STARIMA and kNN, even if the dataset seems to favor ML methods. Furthermore, the feature selection of RF methods seems more well-suited to the dataset-specific characteristics (e.g. many zero samples).

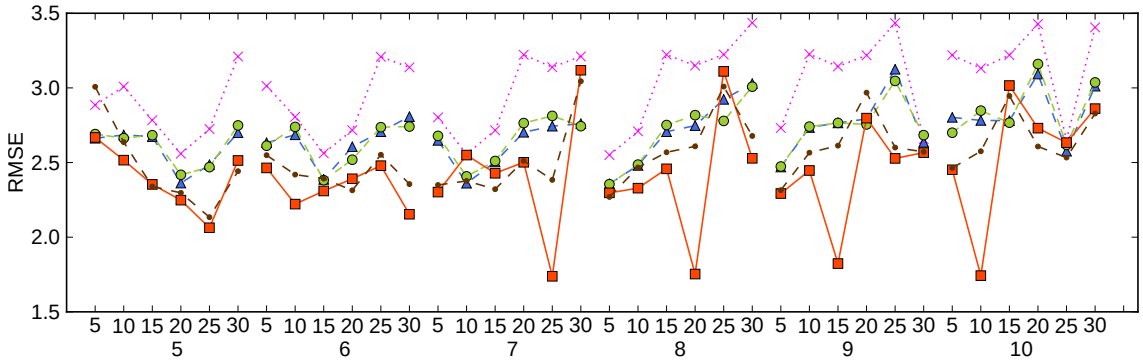
The total results of RMSE for all algorithms are given in Table 3. They were obtained by averaging over all RMSE values (using (2)). Given these results, one can once again observe that the RF implementations are very effective as far as the Warsaw dataset is concerned. However, the STARIMA and the lag-based STARIMA perform better in the case of Berlin. This is rather expected since the average CoD (over all combinations of roads) for Warsaw and Berlin is 26.84% and 83.11% respectively. These metrics provide a hint that the Berlin dataset is more "well-correlated" than the Warsaw one. This is generally expected since the Berlin dataset is real, whereas the Warsaw dataset is generated.

Table 3: Experimental results of traffic prediction algorithms for Warsaw[2]and Berlin, which were obtained by averaging over all RMSE values for all dataset intervals.

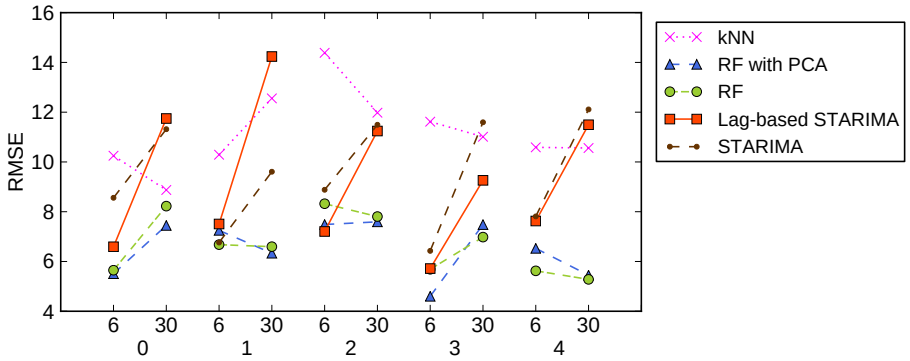
	Datasets	
	Warsaw	Berlin
kNN	$11.818^{\pm 2.10}$	$3.112^{\pm 0.27}$
RF with PCA	$6.460^{\pm 1.35}$	$2.803^{\pm 0.24}$
RF	$7.171^{\pm 1.59}$	$2.809^{\pm 0.24}$
Lag-based STARIMA	$9.539^{\pm 2.95}$	$2.499^{\pm 0.30}$
STARIMA	$10.776^{\pm 3.10}$	$2.655^{\pm 0.22}$

Finally, Figure 6(a) and Figure 6(b) provide an example error illustration for the datasets of Berlin and Warsaw respectively. As noted by the ticks of the x axis for each of the continuous fragments, Berlin predictions range from five to 30 minutes ahead, i.e. from one to six intervals ahead of present time. Concerning the Warsaw dataset, data is split into blocks. For each block two predictions are given, one for six and one for 30 minutes ahead of present time, i.e. for one and five intervals ahead respectively. Finally, Figure 7 illustrates the results given in Table 3. The information of these figures clarifies not only the relative performance of the algorithms, but also

[2]Note that small deviations from the results of the IEEE ICDM contest ([25]) are reasonable for the RF implementation [16] not only because of rounding errors but also because the map-matching procedures differ.



(a) RMSE of Traffic Prediction Algorithms for Berlin



(b) RMSE of Traffic Prediction Algorithms for Warsaw

Figure 6: Graphs showing (a) the RMSE for a sample 1-hour interval of Berlin, and (b) the RMSE for one and five intervals ahead for five blocks of Warsaw. The lower x tick labels denote the current interval at present time (i.e. 6, 7, ...), and the upper x tick labels denote how many minutes ahead of present time are predicted (i.e. 5, 10, 15, ...). For Warsaw, equivalently, the lower x tick labels denote the current block at present time (i.e. 0, 1, ...), and the upper x tick labels denote how many minutes ahead of present time are predicted (i.e. six or 30).

certain interesting features they appear to have.

Concerning Figure 6(a), labeled x-axis items denote intervals, whereas non-labeled ones denote how much intervals ahead are to be predicted (one to six, i.e. five to 30 minutes). At first, one can observe that the prediction error is smaller for one or two intervals ahead than it is for more intervals. This is quite reasonable since e.g. the traffic at interval three may also depend on intervals one and two. Concerning the performance of the algorithms, lag-based STARIMA seems to perform quite well with respect to all other algorithms. The global CoD indeed succeeds in identifying the well-correlated roads. By contrast, the PCA seems to have little impact on such a well-correlated dataset, as the RF algorithm performs quite similarly with and without PCA. Finally, the "plain" STARIMA implementation seems also strong, achieving to capture the trend of traffic, whereas kNN performs worse with respect to all algorithms, possibly because it does not use a weighting method to isolate the most useful series. Specifically, the input vector of kNN takes into account excessive information for all time series, without considering lag (like lag-based STARIMA) or using weights (like STARIMA).

Figure 6(b) depicts the error for a 5-block subset of the target (input) data of the Warsaw dataset. Labeled x-axis items denote the prediction error for the 1st interval ahead (i.e. five min-

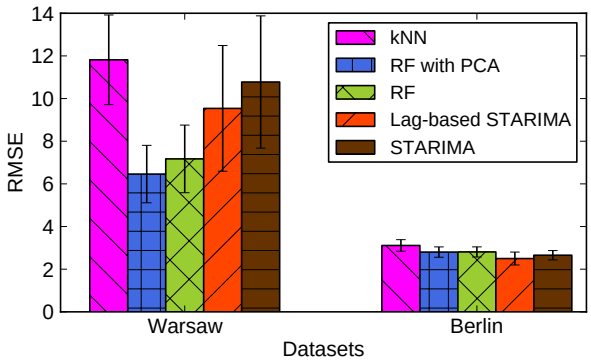


Figure 7: Graph showing average RMSE of algorithms for the Warsaw and Berlin datasets, calculated as the average of all intervals.

utes), whereas non-labeled ones denote the error for the 5th interval ahead (i.e. 30 minutes). At first, it is obvious that all algorithms perform better when forecasting one rather than five intervals ahead. The performance of the RF algorithm is quite satisfactory, and the PCA seems to have some impact on it. Although the effectiveness of the two RF methods does not seem significantly apart, since e.g. a sign test could unveil that they are equally effective, in terms of total RMSE (see Table 3), PCA seems as a well-justified option. Both STARIMA implementations do not fully exploit the data, since the simulated dataset poses a problem seemingly directed towards ML classification algorithms, such as the RF, rather than Time Series Analysis. However, as shown in Figure 6(b), the STARIMA methods perform satisfactorily for one interval ahead, probably because they capture more easily the trend of the time series for short number of intervals ahead. Finally, kNN seems rather unstable, which is expected since the quality of the data (input vectors) given is arbitrary for different roads and time moments. Thus, its error is actually relative to data quality for a given block.

3.3 Conclusion

With respect to the results and analysis given in the previous subsection, the lag-based STARIMA is proven to be a satisfactory approach when it comes to real datasets, such as the one of Berlin. Although the RF approach was more successful for the Warsaw dataset, the lag-based STARIMA is rather adequate concerning the ICDM contest posed a significantly difficult ML problem, with noisy and sparse data.

Specifically, the lag-based STARIMA approach is efficient, because it captures the spatio-temporal nature of the data. The lag element of the algorithm is crucial since it successfully models the temporal characteristics of road networks. Furthermore, identifying well-correlated inputs proved optimal with respect to using data from neighboring roads.

Concerning the effect of using either local or global metrics, the latter proved to be quite useful for both datasets. Generally, the CoD is much more effective when there are strong correlation relationships among the roads of the network, which was the case for Berlin. By contrast, the PCA successfully isolates the "noisy" dimensions of the data, thus it is optimal for the Warsaw dataset.

In conclusion, not only the lag-based STARIMA but also the other algorithms may be improved and tested in different circumstances. For instance, the global CoD could be used along with RF or kNN, while new ideas lie also in hybrid solutions. Further to this, generalizing the findings about STARIMA and RF to parametric and non-parametric algorithms respectively is an interesting, yet complex task to be explored further in future work. In any case, the analysis performed should be a solid first step towards future research on travel time forecasting.

4 Time-Dependent Shortest Paths

4.1 Time-Dependent Approximation Methods

Introduction. Given a directed graph $G = (V, A)$, a set of time-dependent arc-delay functions $D[a](t)$, $a \in A$, and a departure time t from an origin (source) vertex $o \in V$, we introduce a single-pair and a single-source algorithm [19] for approximating the shortest delay (travel time) functions, $D : V \times V \times \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$, from o to one or multiple destination vertices $d \in V$. In the same context, additionally, we provide the resulted approximated shortest path (sub) trees, rooted at o, for any $t \in [0, T]$, where T denotes the time period.

Throughout the section 4.1, for simplicity in notation, we drop the dependence of all the functional descriptions from the departure-time t. Also, due to the assumption for periodicity, we restrict the functions within a single time period.

The proposed algorithms, in [19], are used in our experimental research, in the case of networks with, in general, piecewise linear arc-delay functions $D[a](t)$ and time period $T = 24h$. The first algorithm regards approximating $D[o, d](t) \equiv D[o, d]$, from an origin vertex o to a destination vertex d. The second regards approximating $D[o, *](t) \equiv D[o, *]$, from a common origin vertex o to multiple destination vertices. Both of them are useful, when it is necessary to avoid an expensive computation of a shortest delay function D, without indispensably demanding high precision (although in quite many cases this can be achieved by default). In any case, however, it is required to ensure that the approximation error is bounded by a certain small threshold.

The main idea of the used approximation methods is to keep tracing and sampling the (unknown) shortest delay function $D[o, d]$, at specific time points, thus creating breakpoints, for a lower-approximating function $\underline{D}[o, d]$, and more importantly, for an upper-approximating function $\overline{D}[o, d]$, until the required approximation guarantee is assured. On this purpose, the knowledge of how function is changing during a time interval can be valuable:

- Constant functional form. If the delay function $D[o, d]$ doesn't change (in terms of slope and constant-offset), then the approximation error is zero and thus there is no need for sampling it at intermediate time points of the interval.
- Decreasing slope. When the delay function $D[o, d]$ is concave, its slope is monotonically decreasing until the endpoint of the interval. This info enables the ability of computing straightforward the approximation error between the samples within the interval. In general, if the computed error is enough small, we stop the sampling. Otherwise, we take additional samples inside the interval, until the error becomes small.

Sampling methods. In our implementations, we perform the sampling, computing concurrently the exact worst-case absolute error. Furthermore, we roughly make half the samples compared to the method of Foschini et al [13], while keeping (as breakpoints of D) among them only those that are really necessary in order to assure the approximation guarantee. Additionally, we are only interested in candidate breakpoints, which are beyond the next certificate failure *tfail* (for further details, see Deliverable 2.2 - section 3.4.2), a quantity that can be computed during each Dijkstra-run (TDD) for the next sample point. This is safe, because until *tfail* we already know that $\underline{D}[o, d]$ and $\overline{D}[o, d]$ will be identical to $D[o, d]$.

Our sampling and the approximation error computation take place at intervals where the delay function is known to be concave. These intervals can be identified on the prepossessing phase, that we discuss later. The exact worst-case absolute error is based on the partial derivatives of each consecutive samples and the resulted maximum distance between $\underline{D}[o, d]$ and $\overline{D}[o, d]$ (see Deliverable 2.2 - section 3.4.1 and [19]).

Single pair approximation. The sampling procedure that we use provides the explicit representation of the breakpoints of $\underline{D}[o, d]$ and $\overline{D}[o, d]$: $D[o, d](t) \cdot (1 - \varepsilon) \leq \underline{D}[o, d](t) \leq D[o, d](t) \leq$

$\overline{D}[o,d](t) \leq (1+\varepsilon) \cdot D[o,d](t)$, $o,d \in V$ and $t \in T$, in such a way that both the required approximation ratio is guaranteed and the output (in terms of samples) is indeed asymptotically space-optimal. In the single-pair algorithm, we use two distinct (sampling) phases, each depending on the slope of $D[o,d]$, and also through them a concurrently certificate checking routine. We give a brief description of them :

- First phase (large slope). This implies that the delay is faster changing over the time axis. Therefore, the error between the samples is estimated to be relatively high. So long as the shortest-travel-time slope (i.e., of function $D[o,d]$) is greater than 1 we shall keep sampling the vertical (travel-time) axis, as follows: Departing from o at time $t_s = t_0$, we compute with a forward call of TDD, the earliest-arrival time at d, $t_0 + D[o,d](t_0)$. We then carefully delay the arrival-time, i.e, we consider the arrival time $t_1 + D[o,d](t_1) = t_0 + (1+\varepsilon)^i \cdot D[o,d](t_0) \mid_{i=1}$, and we perform a backward call of TDD, in order to determine the proper (latest) departure-time t_1 from o. If the error between the samples at t_0 and t_1 is small, we increase i and proceed the sampling at the next departure time t_i. Otherwise, if the error becomes prohibitive, we stop, taking only the second last sample at t_{i-1}, which is the last one that passed the error test. In this case, we set as next waypoint, the computed sample at t_{i-1}. This procedure is repeated until the end of time window is reached, or the slope of $D[o,d]$ drops below 1, in which case the current phase of the approximation algorithm is stopped (see Figure 8, 9).

