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

D2.2 - New Algorithms for eco-friendly vehicle routing

Workpackage:	WP2 - Algorithms for Vehicle Routing
Due date of deliverable:	30 April 2013
Actual submission date:	26 April 2013
Responsible Partner:	ETHZ
Contributing Partners:	CERTH, KIT, CTI, PTV, TomTom

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
0.1	17.04.2013	First Draft	Sandro Montanari, ETHZ
1.0	20.04.2013	Sent to internal reviewers and PBQ	Sandro Montanari, ETHZ
1.1	23.04.2013	Reviewers' and PBQ's comments incorporated (sent to PQB again)	Sandro Montanari, ETHZ
1.2	25.04.2013	Further PQB's comments incorporated	Sandro Montanari, ETHZ
1.3	26.04.2013	Final (approved by PQB, sent to the Project Officer)	Christos Zaroliagis, CTI

Deliverable manager

- Sandro Montanari, ETHZ

List of Contributors

- Sandro Montanari, ETHZ
- Tobias Pröger, ETHZ
- Katerina Bohmova, ETHZ
- Rastislav Sramek, ETHZ
- Christos Zaroliagis, CTI
- Kalliopi Giannakopoulou, CTI
- Andreas Paraskevopoulos, CTI
- Spyros Kontogiannis, CTI
- Dimitrios Gkortsilas, CTI
- Dionisis Kehagias, CERTH
- Themistoklis Diamantopoulos, CERTH
- Julian Dibbelt, KIT

List of Evaluators

- Spyros Kontogiannis, CTI
- Michael Marte, TomTom
- Damianos Gavalas, CTI

Summary
The purpose of this deliverable is to present the research results obtained by the project partners in the first 18 months of the project. The focus is on presenting algorithms and solutions developed for problems concerning routing of private vehicles and fleet of vehicles in urban areas and not on their experimental evaluation. For each problem considered, we briefly present state-of-the-art techniques for dealing with it, and we illustrate the new solutions developed within the scope of the project.

Contents

1 Introduction

This deliverable presents the research results obtained by the project's partners in the first 18 months of the project with respect to eco-aware routing for private vehicles and fleet of vehicles. It describes the algorithmic solutions developed so far for the problems related to WP2.

1.1 Objectives and scope of D2.2

The goal of WP2 is to develop novel algorithmic methods for optimization of problems related to routing of vehicles and fleet of vehicles in urban areas, considering the environmental impact as one of the main parameters of the optimization objective. This document summarizes 18 months of research within this workpackage, and introduces the algorithmic solutions developed so far.

The present deliverable is the outcome of the following tasks:

Task 2.2 Eco-friendly private vehicle routing algorithms.

Task 2.3 Eco-friendly routing algorithms for fleet of vehicles.

Task 2.2 aims at designing routing algorithms for private vehicles. The computed routes should be optimized also with respect to their environmental footprint, and they should take into consideration traffic prediction techniques as well. Furthermore, the trade-off between data precision and solution robustness is also investigated in the context of this task.

Task 2.3 aims at designing routing algorithms for fleets of vehicles. The application scenario for this task is a transportation company wishing to schedule the delivery or collection of goods in the most efficient and environmentally-friendly way as possible.

The algorithms developed for Task 2.2 and Task 2.3 should be designed such that the environmental impact of the computed route is minimal, while aiming at outperforming the state-of-the-art techniques for classical routing problems in terms of quality (i.e., precision) and efficiency. Furthermore, dynamic scenarios should be taken into account, wherein the input is not statically predetermined but depends on several factors, like the time at which a query has been issued, or the current road traffic conditions. In scenarios where deriving optimum solutions in an efficient manner is not feasible, the computation of approximate solutions is taken into account.

1.2 Structure of the Document

The main body of this document are Sections 2 to 6, presenting the algorithms developed within the scope of the project. Section 2 describes traffic prediction techniques. Section 3 describes answering shortest path queries in the dynamic scenario where the edge costs of the network depend on the time of the day at which the query has been asked. Section 4 describes techniques for computing routes that can be proposed to users as meaningful alternatives to the fastest route. Section 5 describes the issues arising in the computation of routes when the data is noisy or not completely reliable, namely, it addresses computation of so-called "robust routes". Section 6 illustrates the eCOMPASS approach for the computation of routes for delivery companies that need to schedule the delivery of goods over fleets of vehicles. Finally, Section 7 concludes this document, and outlines the work plan for the next reporting period.

2 Traffic Prediction

2.1 Introduction

Nowadays, the interest for developing intelligent transportation systems (ITS) is constantly increasing. The optimization of transport, either in terms of individuals or fleets, has emerged as an important problem that may have significant impact socially, economically, and in terms of eco-friendliness. The impact of traffic conditions on road networks has grown to be a major parameter, affecting every aspect of transport, either uni-modal or multi-modal, i.e. using one or more different means of transport, respectively. Although detectors could help alleviate certain traffic problems, traffic is constantly altering, e.g. within a 30-minute time interval. Thus, early in the morning the traffic of an area may rise considerably as a result of people heading towards their work sites.

Consequently, the problem of forecasting traffic within short time intervals ahead of present time has arisen as a crucial, yet, challenging task; its thorough investigation could result in improved routing decisions. Although traffic may be interpreted in various ways, the travel time required to traverse a road is the main metric, since it is comprehensive (concerning public), as well as directly associated with routing methods.

The problem of traffic prediction receives various interpretations, since many parameters such as data sources or the actual implementation area may lead to a totally different research question. The main classification criterion is typically the selection of a *traffic descriptor*. Traffic descriptors quantify the performance of the transportation network, so that algorithms may be used on a particular metric or combinations of metrics. Commonly employed metrics include *traffic flow* (vehicles/hour), *density* (vehicles/km), *occupancy* (percentage), *mean speed* (for a time interval km/h) and *travel time* (time used to traverse a link). The aforementioned traffic descriptors are highly relevant to the source of the input data (e.g. loop detectors provide occupancy as the percentage of time that vehicles traverse a link, whereas GPS-enabled devices provide with instantaneous speed measurements per link). In addition, the proper representation of the prediction algorithm's output is usually a matter of choice, highly dependent on the cause of the prediction.

Travel time per link is one of the most commonly required output measurements as it is considered highly relevant to routing. Input, however, may be provided by different sources in various formats. In traffic prediction-related research work done in the project, we considered that the input is given in form of instantaneous vehicles speeds, as observed using GPS devices (see Section 2.3). For the majority of traffic prediction techniques, handling different descriptors is more or less applicable, as far as feature selection is properly taken into consideration.

Since the problem of traffic prediction is widely studied in the literature, different kinds of techniques have emerged in order to tackle it. Upon referring to certain interesting techniques, the problem is analyzed into two main subproblems, data preprocessing and forecasting. The preprocessing step aims to isolate useful data and remove the noise, while the forecasting step involves the prediction of future travel time values.

This section reports on the results of recent research conducted within Tasks 2.2 and 2.3 in order to investigate the effect of various techniques on the quality of traffic prediction when applied to different datasets. Local (i.e. concerning specific road neighborhoods) and global (i.e. concerning the whole road network) measures are used to preprocess the data so that algorithm effectiveness is maximized. From this perspective, the applicability and effectiveness of various algorithmic approaches with respect to specific data characteristics are further discussed. As a result of the previous research, a new algorithm is proposed, drawn from Time Series Analysis, and its effectiveness is discussed against known literature solutions.

The remainder of this section is organized as follows. Subsection 2.2 summarises the most representative state-of-the-art algorithmic approaches in the literature (a more detailed survey can be found in Deliverable D2.1). Subsection 2.3 briefly describes the datasets used, and focuses on their similarities and differences. In subsection 2.4, local and global measures are presented and discussed, whereas in subsection 2.5, the application of known literature algorithms to the datasets at hand is illustrated and a new approach is discussed. Finally, section 2.6 summarises and draws conclusions on the traffic prediction techniques presented in this Section.

2.2 Models for Travel Time Forecasting

The techniques used for traffic prediction are classified into two main categories, the *parametric methods* and the *non-parametric methods*. The term "parametric" determines whether the model of a particular method is selected in advance or is unknown and selected during the training procedure [72].

2.2.1 Parametric Methods

Parametric methods are based on specific pre-determined models, which are trained in order to deduce the parameters of the model in an optimal manner. The most common methods of this category consist of *Auto-Regressive Integrated Moving Average (ARIMA)* and its variations, of which the theory is given by Time Series Analysis. The generic *Auto-Regressive Moving Average (ARMA)* model comprises of two components [11]:

- The *Auto-Regressive (AR)* part provides the current value X_t as the linear aggregate of p previous values:
$$X_t = \sum_{k=1}^{p} \phi_k X_{t-k} + \epsilon_t \tag{1}$$
where ϵ_t is the error term and follows a Gaussian distribution of type $(0, \sigma_\epsilon^2)$ (i.e., white Noise).
- The *Moving Average (MA)* part provides the current value X_t as the aggregate of q previous error terms:
$$X_t = \sum_{k=1}^{q} \theta_k \epsilon_{t-k} + \epsilon_t \tag{2}$$

Hence, according to (1) and (2), the ARMA(p, q) model is given by:

$$X_t = \sum_{k=1}^{p} \phi_k X_{t-k} + \sum_{k=1}^{q} \theta_k \epsilon_{t-k} + \epsilon_t \tag{3}$$

or equivalently:

$$1 - \sum_{k=1}^{p} (\phi_k B^k) X_t = 1 + \sum_{k=1}^{q} (\theta_k B^k) \epsilon_t \tag{4}$$

where B is the *backwards shift* operator ($B^k X_t = X_{t-k}$). Upon differencing the series at the dth degree ($(1 - B)^d X_t$):

$$1 - \sum_{k=1}^{p} (\phi_k B^k)(1 - B)^d X_t = 1 + \sum_{k=1}^{q} (\theta_k B^k) \epsilon_t \tag{5}$$

Finally, (5) describes an ARIMA(p, d, q) model. The ARIMA model was first introduced by Box and Jenkins [9] to forecast the next values of a time series given its past values.

By contrast with the *univariate* analysis performed for a time series using the ARIMA model, its *multivariate* counterpart, the *Space-Time Auto-Regressive Integrated Moving Average (STARIMA)* takes into account several time series, supposed to be related to each other. STARIMA was first introduced by Pfeifer and Deutsch [68] for studying the spread of diseases, however its applicability to problems with multiple time series is generic enough so that it can be used for predicting travel times. The model is given by:

$$\phi_{p,\lambda}(B)\Phi_{P,\Lambda}(B^S)(1-B)^d(1-B^S)^D X_t = \theta_{Q,M}(B^S)\epsilon_t \tag{6}$$

where:

$$\begin{aligned}\phi_{p,\lambda}(B) &= 1 - \textstyle\sum_{k=1}^{p}\sum_{l=0}^{\lambda^k}\phi_{k,l}W_l B^k\\ \Phi_{P,\Lambda}(B^S) &= 1 - \textstyle\sum_{k=1}^{P}\sum_{l=0}^{\Lambda^k}\Phi_{k,l}W_l B^{kS}\\ \theta_{q,m}(B) &= 1 - \textstyle\sum_{k=1}^{q}\sum_{l=0}^{m^k}\theta_{k,l}W_l B^k\\ \Theta_{Q,M}(B^S) &= 1 - \textstyle\sum_{k=1}^{Q}\sum_{l=0}^{M^k}\Theta_{k,l}W_l B^{kS}\end{aligned}$$

As one might observe, (5) and (6) are quite similar. STARIMA actually introduces new parameters to take into account the spatial and temporal lags of the multiple time series. Hence, k and l denote the temporal and spatial lag respectively, while $\phi_{k,l}$ and $\theta_{k,l}$ are the auto-regressive and moving average non-seasonal parameters. In accordance, capital letters denote the seasonal parameters. Preliminary results for predicting traffic flow using STARIMA seem quite promising [51], while travel time forecasting is applied in this work (see subsection 2.5.3).

Finally, a quite different line of research concerning parametric methods used for predicting traffic is the use of *Kalman Filters* to predict traffic flow [66]. Kalman Filters, which were introduced by (and named after) Kalman [50], are based on updating a state variable upon exploiting every new measurement (i.e. one measurement at a time). Although no further information is provided, since it deviates from the purposes of this deliverable, the interested reader is referred to [50] and [66].

2.2.2 Non-parametric Methods

The non-parametric methods can be categorized to *memory-based* and *model-based* ones. Memory-based methods have to retain historical samples in order to perform the prediction, whereas model-based ones exclusively need the extracted model, thus historical data are discarded upon the training phase.

The most typical example of a memory-based model is the one extracted using the *k-Nearest Neighbor (kNN)* methodology. Predicting the next value of a time series using kNN is simple; a hybrid state vector may be defined as in [72]:

$$x(t) = [V(t), V(t-1), V(t-2), V_h(t), V_h(t+1)] \tag{7}$$

where $V(t)$ is the traffic descriptor value at time interval t and $V_h(t)$ denotes historical values of V. The respective output of the above vector is:

$$y(t) = V(t+1) \tag{8}$$

The algorithm creates input vectors $(x_1, x_2, x_3, \dots)$ using the training set. When a prediction is requested, the k nearest vectors to the current vector (i.e. the vector x_p at present time where y_p is unknown) are found and an averaging technique is used to calculate the corresponding output value (y_p)[1]. Although simplistic, kNN seems to provide with satisfactory results, such as in [71], thus giving a hint that performance lies mainly in properly representing dataset features rather than blindly applying a robust algorithm. Interestingly, both first and second runner-up implementations of the GPS task of the IEEE ICDM contest [79] were based on applying variations of KNN to preprocessed (or postprocessed) data.