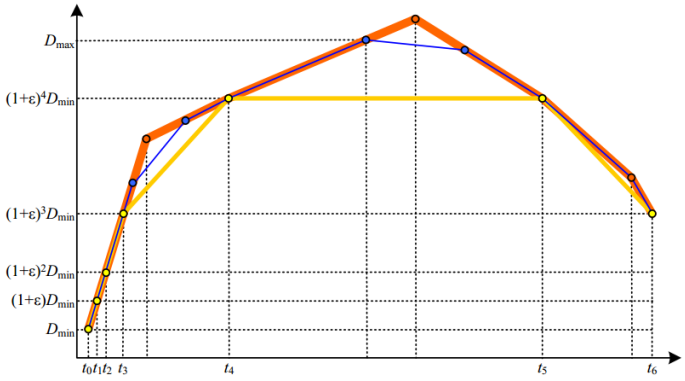


Figure 8: Sampling along the (vertical) delay axis.

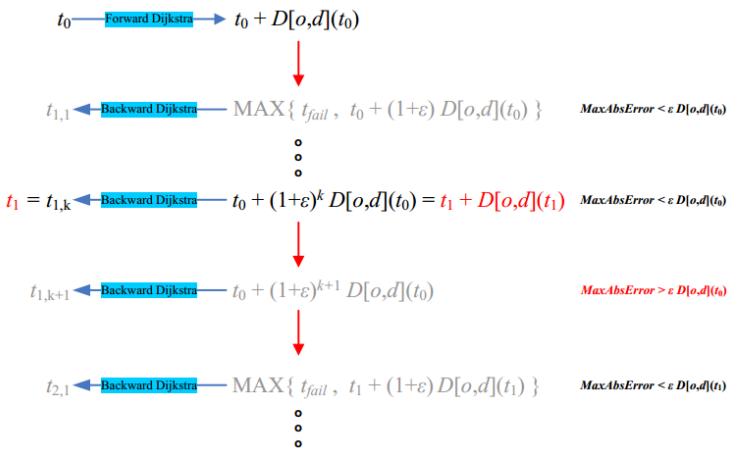


Figure 9: Singe pair approximation. First Phase.

- Second phase (small slope). For slope near to zero, this implies that the delay is slowly changing over the time axis. Therefore the error between the samples is estimated to be relatively small. In this case, the sampling is performed with the classic bisection method. We divide each time interval $[t_1, t_2]$ into two subintervals $[t_1, t_{mid}]$ and $[t_{mid}, t_2]$, computing the additional sample at t_{mid}. We keep bisecting the time-axis until the error between the samples, at the endpoints of the subintervals, become small enough and the required approximation guarantee is achieved [19].

- Certificate failure inspection (constant functional form). If the slope and the offset-constant of the delay function $D[o, d]$ remains unchanged, then no error can arise. The algorithm can learn online a such interval from the (primitive and minimization) shortest path failure certificates. In this case, the minimum certificate of o-d path is used in order to compute the expiration time, when this path and its delay function stop being optimal to a future departure time *tfail* from o.

Single source approximation. Similarly, as before, we want to provide an upper-approximation $\overline{D}[o, *]$ of a vector function $D[o, *]$, in such a way that $\overline{D}[o, v]$ and $\underline{D}[o, v]$, is a $(1 + \varepsilon)$-upper-approximation and $(1 - \varepsilon)$-lower-approximation of $D[o, v]$, for any destination vertex v, reachable from the origin o. The overall goal is to produce upper-approximating delay functions, such that : $\underline{D}[o, v](t) \leq D[o, v](t) \leq \overline{D}[o, v](t) \leq (1 + \varepsilon) \cdot D[o, v](t)$, $\forall o, v \in V$ and $t \in T$.

We start with a generalization of the bisection method, involving all destination vertices. Our method creates concurrently (i.e., within the same bisection) all the required approximated functions of $\overline{D}[o, *]$ and $\underline{D}[o, *]$. This is possible because the bisection is performed on the departure-time axis (common for all travel-time functions $D[o, *]$) from the same origin o. For each destination vertex $v \in V$, we only keep as breakpoints of $D[o, v]$, those sample points, which are indeed necessary for the required approximation guarantee, thus achieving an asymptotically optimal space-complexity of our method. We note that two consecutive samples to each $D[o, v]$ may arise different error magnitude. Therefore we let the bisection, at each level, only for the $D[o, v]$ that need to be approximated better. In any case, we use again the evaluation of the (worst-case) approximation error, but this time per destination vertex [19]. This guides the sampling of any $D[o, v]$, through the bisected time subintervals. Moreover, all the delays to be sampled at a particular bisection point are calculated by a single TDD execution. The time complexity of this approach will be asymptotically equal to that of the single-pair algorithm for the more demanding origin-destination pair.

Preprocessing. As in Foschini et al. [13], our algorithms are applied in subintervals $[t_s, t_f] \subseteq [0, T]$, in which $D[o, d]$ is a concave function. Because only the arc-delay functions $D[a]$ carry the responsibility for the concavity of $D[o, d]$, we search their local (primitive) concavity spoiling breakpoints, b_{cs} (see Figure 10). We then project the traced $O(mK)$ b_{cs} breakpoints to the respective latest departure-times (called concavity spoiling primitive images PI_{cs}), at any origin $o \in V$. The latest departure-times can be obtained by applying backwards TDD probes from all the tails of the arcs a, possessing b_{cs} in their delay function $D[a]$.

For each b_{cs}, we need to store $O(n)$ PI_{cs} images. The overall space complexity of PI_{cs} heuristics is $O(mnK)$. But we note, that usually b_{cs} are few in number, and thus the space that is required is $\Omega(n)$.

The PI_{cs} heuristics provide the material for producing the required subintervals $[t_s, t_f]$. Particularly, in order to preserve the concavity of $D[o, d]$, one has to project to departure-times from the origin o, only between PI_{cs}.

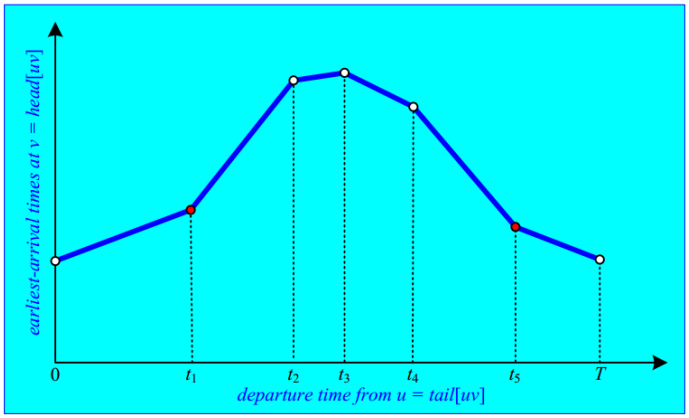


Figure 10: Concavity-spoiling breakpoints of arc-delay functions. Only the (red) delay-Concavity-Spoilers primitive breakpoints t_1, t_5 spoil the concavity of $D[uv]$, and possibly the concavity of $D[o, d]$. These breakpoints have to be projected (as primitive images) to departure-times from the origin o, for every arc in the network. Observe that the (positive) slope at time t_1 increases, while the (negative) slope at t_5 also increases. In all other (non-concavity-spoiling) breakpoints the arc-delay slopes decrease.

The PI_{cs} heuristics can be precomputed and then be considered as part of the input of the above algorithms. During a shortest path query, we should take care about selecting only the necessary PI_{cs}, for the corresponding $D[o, d]$ computation. Note that we want only PI_{cs} from arcs that are part of all (initially unknown) shortest o-d paths, over the time period $[0, T]$. Our simple approach, for filtering the PI_{cs}, is as follows: In the preprocessing phase, except of the PI_{cs}-departure times, we also compute and store the shortest free flow delay from any vertex v to the head of the arc that yields the PI_{cs}. This can be acquired by applying backwards Dijkstra using free flow arc-delays. After, during the query, at first, we compute the shortest worst congestion delay, in a similar manner as free flow one, from the origin vertex o to the destination vertex d (or the farthest destination, in case of multiple ones). Then we collect only the PI_{cs}, which have shortest free flow delay from o to their arc's head less or equal than the shortest worst congestion delay from o to d. Therefore this reduces (depending on the distance of o-d) an important number of neutral PI_{cs}, from farthest arcs, that they cannot belong to any shortest o-d path, during all the time period $[0, T]$.

Experimental setup. The experiments were conducted on an Intel(R) Core(TM) i5-2500K CPU @ 3.30GHz, with a cache size of 6Mb and 8Gb of RAM. Our implementations were written in C++ and compiled by GCC version 4.6.3, with optimization level 3. For the graph representation, we

used the Packed-Memory Graph (PMG) structure [20].

We tested our implementation on the road network of Berlin, with $n = 117839$ vertices and $m = 310152$ arcs. The data set was provided by TomTom [2]. The cost on the arcs is time dependent travel time, and it is defined by piecewise linear delay functions. Each arc-delay piecewise linear function consists of a sequence of leg-functions {slope, offset-constant, time interval}, whose generation is based on the provided TomTom's speed profiles.

In our approach, for reducing the number of breakpoints, we have replaced successive time slots, which their yielded delay time differs 1-3 seconds, with a single arc-delay, as the linear interpolation through them. The total number of the legs/breakpoints, from all arc-delay functions, is 359225. The most arcs have delay functions with zero slope, providing a constant travel time, during all the time period. Furthermore, 18833 arcs have an average of 5 legs (and maximum 46), while the rest 369235 arcs only one. The slope of the delay functions is in $[-0.5, 0.5]$.

In the experiments, we ran 100 queries, where the origin vertex o and the destination vertex d were selected uniformly at random among all vertices. For the time period, we selected Tuesday, as we noticed that is one of the days of week, containing the most non-constant delay functions.

Input parameters. For any query, we use the approximation error ratio ε, such that $D[o,d] \cdot (1-\varepsilon) \leq \underline{D}[o,d] \leq D[o,d] \leq \overline{D}[o,d] \leq (1+\varepsilon) \cdot D[o,d]$. In the experiment, we set ε =0.01, 0.05, 0.1 and 0.25. This means that the resulted $\overline{D}[o,d]$ delay is at most 1%, 5%, 10% and 25%, respectively, higher than $D[o,d]$.

In the case of single-source queries, instead of considering all vertices of graph, as individual targets, we focus on fewer. This can be advantageous, because the computation cost is depending on the number of the target vertices, and we don't always need to process the full graph, but only a subgraph around the selected origin vertex o. In our results, we report the number of the target vertices (Figure 15 and 16).

In order to define the target vertices, we have used a free-flow time horizon thz (among several criteria). This time horizon defines which vertices in graph can be reached from the selected origin vertex o in at most thz delay, via the best traffic case. However, in this matter, we clarify that along the time period $[0, T]$ the delay can become greater than thz, especially during rush hours. The process for collecting the target vertices is performed by running TDD, with free flow arc-delays. In this case, the settled nodes in at most thz delay, will be on the resulted free-flow shortest path tree. However, at the sampling phase of our approximation algorithm, we shall not restrict the exploring of TDD only in the subgraph, that contains the target-destination vertices. In order to guarantee that our approximation algorithm will perform the sampling on the optimal delay functions and use the correct certificates, over all the time period, we also need to build another shortest path tree. This time using the worst congestion delays, and at a distance such that just all target vertices are settled. We then extend the free flow shortest path tree, up to the max shortest congested delay to the farthest target vertex. From this tree the additional settled nodes may be required, at some time point of $[0, T]$, to take part on the building of the shortest paths and their delay functions to a vertex destination.

On the computation of $D[o,d]$, we tested the algorithms not only for the full time period $[0, T]$, but also in small to large subintervals. In this matter, we provide an option on selecting a time window $tw = [t_s, t_f] \subseteq [0, T]$ for specific departure time intervals, that user interests.

Experimental Results. For the experimental evaluation of the proposed algorithms, we present: a) the execution time and b) the number of samples that were obtained during the approximation of the shortest delay $D[o,d]$ functions. Also, we associate the performance of the algorithms with the number of the target-vertices (for single-source queries), the travel time horizon, and the time window, regarding the departure time range from origin o.

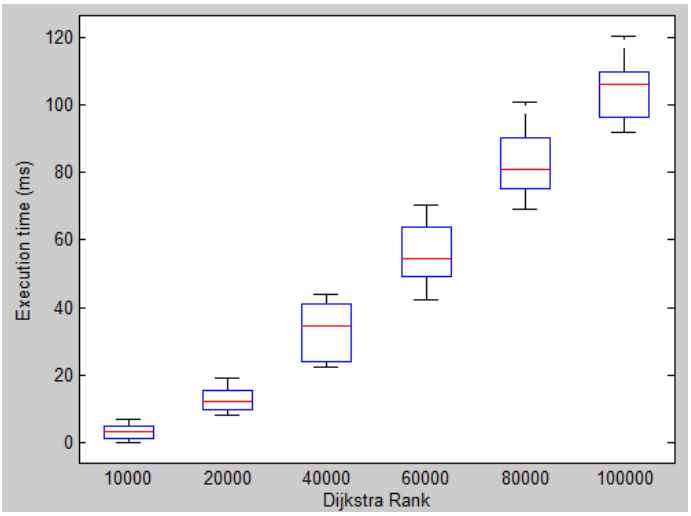


Figure 11: Execution time on single-pair shortest path queries. Time window: [0:00-12:00].

Figures 11, 12, 13, 14 shows the execution time by performing single-pair queries. In these figures, we group the queries, by their computation difficulty, which is reflected by Dijkstra rank. This rank denotes the average number of the nodes required to be settled by TDD, throughout the approximation procedure, in the given departure time window.

About the performance of both approximation algorithms, firstly, we observe that the execution time is not necessarily increased as the departure time window is growing up. This is mainly due that we let our algorithms take into account the shortest path failure certificates. Therefore, if the delay functional form remains constant for large time intervals, the approximation error is zero, and the sampling work minimum. This can be possible at certain departure times, e.g. when there is low congestion traffic (Figure 11). In our experiments, such cases concentrated when departing from evening to morning hours. In Figure 11, we take almost the same execution time results for the time window [14:00, 22:00].

Moreover, we observe that when time window enters in high volatility congestion time ranges, this has a great influence on the performance of the algorithms. More specifically, during peak periods, and mostly for late morning and early afternoon hours (Figure 12), a high percentage of legs of the arc-delay functions is triggered. In this case, the functional form of $D[o, d]$ is more likely to change, making the sampling work of the algorithms more difficult. Also, the certificates become less valuable, because of the small expiration - time horizons of the mutable delay functions.

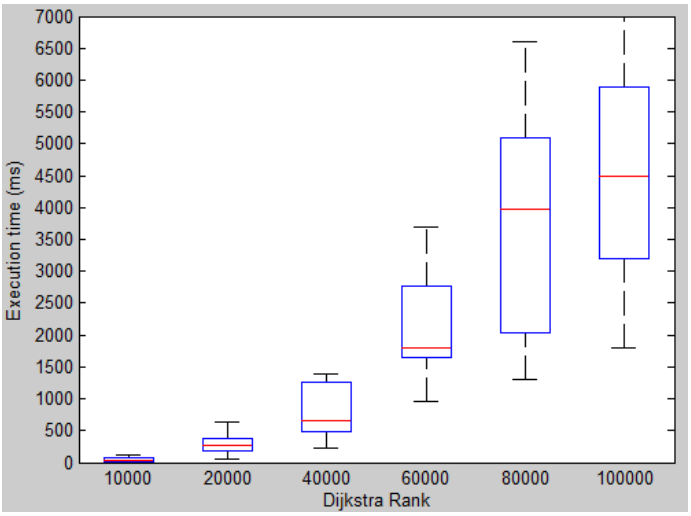


Figure 12: Execution time on single-pair shortest path queries. Time window: [12:00-13:00].