As opposed to memory-based methods, model-based ones perform offline training using the training data in order to construct a model. The methods of this category generally lie in the

[1]The averaging may simply be the average, or it may be adjusted e.g. by distance or by current condition. For an extensive review of possible averaging methods, see [72].

area of *Machine Learning (ML)*. Virtually any ML method may be used as a predictor. Typical examples include *Random Forests (RFs)*, *Artificial Neural Networks (ANNs)*, and *Support Vector Machines (SVMs)*.

RFs were introduced by Breiman [10] as an ensemble of decision trees. The training data given to the algorithm is used so that the branches of the trees are created. Thus, any new sample can be easily classified correctly by simply traversing the trees and outputting the value voted by the majority if trees, or the average of all trees, in case of regression. The theory of RFs has been successfully applied to traffic prediction in the IEEE ICDM contest of 2010 (see Section 2.3) by Hamner [41], with an implementation that ranked first in the GPS task. Subsection 2.5.2 illustrates an application of the algorithm, where feature selection is the main task to be performed.

The applicability of ANNs to a wide range of problems, including pattern classification and recognition, is remarkable [45]. ANNs, in particular *Multi-Layered Perceptrons (MLPs)*, have been broadly applied in the area of traffic prediction, both for traffic flow [78] and travel time forecasting [47] mainly due to their flexible structure as well as their *generalization ability*, i.e. forecasting previously unforeseen conditions. Recently, a *Restricted Boltzmann Machine (RBM)*, which is a kind of stochastic ANN, was used as a model for predicting traffic flow, with the respective implementation ranked first in the Traffic task of the IEEE ICDM contest [79].

Finally, SVMs, a classification method introduced by Vapnik [77], is based on a straightforward idea: construct a hyperplane that sets apart the classes of the data. Thus, SVMs have also been used in terms of travel time forecasting (e.g. [80]), mainly exploiting their high accuracy when trained with a satisfactorily large dataset.

2.3 Traffic Data

At the time our research on traffic prediction initiated, no dataset was provided. Thus, the initial dataset used is taken from the 3rd task of the *IEEE ICDM Contest: TomTom Traffic Prediction for Intelligent GPS Navigation* [79] and concerns the city of Warsaw. Later on, a dataset for the city of Berlin was provided by TomTom, a partner company of this European-funded research project. The two different datasets are thereafter named as the *Warsaw dataset* and the *Berlin dataset*, corresponding to data for the two cities. The characteristics of the datasets are further discussed in this subsection.

The *IEEE ICDM Contest: TomTom Traffic Prediction for Intelligent GPS Navigation* [79] that took place in 2010 provided interesting insight concerning the raw form of the data and the noise it may include. Despite relying on simulated data for the city of Warsaw, the scenarios of the contest are quite realistic. In terms of this section, the 3rd task of the contest, i.e. GPS, is studied as a realistic problem of acquiring sparse data from GPS navigators all over a road network, and requiring travel time forecasting for 5 and 30 minutes ahead of present time. The data is given in raw form, as instantaneous vehicle speeds at given coordinates, so the first step of preprocessing concerns matching the GPS-observed data to links on the map. The map-matching procedure used is the one proposed in [40] by J. Greenfeld. In brief, the procedure includes storing a series of consecutive instantaneous speed records for every car in order to create its *trajectory*. The trajectory is used to map the speeds to the "nearest" link. The proximity of the link is measured using the distance of the trajectory points from the start, the middle, and the end of the link. The interested reader is referred to [40] for a more thorough explanation of the map-matching methodology as well as a comparison with other known methods.

The second dataset concerns the city of Berlin and, although drawn by GPS locators, it was already matched to map links, before it was provided to us. As opposed to the Warsaw dataset, Berlin data consist of real historic speeds, hence comparing and contrasting between the results for the two datasets could be quite insightful.

Upon map-matching, both datasets contain instantaneous speeds that correspond to road links, also including the vehicle's direction along the link. Applying a traffic prediction algorithm on the data is restrictive not only due to scalability issues, but also because the data is sparse, e.g. a link may have no data for different time moments. Nodes are defined as intersections of two or more links (i.e. straight lines). Road segments, hereafter called roads, are defined as segments between nodes with more than two links. Hence each road contains arbitrary number of links.

Furthermore, the speeds for each road are not kept as individual records, yet their arithmetic average, their harmonic average, and certain other statistic measures are calculated for specific intervals (5-minute for Berlin and 6-minute for Warsaw) and stored thereafter. This coarse-grained approach ensures not only that data are tolerant when samples are few, but also that algorithm scalability shall be satisfactory.

The algorithms described in the following subsection may use different statistic variables. However, the harmonic average for each specific time interval seems the best approach, since, according to [79], it corresponds better to travel times.

2.4 Data Preprocessing

Applying a multivariate travel time forecasting algorithm involves selecting several road time series corresponding to road speed values and use them to extract future values of the road of interest. A major research question studied by the community (see [16] for a review of different methodologies) lies in determining which series affect most the speed value of that road. Existing solutions include using data from neighboring roads (e.g. as in [51]) and/or using global network data (as in [41]). However, traffic data is usually sparse and full of noise. In addition, local or global road data may or may not affect the forthcoming values for a network's road.

Consider that the trend of some road time series may appear to be proportional (or inversely proportional) with the one of the road to predict, whereas others may exhibit no such relation, or even not have any data. Considering a traffic congestion in roads $n_1, n_2, \dots$, the road r is likely to have similar traffic over the next interval, especially if roads $n_1, n_2, \dots$ are neighboring to road r or if they are major enough to affect it even if they are not local. Such relations are usually captured using correlation metrics.

At first, a simple, yet powerful, method used to discover relations between time series, is the *Pearson Product-Moment Correlation Coefficient (PPMCC)*. Let x and y be two time series, where each series value is the harmonic speed of the road for a specific time interval, the coefficient for the two time series for time interval t is calculated based on the following equation:

$$PPMCC_{xy} = \frac{E\left[(x_t - \mu_x)(y_t - \mu_y)\right]}{\sigma_x \sigma_y} \tag{9}$$

where μ_x is the mean and σ_x is the standard deviation of time series x. However, as also noted in [16], the PPMCC of two time series fails to capture the temporal characteristics between the time series. For example, the value for road n_1 at time t may not be well correlated with the value of the road to predict r at the same moment in time (t). In contrast, if vehicles travel from road n_1 to road r within 1 interval, then the $(t-1)$-th value of the time series of n_1 shall be well correlated with the value of road r series at time t. Such a relation can be identified by metrics that take into account the possible *lag* between the time series. One of them is the *Cross-Correlation Coefficient (CCF)*, defined as:

$$CCF_{xy}(k) = \frac{E\left[(x_t - \mu_x)(y_{t+k} - \mu_y)\right]}{\sigma_x \sigma_y},\ k = 0, \pm 1, \dots \tag{10}$$

where k is the lag between the time series. A clear interpretation of the above metric is given by taking its squared value and multiplying it by 100, so as to define the *Coefficient of Determination (CoD)* between time series x and y at lag k:

$$CoD_{xy}(k) = 100 \left[\frac{E\left[(x_t - \mu_x)(y_{t+k} - \mu_y)\right]}{\sigma_x \sigma_y}\right]^2,\ k = 0, \pm 1, \dots \tag{11}$$

Equation (11) provides with a generic way of determining the percentage of variance between two time series. Concerning traffic prediction, the interest mainly lies towards finding the correlation between the present (and future) values of the time series to be predicted and the past (and present) values of the neighboring series. Hence, let x be the time series of the road and y the series of a road, the lag k shall receive only negative values.

Intuitively, neighboring roads are quite prone to have some relations, thus the metrics may also be applied locally. However, global metrics are expected to outperform local ones, since traffic depends on several diverse parameters. Moreover, the traffic of a major road may heavily depend on another major road in the same city even if their distance is relatively long. For instance, upon a congestion on a ring road, roads in distant areas may shortly exhibit increasing traffic.

2.5 Application of Traffic Prediction Methods

This subsection illustrates the application of several state-of-the-art techniques to the datasets analyzed in Section 2.3. The application of four different methods including KNN, RFs, STARIMA, and lag-based STARIMA is described in the following paragraphs (2.5.1, 2.5.2, 2.5.3 and 2.5.4, respectively).

2.5.1 *k*-Nearest Neighbors

A univariate approach for predicting traffic using kNN has already been described in subsection 2.2.2. This method is enhanced using multiple time series corresponding to neighboring roads of each road to be predicted. Thus, let r be the road for which the forthcoming travel time is requested and $n1, n2, \dots$ be the neighboring roads, the vector of (7) becomes:

$$x(t) = [V_r(t), V_r(t-1), V_r(t-2), V_{n1}(t), V_{n1}(t-1), V_{n1}(t-2), V_{n2}(t), V_{n2}(t-1), V_{n2}(t-2), \dots] \quad (12)$$

and the outcome, with respect to (8), is:

$$y(t) = V_r(t+1) \quad (13)$$

where $V_i(t)$ corresponds to the harmonic average of interval t for road i. Equation (13) is similar for different time intervals ahead (e.g. for 2 intervals ahead, it shall be $V_r(t+2)$).

Thus, the training procedure includes storing a number of vectors $x(t_1), x(t_2), \dots$ along with the respective outcomes $(y(t_1), y(t_2), \dots)$. Predicting the harmonic average of the road for interval t_p+1 (where t_p denotes present time) comes down to creating the input vector $x(t_p)$ and comparing it to the training vectors $x(t_1), x(t_2), \dots$, using euclidean distance as the metric. Intuitively, each vector value corresponds to a dimension. Different distance metrics have also been tested without, however, producing significant results. The k (in our configuration 4) nearest vectors are found, and the respective output values are averaged using straight average:

$$y(t_p) = \frac{1}{k} \sum_{i=1}^{k} y_i(t) \quad (14)$$

As one may observe, the application of the kNN algorithm using (12), (13), and (14) is rather straightforward. However, selecting the neighboring roads that should be taken into account when training (and running) the algorithm is a quite interesting problem. The CoD (see (11)) was applied to the neighboring roads (of orders 1 up to 3 as proved to be optimal), in order to select those that are most well correlated. A threshold was set so that any road surpassing it to be considered weakly correlated. Thus, roads $n1, n2, \dots$ of (12) are the ones of which the correlation is satisfactory.

2.5.2 Random Forests

The second algorithm that was implemented to predict the travel time of each road is a RF-based algorithm [41], that ranked first in the GPS task of the IEEE ICDM contest [79], held in 2010. Although the algorithm was specifically adjusted to the context of the Warsaw dataset, applying it also to the Berlin dataset was rather intuitive in terms of feature selection.

Each RF corresponds to one road and its features consist of global inputs, neighborhood (or local) inputs, and inputs for the road. These features are defined as follows:

- Global inputs refer to statistics for all roads for a 30-minute interval, including number of non-zero samples, number of zero samples, and arithmetic mean of speeds. The selection of these metrics/dimensions is actually quite reasonable for the Warsaw dataset, due to it being highly noisy and containing many zeros. Concerning, however, the Berlin dataset, the number of zero samples is dropped, as the results were more satisfactory. The difference between the two datasets is clarified further in the conclusion of this chapter. Finally, a *Principal Component Analysis (PCA)* was undertaken on these features in order to isolate the most important features for each of the three (or two) dimensions.

- Neighborhood inputs are received in a manner similar to global inputs. However, no PCA projection is applied on the data of first-order neighbors; only first-order neighbors were taken into account, as in [41].

- The inputs of the road to be predicted consist once again of the three (or in the case of Berlin two) aforementioned metrics, as well as the harmonic average of the road intervals beyond the 30-minute window. The latter are actually used as target output data in order to train the RF.

The parameters of the RFs were adjusted such that they resemble the ones stated in [41]. Concerning both datasets, the number of trees per forest was set to 100.

2.5.3 STARIMA

Determining the main parameters of a STARIMA model is a complex procedure that involves both intuition and use of statistics on the data at hand. An interesting approach on the subject is given by Kamarianakis and Prastacos in [51]. Upon analyzing the network, the authors build a STARIMA model that describes their problem (traffic flow prediction for 25 loop detectors in the city of Athens) sufficiently.

In accordance with (6), the analysis followed determines the orders p, d, and q[2]. A simplified generic form of STARIMA, similar to the one of the authors, is defined by the following equation:

$$Z_{t+1} = \phi_{00} \cdot Z_t + \phi_{10} \cdot Z_{t-1} + \phi_{20} \cdot Z_{t-2} + \phi_{11} \cdot W_1 Z_t + \phi_{12} \cdot W_2 Z_t + \dots \tag{15}$$

where Z_t represents road speed(s) at time t, W_o is the neighbor matrix of order o and ϕ_{to} is the parameter(s) for road(s) of order o at interval t. Intuitively, (15) denotes that the speed of a road at a specific time interval ahead (in (15) it is 1, but it could also be 2, 3, etc.) is a linear combination of the speed values for the three previous intervals as well as the values of the first, second, etc. order neighbors.