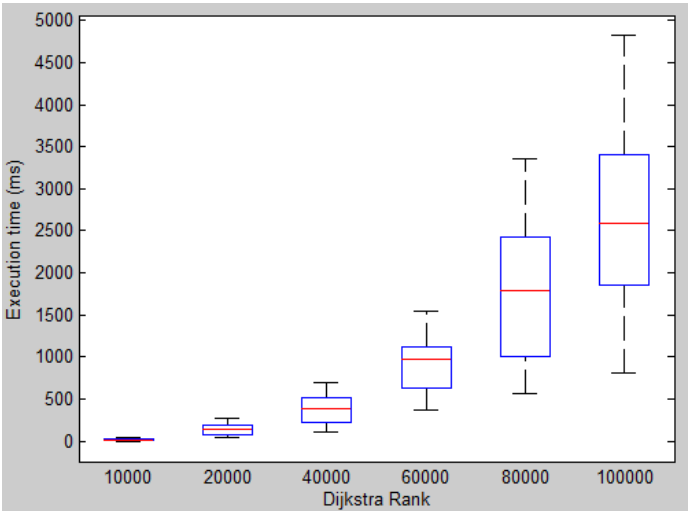


Figure 13: Execution time on single-pair shortest path queries. Time window: [0:00-12:00].

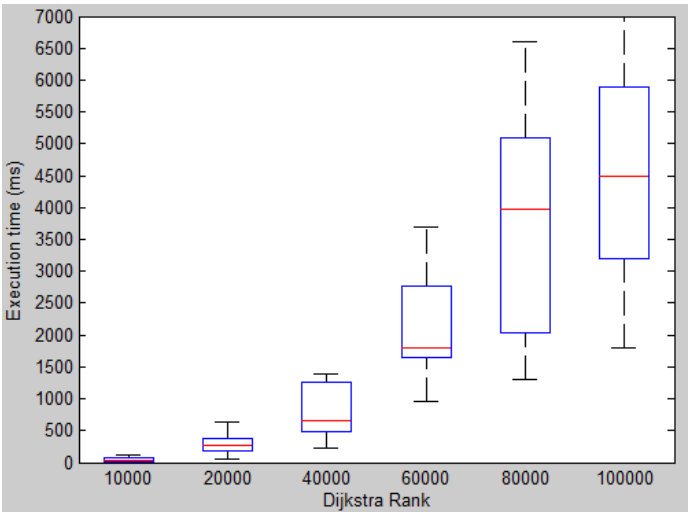


Figure 14: Execution time on single-pair shortest path queries. Time window: all day.

The single-source algorithm, Figures 15 and 16, is more expensive than the single-pair one, because the delay function sampling is extended to more than one destination vertex. In this case, as we increase the number of the targets, the execution time gets bigger. Again the performance of the algorithm is dependent on the how much delay function is changeable, than the length of the time window, that we consider. In figures, the number of the targets are obtained for free flow delay thz, up to 15 mins.

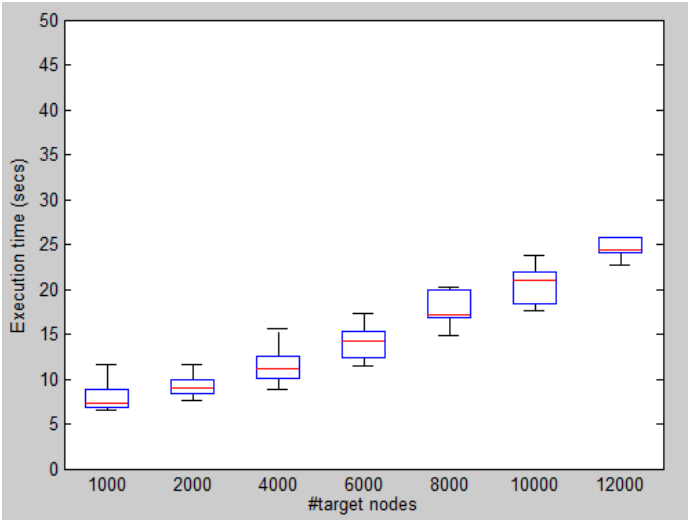


Figure 15: Execution time on single-source shortest path queries. Time window: [0:00-12:00].

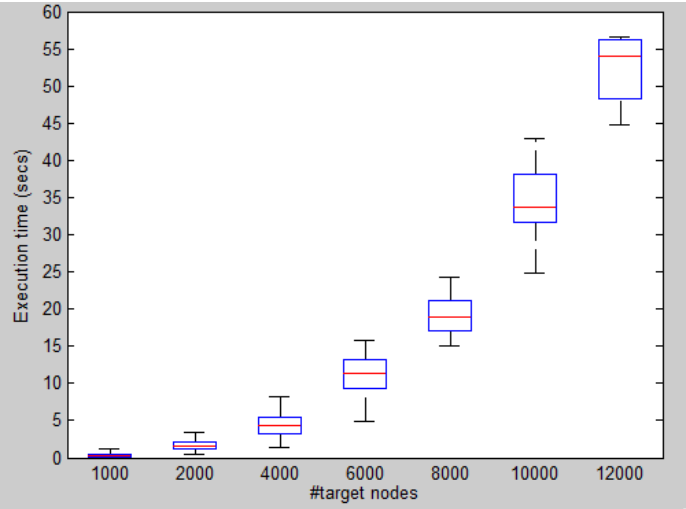


Figure 16: Execution time on single-source shortest path queries. Time window: all day.

In the experiments, we notice that many concavity spooling breakpoints are triggered at rush hours. Therefore their yielded primitive images lead to dividing the departure time window into several small subintervals, densely populated at times with sharp congestion. These small in width subintervals, force the algorithms collecting a large number of samples, is some cases, more than could probably needed. This additionally results on a high approximation of the delay function, independently the selected approximation ratio.

ε	0.01	0.05	0.1	0.25
$0 < targets \leq 2000$				
$\#samples/target$	236.25	233.86	233.20	232.82
$2000 < targets \leq 4000$				
$\#samples/target$	1143.38	1140.30	1139.87	1139.64
$4000 < targets \leq 8000$				
$\#samples/target$	2609.71	2606.55	2606.19	2606.00
$8000 < targets \leq 10000$				
$\#samples/target$	4234.74	4231.46	4231.08	4230.95

Table 4: Single-source approximation. Average number of samples per target based on the approximation error ratio ε.

4.2 Dynamic Time-Dependent Customizable Route Planning

In this section, we evaluate the performance of our separator-based approach to time-dependent route planning. For details on the algorithmic approach and a discussion of related work (e. g., [10]), please refer to Deliverable D2.2.

Experiments. We implemented all algorithms in C++ using g++ 4.7.1 (flag -O3) as compiler. Experiments were conducted on a dual 8-core Intel Xeon E5-2670 clocked at 2.6 GHz, with 64 GiB of DDR3-1600 RAM, 20 MiB of L3 and 256 KiB of L2 cache. Unless otherwise noted, we ran our implementation sequentially.

Table 5: Network properties. We report the number of vertices and arcs of the routing graph, and as a measure of time-dependency, the total amount of break points in the whole network as well as the average arc complexity.

Network	Time resol.	# Vertices	# Arcs	# Break points	Avg. # BPs per arc
Berlin	1 s	443 369	988 686	2 087 913	2.11
Berlin	0.1 s	443 369	988 686	11 438 281	11.57
Germany	1 s	4 688 214	10 795 826	8 527 620	0.79
Germany	0.1 s	4 688 214	10 795 826	10 970 201	1.02

Table 6: Customization performance.

Network	Time resol.	# Thr.	# Add. break points			Customization time [sec]			
			Lvl 1	Lvl 2	Lvl 3	Lvl 1	Lvl 2	Lvl 3	Total
Berlin	1 s	16	5 740 292	15 849 458	25 388 351	0.67	1.95	34.55	37.44
Berlin	0.1 s	16	30 049 724	78 872 663	89 802 847	1.12	9.29	119.88	131.33
Germany	1 s	16	19 827 634	40 637 476	61 630 170	17.59	6.36	18.65	43.26
Germany	0.1 s	16	27 693 336	58 718 516	91 781 882	19.23	6.59	26.02	52.85

Input Data and Methodology. Our test instances are based on the road network of Germany, kindly provided by PTV AG, and the road network of Berlin/Brandenburg, kindly provided by TomTom. Both inputs are specified in a similar way, essentially consisting of a list of road segments of given length, free flow speed, and a reference to a delay profile. This profile describes the time-dependent behavior of the road segment as the relative change in speed as a function of time. For details, see Section 2.

For our purposes, we instead associate each road segment with a piecewise linear function mapping time of day to travel time (c. f. [10]). During the conversion from the input, we observe that for short road segments of high free flow speed (e. g., 1 m length, 67 km/h speed), the resulting travel time functions are almost constant, deviating only in milliseconds and below. Hence, during the conversion to travel time, we may assume fixed time resolutions on arc delay functions such as one second or one 10th of a second. For both instances, we extracted the 24 hour profile of a Tuesday. See Table 5 for the resulting network statistics.

For partitioning, we used BUFFOON [22], which is explicitly developed for road networks and aims at minimizing the number of boundary arcs. For both instances, we computed a nested 3-level partition with 2^5, 2^{10}, 2^{15} cells. Computing partitions takes less than one hour, each. Considering that road topology changes rarely (i. e., the partition needs be updated only when roads are built or (permanently) closed), this is sufficiently fast in practice. For a detailed evaluation of the trade-off between partitioning speed and quality, see [8].

Performance Evaluation. In Table 6, we report the performance of customization for different instances. Total customization time is fast enough to incorporate frequent metric updates, ranging from half a minute to 2.2 minutes computation time. Detailed analysis shows that most effort is spent on the highest level, which also introduces the largest amount of additional number of break points (up to 12 times more than the number of break points on original arcs). In Table 7, we show detailed statistics on the arc complexity after customization. Note that the time resolution chosen for the input has a profound effect on the number of resulting break points and hence the customization performance.

Customization is easy to parallelize between cells. In Table 8, we report details on the cus-

Table 7: Histogram of arc complexity after customization.

# BPs	Berlin-1s	Berlin-0.1s	Germany-1s
constant	343 716	325 542	1 438 871
< 100	224 192	107 003	1 546 863
< 200	32 495	74 971	185 805
< 300	22 187	36 549	84 648
< 400	11 340	16 161	43 885
< 500	5 128	8 305	17 390
< 600	3 141	5 812	5 296
< 700	2 663	5 795	1 622
< 800	2 761	5 814	662
< 900	2 744	5 937	126
< 1 000	2 502	5 949	11
< 1 100	2 219	5 823	—
< 1 200	2 015	5 866	—
< 1 300	1 767	5 512	—
< 1 400	1 527	4 984	—
< 1 500	1 273	4 421	—
< 1 600	934	3 831	—
< 1 700	526	3 100	—
< 1 800	307	2 383	—
< 1 900	142	1 928	—
< 2 000	52	1 437	—
< 3 000	9	9 483	—
< 4 000	—	12 071	—
< 5 000	—	4 896	—
< 6 000	—	67	—

tomization performance for increasing number of threads. While the speedup on Level 2 is very good (factor of ≈ 13), it deteriorates on the most expensive Level 3 (factor of ≈ 3). This is explained by the fact that, for the current graph partitions, cells are rather unbalanced wrt. to break point complexity of their contained arcs; since the highest level only has 32 cells, i. e., 2 per core when parallelized on 16 thread, this results in a rather unbalanced utilization of the threads.

In Table 9, we report figures on query performance, observing that multi-level queries (CRP) are between a factor of 21 to 107 faster than traditional Dijkstra. Note that CRP even scans two to three orders of magnitude less vertices than Dijkstra, but that the number of scanned break points is only up to 1 order of magnitude lower. Most interestingly, when comparing CRP and Dijkstra query times, CRP performance seems to be more robust over different input instances.

Since customization of the the highest level of the partition is most expensive, we also evaluated the performance of CRP when ignoring level 3 in the query phase (removing it from the partition). On Berlin-0.1 s, this increases the number of scanned vertices by a factor of 5 but the query time only by a factor of 2-3, offering an interesting customization–query-time trade-off.

Conclusion. Our experimental evaluation so far shows that a multi-level separator approach to time-dependent route planning is not only promising but indeed practical. There are several aspects of future work. We would like to evaluate a more diverse set of test instances, taking into account bigger road networks. Naturally, we expect to see greater performance gains over Dijkstra on such instances. To this end, we also plan to consider full-week profiles instead of only 24-hour profiles, which to our knowledge has not been done in published work so far. Since customization effort

Table 8: Multi-core customization performance.

Network	Time resol.	# Thr.	Customization time [sec] Lvl 1	Lvl 2	Lvl 3	Total
Berlin	1 s	1	1.38	25.54	115.41	142.56
Berlin	1 s	2	0.76	13.07	59.32	73.39
Berlin	1 s	4	0.42	6.74	36.91	44.30
Berlin	1 s	8	0.28	3.61	34.84	38.97
Berlin	1 s	16	0.67	1.95	34.55	37.44

Table 9: Query performance.

Network	Time res.	Algorithm	# Scn. vertices	# Scn. BPs	Time [ms]
Berlin	1 s	Dijkstra	222 359	639 111	53.54
Berlin	0.1 s	Dijkstra	222 410	1 909 451	74.03
Germany	1 s	Dijkstra	2 489 829	1 832 060	534.90
Germany	0.1 s	Dijkstra	2 490 073	2 070 130	550.88
Berlin	1 s	CRP	1 140	353 384	2.54
Berlin	0.1 s	CRP	1 141	505 344	3.32
Germany	1 s	CRP	3 478	111 957	4.98
Germany	0.1 s	CRP	3 479	136 630	5.20

increases quickly from level to level, we would like to apply approximation techniques between levels, reducing the stored amount of break points on overlay arcs, before continuing with the next level of customization. This could also eliminate the imbalance that currently hinders parallelization for higher number of threads. In this regard, we are also interested in evaluating different partitioning schemes. Finally, we will consider the scenario of local updates due to live traffic data and traffic prediction more carefully. It seems reasonable to assume that most of the customization data, especially on higher levels, does not need to be updated in such cases, resulting in much faster customization time.