An element in an o-th neighbor matrix is defined by:

$$w_{ij}^o = \begin{cases} \dfrac{1}{\sum_{j=1}^{N} w_{ij}^o} & \text{, if } i \text{ and } j \text{ are } o\text{-th neighbors} \\ 0 & \text{, otherwise} \end{cases} \tag{16}$$

[2]See [51] for a detailed analysis on determining the space and time parameters, based on space-time cross-covariance.

Neighbor parameters can either be as many as the neighbors or as many as the orders of the neighbors taken into account. Concerning the datasets, averaging over the neighbors provided better results, so this is used for the experiments. The order of neighbors to take into account is also an interesting choice. Upon testing, it was determined that taking up to third-order neighbors into account is optimal.

In any case, the procedure is similar. The training data are used to construct different instances of (15), i.e. different Z vectors for the time intervals $(t_1, t_2, \dots)$. Upon creating all equations, the algorithm is trained and a least square estimate is used to determine the values of the ϕ vector:

$$\phi = [\phi_{00}\ \phi_{10}\ \phi_{20}\ \phi_{11}\ \phi_{12}\ \dots] \tag{17}$$

Forecasting a future value for a road is straightforward. The Z array is constructed for the present interval t_p and the next value is estimated by calculating the value of (15). Of course, the ϕ parameter vector is known at run time.

2.5.4 Lag-based STARIMA

The STARIMA implementation shown in the previous subsection is found to be quite successful when the neighborhood of each road is actually indicative of the values of the respective time series. However, the locality hypothesis of this algorithm (something that is also present in KNN- see subsection 2.5.1) may not stand for all possible cases.

Hence the intuition indicates the need for a global metric, such as the PCA analysis used in the RF algorithm (see subsection 2.5.2). In general, PCA is optimal when the data is noisy and hard to interpret, thus using it for the Warsaw dataset is quite reasonable. However, the Berlin dataset seems much more noise-free, due mainly to it being real (instead of simulated). Since the Berlin dataset is the main scope of this work, the selected global metric that should reveal the hidden dependencies among roads is the CoD. Correlation metrics are known to be quite effective in analyzing multivariate time series, especially if data is dense, with meaningful analogies.

Thus, the CoD is applied globally between any two roads of the network, using (11). As a result, the most well-correlated time series for each road, which are not necessarily neighboring, are identified and given to equation (15) of the STARIMA algorithm. However, since neighborhood information is no longer important, the parameters are altered in order to reflect the lag between each pair of time series. Consequently, the spatio-temporal properties of the data are interpreted in a more meaningful way.

In particular, the speeds in (15), denoted by Z, are now adjusted to reflect lags:

$$Z_{t+1} = \phi_{00} \cdot Z_t + \phi_{10} \cdot Z_{t-1} + \phi_{20} \cdot Z_{t-2} + \phi_{11} \cdot W_1 Z_{t-1} + \phi_{12} \cdot W_2 Z_{t-2} + \dots \tag{18}$$

where W_l is the neighbor matrix of lag l and ϕ_{tl} is the parameter(s) for road(s) of lag l at interval t. Note also that the Z values correspond to lags, in contrast with (15). In accordance with (16), the weight matrix becomes:

$$w_{ij}^l = \begin{cases} \dfrac{1}{\sum_{j=1}^{N} w_{ij}^l} & \text{, if } CoD_{ij}(l) \in M_l \\ 0 & \text{, otherwise} \end{cases} \tag{19}$$

where M_l is the set with the largest CoDs for lag l. Aiming for robustness, three lag values are considered, concerning 3 intervals before the current one to be predicted. In addition, the m largest values are taken into account, without further restrictions about their lag. Setting m to 10 provided optimal results, even though most reasonable values performed well.

2.6 Conclusion

Traffic has lately grown to be a major problem with social, economical, and the ecological effects. In terms of this deliverable, the problem of forecasting travel times was analyzed and several literature methods were illustrated. Furthermore, novel ideas were discussed in order to improve on the application of these methods to the problem at hand. The useful conclusions on the research topic of forecasting travel times are analyzed hereafter.

At first, the algorithms were drawn from a plethora of scientific areas, thus their application shall provide with interesting insight concerning their appropriateness on solving complex tasks, with respect to the data. In particular, since two different datasets were used, one could comment on the applicability and effectiveness of the algorithms in each dataset. Generally, the datasets are quite diverse. The Warsaw dataset, being generated, was formed as a difficult ML problem, thus handling it using “pure” ML implementations, such as the RF approach seems reasonable. However, the STARIMA implementations usually prove quite effective in terms of real data, with trend and periodicity, seen mainly in the Berlin dataset.

Concerning the complexity of detecting trends in multivariate time series, it grows exponentially, thus making the task quite complex, especially when requiring global information. Therefore, global metrics prove necessary for isolating the “useful” chunks of data with respect to the roads of which future travel time values are predicted. PCA seems to be better suited for “noisy” and sparse datasets, such as the Warsaw one, since it reveals hidden dependencies among different space-time instances of the dataset. On the contrary, correlation metrics, such as the CoD, are generally effective when there are strong correlation relationships among the roads of the network, which is the case for Berlin.

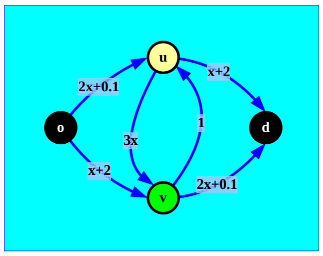
Finally, generalizing the analysis of this chapter to parametric and non-parametric algorithms is an interesting, yet, complex task which may be explored further in future work. Furthermore, novel ideas may arise by combining the algorithms to form generally effective hybrid solutions. In terms of the eCOMPASS project, this chapter provided useful insight, since it is derived from a hands-on approach on the task of traffic prediction.

3 Time-Dependent Shortest Paths

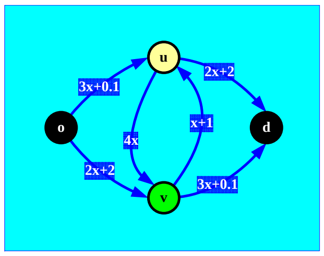
3.1 Preliminaries and Problem Statements

Computing shortest paths in graphs is a core task in many real-world applications, such as route planning in transportation networks, routing in communication infrastructures, etc. Typically the underlying graph is accompanied with an arc-cost function, assigning a *fixed* cost value to every arc, representing average travel-time, distance, fuel consumption, etc. The path of a particular cost is then the aggregated arc costs along it.

However, in real-world applications the cost of each arc should not be considered as a fixed value, since it undergoes frequent updates. These updates may be instantaneous: Unpredictable changes in traffic may occur, e.g., due to a sudden change of weather conditions, or a car accident that blocks a road segment or junction. But also anticipated updates may occur, due to periodic changes of the network characteristics over time. For example, the travel-time of a road segment may depend on the real-time congestion upon traversal, and thus on the departure time from its tail: During rush hours it is anticipated that it will be much longer than the free-ride travel-time, which is usually valid only for particular departure times (e.g., during the weekend, or late at night). This latter case of networks in which the characteristics of the network *change in a predictable fashion* over time, are called *time-dependent networks*. In the present section our focus is on shortest path computations on these networks, therefore, we assume that the behavior of each arc-cost (e.g., arc-travel-time / delay) is described by a function of the departure time from the origin, whose exact shape comes from statistical analysis of historical traffic information. For example, the travel-time of a particular road segment may be the average of sampled delays at particular times during a day from the historical traffic information, say per 2 minutes during rush hours and more rarely for the remaining periods of the day; the corresponding arc-cost function is then considered to be the (continuous) *interpolant* of all these averaged sampling points. Simply taking a snapshot of the entire network (if possible) and solving the corresponding (static) shortest path problem is clearly not the proper way to provide a route plan in a time-dependent network. In the following we shall interpret the arc-cost functions as travel-time (or delay) functions. The problem is then to compute a truly shortest path between an origin-vertex o and a destination-vertex d in the network taking into account not only the departure time from the origin, but also the consequent departure time from the tail of any other arc that is to be used by the adopted $od-$path towards the destination. The problem was introduced in [18]. We explain the nature of questions that might be of interest via a particular example and consequently we formally determine the algorithmic challenges to address. Assume that one has to move from a location o (the origin) to a location d (the destination) in a directed graph whose arc-travel times are determined by (continuous) functions of departure time from $tail[a] = u$ of arc $a = uv$, $D[a] : \mathbb{R}_{\geq 0} \rightarrow \mathbb{R}_{>0}$, as shown for example in figure 1.(a). How



(a) A time-dependent network with (linear in this case) arc-delay functions.



(b) The corresponding arc-arrival functions.

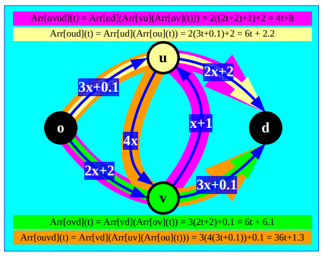

Arr[ovud](t) = Arr[ud](Arr[vu](Arr[ov](t))) = 2((2t+2)+1)+2 = 4t+8
Arr[oud](t) = Arr[ud](Arr[ou](t)) = 2(3t+0.1)+2 = 6t + 2.2
u
o
d
v
3x+0.1
2x+2
4x
x+1
2x+2
3x+0.1
Arr[ovd](t) = Arr[vd](Arr[ov](t)) = 3(2t+2)+0.1 = 6t + 6.1
Arr[ouvd](t) = Arr[vd](Arr[uv](Arr[ou](t))) = 3(4(3t+0.1))+0.1 = 36t+1.3
(c) The resulting path-arrival functions.

Figure 1: A motivating example for time-dependent shortest paths.

should the traveler commute *as fast as possible* from o to d, for a given departure time (from o), say $t_o = 0$ or $t_o = 5$? What if he is not certain about the exact departure-time and would rather prefer to have an answer for *every possible departure time*? Rather than considering the arc-delay functions $D[a](t)$ given by the description of the instance, it is more convenient to consider the arc-arrival-time functions $Arr[a](t) = t + D[e](t)$ to $head[a]$, as functions of the departure times t from $tail[a]$ of each arc a, as shown in figure 1.(b). The commuter has to compare the *arrival-times* provided by the four distinct $od-$paths in the example, shown by different colors in in figure 1.(c). For every $od-$path p, the commuter can calculate the corresponding *path-arrival-time* function, $Arr[p] : \mathbb{R} \to \mathbb{R}_{>0}$ that can be easily be shown to be the *composition* of the path p's constituent arc-arrival functions. It is then straightforward to answer both the above questions, by considering the *minimum* over all these path-arrival functions, to determine the *earliest-arrival-time* function $Arr[o,d](t_o) = \min_{p \in P_{o,d}}\{Arr[p](t_o)\}$ for any given departure time t_o. In the above mentioned example this function is:

$$Arr[o,d](t_o) = \begin{cases} 36t_o + 1.3 & \text{(ie, orange path), if } t_o \in [0, 0.03] \\ 6t_o + 2.2 & \text{(ie, yellow path), if } t_o \in [0.03, 2.9] \\ 4t_o + 8 & \text{(ie, purple path), if } t_o \in [2.9, +\infty) \end{cases}$$

Having the (exact, in the above example) functional description of $Arr[o,d]$, the commuter can easily decide what to do, either for a particular departure time, or for various potential departure times from the origin.

Of course, due to the exponential number of $od-$paths in the network, such an approach is not plausible. Still, it might be useful for the commuter (and also for the designer of an approximate distance function in the network, or a time-dependent distance oracle) to be able to have instantaneous descriptions of the earliest-arrival functions around given departure-times from the origin. Therefore, two main algorithmic challenges that we have to pursue are the following:

INPUT:	A directed graph $G = (V, A)$ with (continuous, positive) *arc-delay* functions, ie, $\forall a \in A,\ D[a] : \mathbb{R}_{\geq 0} \to \mathbb{R}_{>0}$ is the *arc-travel-(delay)-time* function. $Arr[a](t) = t + D[a](t)$ is the *arc-arrival-time* function.
DEFINITIONS:	*Path-Arrival/Delay Function:* $\forall (o,d) \in V \times V,\ \forall p = (a_1, \ldots, a_k) \in P_{o,d},\ \forall t \geq 0,\ Arr[p](t) = Arr[a_k] \circ Arr[a_{k-1}] \circ \cdots \circ Arr[a_1](t) = Arr[a_k](Arr[a_{k-1}](\cdots(Arr[a_1](t))\cdots))$ (the *composition* of the involved arc-arrivals in p in reverse order) is the path-arrival-time function of p. $\forall t \geq 0,\ D[p](t) = Arr[p](t) - t$ is the path-travel-time function along p. *Earliest-Arrival/Delay Function:* $\forall (o,d) \in V \times V,\ \forall t_o \geq 0,\ Arr[o,d](t_o) = \min_{p \in P_{o,d}} \{\ Arr[p](t_o)\ \}$ is the *earliest-arrival-time* function and $D[o,d](t_o) = Arr[o,d](t_o) - t_o$ is the *shortest-travel-time* function, from o to d.
(SOEAT)	Single-Origin Earliest-Arrival-Times: For given departure-time $t_o \in \mathbb{R}_{\geq 0}$ from o, determine the *value* $t_d = Arr[o,d](t_o)$.
(SOEAF)	Single-Origin Earliest-Arrival-Functions: Provide a *succinct representation* of the *function* $Arr[o,d]$ (or equivalently, $D[o,d]$), for a given origin o and any destination d reachable from it.

But these are *not* the only challenges that we seek to tackle for time-dependent shortest path computations within eCOMPASS. Our primary concern is to provide quite fast (e.g., polylogarithmic / sublinear / within milliseconds) response-time per query, when several shortest-path queries for arbitrary $od-$pairs and departure times are submitted by the commuters and have to be served *in real-time*. In order to achieve this, one has to avoid processing directly (and solely) the raw traffic data. A similar approach has extensively been studied in the literature for static (time-independent) networks (see, for example, a nice overview of speed-up techniques and distance oracles for static networks in deliverable D2.1 of eCOMPASS). This is because urban road-network instances are typically of huge scale, therefore rendering impractical the real-time computation of shortest paths directly on the raw data, particularly when an urban traffic server has to respond to decades or even hundreds of shortest-path queries per second.

The idea is to somehow have the raw traffic information *preprocessed offline* (i.e., solve SOEAF for selected locations in the network) in order to create distance-metric summaries, in order to exploit this metadata to speed-up the responses to arbitrary shortest-path queries submitted by the commuters in real-time (i.e., assure really fast solutions to arbitrary, practically concurrently submitted SOEAT queries). In what follows we proceed as follows: We start by commenting on the kind of raw traffic data we anticipate within eCOMPASS. Then we discuss the main ingredients for responding to time-dependent shortest path queries, namely, how to solve either exactly or approximately SOEAT and SOEAF for selected origin / origin-destination nodes. Finally, we conclude with the overview of a novel approach for providing a time-dependent distance oracle, that supports provably sublinear shortest-path queries.

3.2 History and Related Literature

3.2.1 FIFO vs Non-FIFO Networks

A fundamental parameter for time-dependent networks is whether or not the arc delay functions possess the *(strict) FIFO property*, according to which there is no need to (we should not) wait at the tail of an arc, hoping for an earlier arrival-time at the head by waiting and then traversing the arc in the future. This is equivalent to requiring that the arc-arrival functions are non-decreasing (i.e., strictly increasing). The property is also known as the *non-overtaking property* because it is equivalent to assuming that no car may overtake another car in the urban network, e.g. because we assume that every commuter moves at the alleged speed per arc. This seems to be plausible in network representing urban-traffic road networks, but there are other cases in which it may not be applicable. For instance, one should probably wait for the next (faster) IC train, than use the immediately available (slower) local train, in a time-dependent *public transport* network. Figure 2 demonstrates a FIFO (on the left) and a non-FIFO arc-delay function. It is not hard to observe

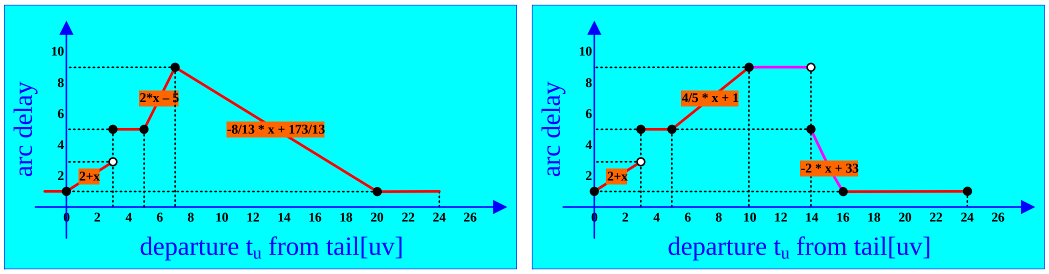


Figure 2: Example of a FIFO (on the left) and a non-FIFO (on the right) arc-delay function.

that the FIFO property (only defined for arc-delay / arc-arrival functions) is also extended to path-delay/path-path arrival functions. When the FIFO property does not hold, it may be meaningful for a commuter to also adopt a *waiting policy* at intermediate nodes while traveling. It is not always reasonable to assume waiting unrestrictedly at nodes of the network, in order to achieve the earliest-arrival time at the destination. Indeed, it may even be the case that there is no optimal waiting time that would assure that (e.g., due to discontinuities of some arc-delay functions). Even if the optimal waiting times at nodes are always well-defined, it may occur that computing the optimal waiting policies is indeed a hard problem. Within eCOMPASS WP2 we focus on route planning of private cars and fleets of vehicles. It is therefore quite reasonable to assume that the FIFO property holds, and this assumption is also justified by the experimental traffic data for the urban traffic network of Berlin, provided to the project by TomTom.

3.3 Solving SOEAT in Real Time

The contribution of this section originates from [55]. For a given $od-$pair (or, given origin only), and a given departure time t_o from o, it is known since 1969 [30] that, when unrestricted waiting-at-nodes is allowed unconditionally and an optimal waiting value always exists, a time-dependent shortest paths (TDSP) tree from o, along with the earliest-arrival-time *values* (i.e., a solution to SOEAT), is polynomial-time computable via a simple variant of Dijkstra's algorithm. I has also been demonstrated [67] that non-FIFO networks with arbitrary waiting-at-nodes are polynomially-equivalent to a FIFO networks in which of course there is no meaning in waiting at nodes. Moreover, it is known [67] that, for *affine* arc-delays (with FIFO property, or non-FIFO property but allowing unrestricted waiting at nodes), also the algorithm of Bellman-Ford works. But, if waiting-at-nodes is restricted/forbidden and the arc-delays functions may not preserve the FIFO property, then subpath optimality of shortest paths is not necessarily preserved and the problem becomes hard. For a more detailed overview of the problem see deliverable D2.1 of eCOMPASS.

3.3.1 Succinct Representation of Arc Delay Functions

In eCOMPASS the raw-traffic data for experimentation provided by TomTom are indeed created by processing historical traffic data for the city of Berlin, and are represented as frequent sample points (per arc) on a weekly basis. Therefore, we consider that $\forall a = uv \in A$, the *forward arc-delay* function $\overrightarrow{D}[a] : \mathbb{R} \to \mathbb{R}_{\geq 0}$ is a periodic piecewise-linear (pwl) function (of departure times t_u from u) expressed as the interpolant of these sampling points, for a time period $\Pi = [0, T]$, such that $\forall k \in \mathbb{Z}, \forall t_u \in \Pi, \forall a \in A, \overrightarrow{D}[a](t_u + k \cdot T) = \overrightarrow{D}[a](t_u)$. The periodicity, along with the piecewise-linearity of the arc-delay functions, assure succinctness of the representation of the raw traffic information. An example of such an arc-delay function is the following (its plot is shown in figure 3).

$$\forall t_u \in \mathbb{R},\ \overrightarrow{D}[uv](t_u) = \begin{cases} \frac{4}{3}t_u + 1, & 0 \leq t_u \mod T \leq 3 \\ 5, & 3 \leq t_u \mod T \leq 5 \\ 2t_u - 5, & 5 \leq t_u \mod T \leq 7 \\ -\frac{8}{13}t_u + \frac{173}{13}, & 7 \leq t_u \mod T \leq 20 \\ 1, & 20 \leq t_u \mod T \leq 24 \end{cases}$$

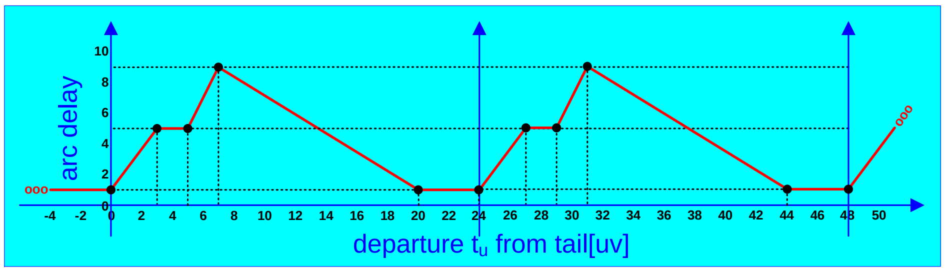


Figure 3: Example of a continuous pwl (forward) arc-delay function for an arc $a = uv \in A$, whose period is $T = 24h$.

For notational reasons we assume that $\forall t_u \in \Pi, \forall u \in V, \overrightarrow{D}[uu](t_u) = 0$ and $\forall uv \notin A \Rightarrow \overrightarrow{D}[uv](t_u) = +\infty$. Moreover, rather than defining the arc-delay functions as functions of *departure-time from the tail*, we may also prefer to express them as functions of *arrival-times at the heads*. We use the notation $\overleftarrow{D}[uv] : \mathbb{R} \to \mathbb{R}_{\geq 0}$ for these *reverse arc-delay* functions. For example, the reverse arc-delay function corresponding to the forward arc-delay of figure 3 is the following (see figure 4):

$$\forall t_v \in \mathbb{R},\ \overleftarrow{D}[uv](t_v) \quad = \quad \begin{cases} \frac{4}{7}t_v + \frac{3}{7}, & 1 \leq t_v \mod T \leq 8 \\ 5, & 8 \leq t_v \mod T \leq 10 \\ \frac{2}{3}t_v - \frac{5}{3}, & 10 \leq t_v \mod T \leq 16 \\ -\frac{8}{5}t_v + \frac{173}{5}, & 16 \leq t_v \mod T \leq 21 \\ 1, & 21 \leq t_v \mod T \leq 24 \vee 0 \leq t_v \mod T \leq 1 \end{cases}$$

It is mentioned that for the reverse expression of the arc-delay function to exist, it must be the

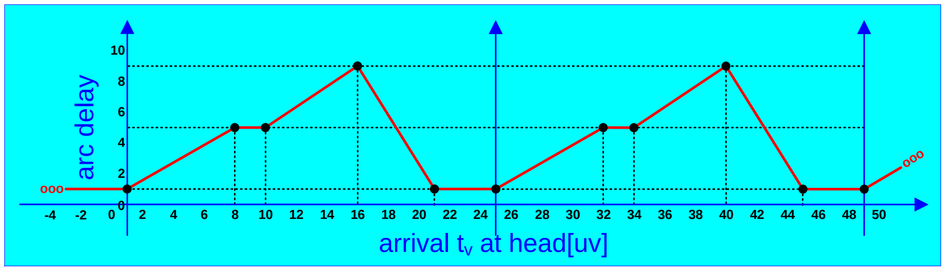


Figure 4: The reverse arc-delay function corresponding to the (forward) arc delay function of figure 3.

case that the original (forward) arc-delay does not have any leg of slope less or equal to -1, i.e. the FIFO property holds. In particular, when this is the case, we can invert the (monotone in this case) arrival-time function $t_v = Arr[uv](t_u) = t_u + \overrightarrow{D}[uv](t_u)$ to get the *latest-departure-time* function from the tail u, $Dep[uv] = (Arr[uv])^{-1}$, and then compute $\overleftarrow{D}[uv](t_v) = t_v - Dep[uv](t_v) = Arr[uv](t_u) - t_u = \overrightarrow{D}[uv](t_u)$.

3.3.2 How to Solve PWL-SOEAT

One can solve SOEAT with pwl arc-delays, assuming that the (strict) FIFO property holds. A simple variant of Dijkstra's algorithm indeed works also for the computation of shortest $od-$paths and earliest-arrival-time *values* (for given departure time from the origin) in any time-dependent network possessing the FIFO property [30]. We denote such a time-dependent variant by TDD. To avoid tricky situations in which the algorithm (even for static networks) might fail, we suppose that all the arc-delay functions are always *non-negative*. Put it differently, we consider as the actual arc-delay to be the maximum of zero and the declared arc-delay function, for any departure time from the tail.

3.3.3 Evaluating Arc-Delay Values

During the execution of TDD on a FIFO network with (non-negative) arc-delay functions, the delay value of every arc has to be estimated upon its (unique) relaxation, when its head is settled. For SOEAT with *linear* arc delay functions, this would definitely have cost $\mathcal{O}(1)$. The case of pwl arc delays, arc-delay evaluation is a little bit more complicated: When referring to the succinct representation (e.g., as a collection of breakpoints) of an arc-delay function for the arc $a = uv$ that is currently being relaxed for a given departure time $t_u = Arr[o, u](t_o)$, the arc-delay evaluation operation is not constant anymore, but costs either $\mathcal{O}(\log(K_a))$ (e.g., by maintaining a binary search tree of breakpoints) or even $\mathcal{O}(\log(\log(K_a)))$ if one employs more advanced data structures (e.g., fast tries of breakpoints) in order to determine the appropriate leg of the (pwl) arc-delay function $D[a]$ which is appropriate for t_u. K_a is the space-complexity (number of breakpoints) of $D[a]$. Since every arc is relaxed at most once, in overall TDD will have time-complexity $\mathcal{O}(n\log(n) + m \cdot \log(\log(K_{\max})))$ to solve SOEAT for pwl arc-delays, where $K_{\max} = \max_{a\in A} K_a$.

3.3.4 Computing Latest Departure Times from Origins to a Single Destination

Knowing how to solve SOEAT (eg, for pwl arc-delays) also assures a solver for the complementary problem of computing latest-departure-times from origins to a single destination (SDLDT): One has to consider the same graph but also consider the incoming (rather than the outgoing) arcs per vertex behaving according to the reverse arc delay functions, and then run TDD, the only differences being that:

- Within the priority queue Q, the objects are ordered in decreasing departure-times from the tails of the arcs.
- Each $Q.pop()$ operation retrieves the object with the maximum key.
- Relaxation of arc $vu \in \overleftarrow{A}$ occurs during the settlement of the *tail vertex* v, when the *subtraction* of the reverse-arc-delay value $\overleftarrow{D}[uv](t_v)$ from the actual arrival time t_v at v is *greater than* the current value of vertex v in Q.