5 Alternative Route Planning

5.1 Introduction

Route planning services – offered by web-based, hand-held, or in-car navigation systems – are heavily used by more and more people. Typically, such systems (as well as the vast majority of route planning algorithms) offer a best route from a source (origin) s to a target (destination) t, under a single criterion (usually distance or time). Quite often, however, computing only one such s-t route may not be sufficient, since humans would like to have choices and every human has also his/her own preferences. These preferences may well vary and depend on specialized knowledge or subjective criteria (like or dislike certain part of a road), which are not always practical or easy to obtain and/or estimate on a daily basis. Therefore, a route planning system offering a set of good/reasonable alternatives can hope that (at least) one of them can satisfy the user, and vice versa, the user can have them as back-up choices for altering his/her route in case of emergent traffic conditions. This can be particularly useful in several cases. For example, when the user has to choose the next optimal alternative path, because in the current one, adverse incidents are occurred, like traffic jams, accidents or permanent unavailability due to construction work.

The aggregation of alternative paths between a source s and a target t can be captured by the concept of the *Alternative Graph* (AG), a notion first introduced in [5]. Storing paths in an AG makes sense, because in general alternative paths may share common nodes (including s and t) and edges. Furthermore, their subpaths may be combined to form new alternative paths.

In general, there may be several alternative paths from s to t. Hence, there is a need for filtering and rating all alternatives, based on certain quality criteria. The study in [5] quantified the quality characteristics of an alternative graph (AG), captured by three criteria. These concern the non-overlappingness ($totalDistnace$) and the stretch ($averageDistnace$) of the routes, as well as the number of $decisionEdges$ (sum of node out-degrees) in AG. For more details, see Deliverable D2.2. As it is shown in [5], all of them together are important in order to produce a high-quality AG. However, optimizing a simple objective function combining just any two of them is already an NP-hard problem [5]. Hence, one has to concentrate on heuristics. Four heuristic approaches were investigated in [5] with those based on Plateau [3], Penalty [7], and a combination of them to be the best.

In this deliverable, for the sake of completeness, we present our final improved methods for computing a set of alternative source-to-destination routes in road networks in the form of an alternative graph, which appear to be more suitable for practical navigation systems [4, 18]. These methods appeared in [21]. The resulting alternative graphs are characterized by minimum path overlap, small stretch factor, as well as low size and complexity. Our approach improves upon a previous one by introducing a new pruning stage preceding any other heuristic method and by introducing a new filtering and fine-tuning of two existing methods.

We extend the approach in [5] for building AGs in two directions. First, we introduce a pruning stage that precedes the execution (and it is independent) of any heuristic method, thus reducing the search space and hence detecting the nodes on shortest routes much faster. Second, we provide several improvements on both the Plateau and Penalty methods. In particular, we use a different approach for filtering plateaus in order to identify the best plateaus that will eventually produce the most qualitative alternative routes, in terms of minimum overlapping and stretch. We also introduce a practical and well-performed combination of the Plateau and Penalty methods with tighter lower-bounding based heuristics. This has the additional advantage that the lower bounds remain valid for use even when the edge costs are increased (without requiring new preprocessing), and hence are useful in dynamic environments where the travel time may be increased, for instance, due to traffic jams.

Finally, we conducted an experimental study for verifying our methods on several road networks of Western Europe. Our experiments showed that our methods can produce AGs of high quality pretty fast.

The rest of this section is organized as follows. In subsection 5.2, we provide the main background information, from Deliverable 2.2. In subsection 5.3, we present our proposed improvements for producing AGs of better quality. In subsection 5.4, we report a thorough experimental evaluation of our improved methods. In subsection 5.5, we demonstrate some of the visualized results we got with our alternative route planning implementation.

5.2 Preliminaries

A *road network* can be modeled as a *directed graph* $G = (V, E)$, where each node $v \in V$ represents intersection points along roads, and each edge $e \in E$ represents road segments between pairs of nodes. Let $|V| = n$ and $|E| = m$ and $d(u, v) \equiv d_G(u, v)$ be the shortest distance from u to v in graph G.

We consider the problem of tracing alternative paths from a source node s to a target node t in G, with edge weight or cost function $w : E \to \mathbb{R}^+$. The essential goal is to obtain sufficiently different paths with optimal or near optimal cost. We proceed with the definitions of an alternative graph and its quality indicators.

Alternative Graph. Formally, an AG $H = (V', E')$ [5] is a graph, with $V' \subseteq V$, and such that for all $e = (u, v) \in E'$, there is a P_{uv} path in G and a P_{st} path in H, so that $e \in P_{st}$ and $w(e) = w(P_{uv})$, where $w(P_{uv})$ denotes the weight or cost of path P_{uv}. Let $d_H(u, v)$ be the shortest distance from u to v in graph H.

Quality indicators. For filtering and rating the alternatives in an AG, we use the following indicators, as in [5]:

$$totalDistance = \sum_{e=(u,v)\in E'} \frac{w(e)}{d_H(s,u) + w(e) + d_H(v,t)} \qquad (overlapping)$$

$$averageDistance = \frac{\sum_{e\in E'} w(e)}{d_G(s,t) \cdot totalDistance} \qquad (stretch)$$

$$decisionEdges = \sum_{v\in V'\setminus\{t\}} (outdegree(v) - 1) \qquad (size\ of\ AG)$$

In the above definitions, the *totalDistance* measures the extend to which the paths in AG are non-overlapping. Its maximum value is *decisionEdges*+1. This is equal to the number of all s-t paths in AG, when these are disjoint, i.e. not sharing common edges.

The *averageDistance* measures the average cost of the alternatives compared with the shortest one (i.e. the stretch). Its minimum value is 1. This occurs, when every s-t path in AG has the minimum cost.

The *decisionEdges* measures the size complexity of AG. In particular, the number of the alternative paths in AG, depend on the "decision branches" are in AG. For this reason, as high the number o *decisionEdges*, the more confusion is created to a typical user, when he tries to decide his route. Therefore, it should be bounded.

Consequently, to compute a qualitative AG, one aims at high *totalDistance* and low *averageDistance*. Examples of the use of the above quality indicators can be found in Deliverable D2.2.

5.3 Our Improvements

In Deliverable 2.2, we reviewed the previous approaches for computing alternative graphs, and briefly highlighted our improved methods. In this deliverable, we present in detail these improved methods by extending the Plateau and Penalty approaches. Our improvements are twofold :

A) We introduce a pruning stage that precedes the Plateau and Penalty methods in order to a-priori reduce their search space without sacrificing the quality of the resulted alternative graphs.

B) We use a different approach for filtering plateaus in order to obtain the ones that generate the best alternative paths. In addition, we fine tune the penalty method, by carefully choosing the penalizing factors on the so far computed P_{st} paths, in order to trace the next best alternatives.

5.3.1 Pruning

We present two bidirectional Dijkstra-based *pruners*. The purpose of both of them, is to identify the nodes that are in P_{st} shortest paths. We refer to such nodes, as the useful search space, and the rest ones, as the useless search space. Our goal, through the use of search pruners, is to ensure: (a) a more quality-oriented build of the AG and (b) a reduced dependency of the time computation complexity from graph size. The latter is necessary, in order to acquire fast response on queries. We note that the benefits are notably for the Penalty method. This is because, the Penalty method needs to run iteratively several s-t shortest path queries. Thus, having put aside the useless nodes and focussing only on the useful ones, we can get faster processing. We also note that, over the P_{st} paths with the minimum cost, it may be desired as well to let in AG paths with near optimal cost, say $\tau \cdot d_s(t)$, which will be the maximum acceptable cost $w(P_{st})$. Indicatively, $1 \leqslant \tau \leqslant 1.4$. Obviously, nodes far away from both s and t, with $d_s(v) + d_t(v) > \tau \cdot d_s(t)$, belong to P_{st} paths with prohibitively high cost. In the following we provide the detailed description of both pruners, which are illustrated in Figures 17 and 18.

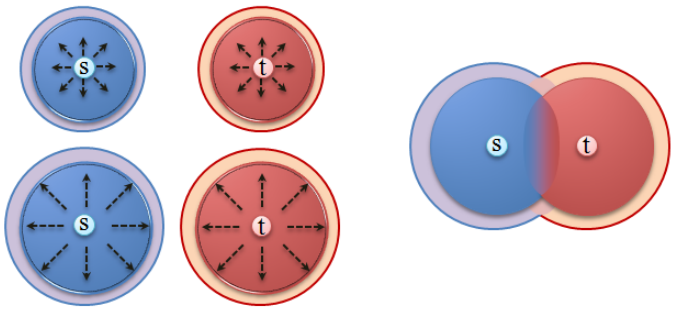


Figure 17: The forward and backward searches meet each other. In this phase the minimum distance $d_s(t)$ is traced.

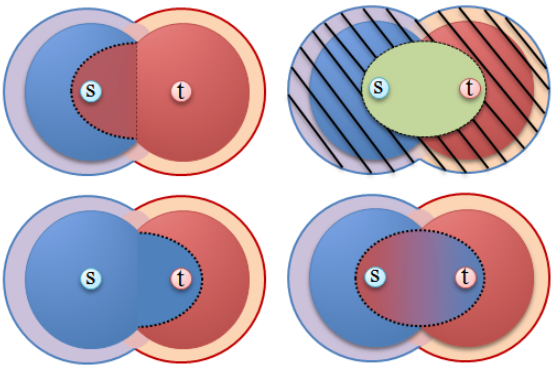


Figure 18: The forward and backward settles only the nodes in the shortest paths, taking account of the overall $d_s(v) + d_t(v)$.

Uninformed Bidirectional Pruner. In this pruner, there is no preprocessing stage. Instead, the used heuristics are obtained from the minimum distances of the nodes enqueued in Q_f and Q_b, i.e. $Q_f.minKey() = \min_{u \in Q_f}\{d_s(u)\}$ and $Q_b.minKey() = \min_{v \in Q_b}\{d_t(v)\}$.

We extend the regular bidirectional Dijkstra, by adding one extra phase. First, for computing the minimum distance $d_s(t)$, we let the expansion of forward and backward search until $Q_f.minKey() + Q_b.minKey() \geq d_s(t)$. At this step, the current forward T_f and backward T_b shortest path trees produced by the bidirectional algorithm will have crossed each other and so the minimum distance $d_s(t)$ will be determined. Second, at the new extra phase, we continue the expansion of T_f and T_b in order to include the remaining useful nodes, such that $d_s(v) + d_t(v) \leq \tau \cdot d_s(t)$, but with a different mode. This time, we do not allow the two searches to continue their exploration at nodes v that have $d_s(v) + h_t(v)$ or $h_s(v) + d_t(v)$ greater than $\tau \cdot d_s(t)$. We use the fact that Q_f and Q_b can provide lower-bound estimates for $h_s(v)$ and $h_t(v)$. Specifically, a node that is not settled or explored from backward search has as a lower bound to its distance to t, $h_t(v) = Q_b.minKey()$. This is because the backward search settles the nodes in increasing order of their distance to t, and if u has not been settled then it must have $d_t(u) \geq Q_b.minKey()$. Similarly, a node that is not settled or explored from forward search has a lower bound $h_s(v) = Q_f.minKey()$. Furthermore, when a search settles a node that is also settled from the other search we can calculate exactly the sum $d_s(u) + d_t(u)$. In this case, the higher the expansion of forward and backward search is, the more tight the lower bounds become. The pruning is ended, when Q_f and Q_b are empty.

Before the termination, we exclude the remaining useless nodes that both searches settled during the pruning, that is all nodes v with $d_s(v) + d_t(v) > \tau \cdot d_s(t)$.

Informed ALT bidirectional pruner. In the second pruner, our steps are similar, except that we use tighter lower bounds. We acquire them in a one-time preprocessing stage, using the *ALT* approach. In this case, the lower bounds that are yielded can guide faster and more accurately the pruning of the search space. We compute the shortest distances between the nodes in G and a small set of landmarks. For tracing the minimum distance $d_s(t)$, we use *BLA* as base algorithm, which achieves the lowest waste exploration, as experimental results showed in [14, 20]. During the pruning, we skip the nodes that have $d_s(v) + h_t(v)$ or $h_s(v) + d_t(v)$ greater than $\tau \cdot d_s(t)$.

The use of lower-bounding heuristics can be advantageous. In general, a heuristic stops being valid when a change in the weight of the edges occurs. But note that in the penalty method, we consider only increases on the edge weights and therefore this does not affect the lower bounds on the shortest distances. Therefore, the combination of the *ALT* speedup [20, 14] with Penalty is suitable. However, depending on the number and the magnitude of the increases the lower bounds can become less tight for the new shortest distances, leading to a reduced performance on computing the shortest paths.

5.3.2 Filtering and Fine-tuning

Over the standard processing operations of Penalty and Plateau, we introduce new ones for obtaining better results. In particular:

Plateau. We use a different approach on filtering plateaus. Specifically, over the cost of a plateau path we take into account also its non-overlapping with others. In this case, the difficulty is that the candidate paths may share common edges or subpaths, so the *totalDistance* is not fixed. Since at each step an insertion of the current best alternative path in AG may lead to a reduced *totalDistance* for the rest candidate alternatives, primarily we focus only on their unoccupied parts, i.e., those that are not in AG. We rank a x-y plateau $\overline{P}$ with $rank = totalDistance - averageDistance$, where $totalDistance = \frac{w(\overline{P})}{d_s(x)+w(\overline{P})+d_t(y)}$ is its definite non-overlapping degree, and $averageDistance = \frac{w(\overline{P})+d_s(t)}{(1+totalDistance)\cdot d_s(t)}$ is its stretch over the shortest s-t path in G. During the collection of plateaus, we insert the highest ranked of them via its node-connectors $v \in \overline{P}$ in T_f and T_b to a min heap with fixed size equal to *decisionEdges* plus an offset. The offset increases the

number of the candidate plateaus, when there are available, and it is required only as a way out, in the case, where several P_{st} paths via the occupied plateaus in AG lead to low $totalDistance$ for the rest P_{st} paths via the unoccupied plateaus.

Penalty. When we "penalize" the last computed P_{st} path, we adjust the increases on the weights of its outgoing and incoming edges, as follows:

$$w_{new}(e) = w(e) + (0.1 + r \cdot d_s(u)/d_s(t)) \cdot w_{old}(e), \qquad \forall e = (u,v) \in E:\ u \in P_{st},\ v \notin P_{st}$$
$$w_{new}(e) = w(e) + (0.1 + r \cdot d_t(v)/d_t(s)) \cdot w_{old}(e), \qquad \forall e = (u,v) \in E:\ u \notin P_{st},\ v \in P_{st}$$

The first adjustment puts heavier weights on those outgoing edges that are closer to the target t. The second adjustment puts heavier weights on those incoming edges that are closer to the source s. The purpose of both is to reduce the possibility of recomputing alternative paths that tend to rejoin directly with the previous one traced.

An additional care is given also for the nodes u in P_{st}, having $outdegree(u) > 1$. Note that their outgoing edges can form different branches. Since the edge-branches in G constitute generators for alternative paths, they are important. These edges are being inserted to AG with a greater magnitude of weight increase than the rest of the edges.

The insertion of the discovered alternative paths in G and the maintenance of the overall quality of AG should be controlled online. Therefore, we establish an online interaction with the AG's quality indicators, described in subsection 5.2, for both Plateau and Penalty. This is also necessary because at each step an insertion of the current best alternative may lead to a reduced value of $totalDistance$ for the next candidate alternative paths that share common edges with the already computed AG.