3.4 Efficient Approximation Algorithms for Travel Profiles

The contribution of this section originates from [55]. In this subsection we gradually move from SOEAT to SOEAF. The solution to such a problem will be valuable for the provision of distance summaries, to be used either for speed-up techniques, or by query-algorithms of a distance oracle.

3.4.1 Instantaneous Descriptions of Earliest Arrival Functions at Sampled Points

We start with explaining how one can gather additional information (apart from earliest-arrival *values* provided by a SOEAT solution), concerning the instantaneous functional descriptions of the earliest-arrival-time *functions* at nodes reachable from the origin o, with respect to an arbitrary sampling departure-time t_o. In particular, our goal is to provide the description of the (affine) earliest-arrival-time functions:

$$\begin{aligned} Arr^{-}[o,v](x) &= A^{-}[o,v](t_o)\cdot x + B^{-}[o,v](t_o),\ x\in(t_o-\delta,t_o] \\ Arr^{+}[o,v](x) &= A^{+}[o,v](t_o)\cdot x + B^{+}[o,v](t_o),\ x\in[t_o,t_o+\delta) \end{aligned}$$

to each node $v \in V$, for arbitrarily small $\delta > 0$. The affinity of these functions is due to the fact that all the earliest-arrival functions we want to compute / succinctly represent are pwl functions, since they are defined as the minimization operation over path-arrival functions, which in turn are compositions of pwl functions (the arc-arrival functions). This kind of information will become useful later, when we shall seek for *approximate* shortest-travel-time functions in the network.

The main idea is, after having executed a time-dependent Dijkstra run from an origin (o, t_o), $TDD(G, D, o, t_o)$, which creates not only the earliest-arrival-time *values* at the final vertex labels $L[v]$, but also the instantaneous shortest-paths tree $T = SPT[o](t_o)$ assuring them, to execute a BFS scan in T (starting from the origin o) in order to recursively compute the above mentioned functional descriptions of earliest-arrival-time functions. Before describing the appropriate formula, we need some additional notation. The set:

$$P[o,v](t_o) = \{u \in V : (u,v) \in A \wedge L[v] = L[u] + D[uv](L[u])\}$$

contains all the parents of v in shortest $ov-$paths with departure time t_o. $L[x]$ is the final label-value of vertex x at the end of the Dijkstra run.

$$A^{\pm}[o,o](t_o) = 1,\ B^{\pm}[o,o](t_o) = 0$$

$\underline{P[o,v](t_o) = \{u\},\ \text{for some } u \in V:}$ /* unique shortest $ov-$path parent for departure-time t_o */

$$A^{\pm}[o,v](t_o) = (1 + \lambda^{\pm}[uv](Arr^{\pm}[o,u](t_o))) \cdot A^{\pm}[o,u](t_o)$$
$$B^{\pm}[o,v](t_o) = (1 + \lambda^{\pm}[uv](Arr^{\pm}[o,u](t_o))) \cdot B^{\pm}[o,u](t_o) + \mu^{\pm}[uv](Arr^{\pm}[o,u](t_o))$$

$\underline{|P[o,v](t_o)| \geq 2:}$ /* multiple shortest $ov-$path parents for departure-time t_o */

$$A^{-}[o,v](t_o) = \max_{u \in P[o,v](t_o)} \{(1 + \lambda^{-}[uv](Arr^{-}[o,u](t_o))) \cdot A^{-}[o,u](t_o)\}$$
$$B^{-}[o,v](t_o) = \min_{u \in P[o,v](t_o)} \{(1 + \lambda^{-}[uv](Arr^{-}[o,u](t_o))) \cdot B^{-}[o,u](t_o) + \mu^{-}[uv](Arr^{-}[o,u](t_o))\}$$
$$A^{+}[o,v](t_o) = \min_{u \in P[o,v](t_o)} \{(1 + \lambda^{+}[uv](Arr^{+}[o,u](t_o))) \cdot A^{+}[o,u](t_o)\}$$
$$B^{+}[o,v](t_o) = \max_{u \in P[o,v](t_o)} \{(1 + \lambda^{+}[uv](Arr^{+}[o,u](t_o))) \cdot B^{+}[o,u](t_o) + \mu^{+}[uv](Arr^{+}[o,u](t_o))\}$$

where the arc-travel-time function of an arc $a = uv$ may also have an affine prior-description $D^{-}[a](t_u) = \lambda^{-}[uv](t_u) \cdot t_u + \mu^{-}[uv](t_u)$ described by $(\lambda^{-}[uv](t_u), \mu^{-}[uv](t_u))$ and a corresponding post-description $D^{+}[a](t)$ given by $(\lambda^{+}[uv](t_u), \mu^{+}[uv](t_u))$, if we depart from u in a small neighborhood around t_u. Observe that if t_u corresponds to a *breakpoint* of the pwl function $D[a]$, then $(\lambda^{-}[uv](t_u), \mu^{-}[uv](t_u)) \neq (\lambda^{+}[uv](t_u), \mu^{+}[uv](t_u))$, otherwise $(\lambda^{-}[uv](t_u), \mu^{-}[uv](t_u)) = (\lambda^{+}[uv](t_u), \mu^{+}[uv](t_u))$.

Therefore, for a time-dependent instance $\langle G = (V, A), (D[a] : [0, T] \to \mathbb{R}_{>0})_{a \in A} \rangle$ with *strictly positive* arc-travel-time functions, if the instantaneous functional descriptions of earliest-arrival-time functions are computed as described above, after the completion of the time-dependent Dijkstra run with departure time t_o from the origin, and by considering the vertices of the graph exactly in the same order as they were settled (i.e., as if we scan the shortest paths tree in bfs order, with the vertices of each level ordered by increasing settling times, then it holds that:

$$\begin{aligned} Arr^{-}[o,v](x) &= A^{-}[o,v](t_o) \cdot x + B^{-}[o,v](t_o),\ x \in (t_o - \delta, t_o] \\ Arr^{+}[o,v](x) &= A^{+}[o,v](t_o) \cdot x + B^{+}[o,v](t_o),\ x \in [t_o, t_o + \delta) \end{aligned}$$

to each node $v \in V$, for arbitrarily small $\delta > 0$.

The explanation of this property is based on an inductive argument on the vertices whose functional descriptions have already been computed. The basis of the induction concerns the root itself, whose functional description is trivially correct. Assume now that all the vertices that have been processed so far already have the correct instantaneous descriptions of their earliest-arrival functions. Consider the next vertex $v_k \in V$ to process. Let $V_k = \{o = v_1, v_2, \ldots, v_{k-1}\}$ be the set of already processed vertices. Clearly, $\forall 1 \leq i < k < j \leq n$ it holds that $L[v_i] \leq L[v_k] \leq L[v_j]$, by correctness of time-dependent Dijkstra. Due to the *positivity* of the arc-delay values, it certainly holds that $P[o, v_k](t_o)$ may only contain nodes from V_k, whose instantaneous descriptions of earliest-arrival-time functions have already been previously computed, since we conduct a BFS scan of the produced Dijkstra tree. In case that $P[o, v_k](t_o) = \{u\}$ for some vertex $u \in V$, there is a sufficiently small $\delta > 0$ such that u is the unique shortest-path parent of v_k, for all departure times $t \in (t_o - \delta, t_o + \delta)$, by continuity of the earliest-arrival functions. Thus, by the (inductively assumed) correctness of the instantaneous functional description for vertex u's earliest arrival, it certainly holds that v_k's functional description is also correct. When $|P[o, v_k](t_o)| \geq 2$, then for each $u \in P[o, v_k](t_o)$ we have a different earliest-arrival-time-via-u function, $Arr^{\pm}[o, v|u](t)$. All these are *affine* functions that meet at the point $(t_o, L[v])$. We want to express $Arr^{\pm}[o, v]$ around t_o as the minimum over these affine functions. Clearly, $Arr^{-}[o, v](t)$ has the largest slope and the smallest constant, whereas $Arr^{+}[o, v](t)$ has the smallest slope and the larger constant among these affine functions, as shown also in Figure 5:

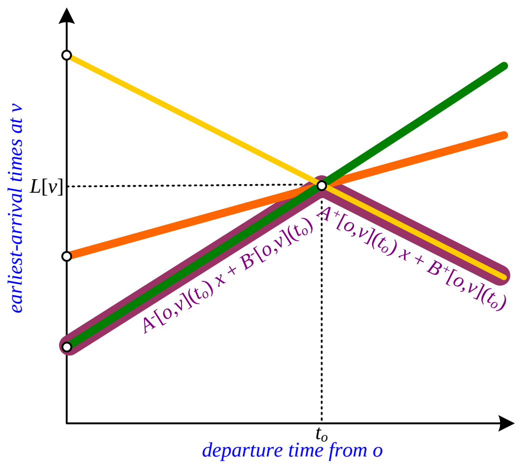


Figure 5: Instantaneous functional descriptions of the earliest-arrival-time function around the sampling departure-time t_o from the origin.

3.4.2 Time Horizon of Combinatorial Structures of Given Departure Times

After running TDD for a given departure time t_o and consequently computing the instantaneous functional descriptions of the earliest-arrival functions to destinations, our last task is to determine the *time horizon* $\hat{t}_o > t_o$ until which this information will remain valid. The reasons for such a change might be either a future arc-delay breakpoint activation, or the effect of a minimization operation at a destination vertex. Our approach for this computation is inspired by the output-sensitive algorithm of Foschini et al. [36], which introduced the related notion of *certificates* (we provide in the following a formal definition of a certificate). The time-horizon that we seek is exactly the earliest certificate failure that we shall discover in the graph.

In order to discover these discrete points at which the combinatorial structure (shortest path tree) and / or some earliest-arrival-time functions change (due to appearance of new breakpoints), we compute a set of *certificates* (one per vertex and edge) to indicate the next failure time of some earliest-arrival function, triggered by a particular element of the graph as if nothing else would change in the future.

The notion of *minimization (vertex) certificates*, one per vertex $v \in V$, provides estimations on the earliest future departure-time $t_{fail}[v](t_o)$ from the origin with respect to t_o, at which the shortest $ov-$path would change, due to the application (at v) of the minimization operation at the earliest arrival functions via different parents, assuming that no other functional description would change in the graph. Similarly, the notion of *primitive (arc) certificates*, one per arc $a = uv \in A$, indicates the projection $t_{fail}[a]$ (to a departure-time from the origin) of the next breakpoint-time (after $Arr[o,u](t_o)$) to the arc-travel-time function $D[a]$.

In particular, assume that for every vertex $v \in V$ its in-neighborhood is $IN[v] = \{u \in V : uv \in A\}$ and let for convenience $u_v = p[v](t_o)$ be the current parent of v in the shortest paths tree for departure time t_o. Recall that the instantaneous *earliest-arrival-time-via* functions are determined as follows:

$$\begin{aligned}&\forall u \in IN[v],\ Arr^+[o,v|u](t) = Arr^+[o,u](t) + D^+[uv](Arr^+[o,u](t))\\ &\quad = \quad (1+\lambda^+[uv](t_o)) \cdot A^+[o,u](t_o) \cdot t + (1+\lambda^+[uv](t_o)) \cdot B^+[o,u](t_o) + \mu^+[uv](t_o)\end{aligned}$$

where, $Arr^+[o,v](t_o) = Arr^+[o,v|u_v](t_o) < Arr^+[o,v|u](t_o),\ \forall u \in IN[v]$. Assuming that the recursively called earliest-arrival-via functions at $Arr^+[o,v|u](t) : u \in IN[v]$ would remain affine from t_o and beyond, the next *minimization certificate* at vertex v is relatively simple to compute: It is the earliest point $t_{fail}[v] > t_o$ (if any) at which any of the (suboptimal at t_o) alternative earliest-travel-time functions become equal to the value of $Arr[o,v](t_{fail}[v])$. For simplicity in notation we drop the dependence of all the functional descriptions from the departure-time t_o: $\forall u \in IN[v]\backslash\{u_v\}$,

$$t_{fail}[v|u](t_o) = \begin{cases} +\infty, & \begin{array}{c} A^+[o,v] \leq (1+\lambda^+[uv]) \cdot A^+[o,u] \\ \vee \\ B^+[o,v] \geq (1+\lambda^+[uv]) \cdot B^+[o,u] + \mu^+[uv] \end{array} \\ \frac{(1+\lambda^+[uv])\cdot B^+[o,u]+\mu^+[uv]-B^+[o,v]}{A^+[o,v]-(1+\lambda^+[uv])\cdot A^+[o,u]}, & \text{otherwise.} \end{cases}$$

The certificate failure of vertex v is then the earliest failure indication from all the possible alternative routes to reach v:

$$t_{fail}[v](t_o) = \inf_{u\in IN[v]\backslash\{u_v\}} \{\ t_{fail}[v|u](t_o)\ \} \tag{20}$$

We must also deal with the future primitive (arc) breakpoints of the earliest-arrival functions, which are indeed caused by the fact that the arc-travel-time functions are themselves piecewise linear. For this reason, for every arc $a \in A$ and departure-time t_o from o, we must know the closest future departure time from o (if any) for which the earliest arrival time at $tail[a]$ is the time-coordinate of a breakpoint in $D[a]$, *assuming* that no other breakpoint would affect the earliest arrival function at $tail[a]$. We call this departure time from o a *primitive certificate* for arc a. This is defined as follows:

$$\begin{aligned} &t_{fail}[a](t_o) \\ &= \begin{cases} +\infty, & A[tail[a]](t_o)\cdot t_o + B[tail[a]](t_o) > t^a_{k_a} \\ \min\left\{t \geq t_o : A[tail[a]](t_o)\cdot t + B[tail[a]](t_o) \in \{t^a_1,\ldots,t^a_{k_a}\}\right\}, & \text{otherwise.} \end{cases} \end{aligned} \tag{21}$$

It is mentioned that, in order to compute the instantaneous earliest-arrival-time functional descriptions *and* the tentative certificate failure times, one has to perform two sequential passes (in BFS order) over the vertices of the shortest paths tree produced by the execution of the time-dependent Dijkstra. This is because the first pass will compute the earliest-arrival-time functions, and only then can one (in the second pass of the same tree, in any order) recalculate the correct values of all the tails of arcs headed from vertices of the subtree. Figure 6 illustrates the necessity for two passes.