In order to get the best alternatives, we seek to maximize the $targetfunction = totalDistance - \alpha \cdot averageDistance$, where α is a balance factor that adjusts the stretch magnitude rather than the overlapping magnitude. Maximization of the target function leads to select the best set of low overlapping and shortest alternative paths.

Since the penalty method can work on any pre-computed AG, it can be combined with Plateau. In this way, we collect the best alternatives from Penalty and Plateau, so that the resulting set of alternatives maximizes the target function. In this matter, we can extend the number of decision edges and after the gathering of all alternatives, we end by performing thinout in AG. Moreover, in order to guide the Penalty method to the remaining alternatives, we set a penalty on the paths stored by Plateau in AG, by increasing their weights. We also use the same pruning stage to accommodate both of them.

5.4 Experimental Results

The experiments were conducted on an Intel(R) Xeon(R) Processor X3430 @ 2.40GHz, with a cache size of 8Mb and 32Gb of RAM. Our implementations were written in C++ and compiled by GCC version 4.6.3 with optimization level 3.

The data sets of the road networks in our experiments were acquired from OSM [1] and TomTom [2]. The weight function is the travel time along the edges. In the case of OSM, for each edge, we calculated the travel time based on the length and category of the roads (residential street, tertiary, secondary, primary road, trunk, motorway, etc). The data set of the Greater Berlin area was kindly provided by TomTom in the frame of the eCOMPASS project [4]. The size of the data sets are reported in Table 10.

map		n	m
B	Berlin	117,839	310,152
LU	Luxembourg	51,576	119,711
BE	Belgium	576,465	1,376,142
IT	Italy	2,425,667	5,551,700
GB	GreatBritain	3,233,096	7,151,300
FR	France	4,773,488	11,269,569
GE	Germany	7,782,773	18,983,043
WE	WesternEurope	26,498,732	62,348,328

Table 10: The size of road networks, where n denotes the number of nodes and m denotes the number of edges.

For our implementations, we used the packed-memory graph (PMG) structure [20]. This is a highly optimized graph structure, part of a larger algorithmic framework, specifically suited for very large scale networks. It provides dynamic memory management of the graph and thus the ability to control the storing scheme of nodes and edges in memory for optimization purposes. It supports almost optimal scanning of consecutive nodes and edges and can incorporate dynamic changes in the graph layout in a matter of μs. The ordering of the nodes and edges in memory is in such a way that increases the locality of references, causing as few memory misses as possible and thus a reduced running time for the used algorithms.

We tested our implementations in the road network of the Greater Berlin area, the Western Europe (Austria, Belgium, Denmark, France, Germany, Italy, Luxembourg, Netherlands, Norway, Portugal, Spain, Sweden, Switzerland, and Great Britain), as well as in the network of each individual West European country. In the experiments, we considered 100 queries, where the source s and the destination t were selected uniformly at random among all nodes. For the case of the entire Western European road network, the only limitation is that the s-t queries are selected, such that their geographical distance is at most 300 kilometers. This was due to the fact that although modern car navigation systems may store the entire maps, they are mostly used for distances up to a few hundred kilometers.

For far apart source and destination, the search space of the alternative P_{st} paths gets too large. In such cases, it is more likely that many non-overlapping long (in number of edges) paths exist between s and t. Therefore, this has a major effect on the computation cost of the overall alternative route planning. In general, the number of non-overlapping shortest paths depends on the density of the road networks as well on the edge weights.

There is a trade-off between the quality of AG and the computation cost. Thus, we can sacrifice a bit of the overall quality to reduce the running time. Consequently, in order to deal with the high computation cost of the alternative route planning for far apart sources and destinations we can decrease the parameter τ (max stretch). A dynamic and online adjustment of τ based on the geographical distance between source and target can be used too. For instance, at distance larger than 200km, we can set a smaller value to τ, e.g. close to 1, to reduce the stretch and thereby the number of the alternatives. We adopted this arrangement on large networks (Germany, Western Europe). For all others, we set $\tau = 1.2$, which means that any traced path has cost at most 20% larger than the minimum one. To all road networks, we also set $averageDistance \leq 1.1$ to ensure that, in the filtering stage, the average cost of the collected paths is at most 10% larger than the minimum one.

In order to fulfill the ordinary human requirements and deliver an easily representable AG, we have bounded the *decisionEdges* to 10. In this way, the resulted AG has small size, $|V'| \ll |V|$ and $|E'| \ll |E|$, thus making it easy to store or process. Our experiments showed that the size of an AG is at most 3 to 4 times the size of a shortest s-t path, which we consider as a rather acceptable solution.

Our base *target function* [3] in Plateau and Penalty is $totalDistance - averageDistance + 1$. Regarding the pruning stage of Plateau and Penalty, we have used the ALT-based informed bidirectional pruner with at most 24 landmarks for Western Europe.

In Tables 11, 12, and 13, we report the results of our experiments on the various quality indicators: targetFunction (*TargFun*), totalDistance (*TotDist*), averageDistance (*AvgDist*) and decisionEdges (*DecEdges*). The values in parentheses in the header columns provide only the theoretically maximum or minimum values per quality indicator, which may be far away from the optimal values (that are based on the road network and the *s-t* queries).

In Tables 11, 12, and 13, we report the average value per indicator. The overall execution time for computing the entire *AG* is given in milliseconds. As we see, we can achieve a high-quality *AG* in less than a second even for continental size networks. The produced alternative paths in *AG* are directly-accessible for use (e.g., they are not stored in any compressed form).

Due to the limitation on the number of the decision edges in *AG* and the low upper bound in stretch, we have chosen in the Penalty method small penalty factors, $p = 0.1$ and $r = 0.1$. In addition, this serves in getting better low-stretch results, see Table 12. In contrast, the *averageDistance* in Plateau gets slightly closer to the 1.1 upper bound.

In our experiments, the Penalty method clearly outperforms Plateau on finding results of higher quality. However it has higher computation cost. This is reasonable because it needs to perform around to 10 shortest *s-t* path queries. The combination of Penalty and Plateau is used to extract the best results of both of the methods. Therefore in this way the resulted *AG* has better quality than the one provided by any individual method. In Tables 11, 12, and 13, we also report on the *TargFun* quality indicator of the study in [5]. The experiments in that study were run only on the LU and WE networks, and on data provided by PTV, which concerned smaller (in size) networks and which may be somehow different from those we use here [1]. Nevertheless, we put the *TargFun* values in [5] as a kind of reference for comparison.

map	TargFun		TotDist	AvgDist	DecEdges	Time
	(max:11)	in [5]	(max:11)	(min:1)	(max:10)	(ms)
B	3.82	-	3.91	1.09	9.95	45.61
LU	4.44	3.05	4.49	1.05	9.73	37.05
BE	4.83	-	4.87	1.04	10.00	85.08
IT	4.10	-	4.14	1.04	9.92	114.29
GB	4.36	-	4.40	1.04	9.93	180.12
FR	4.22	-	4.26	1.04	9.97	159.93
GE	4.88	-	4.92	1.04	10.00	286.40
WE	4.35	3.08	4.37	1.02	9.88	717.57

Table 11: The average quality of the resulted *AG* via Plateau method.

[3] We have been very recently informed [9] that this is the same target function as the one used in [5] and not the erroneously stated $totalDistance - averageDistance$ in that paper.

map	TargFun		TotDist	AvgDist	DecEdges	Time
	(max:11)	in [5]	(max:11)	(min:1)	(max:10)	(ms)
B	4.16	-	4.23	1.07	9.92	49.34
LU	5.14	2.91	5.19	1.05	9.23	41.56
BE	5.29	-	5.33	1.04	9.54	159.71
IT	4.11	-	4.14	1.03	9.47	105.84
GB	4.38	-	4.41	1.03	9.87	210.94
FR	4.11	-	4.16	1.05	9.32	192.44
GE	5.42	-	5.46	1.04	9.91	388.97
WE	5.21	3.34	5.24	1.03	9.67	776.97

Table 12: The average quality of the resulted AG via Penalty method.

map	TargFun		TotDist	AvgDist	DecEdges	Time
	(max:11)	in [5]	(max:11)	(min:1)	(max:10)	(ms)
B	4.55	-	4.61	1.06	9.97	54.12
LU	5.25	3.29	5.30	1.05	9.81	43.69
BE	5.36	-	5.41	1.05	9.89	163.75
IT	4.37	-	4.41	1.04	9.79	178.08
GB	4.67	-	4.71	1.04	9.86	284.38
FR	4.56	-	4.60	1.04	9.86	217.30
GE	5.50	-	5.54	1.04	9.89	446.38
WE	5.49	3.70	5.52	1.03	9.94	987.42

Table 13: The average quality of the resulted AG via the combined Penalty and Plateau method.

We would like to note that if we allow a larger value of τ (up to 1.2) for large networks (e.g., WE) and for s-t distances larger than 300km, then we can achieve higher quality indicators (intuitively, this happens due to the many more alternatives in such a case). Indicative values of quality indicators for WE are reported in Table 14, 15.

map WE	TargFun	TotDist	AvgDist	DecEdges	Time(ms)
Plateau	4.57	4.59	1.02	10.00	1564.28
Penalty	4.36	4.38	1.02	9.95	2588.31
Plateau & Penalty	6.29	6.31	1.02	9.97	2692.56

Table 14: Random alternative route queries in the road network of Western Europe, with geographical distance up to 400km.

map WE	TargFun	TotDist	AvgDist	DecEdges	Time(ms)
Plateau	4.71	4.73	1.02	10.00	2171.13
Penalty	4.78	4.80	1.02	9.97	3536.76
Plateau & Penalty	6.46	6.48	1.02	9.98	3806.92

Table 15: Alternative route queries in the road network of Western Europe, with geographical distance up to 500km.

5.5 Visualization of Alternative Graphs

In Figures 19, 20, 21 and 22, we demonstrate some of the visualized results [4] we got with our alternative route planning implementation.

Figure 19: Improved Penalty method. Shape of AG in Italy.

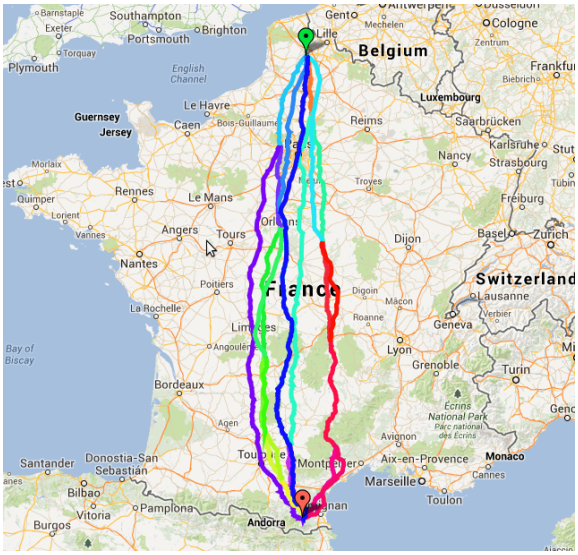


Figure 20: Improved combination of Penalty and Plateau methods. Shape of AG in France.

[4] The images produced by Google Maps © mapping service.

Figure 21: Improved Plateau method. Shape of AG in Spain.

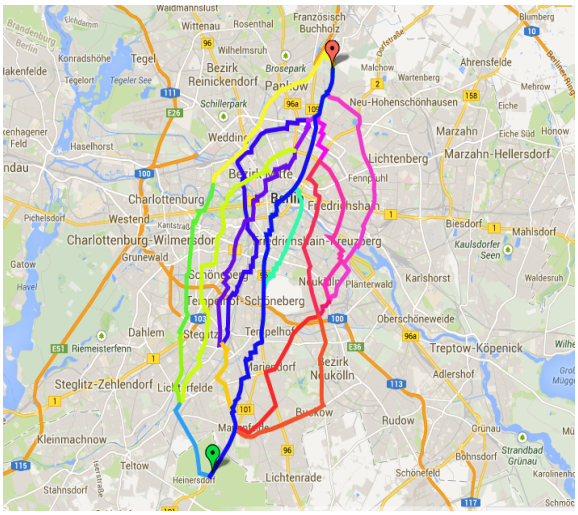


Figure 22: Improved combination of Penalty and Plateau methods. Shape of AG in Berlin.

6 Robust Route Planning

In real-world route planning in road networks, traffic situation and road congestion have an impact on the decision of which route to follow when traveling from one location to another. Unfortunately, if the travel times on the roads (the edge costs of the underlying graph) change with time in an unpredictable way because of traffic or congestion, we cannot hope to safely identify the route that will be the fastest for any future moment. A possible approach to handle these uncertain situations is to compute a *robust route* instead of a fastest route.

Loosely speaking, we say that a route is robust for a future time moment if its cost is "close" to the cost of the actual fastest route in that moment. There are various ways to formally define what "close" in this context means; we refer the reader to the eCOMPASS deliverable series [12] for more details on the topic.

Similarity-based approach. Within the project, we follow the similarity-based method proposed by Buhmann *et al.* [6]. Adapted to the time-dependent scenario, the method works as follows. We are given a simple, i.e., without parallel edges, graph $G = (V, E)$ with time-dependent edge cost functions $c_e : T \to \mathbb{Q}^+$ for every $e \in E$, and two vertices $s, t \in V$. In this context, the set T represents absolute time in the past, i.e., every $d \in T$ corresponds to a unique well-defined moment in the past[5]. We assume $c_e(d)$ to be a reliable measurement of the actual travel time on road e when departing at time d. The *travel time* of a path $p = (v_1, \ldots, v_l)$ departing v_1 at time d is defined as

$$c(p, d) = \sum_{i=1}^{l-1} c_{(v_i, v_{i+1})}(d_i), \tag{4}$$

where d_i is the departure time from the vertex v_i of p. Note that we assume that waiting at vertices is not allowed, therefore, for every vertex $v_2, \ldots, v_{l-1}$, the departure time is equal to the arrival time. This can be safely assumed if the time-dependent edge costs satisfy the *FIFO property* [11], stating that departing from a vertex at a later moment will never result in an earlier arrival time. For a given $d \in T$, a *fastest route* from s to t, i.e., a path minimizing the travel time when leaving s at time d, can be computed using standard Dijkstra's-like techniques [11].

Our goal is to find an st-path that is robust with respect to a future departure time from s, for which we do not have a measurement of the travel times on the roads. Ideally, this path should be *simple*, i.e., without cycles. To solve this problem, we consider two departure times $d_1, d_2 \in T$ in the past that are "related" to the future time moment; in the following sections we will make more precise what "related" means. For a given value $\rho \in [1, \infty)$, we define the *ρ-similarity* between d_1 and d_2 to be the ratio

$$\frac{|A_\rho(d_1) \cap A_\rho(d_2)|}{|A_\rho(d_1)| \cdot |A_\rho(d_2)|}, \tag{5}$$

where $A_\rho(d)$ is the set of all simple st-paths whose travel time when departing s at time d is at most ρ times the travel time of a fastest st-path departing s at time d. To compute a robust route, the method requires to find a value $\rho^* \in [1, \infty)$ maximizing (5), and subsequently pick an st-path uniformly at random from the set $A_{\rho^*}(d_1) \cap A_{\rho^*}(d_2)$. We refer to the eCOMPASS deliverable series [12] for a more formal explanation of the method and its properties.

Issues. One of the main drawbacks of computing robust routes using the above method is the requirement of knowing the size of the approximation sets $A_\rho(d_1)$ and $A_\rho(d_2)$. This is a #P-hard problem [23] already for the non time-dependent scenario and, at the best of our knowledge, no exact, approximate, or randomized algorithms solving it in less than exponential time are known.

[5] Of course, this representation is impossible in practice, therefore in our implementation we will restrict T to a suitably large, but closed and finite, time interval (for example, Unix Time).

If the counting is extended to simple and non-simple st-paths in G, there exists an exact algorithm [12] with pseudo-polynomial running time.

Given the above difficulties, we decided to split the experimental evaluation of robust routes in two parts. In Section 6.1, we do not focus on the computational issues of computing robust routes, and we design experiments aiming only at assessing the quality of the computed route. In particular, we compute and assess the quality of the routes produced by the above method on real data using an exponential time algorithm that finds ρ^* by enumerating all st-paths in G.

In Section 6.2, we evaluate the pseudo-polynomial time algorithm on the same data to investigate whether in practice it runs faster than enumeration in exponential time. We also design and evaluate several heuristics that improve the practical running time of both algorithms.

Computation environment. The experiments were performed on the high-performance cluster of ETH Zurich, Brutus [24]. Each experiment was run on a single core of a computation node of the cluster. The results shown in the following refer to computation nodes with AMD Opteron 8380 processors clocked at 2.5 GHz and 32 GB main memory. The code was written in C++ and compiled using GNU C++ compiler 4.4.7 with default compiler optimization switch (no optimization).

6.1 Computation and Assessment of Robust Routes

Computing a robust route using the similarity-based method explained above requires to be able to compute the values $|A_\rho(d_1) \cap A_\rho(d_2)|, |A_\rho(d_1)|$ and $|A_\rho(d_2)|$ for given $d_1, d_2 \in T$ and $\rho \in [1, \infty)$. Furthermore, we must develop an algorithm that finds a value ρ^* maximizing (5). In the following, we first present an exponential-time algorithm that computes $|A_\rho(d)|$ for given $d \in T$ and $\rho \in [1, \infty)$. Then, we show how to generalize the exponential-time algorithm to evaluate $|A_{\rho'}(d)|$ for *any* $\rho' \in [1, \rho]$ for a given $\rho \in [1, \infty)$. Finally, we explain how to generalize the algorithm to evaluate (5) for $d_1, d_2 \in T$ and $\rho \in [1, \infty)$, and how to find ρ^*.

Exponential-time enumeration. In the following, let p^* be a fastest st-path when departing s at time $d \in T$. The enumeration algorithm works as follows: At the beginning, a counter l is set to 0. Then, all st-paths in G are enumerated. Every time a path whose travel time is at most $\rho \cdot c(p^*, d)$ is found, the counter l is increased by 1. Once every path has been processed, the counter l is equal to $|A_\rho(d)|$. To heuristically speed-up the enumeration, we use a branch-and-bound technique.

The branching step is similar to a depth-first search from s; At the beginning, s is marked "active" and every vertex in $V \backslash \{s\}$ is marked "inactive". Then, we consider all the outgoing neighbors from s. For every inactive neighbor v, we mark v as active and we proceed recursively from it; once all neighbors of s have been processed, s is marked inactive and we return. Every step of the branching corresponds to a different path from s to the last vertex marked active, and it can be proven that this procedure generates all simple paths from s. During the execution, every time t is reached, a new st-path has been found, and the counter l is increased by 1 if its travel time is at most $\rho \cdot c(p^*, d)$.

The bounding step requires to know, for every vertex v in G, a lower bound $c_{vt}(d')$ on the travel time of a fastest path from v to t departing v at time $d' := d + c(p, d)$, where p is the path from s to v corresponding to the current step of the branching procedure. Every time a vertex v is reached by the branching step, we check whether $c(p, d) + c_{vt}(d')$ is at most $\rho \cdot c(p^*, d)$. If it is, we continue branching from v. Otherwise, no st-path with travel time smaller than $\rho \cdot c(p^*, d)$ having p as prefix exists, and we avoid branching from v.

The bounding step is more efficient the tighter the bound $c_{vt}(d')$ is, because it allows earlier pruning of non-promising branches. Since each st-path with v as intermediate vertex may reach v at different times d', we cannot in general compute $c_{vt}(d')$ for every possible departure time from v because they might be too many. In our experiments, this issue was solved by developing and applying a heuristic based on the following observation. For a given $\rho \in [1, \infty)$, we know by

definition that every path in $A_\rho(d)$ cannot arrive at t later than $d + \rho \cdot c(p^*, d)$. In our heuristic, for every edge $e \in E$ we find the time instant minimizing c_e in the time window $[d, d + \rho \cdot c(p^*, d)]$; let this time instant be d_e^*. Then, we build a static, i.e., non time-dependent, edge cost function by setting the cost of each edge $e \in E$ to be d_e^*. For every $v \in V$, we set c_{vt} to be constantly equal to the cost of a shortest path from v to t using the static cost function.

The above algorithm computes $|A_\rho(d)|$ for a given $d \in T$ and $\rho \in [1, \infty)$. We can extend the algorithm to compute $|A_{\rho'}(d)|$ for every $\rho' \in [1, \rho]$ for a given ρ by introducing additional counters as follows. Every time a path $p \in A_\rho(d)$ is found, we evaluate the ratio $\hat{\rho} := c(p, d)/c(p^*, d)$. If a counter $l_{\hat{\rho}}$ already exists, we increment it by 1. Otherwise, we create the counter $l_{\hat{\rho}}$ and we set it to 1. Once all paths in $A_\rho(d)$ have been enumerated, to evaluate $A_{\rho'}(d)$ for any $\rho' \in [1, \rho]$ we only need to sum all the counters $l_{\hat{\rho}}$ with $\hat{\rho} \leq \rho'$. Note that, in the worst-case, this may require to create a counter for every path in $A_\rho(d)$. Since these paths may be exponentially many, this would imply an exponential increase in the running time and in the space requirement of the algorithm. To overcome this issue, we can consider only a finite set of values for ρ and create correspondingly many counters. For example, the interval $[1, \rho]$ can be split into k sub-intervals of constant size, for some fixed $k \in \mathbb{N}$, and create k counters $l_1, \ldots, l_k$ initialized to 0. Every time a path p is found, we compute $\hat{\rho} := c(p, d)/c(p^*, d)$ and we increase the counter l_j by 1, where $j \in \{1, \ldots, k\}$ is the first index such that $\hat{\rho} \leq \frac{j}{k} \cdot \rho$.

Evaluating similarity. We now explain how to extend the enumeration algorithm to compute the ρ-similarity for given $d_1, d_2 \in T$ and $\rho \in [1, \infty)$. This generalization can be done trivially by keeping three counters, one for each set $A_\rho(d_1)$, $A_\rho(d_2)$ and $A_\rho(d_1) \cap A_\rho(d_2)$. Every time the branching step finds an st-path p, we evaluate $c(p, d_1)$ and $c(p, d_2)$, and we increase the counters accordingly. The bounding step must be extended to prune the branching from a vertex if the current path cannot reach t with travel time at most ρ times the fastest path for both departure times d_1 and d_2. Once all paths have been enumerated, we can evaluate the ρ-similarity exactly by computing the ratio (5). Using the above technique, this algorithm can be extended to evaluate (5) for every $\rho' \in [1, \rho]$, for given $d_1, d_2 \in T$ and $\rho \in [1, \infty)$.

Finding ρ^*. Computing ρ^* maximizing (5) can be done trivially if an upper bound $\rho_{\max}$ on ρ^* is known in advance. If this is the case, we can run the enumeration algorithm to evaluate the similarity for every $\rho' \in [1, \rho_{\max}]$ and pick a ρ' maximizing (5). A suitable upper bound on ρ^* can be found experimentally. From our experiments, it turns out that for the data available it is sufficient to set $\rho_{\max} := 1.1$.

6.1.1 Experiments

In the data provided by TomTom for the project, the travel times on roads are described in two different ways. In the *speed profiles*, travel times on roads are assumed to be periodic with a period of one week. For a given road, the associated speed profile describes, for every 5 minute window of each day of a week, an average travel time on that road. The average is computed over live measurements recorded for a period of 2 years. Additionally to the speed profiles, we are given some of the recorded live measurements, denoted as *speed probes*. A speed probe is associated to a road $e \in E$ and consists of a timestamp t and a travel time s. The meaning of a probe is that the travel time on road e has been measured at time t and found out to be equal to s. Clearly, speed probes provide more accurate measurements than speed profiles for the time at which they are associated. However, speed probes are not available for every road at any time, therefore we need to resort to use speed profiles in case a probe is not available. The probes we are given cover a period of two weeks, from March 18th 2012 to March 31st 2012. For every 30 minutes window of this two weeks period, an average of 227834 probes is available. Since the whole road network contains 1038288 edges, and some probes refer to the same road at different moments, in a 30 minutes window the travel times of less than 22% of the edges of the road network is measured.

Departure times. To assess the quality of robust routes, we need to pick suitable departure times d_1 and d_2, and a start and a target vertex s and t, respectively. Deciding the departure times is a most critical issue because, according to the method of Buhmann *et al.*, the more "related" the departure times are, the more robust the resulting route will be. The term "related" is not formally defined, and the only way we know to decide how to pick the departure times is by an ad-hoc decision tuned to the data available.

Since robust routes are most needed in periods of high congestion of the road network, a natural choice is to pick departure times within rush hours periods. Ideally, the best choice for d_1 and d_2 would be two departure times having the highest correlation possible. For example, two consecutive Tuesdays at 17:00. We could then check how robust the computed route is with respect to a third Tuesday at 17:00 of the following week. However, given that the available probes cover only a two weeks period, we cannot set d_1 and d_2 in this way. In the experiments, we set d_1 to be Tuesday March 20th 2012 at 17:00, and d_2 to be Wednesday March 21st 2012 at 17:00. To assess the quality of a route, we evaluated its travel time for a departure time d_3 on a later day, to see how well it predicts the behavior of traffic for that moment. As departure time d_3 we picked another rush-hour period. In particular, we set d_3 to Thursday March 22nd 2012 at 17:00.

Suitable start and target vertices could be picked uniformly at random over the set of vertices V. However, the provided road network is too big to allow enumeration of st-paths for any pair of start and target vertices (it contains 478989 vertices and 1038288 edges). For this reason, we considered a subgraph of the whole network and picked s and t uniformly at random in the subgraph. As an additional benefit, this allows assessing the quality of robust routes with respect to different portions of the network. For example, comparing the quality of routes computed in the city center, and in the suburbs of the city. The subgraph for the center of Berlin contains 4856 vertices and 12298 edges, while the subgraph for the suburbs of Berlin contains 4761 vertices and 11638 edges.

Assessment. The assessment of a route is done with respect to the fastest route between s and t for departure time d_3. Given a route, we check how slower it is (in percentage) with respect to the actual fastest st-path for d_3; we refer to this value as *quality of the prediction*. Our assessment is done in comparison with a different method for computing robust routes, denoted "AVG".

The AVG method is an extension of time-dependent Dijkstra's algorithm receiving as input two departure times instead of one. Every time a road $e \in E$ is evaluated for departures $d, d' \in T$, the average travel time $(c_e(d) + c_e(d'))/2$ is returned. Running the AVG method between s and t with departure times d_1 and d_2 results in a route whose quality as a prediction can be evaluated for d_3.

We performed two types of experiments: In the first type, speed probes are not considered when computing travel times. In the second type, we replace the values given by speed profiles with the values provided by probes when available. Our aim is to show that the quality of routes computed using the similarity-based method increases if more reliable measurements of travel times are available. To obtain more meaningful results, each probe is considered to be valid for a 5 minutes window starting at the time indicated by its timestamp. In case of overlapping probes, we consider only the closest one in time when evaluating the travel time of a road.

Table 16 shows the average quality of the predictions of the routes obtained using the two methods above, and the corresponding variance. The first column shows the results when using no probes, while the second column shows the results using probes. The results shown are further categorized according to the subgraph considered in the computation.

It can be seen from Table 16 that the average quality as a prediction of the routes obtained using AVG is better than using the similarity-based method (labeled SIM in the table) with and without probes. However, introducing probes the variance of SIM is lower than the variance of AVG. This implies that the solutions returned by SIM can be considered less subject to variations than those returned by AVG, and in this sense they may be considered more stable. Furthermore, the average quality of the prediction returned by AVG worsen when probes are used, while the quality of the prediction of SIM improves. This is especially evident if we consider the routes computed on the

		Without probes		5min probes	
		quality	variance	quality	variance
AVG	city center	0.396%	0.790%	0.548%	2.016%
	suburbs	0.354%	0.335%	0.615%	8.776%
SIM	city center	1.878%	26.87%	0.754%	0.951%
	suburbs	0.940%	3.315%	0.746%	0.920%

Table 16: Average quality of prediction and variance

subgraph of the road network located in the suburbs.

6.2 Evaluation of the Label Propagating Algorithm

The previous experiments aim to assess the quality of the robust routes computed using the similarity-based method of Buhmann *et al.* [6]. These experiments were performed using an exponential-time algorithm enumerating all *st*-paths of the input road network. A natural question is to ask for an efficient algorithm for computing robust routes.

Such an algorithm should be able to evaluate, exactly or approximately, the ratio (5) efficiently, therefore estimating the values $|A_\rho(d_1) \cap A_\rho(d_2)|, |A_\rho(d_1)|$ and $|A_\rho(d_2)|$ for given departure times d_1, d_2, and $\rho \in [1, \infty)$. Since the problem of counting all paths with cost at most ρ times the cost of a fastest path is #P-hard [23], we cannot hope to solve it efficiently. Furthermore, at the best of our knowledge, no exact, approximate, or randomized algorithms solving it in less than exponential time are known. As shown in the eCOMPASS deliverable series [12], a partial answer to this problem can be found if we allow the running time to be pseudo-polynomial and we extend the counting to both simple and non-simple paths, i.e., containing cycles. The purpose of this section is to evaluate this pseudo-polynomial algorithm by experimentally comparing its running time and number of paths reported against the running time and number of paths reported by the enumeration algorithm. We also introduce and evaluate heuristics improving the running time of the pseudo-polynomial time algorithm while introducing an error on the number of paths reported.

Label-propagating algorithm. In the following, we recall briefly the pseudo-polynomial time algorithm [12] adapted for time-dependent edge cost functions; given its nature, this algorithm is sometimes referred to as a label-propagating algorithm.