3.5 From SOEAT to (approximations of) SOEAF

The contribution of this section originates from [55]. Rather than computing earliest-arrival-time *values*, one may need to provide succinct representations of earliest-arrival-functions from a given origin o. Unfortunately, even for *affine* arc-delay functions, Foschini et al. [36] recently proved that there may be too many (superpolynomial) number of breakpoints, excluding a succinct representation of these functions. On the other hand, in the same work a quite interesting output-sensitive algorithm was proposed to explicitly construct these representations.

Nevertheless, the main goal is to exploit (at least approximate) solutions of SOEAF as subroutines for the creation of distance summaries (e.g., from/to selected landmark nodes) at the preprocessing step of an oracle, whose online queries will exploit these summaries to support real-time fast responses for arbitrary queries $(o,d,t_o) \in V\times V\times[0,T]$ arriving online. A crucial aspect is, therefore, to assure not just succinct, but also space-efficient approximations of the distance metric. Since it is not meaningful to provide approximate earliest-arrival-time functions (they cannot be independent of time shifts), we consider only approximations of shortest-travel-time functions. [36] et al. proposed a polynomial-time algorithm that constructs *point-to-point* approximate distance

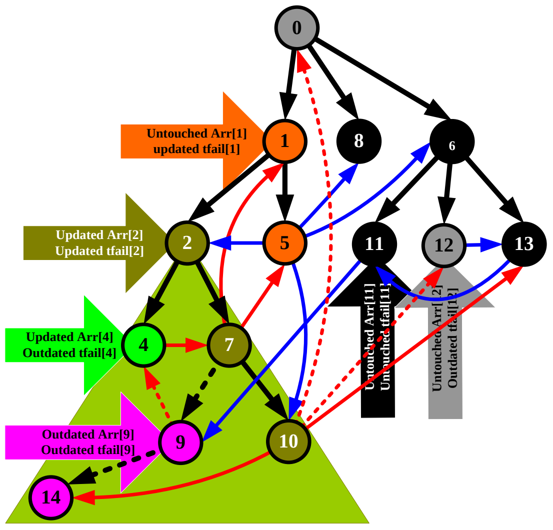


Figure 6: Explanation of the necessity for two passes of the acrs headed by vertices in the subtree of the graph element causing the current certificate failure. Solid lines demonstrate arcs that have already been scanned for the instantaneous functional description of arrival-time functions at their heads. Dashed lines demonstrate arcs that have not been visited yet.

functions, which only uses a number of breakpoints for its succinct representation that is *independent of the network size.* The main idea of the algorithm is to keep sampling the (unknown) distance function $D[o,d]$, thus creating breakpoints for an upper-approximating function $\overline{D}[o,d]$, until the required approximation guarantee of $1+\varepsilon$ is assured. This algorithm keeps sampling the function on the delay axis, so long as the partial derivative (slope) of the distance function becomes greater than 1 (i.e., the distance values change faster than departure times). As soon as the slope of $D[o,d]$ at a sample point drops below this value, the sampling procedure continues by considering samples along the time axis, via a simple bisection. Despite its small number of breakpoints, the approximate distance function is provably (asymptotically) space-optimal only for the second phase (the bisection). It is also not explained how one can gather the required information at given sampling points, such as the partial derivatives of $D[o,d]$ at the sampling points. Finally, it is not possible for the approximation algorithm of Foschine et al. to compute all the approximate distance functions from the common origin concurrently, as is done for example by the output sensitive algorithm.

Polynomial-time approximation algorithms are proposed in [55] for the succinct representation of upper-bounding functions $\overline{D}[o,d]$, for all destination vertices reachable from o at the same time, that requires asymptotically space-optimal representations, in time-complexity comparable to that of the worst-case point-to-point computation. Our algorithm is based on the exact calculation (given in closed form) of the maximum additive error assured between two consecutive sampling points. This formula is used both for the analysis of the algorithm, but also for keeping (as breakpoints) only those sampling points which are really necessary for the required approximation guarantee. For more details, the reader is referred to the eCOMPASS technical report [55].

3.6 A Time-dependent Approximate Distance Oracle

The contribution of this section originates from [56]. The main rationale of a distance oracle is to create offline a summary of distance that will be useful for responding in sublinear time to arbitrary time-dependent shortest-path queries. To our knowledge, the first time-dependent distance oracle is proposed in [56]. The algorithm works for sparse (but not necessarily planar) directed graphs, possessing two (crucial for the analysis, but also quite natural for urban traffic road networks) properties that we call the *Bounded Delay Slopes* and the *Bounded Opposite Trips*. The former asserts that the maximum slope of a shortest-travel-time function between any arbitrary $od-$pair is upper bounded by a constant $\Lambda_{\max}$. The latter asserts that there is a global constant $\zeta \geq 1$, such that for any given $od-$pair and a given time t, the shortest travel time from o to d is at most ζ times larger that that from d to o. Both these two assumptions are quite natural in realistic urban traffic road networks with time-dependent arc-delays. Our oracle is based on an analogous oracle for static (time-independent) networks described in [3], and consists of three main ingredients:

Preprocessing: A subset of *landmark vertices* $L \subset V$ is randomly chosen. In particular, every vertex $v \in V$ has a probability $\frac{1}{a}$ of being selected as landmark. Consequently, any polynomial-time approximation algorithm for $(1+\varepsilon)-$distance functions is used, to compute distance functions from/to landmark vertices to/from arbitrary vertices. The required space for storing the preprocessed data is $\mathcal{O}(|V| \cdot |L|)$, since we only need a constant number of breakpoints per landmark-to-vertex function.

Constant-Approximation Based on Landmarks: A sublinear-time algorithm for computing (on the fly) constant-approximations to arbitrary $od-$pair distances is proposed. The main challenge is to tackle the departure-times variation, and the lack of undirectedness. Additionally, the algorithm provides two approximate solutions, one based on a Dijkstra ball around the origin, and another one based on a ball around the destination. Both approximations must assure constant ratio.

Query: A recursive algorithm grows Dikjstra balls around the origin and the destination, and each time recurs on the side of the ball with the largest radius until no further recursion is allowed, in which case the constant-approximation algorithm is exploited to connect the exhaustively searched prefix and suffix subpaths and construct an $od-$path whose delay is at most $(1+\varepsilon)$ approximation of the shortest travel-time. The best possible path from all the constructed solutions is returned as the final answer to the query.

For more details, the reader is referred to [56].

3.7 Dynamic Time-Dependent Customizable Route Planning

Road Live Traffic and Historical Knowledge. For several years, commercial routing products have used data collected from sensors in the road infrastructure or GPS traces from mobile devices to provide live (i. e., current) traffic information to users. After careful statistical cleaning of the input data, this information is used twofold: to update directions for drivers currently on the road as well as to build historical knowledge about the changing behavior of street segments during a day of the week (e. g., rush hours).

Research in route planning for road networks has somewhat followed this distinction. Most of the developed techniques concern static routing with scalar edge weights (travel time). They provide quick preprocessing and fast queries [2, 21, 23, 38]. Some techniques can be generalized to dynamically updated but still with scalar edge weights (e. g. [21, 24, 70]). Somewhat less studied is the problem of time-dependent route planning [26]. Here, historical knowledge about traffic patterns is encoded into edge delay functions that map time of day to travel time along the edge. Usually, these functions are piecewise linear. A variant of Dijkstra's algorithm can be used to compute earliest arrival queries on these networks. TCH [7], TD-SHARC [20] and ATCH [8] generalize contraction and/or arc flags preprocessing to the time-dependent setting. Employing over- and under approximation of the delay functions and sophisticated multi-phase queries, practical query times [8] can be achieved on real-world datasets (road network of Germany, kindly provided by PTV). However, preprocessing is rather extensive (>30 min, less if distributed [53]) despite the fact that only 12% of the edges of that German network have time-dependent delay functions.

To tackle the challenges posed by live data, two approaches come immediately to mind: First, one could simply update a scalar graph with current traffic information and apply routing algorithms to this snapshot of the road network. However, this does not take into account traffic development at later stages of a route, or rather, traffic is assumed to not have changed by then. Second, one could generally use preprocessed time-dependent routing data but ignore it within a certain radius of the source location (i. e., the expected time horizon of the current traffic situation). Hence, no speed up would be achieved for the first leg of each route. However, the exact details of applying this scheme efficiently to preprocessing techniques is an open problem. Also, with better traffic prediction that achieves longer time horizons, hence this approach would further degrade.

Instead, we aim to incorporate, as a whole, the current traffic situation, predictions for the near future, as well as historic knowledge of traffic fluctuations as long-term prediction. We propose to treat live traffic and traffic predictions as (partial) delay function (supplied by the eCOMPASS Traffic Prediction Module), updating preprocessed data with these partial functions.

To this end, we extend the Customizable Route Planning (CRP) framework [21] to the dynamic time-dependent scenario. In a preprocessing phase, the routing graph is partitioned into multiple levels of balanced cells with small edge separators. While this is a costly operation, it is only done once. In the time-dependent customization step, a multi-level overlay graph is computed based on this partition (whenever historical knowledge has changed significantly). A specialized time-dependent customization step regularly updates the overlay graph for the current traffic information and short-term traffic prediction (e. g., once per minute). An adapted CRP query returns time-dependent shortest paths based on the customized overlay.

Preprocessing. Given a graph $G = (V, A)$, a *partition* of the vertices V is a family $\mathcal{C} = \{C_1, \ldots, C_k\}$ of *cells* $C_i \subseteq V$, such that each vertex $v \in V$ is contained in exactly one cell C_i. More generally, a *nested multilevel partition* of L levels is a family $\{\mathcal{C}^1, \ldots, \mathcal{C}^L\}$ of partitions with nested cells, that is, for each level $\ell \leq L$ and cell $C_i^\ell \in \mathcal{C}^\ell$ there must exist a cell $C_j^{\ell+1} \in \mathcal{C}^{\ell+1}$ on level $\ell + 1$, such that $C_i^\ell \subseteq C_j^{\ell+1}$ holds. We call $C_j^{\ell+1}$ the *supercell* of C_i^ℓ. For consistency, we define $\mathcal{C}^0 = V$ and $\mathcal{C}^{L+1} = \{V\}$. An arc $(u, v) \in A$ is called a *boundary arc* on level ℓ, iff u and v are in different cells of $\mathcal{C}^\ell$. In this case, u and v are called *boundary vertices* (of level ℓ). Note that a boundary vertex of level ℓ is also a boundary vertex on all lower levels. Many general graph partitioning algorithms are available, several of which aim for balanced cells while minimizing the number of boundary arcs [5]. For road networks, tailored algorithms, such as PUNCH [22] and BUFFOON [69], exist.

The preprocessing phase of CRP computes a multilevel *overlay* [46] of the input graph $G = (V, A)$. An overlay is a graph $G' = (V' \subseteq V, A')$, such that distances between vertices of G' are the same as in G. Overlays are obtained from a nested multilevel partition $(\mathcal{C}^1, \ldots, \mathcal{C}^L)$, as follows. For a fixed level ℓ, the overlay graph of level ℓ consists of exactly the boundary vertices of $\mathcal{C}^\ell$. Besides boundary arcs of G (with respect to $\mathcal{C}^\ell$), the overlay contains for each cell $C \in \mathcal{C}^\ell$ and all pairs of boundary vertices $u, v \in C$ an arc (u, v). This results in a full clique of arcs over the cell's boundary vertices. Similarly to [21], we use a compact representation to store the overlays: Instead of keeping separate graphs, we store a common vertex set for *all* levels (which is equivalent to the boundary vertices of $\mathcal{C}^1$). Only clique arcs are kept in a separate data structure per level, and are organized as matrices of preallocated contiguous memory (note that boundary arcs are already present in the input graph). In contrast to [21], we *reorder* the vertices of G, such that overlay vertices are pushed to the front (order by descending level), breaking ties by cell. Non-overlay vertices are ordered by their level-1 cells. This improves spatial locality for customization and query, and simplifies mapping between overlay and original vertices. Preprocessing must only be rerun if the *topology* of the input changes (significantly). Since this happens infrequently in practice, somewhat higher preprocessing times are not an issue.

Customization. The customization phase uses the output of the preprocessing phase to compute the metric of the overlays, i. e., for each clique arc it must compute its cost function. It proceeds in a bottom-up fashion, starting with the lowest level. Within level ℓ, each cell $C \in \mathcal{C}^\ell$ is processed independently. A cell C is processed by running, for each boundary vertex $u \in C$, a *profile search* (i. e., SOEAF search) from u. The search is, thereby, restricted to cell C, i. e., it does not relax any arcs pointing outside C. At every boundary vertex $v \in C$, this results in an earliest-arrival-time function f_v, which is assigned to the clique arc (u, v) of cell C. Customization can be parallelized by distributing different cells (on a level) among processors. In contrast to [21], the complexity of the cost functions is not known in advance. In fact, our overlay uses a (single) dynamic adjacency array to store interpolation points of clique arcs. Updates to this data structure must be synchronized. A common approach is using locks, which is costly. Instead, each thread locally maintains a log of the clique arc functions it has computed. These logs are sequentially merged at the end of processing level ℓ. Unlike the preprocessing phase, customization is much faster. Note that, like [21], when processing level $\ell + 1$, we make use of the (already computed) overlay of level ℓ, which significantly improves customization time.

Query. For vertices s, t, and departure time τ, the query operates on a search graph consisting of, (a), the overlay graph of the topmost level L, (b), all cells from the overlay that contain s or t, and (c), the subgraph of the original graph induced by the level-one cells that contain s or t. Then, a variant of Dijkstra's algorithm (i. e., SOEAT search) can be run on this search graph to get provably optimal solutions. Note that instead of extracting the search graph, we implicitly determine the level and cell on which arcs are scanned by using the partition data. To obtain the full path description, clique arcs a on level ℓ can be unpacked, by (recursively) running a local (to the cell of a) query on the overlay of level $\ell - 1$ [21].

Next Steps. In the current state, clique arcs on high level have very high complexity (i. e., their piecewise-linear representation has many interpolation points), which results in a high storage overhead in comparison to plain CRP. Also, search during the query phase becomes noticeably more expensive on higher levels for the same reason. For ATCH [8], over- *and* under ε-approximation of the arc delay functions is used to guide a first search stage, doing exact search only in the resulting subgraph. We are currently evaluating different shortcut schemes in the context of CRP customization, where we would also like to know the memory layout of clique arc representation upfront (instead of resorting to a dynamic data structure as described above). Furthermore, we are interested in evaluating different parallelization approaches.

4 Alternative Route Planning

In many areas of route planning, computing only one route from origin to destination is not sufficient. A user may want to choose a different route according to his personal preferences. For example, the subject of the selection could depend on the scenic value of the route, the minimum distance to multiple destinations over the main one, avoidance of adverse incidents like traffic jams and unavailability of some roads due to construction work, floods along a river or traffic accidents. In general, these preferences may vary and depend on specialized knowledge or subjective criteria, which are not always practical or easy to be obtained or estimated (on a daily basis). Therefore, there are many cases in which users would like to have the option of an alternative route. In order to provide many options, we are interested in computing several alternatives over the shortest route. As a result, a complete solution indicates a good set of acceptable alternative routes.

4.1 Problem Statements and Preliminaries

We consider the problem of tracing alternative paths from a source node s to a target node t on a directed graph $G = (V, E)$. Our goal is to obtain sufficiently different (low overlapping) paths with optimal or nearly optimal cost (length, travel time).

The gathering of alternative paths between a fixed source s and target node t can be supported by an *Alternative Graph*, a notion first considered in [6]. An Alternative Graph (AG) is defined as the union of several s-t paths. Formally, let $G = (V, E)$ be a directed graph with edge weight function $w : E \rightarrow \mathbb{R}^+$. An *AG* $H = (V', E')$ is a graph with $V' \subseteq V$ such that for every edge $e = (u, v) \in E'$ there is a path P_{uv} in G and a path P_{st} in H so that $e \in P_{st}$ and $w(e) = w(P_{uv})$ (see Figure 7). We call $w(P_{uv})$ the weight or cost of path P_{uv}.

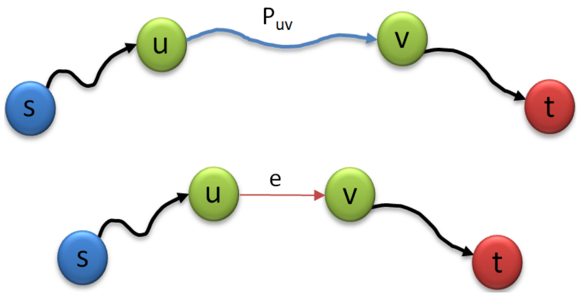


Figure 7: Upper part: Path P_{uv} in graph G. Lower part: The edge in graph H that corresponds to path P_{uv}.

In the general case, there are many alternatives. Hence, there is a need of filtering and rating all alternatives based on certain quality criteria. The main idea of the quality criteria is to discard routes with poor values. For that purpose, the following quality indicators are used [6]:

$$\begin{aligned} totalDistance &= \sum_{e=(u,v)\in E'} \frac{w(e)}{d_H(s,u) + w(e) + d_H(v,t)} \\ averageDistance &= \frac{\sum_{e\in E'} w(e)}{d_G(s,t) \cdot totalDistance} \\ decisionEdges &= \sum_{v\in V'\setminus\{t\}} (outdegree(v) - 1) \end{aligned} \tag{22}$$

In the above definitions, d_G denotes the shortest distance in graph G, d_H the shortest distance in the alternative graph H, *totalDistance* measures the extend to which the paths in the AG are non-overlapping, *averageDistance* measures the stretch (i.e. the average cost of the alternatives compared with the shortest one, as shown in Figure 10), *decisionEdges* measures the complexity of the AG (i.e. how much digestible is for a human).

The *decisionEdges* counts the possible decision “branches” in an AG. The higher the *decisionEdges* the more confusion creates to a typical user, when he tries to decide his route. The maximum value of *totalDistance* for AG is the number of s-t paths, if only all s-t paths are disjoint, i.e. not sharing common edges. The minimum value of *averageDistance* is 1. This occurs when every s-t path in AG has the minimum cost. In this way, for computing a qualitative AG, we aim at high *totalDistance* and low *averageDistance*, i.e. in the best case having acquired disjoint and shortest (i.e. with minimum cost) s-t paths. The number of *decisionEdges* is an input parameter determined by the user, but in general it should be bounded. For example, in Figures 8 and 9 we present the evaluation of the quality of some AGs according to the above indicators.

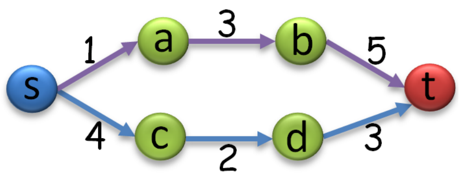


Figure 8: Quality measures of alternative paths.

For the graph in Figure 8, the quality indicators take the following values:

$$\begin{aligned} totalDistance &= (1+3+5)/9 + (4+2+3)/9 = 2 \\ averageDistance &= 18/(9 \times 2) = 1 \\ decisionEdges &= 1 \end{aligned}$$

This figure shows two s-t paths in a AG. The paths are disjoint, because they don’t share edges. Therefore, in this case the *totalDistance* gets the optimum value 2. The minimum cost is 9. So in addition the paths are shortest. The averageDistance of the AG gets the optimum value 1. This means that the cost of the s-t paths is 0% larger than the minimum.

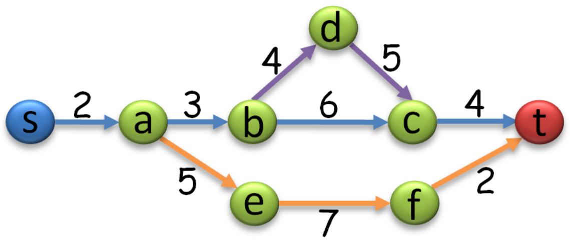


Figure 9: Quality measures of alternative paths.

For the graph of Figure 9, the quality indicators take the values:

$$\begin{aligned} totalDistance &= \frac{2+3+6+4}{15} + \frac{4+5}{5+9+4} + \frac{5+7+2}{2+14+0} \\ &= 1 + 0.5 + 0.875 = 2.375 \\ averageDistance &= \frac{38}{2.375 \times 15} = \frac{1 \times 15 + 0.5 \times 18 + 0.875 \times 16}{(1+0.5+0.875) \times 15} = 1.067 \\ decisionEdges &= 2 \end{aligned}$$

This figure shows three s-t paths in a AG. We depict the paths as $P_1 = s - a - b - c - t$ (blue), $P_2 = s - a - b - d - c - t$ (purple) and $P_3 = s - a - e - f - t$ (orange). Path P_1 has cost 15, P_2 has 18 and P_3 has 16. Therefore, path P_1 has the minimum cost. Over P_1, there are two outgoing edges that forms the alternatives and therefore the *decisionEdges* are 2. P_2 and P_3 paths have overlapping subpaths with P_1, and so their contribution in AG is less than P_1. Therefore, the *totalDistance* is less than 3. The averageDistance is calculated as 1.067, based on the *totalDistance* - contribution of the paths (values 1, 0.5 and 0.875) in AG. The *weighted average* cost is at most 6.7% larger than the minimum.

Figure 10 provides an illustration of the stretch (averageDistance) of an AG. The stretch is based on the cost of the alternative paths.

- Big stretch : the alternative paths have high costs in comparison with the minimum cost.
- Small stretch : the alternative paths have cost nearly to minimum.

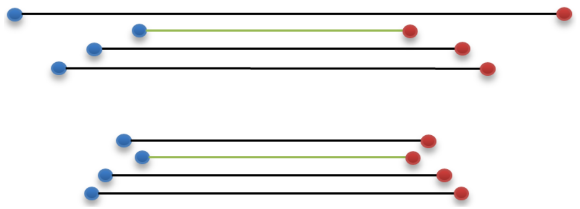


Figure 10: Cost of alternative routes. The green path has minimum cost.

4.2 Related Work

Several algorithms for computing alternative paths use the classical Dijkstra's algorithm. We recall that Dijkstra's algorithm grows a full shortest path tree rooted at a source node s keeping a partition of the node set V into a set of nodes with permanent distances (maintained implicitly), and a set of nodes with tentative distances maintained explicitly in a priority queue Q, where the priority of a node v is its tentative distance $d(s, v)$ from s. In each iteration, the node u with minimum tentative distance $d(s, u)$ is deleted from Q, its adjacent vertices get their tentative distances updated, and u thus becomes settled with $d(s, u) = d_G(s, u)$.

A host of approaches have been considered for computing alternative paths, the most important of which are reviewed in the eCOMPASS Deliverable D2.1. The most promising of them require preprocessed data computed with constant weights on edges. In general, such approaches are not suitable for the purposes of eCOMPASS, since the weight on the edges can change (i.e. due to unavailability of the road) and, moreover, they may be time-dependent. Regarding the generation of alternative graphs without preprocessed data, two approaches for producing them are the classical *k-shortest path* and the *Pareto* approaches, which have been considered in [6].

- *k-Shortest Paths*

  The k-shortest path routing algorithm [32, 82] finds k shortest paths in order of increasing cost. The disadvantage of this approach is that the computed alternative paths share many edges, which makes them difficult to be distinguished by humans. Good alternatives could be revealed for large k, but at the expense of a rather high computational cost.

- *Pareto*

  The Pareto algorithm [43, 60, 25] computes the *Pareto-Optimal* paths using more than one weight function. The weight functions could correspond to travel time, length or fuel consumption. However, even if there is only one function one could define a second function with value equal to edge's weight for the edges included in AG and zero for the edges outside AG. When there are many computed paths, by tightening the domination criteria on Pareto-optimality their number can be decreased.

As the experimental study in [6] showed, both approaches generate alternative graphs of low quality, and hence will not be investigated any further.

4.3 The eCOMPASS Approach for Alternative Routes

The eCOMPASS approach for computing alternative graphs (and hence routes) is an extension of the Plateau [6, 1] and Penalty [6, 15] methods that prune the search space and/or filter the resulting set of paths, resulting in the desired alternative graph. This makes sense, because in general alternative paths may share common nodes (including s and t nodes) and edges. Furthermore, their subpaths may be combined to form new alternative paths. The insertion of paths is determined by the indicators described in Section 4.1. In order to get the best alternatives from each method, we seek to maximize the $targetfunction = totalDistance - averageDistance$.

The Plateau method provides alternative paths by combining pairs of $s - v$ and $v - t$ shortest paths. At first, we have to find the shortest paths between s and t. The $s - v$ shortest paths can be obtained from the shortest path tree (SPT) of s and the $v - t$ shortest paths from the backward shortest path tree of t. Note that there is no need to construct the full shortest path trees. For a node v, let $d_s(v)$ be the shortest distance from s to v, and $d_t(v)$ be the shortest distance from v to t. In order to get shortest paths we limit the search space on nodes with $d_s(v)$ and $d_t(v)$ $\leq d_s(t) = d_t(s)$. The reason for this is because a node v that has $d_s(v)$ or $d_t(v) > d_s(t)$ cannot belong to a s-t shortest path. For this purpose, we can perform an (s, t) single-source single-target shortest path query using the forward Dijkstra algorithm in G, with root the node s and termination condition the settle of node t, and the backward Dijkstra algorithm in G^T (traversing the edges by the reversed direction), with root the node t and terminal condition the settle of node s.