The algorithm receives as input the graph $G = (V, E)$ with edge cost functions $c_e : T \to \mathbb{Q}^+$ for every $e \in E$, two vertices $s, t \in V$, a value $\rho \in [1, \infty)$, and a departure time d. It maintains a counter n_{st} and a set of labels. A label (c_v, v, n_v) represents a lower bound n_v on the number of (simple and non-simple) paths from s to v with cost c_v. At the beginning, the counter n_{st} is set to 0 and the set of labels contains only the initialization value $(0, s, 1)$. At each step, the algorithm extract from the set the smallest label (c_u, u, n_u) in lexicographical order. It can be shown that, at this time, the value n_u is exactly the number of su-paths with cost c_u. If $u = t$, the counter n_{st} is increased by n_u. Otherwise, the algorithm considers every outgoing edge from u. For each edge $e = (u, v)$, its cost is evaluated at the time instant $d + c_u$. If $c_u + c_e(d + c_u)$ is at most ρ times the cost of a fastest route from s to t in G with departure time d, the label (c_v, v, n_v) is created, with $c_v := c_u$ and $n_v := n_u$. If the set of labels already contains a label starting with c_v and v, its third value is updated increasing it by n_v. Otherwise, the label (c_v, v, n_v) is added to the set. As a speed-up heuristic, if a lower bound on the cost to reach t from v when departing at time c_v is known in advance, like the values c_{vt} introduced for the branch-and-bound algorithm, we can avoid processing v. The algorithm ends when the set of labels becomes empty. Once the algorithm ends, n_{st} contains the number of simple and non-simple paths from s to t with cost at most ρ times the cost of a fastest st-path for departure time d. For a more detailed explanation, as well as the analysis of the running time and a discussion on implementation issues, we refer the reader to the

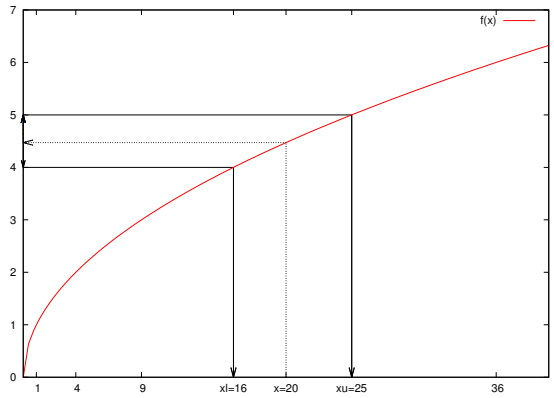


Figure 23: Rounding with h_{sqrt}

eCOMPASS deliverable series [12].

Speed-up heuristics. The running time of the above algorithm is proportional to the overall number of labels contained in the set of labels during the execution. Since each label (c_v, v, n_v) represents a number n_v of paths with the same travel time c_v, the more accurate the measurement of the travel time is, the more labels will be created during the execution of the algorithm. For example, if the travel times are measured in seconds, the overall number of labels will be smaller than the case where travel times are measured in milliseconds. Clearly, less precise measurements introduce an experimental error because we may count, or not count, paths that should not, or should, be counted.

A possible direction for the designing of heuristics is therefore the accuracy used when measuring travel times. In the following, let $x \in \mathbb{N}$ be the exact travel time required to cross the edge $e \in E$ at time $d \in T$, i.e., $x := c_e(d)$. To avoid using floating point arithmetic, we assume x to be an integer representing the travel time in milliseconds.

We define a *cost heuristic* to be a function $h : \mathbb{N} \to \mathbb{N}$ for rounding the travel time x. During the computation, every time the travel time $c_e(d)$ is evaluated, the value $h(x)$ is returned instead of x. To compute the cost heuristic, let $f : \mathbb{R} \to \mathbb{R}$ be an increasing invertible function. Given the travel time $x \in \mathbb{N}$, we compute $x_L = f^{-1}(\lfloor f(x) \rfloor)$ and $x_U = f^{-1}(\lceil f(x) \rceil)$. If $x_L \neq 0$, the value returned by h is the one among x_L and x_U that is closest to x; ties are resolved in favor of x_U. If $x_L = 0$, since we assume the travel times to always be > 0, the heuristics always return x_U. Figure 23 illustrates an example of a cost heuristic with $x = 20$ and $f(x) = \sqrt{x}$.

We will show experimentally that different choices for h lead to different numbers of paths reported by the pseudo-polynomial algorithm, and different running times. Intuitively, the more steep the function f is, the more "aggressive" the rounding of h is, meaning that the number of uniquely different values returned by h will be smaller. The following choices were considered as cost heuristics:

$f(x) = x$ Travel times are not rounded. We refer to this heuristic as h_{ms};

$f(x) = \frac{x}{1000}$ Travel times are rounded to seconds. We refer to this heuristic as h_{sec};

$f(x) = \log_e x$ We refer to this heuristic as $h_{\log}$;

$f(x) = \sqrt{x}$ We refer to this heuristic as h_{sqrt}.

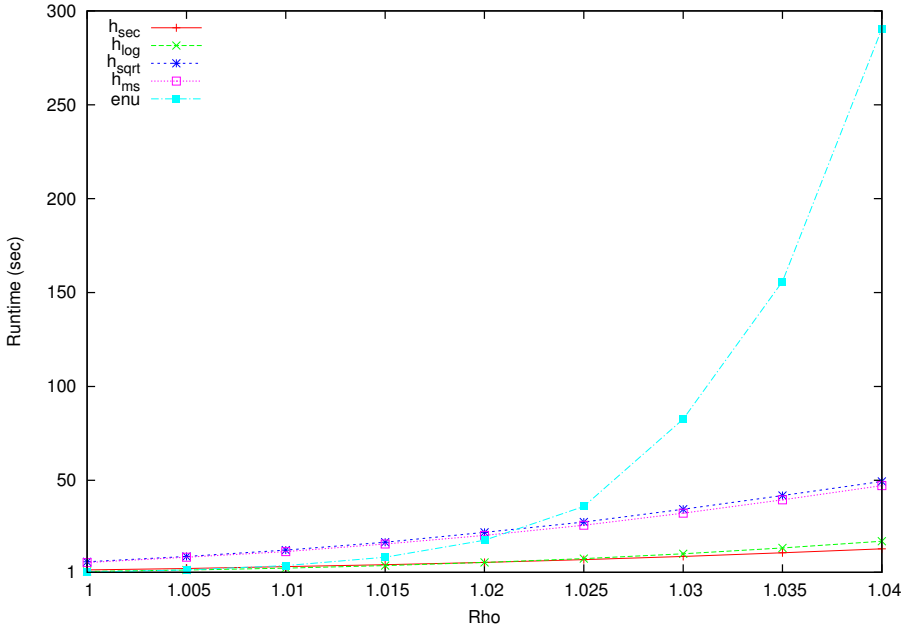


Figure 24: Average runtime of heuristics and enumeration.

6.2.1 Experiments

In the following, we show the results of several experiments aiming to assess the trade-off between the running time of the label propagating algorithm and the number of st-paths reported with different cost heuristics.

Each experiment is designed as follows. At the beginning we set the departure time d to Tuesday March 20th 2012 at 17:00, and we pick a pair of vertices $s, t \in V$ at random. Due to the huge size of the data provided, we restrict the computation to a subgraph of the whole road network containing 4856 vertices and 12298 edges. After s and t have been picked, the label propagating algorithm is run 12 times. At run $i = 1, \ldots, 12$, the algorithm counts the number of st-paths with travel time at most $\rho_i := 1 + (i-1) \cdot 0.005$ times the travel time of the fastest st-path when departing s at time d. The same experiment is repeated for each of the above cost heuristics. Then, the enumeration algorithm of Section 6.1 is run for the same values of ρ_i, s, t and d.

Figure 24 shows the plot of the average running time of the label propagating algorithm using different cost heuristics compared to the average running time of the enumeration algorithm. It can be seen from Figure 24 that counting through enumeration is usually faster than any other solution for smaller values of ρ. However, as ρ increases, the enumeration algorithm becomes slower, and eventually each cost heuristic results in a faster computation time than plain enumeration.

Figure 25 shows, for each cost heuristic, the error introduced by the heuristic on the number of paths reported. Each value is normalized by the number of simple paths as reported by the enumeration algorithm, and the y-axis grows logarithmically. The number of paths reported is, for each heuristic, much higher than the number of simple paths. In particular, even without rounding the travel times (the line labeled h_{ms} in the plot) the number of paths reported grows exponentially with ρ. In other words, the number of non-simple paths is in practice exponentially greater than the number of simple paths. Figure 25 also shows that the most aggressive cost heuristic ($h_{\log}$) introduces the highest error in the number of paths reported.

Surprisingly, a very simple heuristic like h_{sec} seems to have a sort of self-correcting mechanism

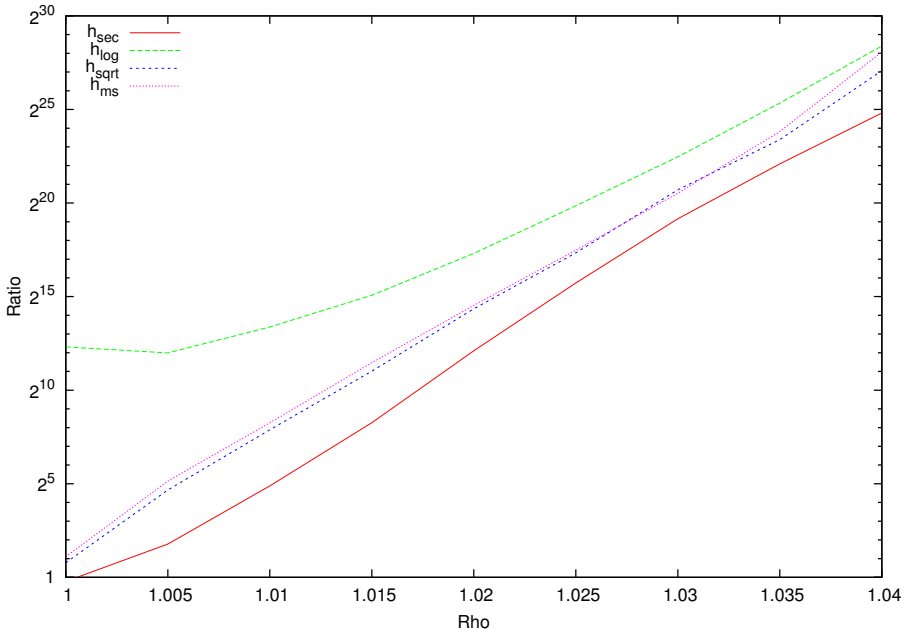


Figure 25: Average ratio of paths reported with respect to enumeration

reducing the number of paths reported with respect to h_{ms}. Since h_{sec} is also the heuristic resulting in the fastest running time for higher values of ρ, it seems promising in the future to inspect if alternative choices of linear scaling yield even better results.

6.3 Conclusion and Discussion

We have presented an experimental study on the computation of robust routes for the eCOMPASS project. We have shown that the routes obtained using the method by *Buhmann et al.* do not provide the best results in terms of expected quality of the prediction, but they tend to have smaller variance and can therefore be considered more stable. Furthermore, we have shown that increasing the number of live measurements on road, the quality of the computed routes seems to improve.

While running these experiments, we also observed that in more than 99% of the experiments, the ρ maximizing ratio (5) is also the first value for which the set $A_\rho(d_1) \cap A_\rho(d_2)$ is not empty. In the future, we plan to develop fine-tuned algorithms for computing robust routes exploiting this relation for decreasing the running time.

Computing robust routes with the above method is in general computationally intensive, and cannot be done in practice on a real-time basis on a limited device like a navigator. For this reason, we developed an algorithm counting simple and non-simple *st*-paths in a given graph. We have shown that this method is in practice faster than plain enumeration but the number of non-simple paths can be exponentially greater than the number of simple paths. In the future, we plan to refine this algorithm in order to avoid cycles of small size, and improve the proposed speed-up heuristics. Furthermore, we want to apply the fast but imprecise label-propagating algorithm for the computation of robust routes.

7 Route Planning for Vehicle Fleets

7.1 Vehicle Routing Problem Data

In this Section, we examine the differences between real world data and synthetic data. The approach followed by eCOMPASS was to first test our algorithm with synthetic data to verify that it behaves as expected. The second step was to test the algorithm with real life datasets, that were provided by PTV.

7.2 Laboratory test data compared to real life data

In both cases, laboratory and real world, the data model is the same but the scope addresses two different worlds. The laboratory test data is used most often to test the function of modules and the performance of the solution. Real world data in contrast to laboratory data has to deal in many cases with complex data sets which describe specific problems. Even the problem analysis stage is not trivial in practice. In practical scenarios, the problems are not clear or well defined. A precise classification is not so easy, as many real problems include characteristics and features of more than just one model. Thus it is a real challenge to provide solution procedures that match real world practitioner needs and expectations.

7.3 Richness of real world problems in VRP

In an operative setting, real world VRP problems do not come with an unlimited homogeneous fleet. Instead we have to deal most often with different types of vehicles and limited availabilities. It goes without saying that this probably imposes new constraints and new aspects on the original problem.

For an acceptance in practical settings it is very important to have a reasonable network model that allows the calculation of reliable distances and driving times. There may exist different routes for different types of vehicles, for different loads and cargo or for different times of the day.

Further aspects of richness are the presence of multiple customer time windows with different kinds of service. In real world problems we distinguish between start of service intervals and full service intervals. The correct handling of working hours regulations increases the degree of complexity considerably.

Often real world problems do not focus on one problem but deal with multiple objectives, such as service level, social criteria, robustness, ecological criteria and visual attractiveness.

7.4 Operative setting of real world problems

In an operative setting, planning is a process. Dispatcher works systematically towards certain objectives. In addition to the algorithm that supports planning, he performs manual operations: Insertions, relocations of customers, assigning a certain vehicle to a tour or vice versa assigning tours to a vehicle.

In operations, he has to deal with modifications or cancellations of orders. Tours may have different states, e.g. special states can limit the degrees of freedom for modifications; e.g. if the loading for a tour has already started it might be desired that this tour shall keep its vehicle. Thus data does not remain static but behaves dynamically.

In professional settings typically the planning is carried out in a multi-user mode. Multiple planners are involved with dedicated tasks and rights. The planner can relax constraints to allow the actions. Of course the planner can overrule each decision of the system. Furthermore, an appropriate IT-infrastructure is required to match all the requirements. The logic model must ensure that it is possible to partition and share the planning data correctly, according to defined rules and concepts. The IT-infrastructure has to physically support and implement the logic model,

e.g. it has to be decided whether the model shall support concurrent-competitive or cooperative work modes.

7.5 Synthetic Laboratory Test Data

Regarding laboratory test data the question to ask is what is needed to perform a meaningful test. Often only a function or a working hypothesis has to be proven. In this case simplistic data without high complexity may be sufficient to perform the verification. Of course some functionality tests, especially regarding performance, may require more complex test data to achieve a meaningful result.

Besides data availability, the main reason to use laboratory data is the possibility to generate data which fulfils all test requirements without introducing additional complexity. In essence laboratory can manipulated to reflect real world problems. For this manipulation, synthetic test data adapts data from real world problems and applies the restrictions and constraints to the synthetic data set.

A further advantage of laboratory tests is, to verify algorithmic functions in a controlled environment without uncontrollable influences.

7.6 eCOMPASS Approach Regarding Fleets of Vehicles

One of the most challenging tasks for eCOMPASS regarding fleets of vehicles, is to develop an algorithm that takes into account the ecological impact of the tours of fleets of vehicles. For this reason, a new three phase approach was developed that tries to group customers together in order to be served by a vehicle. The main idea of the eCOMPASS approach is the following:

- Phase I tries to group together customers regarding their time windows. A graph $G = (V, E)$ is constructed where each customer is represented by a vertex $u \in V$, and there is an edge $e_{u,v}$ connecting nodes (customers) u, v if their time windows are compatible. This means that if a vehicle serves customer u it can also serve customer v without violating any time constraints. At the end of Phase I customers are grouped together into clusters.

- Phase II tries to group together customers taking into account their geographical location. A geographic partition is performed and customers are grouped into cells. All customers that belong to a cell C are close together regarding the real distance among them. At the end of Phase II customers are grouped together in cells.

- Phase III is a refinement phase. It tries to split or merge clusters and cells created from the previous phases. The main idea is that if some customers that are close (regarding real distance) and have compatible time windows are merged together into a final group. On the other hand, if a cell contains customers that are close but have incompatible time windows this cell must be split into two groups.

The ecological aspect is taken into account implicitly. The final clusters that are created have the property that all their customers are close together and have compatible time windows. Thus, a vehicle can serve them without wasting time going back and forth to the depot or travelling with low load. More details regarding eCOMPASS approach can be found in D2.2.

7.7 Experimental Study and Data Sets

The main benefit of the eCOMPASS approach of balanced and compact trips is to provide trip structures are stable during the execution phase in case of unplanned events (e.g., unplanned multiple stops, additional stops). As the available existing solutions do not cover this target in a meaningful way, a direct comparison between the eCOMPASS approach against existing optimized (for a

set of different criteria) VRP solutions is not the focus of our experimental study. Consequently, the experimental study focuses on two aspects: 1) to prove the functionality of the eCOMPASS algorithm at a generic level 2) to achieve an understanding of the tradeoff between compact and balanced eCOMPASS solutions in comparison to baseline solutions of existing state of the art VRP approaches. The experiments therefore compares eCOMPASS solutions of the Munich data sets against baseline instances of PTV which focused on different optimization aspects.

To achieve the above goals, we conducted our experimental study on three real-world data sets provided by PTV. The first, is a data set in the city of Milan (Italy). The other 2 data sets include customers located in the city of Munich. Specifically, one regards a parcel delivery and the other a furniture delivery. All Munich data sets are in urban areas. All data sets provide the following information: total number of customers, a unique customer id, a location of each customer (longitude,latitude), one (or more) time window(s) of each customer, the weight of each customer (a number representing the amount of goods that have to be delivered) and a distance matrix with the real distance among all customers.

The quality measures that are reported are: total driving distance (in km), number of vehicles used for each scenario, number of tours and number of tour stops. For the Milan dataset, a comparison was made between the routes computed with the real distance against the routes computed with the Euclidean distance.

7.7.1 Milan Dataset

The Milan dataset consists of 1000 customers and is the largest dataset on which we conducted experiments. Due to lack of quality measures of other approaches we did not perform a comparison with the eCOMPASS approach. However, we report on the ratio between the total distance travelled using the real distance and the total distance travelled using the Euclidean distance. Our experiments showed that this ratio is 1.75, a number that is acceptable because in urban areas the distance between two points is typically Manhattan, i.e. at least greater than 1.41 times bigger than the Euclidean distance.

7.7.2 Munich Dataset - Parcel Delivery

The tour planning results for the parcel courier express service providers are listed in Table 17. Without traffic information a total tour length of 163.32 km for serving 32 customer orders was calculated. For the process of delivery one vehicle is needed for generated tour. In Figure 26, all 32 customers are shown on the map. Customers are grouped together creating clusters.

	Previous Approach	eCOMPASS Approach
Total km driven	163.32	114.01
Total driving time	4h 12 min	4h 32min
CO_2 emissions	62.45kg	41.33kg
Total vehicles used	1	1
Number of tours	1	1
Tour stops	34	34

Table 17: Munich Dataset: Performance indicators for the parcel delivery scenario. The vehicle type chosen for CO_2 emissions calculation was truck (7,5t).

In Table 17, the eCOMPASS approach achieves a further improvement in total kilometres driven. The generated tour takes 48 minutes longer and part of the tour uses the motorway.

Figure 26: Munich Dataset: Groups created for the parcel delivery scenario. Each customer is represented by a marker. In this case, all customers form one group and are served by one vehicle.

7.7.3 Munich Dataset - Furniture Delivery

The second scenario to be considered is the delivery of furniture, in particular kitchen furniture from a furniture store to various customers in the city centre of Munich. The furniture store with its warehouse is located outside of Munich in the district of Taufkirchen. For this scenario it is assumed that the furniture can be ordered directly in the furniture store by the customer and every piece of furniture is available from the stock. Thus, a suitably short period of time between the point of order and delivery will be accepted. For simplicity the furniture store's warehouse is operating all the time.

After the customers chose the pieces of furniture they wish to receive, the warehouse processes their orders and the delivery will be planned. As furniture is often bulky, the delivery process of the furniture is modelled as mid-size truck operations. We modelled the distribution process with two trucks and assumed 5 tons payload. Furthermore we assumed service time of 15 minutes for a drop per truck stop for unloading the pieces of furniture. As a consequence, a vehicle is not immediately ready for use again after the point of delivery. After finishing the tours the trucks return to the furniture store/warehouse. The vehicle fleet we modelled consists of two mid-size lorries with 5.000 kg payload and an overall weight of about 12.000 kg per lorry.

For the case of furniture delivery the calculations are based on a data set with 150 entries for a time period of about two weeks. The handled information are real, but made anonymous. For the calculation and tour planning two trucks with 5 tons payload were used with an availability of 24/7. The only restrictions for tour planning are the weight of the transported pieces of furniture and a service time per tour stop of 15 minutes to guarantee the unloading process. For simplicity, the delivery time windows, in which customers can receive their furniture were standardised from 08:00 to 18:00 o'clock and Monday to Friday. As mentioned already in the other scenarios the order specifications on each of the both Mondays are identical making them comparable in the case of traffic information. The database contains 31 orders for each Monday. For the initial tour planning solution the results are shown in Table 18.

Based on the given information without any traffic the following tours for the furniture delivery on Monday were generated. There are three tours operated by two vehicles to serve all 31 customers. The visualization of the furniture delivery scenario is shown in Figure 27.

In Table 18, the eCOMPASS approach achieves a further improvement both for total kilometres driven and total driving time. The tours generated do not use the motorway.

	Previous Approach	eCOMPASS Approach
Total km driven	204.36	103.15
Total driving time	4h 29min	4h 06min
CO_2 emissions	115.46kg	57.61kg
Total vehicles used	2	2
Number of tours	3	3
Tour stops	37	37

Table 18: Munich Dataset: Performance indicators for the furniture delivery scenario. The vehicle type chosen for CO_2 emissions calculation was truck (7,5t).

Figure 27: Munich Dataset: Groups created for the furniture delivery scenario. Each customer is represented by a marker. In this case, customers are divided into 3 groups, served by two vehicles that perform three tours.

8 Conclusion and Future Work

In this document, we presented and discussed the outcome of the testing stage for the routing algorithms developed by the eCOMPASS project partners in the first 18 months of the project. The specific goal for the testing stage was to fine-tune the proposed algorithms for practical use by experimenting on both artificial and real-world data sets located in urban areas. The results obtained from these experiments will be used to decide which of the algorithms are already fit enough for turning them into prototypes in WP5 with the final aim to assess their validity for every-day use. In parallel to WP5, research for even more efficient and precise algorithms will go on.

In the following, we summarize experimental results and future research directions.

8.1 Traffic Prediction

In section 3, the lag-based STARIMA approach, which was developed in WP2, was compared to previously existing approaches on both artificial and real-world data. It turned out that lag-based STARIMA performs quite satisfactory in general and outperforms the other approaches on real-world data because of its particular ability to capture the spatio-temporal nature of the data. Future work includes the improvement of the compared approaches, for example by hybridization, and their evaluation in further settings.

8.2 Time-Dependent Shortest Paths

In section 4.1, the time-dependent approximation algorithms provide the ability of computing approximated shortest delay functions and shortest paths between vertices, for any departure time range. In this matter, the accuracy of the results can be determined by appropriate setting the error approximation ratio ε. A small $\varepsilon \in [0.01, 0.1]$ leads to high precision, with large number of samples, and thus higher execution time. On the other hand, a large $\varepsilon \in [0.1, 0.5]$ may lead to low precision, with small number of samples, and thus lower execution time.

In section 4.2, we reported on a multi-level separator approach that turned out to be indeed promising for dynamic, customized, time-dependent route planning. Customization was tested on different instances. Total customization time is fast enough to incorporate frequent metric updates, ranging from 1 minute to 10 minutes computation time. Our analysis showed that most effort is spent on the highest level, which also introduces the largest amount of additional number of break points (up to 12 times more than the number of break points on original arcs). Future work concerns: (i) evaluation of a more diverse set of test instances, taking into account higher time resolutions; (ii) application of approximation techniques between levels, thus reducing the stored amount of break points on overlay arcs; (iii) a careful consideration of the scenario of performing local updates due to live traffic data and traffic prediction.

8.3 Alternative Route Planning

In section 5, the Penalty and Plateau based methods [5] as well as their combination, which was developed in WP2, were extended in several ways. Now a large number of qualitative alternatives can be computed in time less than one second on continental size networks. Future work includes the optimization of these algorithms and the development of even stronger heuristic approaches.

8.4 Robust Route Planning

In section 6, the robust-route planning algorithms developed in WP2 have, like their ancestor, exponential time complexity and are hence not relevant for practical applications. Nevertheless, implementing these algorithms and testing them on real-life data has resulted in valuable insights

and ideas that will guide the future work on these algorithms with the aim to make them faster, for example by trading route quality for runtime.

8.5 Fleet-of-Vehicles Route Planning

In section 7, the eCOMPASS approach for fleets of vehicles achieves results that are comparable to PTV's baseline solutions for the data sets examined. Future work is to optimize the tradeoff between compact and balanced eCOMPASS solutions in comparison with baseline solutions of existing state of the art VRP approaches.

References

[1] Openstreetmap. http://www.openstreetmap.org.

[2] Tomtom. http://www.tomtom.com.

[3] Camvit: Choice routing, 2009. http://www.camvit.com.

[4] eCOMPASS project, 2011-2014. http://www.ecompass-project.eu.

[5] Roland Bader, Jonathan Dees, Robert Geisberger, and Peter Sanders. Alternative route graphs in road networks. In *Theory and Practice of Algorithms in (Computer) Systems*, pages 21–32. Springer, 2011.

[6] Joachim M. Buhmann, Matús Mihalák, Rastislav Srámek, and Peter Widmayer. Robust optimization in the presence of uncertainty. In *ITCS*, pages 505–514, 2013.

[7] Yanyan Chen, Michael GH Bell, and Klaus Bogenberger. Reliable pretrip multipath planning and dynamic adaptation for a centralized road navigation system. *Intelligent Transportation Systems, IEEE Transactions on*, 8(1):14–20, 2007.

[8] Daniel Delling, Andrew V. Goldberg, Ilya Razenshteyn, and Renato F. Werneck. Graph Partitioning with Natural Cuts. In *25th International Parallel and Distributed Processing Symposium (IPDPS'11)*, pages 1135–1146. IEEE Computer Society, 2011.

[9] Daniel Delling and Moritz Kobitzsch. Personal commnication, July 2013.

[10] Daniel Delling and Dorothea Wagner. Time-Dependent Route Planning. In Ravindra K. Ahuja, Rolf H. Möhring, and Christos Zaroliagis, editors, *Robust and Online Large-Scale Optimization*, volume 5868 of *Lecture Notes in Computer Science*, pages 207–230. Springer, 2009.

[11] Daniel Delling and Dorothea Wagner. Time-dependent route planning. In *Robust and Online Large-Scale Optimization*, pages 207–230. 2009.

[12] eCOMPASS. D2.2 – new algorithms for eco-friendly vehicle routing. Technical report, The eCOMPASS Consortium, 2013.

[13] Luca Foschini, John Hershberger, and Subhash Suri. On the complexity of time-dependent shortest paths. In *Proceedings of the Twenty-Second Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 327–341. SIAM, 2011.

[14] Andrew V Goldberg and Chris Harrelson. Computing the shortest path: A* search meets graph theory. In *Proc. 16th ACM-SIAM symposium on Discrete algorithms*, pages 156–165. Society for Industrial and Applied Mathematics, 2005.

[15] Joshua S Greenfeld. Matching GPS observations to locations on a digital map. *In Proc. 81th Annual Meeting of the Transportation Research Board*, pages 164–173, 2002.

[16] Benjamin Hamner. Predicting travel times with context-dependent random forests by modeling local and aggregate traffic flow. In *Proceedings of the 2010 IEEE International Conference on Data Mining Workshops*, ICDMW '10, pages 1357–1359, Washington, DC, USA, 2010. IEEE Computer Society.

[17] Yiannis Kamarianakis and Poulicos Prastacos. Space-time modeling of traffic flow. *Comput. Geosci.*, 31(2):119–133, March 2005.

[18] Felix Koenig. Future challenges in real-life routing. In *Workshop on New Prospects in Car Navigation*. February 2012. TU Berlin.

[19] Spyros Kontogiannis and Christos Zaroliagis. Approximation Algorithms for Time-Dependent Shortest Paths eCOMPASS Project, Technical Report TR-017, April 2013. `http://www.ecompass-project.eu/sites/default/files/ECOMPASS-TR-017_0.pdf`.

[20] Georgia Mali, Panagiotis Michail, Andreas Paraskevopoulos, and Christos Zaroliagis. A new dynamic graph structure for large-scale transportation networks. In *Algorithms and Complexity*, volume 7878 of *LNCS*, pages 312–323. Springer, 2013.

[21] Andreas Paraskevopoulos and Christos Zaroliagis. Improved alternative route planning. In *13th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems*, volume 33 of *OASICS*, pages 108–122, 2013. Also eCOMPASS Project, Technical Report TR-024, July 2013.

[22] Peter Sanders and Christian Schulz. Distributed Evolutionary Graph Partitioning. In *Proceedings of the 14th Meeting on Algorithm Engineering and Experiments (ALENEX'12)*, pages 16–29. SIAM, 2012.

[23] Leslie G. Valiant. The complexity of enumeration and reliability problems. *SIAM J. Comput.*, 8(3):410–421, 1979.

[24] Wikipedia. Brutus cluster. Accessed on 10.06.2013.

[25] Marcin Wojnarski, Pawel Gora, Marcin Szczuka, Hung Son Nguyen, Joanna Swietlicka, and Demetris Zeinalipour. Ieee icdm 2010 contest: Tomtom traffic prediction for intelligent gps navigation. *2012 IEEE 12th International Conference on Data Mining Workshops*, 0:1372–1376, 2010.