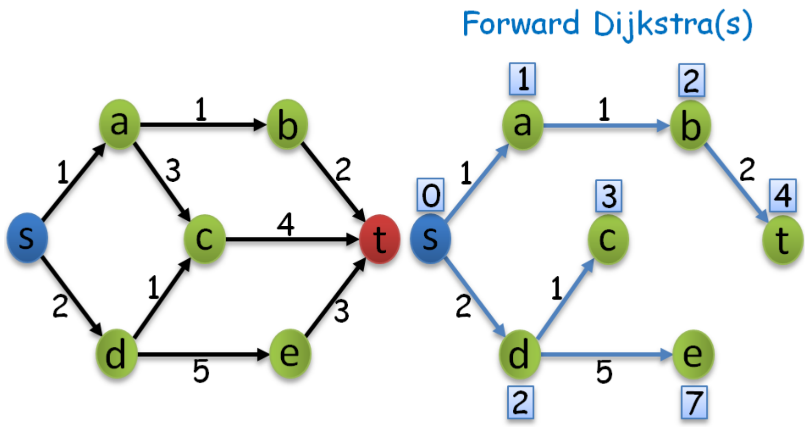


Figure 11: The original graph and the forward SPT from s.

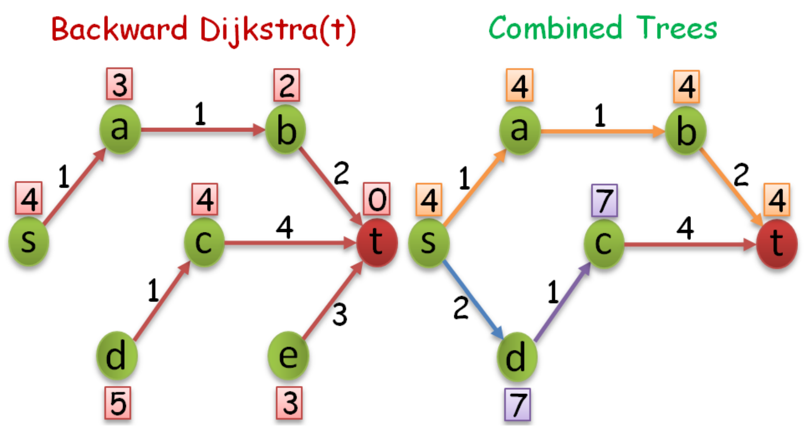


Figure 12: The backward SPT from t and the combined forward and backward SPT.

In order to identify sufficiently long shortest paths we look at the combination of forward and backward shortest path trees. The overlapping of these trees (see Figures 11 and 12) reveal paths, called the *plateaus*, which consist of nodes v which have the same $d_t(v) + d_s(v)$. In this way, a node in a plateau following the predecessor nodes in forward SPT and the successor nodes in backward SPT can form a complete s-t alternative path.

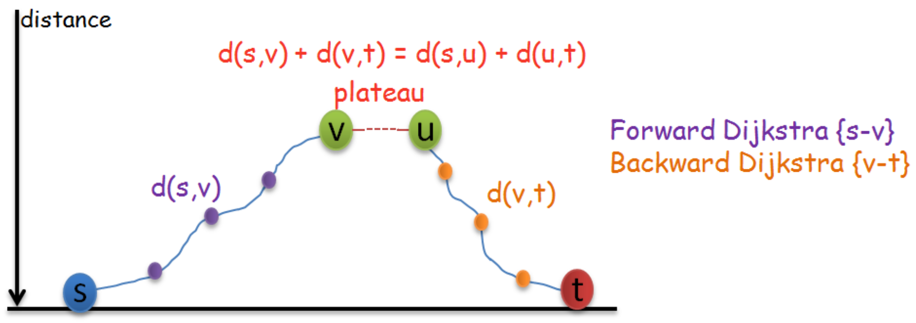


Figure 13: Plateaus.

In order to get the best alternative paths we use two pruning stages. In the first stage, since there may be many possible plateaus, we efficiently need to select the most promising among them. This can be implemented by selecting paths in non-decreasing order of rank, where the rank of a path is defined as $rank = (path\ cost - plateau\ cost)$. Hence, a plateau that corresponds to a shortest path from s to t has rank zero, which is the best value. In the second stage, we interact with the AG's quality indicators. This is 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. Maximization of the target function leads to select the best set of low overlapping and shortest alternative paths.

The Penalty method provides alternative paths by running shortest path queries and adjusting the weight of the edges of the resulted paths. The basic steps are the following. We compute a shortest path P_{st} with Dijkstra's algorithm or a speedup variant of it. Then, we penalize P_{st} by increasing the weight of its edges. Next, we run a new (s,t) query. If the new computed path P'_{st} is short and different enough from the previously discovered s-t paths, we add it to AG. We repeat until a sufficient number of alternative paths (with the desired characteristics) is found, or the weight adjustments of s-t paths bring no better results.

In order to offer the best results we need an efficient and safe way to increase the weights. Our weight adjustment policy is as follows:

- We continue the increase of the weights while the new computed path has cost, based on the original weights, close to the shortest path cost $d_G(s,t)$ or until successive increases do not offer new results.
- Instead of a constant value, we add a fraction (penalty factor) of the initial edge weight to the weight of the edge. We avoid using constant values, because they do not guarantee a balanced adjustment, since in some cases longer edges may be favored over shortest ones. In general, the higher the penalty factor is, the more the new shortest path differs from the last one. On the other hand, the lower the penalty factor is, more shortest path queries can be performed and less alternative paths can be lost.
- We restrict the weight adjustment when it could lead to the loss of good alternatives. Notice that, an unbounded penalty leads to multiple increases on the edge weights and is risky. For example, suppose that there is only one fast highway into a city, whereas there are many alternatives through the city center. If we allow multiple increases on the weights of the highway then its cost will be increased several times during the iterations. For new (s,t) shortest path queries, this may result to new computed paths that now begin from a detour longer than the highway. In this example, any possible alternative inside the city will be lost. Even worse, the algorithm will terminate with poor results if the stretch is going to be much higher. To overcome this problem we limit the number of increases or the penalty factor for the edges already included in AG. Another approach, is to prevent the increase of single edges $e = (u, v)$ with $outdegree(u) = 1$, in the computed s-t paths.
- We extend the weight adjustment to the neighborhood of the computed paths. In some cases, the new computed paths may share many small detours with the previous ones. For example, the first path computed is a fast highway and the new paths are along the highway except that they have one or many outgoing and incoming small detours to the highway. This increases the *decisionEdges* and offers meaningless alternatives. Therefore, when increasing a shortest path's weights, the weights of edges around the shortest path, that leave and join the current AG are additionally penalized (rejoin-penalty). The rejoin-penalty technique leads to high *totalDistance*.

In Figure 14 we illustrate an example of the penalty method.

Since the penalty method works on any pre-computed AG, it can be combined with Plateau. In this way, we can collect the best alternatives from Penalty and Plateau, so that the resulting set of alternatives maximizes the target function. This combination outperforms any of the individual methods.

Independently of the target function, we also bound the *decisionEdges* of an AG, resulting in an AG of typically very small size, $|V'| \ll |V|$ and $|E'| \ll |E|$, thus making it easy to store or process.

Another important issue that has to be taken into account is the optimality according to the cost of the computed paths. This is achieved online by bounding the *averageDistance* indicator, and further in a post-processing phase by setting tighter bounds to the local optimality of the edges or the subpaths of the AG. In the plateau method, the local optimality is guaranteed because the paths are selected from the shortest path trees. In the penalty method, however, the adjustment of the weights can insert non optimal subpaths. To overcome this issue, we perform a global refinement (focusing on the entire s-t path), and an iterative local refinement (focusing on individual edges).

- Global refinement: remove any edge $e = (u, v) : d_G(s, u) + w(u, v) + d_G(v, t) > \delta \cdot d_G(s, t)$, for some $\delta \geq 1$.
- Local refinement: remove iteratively any edge $e = (u, v) : w(u, v) > \delta \cdot d_G(u, v)$, for some $\delta \geq 1$.

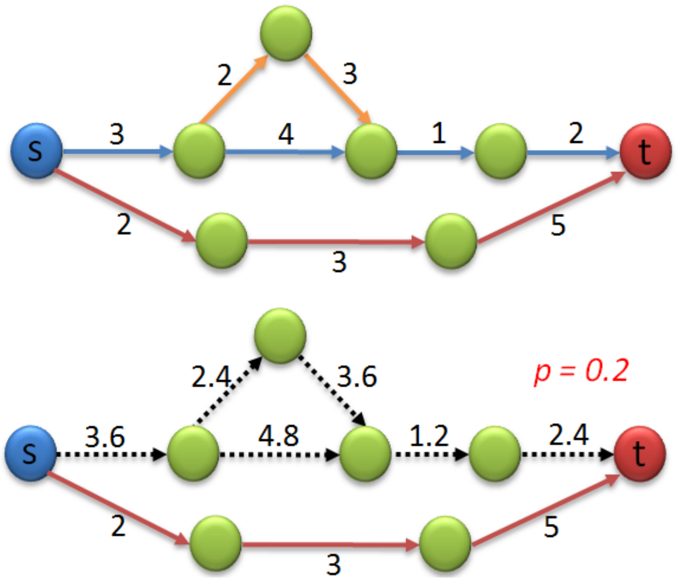


Figure 14: $\forall e \in$s−t path: $w_{new}(e) = w_{old}(e)(1+p)$, $0 \leq p \leq 1$

4.4 Current Status and Future Work

In our preliminary experiments, we used the *ALT* (*A^*+Landmarks+triangle inequality*) speedup technique [59, 39], in order to prune efficiently the search space. Note that in the penalty method, we consider only increases on the weights of the edges and this does not affect the lower bounds on the shortest distances. However, this may decrease the efficiency of the ALT algorithm. In this case, efficiency means the number of nodes on the shortest path divided by the number of settled nodes by the ALT algorithm. This factor has an important effect on the overall execution time.

For both the plateau and the penalty method, we acquired high quality results. For the plateau, we collect the best alternatives that can be formed from the forward and backward shortest path trees. This collection is based on the quality indicators. For the penalty method, in order not to miss a major quantity of alternatives, we developed a new technique in which we prevent the increase of the edges $e = (u, v)$ with $outdegree(u) = 1$ (in contrast to decision edges) in the computed *s-t* paths. Also, we set the penalty factor parameter in $(0, 1]$ and the rejoin penalty factor to an non-constant value depending on the shortest path cost. In the average case, a suitable choice for the penalty factor is 0.2. Setting a larger penalty factor reduces the number of the iterations of the penalty method; however, it may lead to occurrences with missed alternatives at shared parts that belong to previously computed paths. With the above methods we derived disjoint or low-overlap optimal paths that maximize the target function, bounding the parameters $decisionEdges \leq 10$ and $averageDistance \leq 1.1$.

5 Robust Route Planning

5.1 Introduction

Motivation Given two places A and B in a road network, the standard goal in route planning is to compute a fastest route between them. This task can be modeled as the well-known *shortest path* problem: the road network is represented by a graph with vertices corresponding to crossings, edges corresponding to roads connecting the crossings, and the goal is to find a shortest path with respect to edge costs that typically correspond to travel time estimates. However, when a computed route is traveled in reality, the travel time is influenced by various factors such as the weather, the traffic situation, the amount of road work along the route, and so on. Thus, a shortest path with respect to (precomputed) travel time estimates may be a really bad choice in the case of unforeseen accidents. In such situations, one often seeks *robust* routes instead of just fast ones. This part of the project proposes and develops a novel technique introduced by Buhmann *et al.* [12] for finding such robust routes. The only requirement is that two typical instances (e.g., traffic snapshots of yesterday and today) are provided, and the goal is to compute a solution that is likely to be good for a future (yet unknown) instance. An advantage of this method is that we can avoid fine-tuning of parameters. Especially no other knowledge such as the probability of traffic jams, weather conditions, etc., is required. This section describes the details and the issues related with this approach when applied to route planning problems. Note that we focus on the so-called *non-adaptive* scenario: a route is computed in advance and is taken throughout the whole trip. In contrast, the *adaptive* scenario considers (online) algorithms that obtain information addressing uncertainty (delays, traffic jams, etc.) during the execution and are therefore allowed to change the precomputed route.

The shortest path problem under uncertainty Let $G = (V, E, c_G)$ be a directed graph with a cost function $c_G : E \to \mathbb{Q}_0^+$. A *path* P is a sequence $\langle v_0, ..., v_k \rangle$ of vertices $v_i \in V$, $0 \le i \le k$, where $(v_{i-1}, v_i) \in E$ for $i = 1, ..., k$, and P is called a *simple path* iff $v_i \neq v_j$ for each $i \neq j$. For an edge $e = (u, v) \in E$, we write $e \in P$ iff there exists an index $i \in \{1, ..., k\}$ such that $u = v_{i-1}$ and $v = v_i$. The *cost* of a path is the sum of the edge costs. The *shortest path* problem asks, for a given graph $G = (V, E, c_G)$, a *source* $s \in V$ and a *target* $t \in V$, to compute an s-t-path with minimum cost. The set of *feasible solutions* $\mathcal{S}$ is the set of all simple s-t-paths in G. The aforementioned goal translates into finding a simple path $s \in \mathcal{S}$ that minimizes the objective function

$$c : \mathcal{S} \to \mathbb{Q}_0^+, c(s) = \sum_{e \in s} c_G(e). \tag{23}$$

As previously stated, robust paths are not necessarily shortest paths in given costs. Instead, we search for a path that is fairly good for a typical scenario. For the *shortest path problem under uncertainty*, we assume that the underlying graph (i.e., the topology of the road network) is fixed, but the edge costs (i.e., the travel times) are subject to uncertainty. The concrete travel times for a certain time period are denoted as an *instance* I and are given by a cost function $c_I : \mathcal{S} \to \mathbb{Q}_0^+$. A blocked road in an instance can be modeled by assigning large costs to the corresponding edge. Note that all instances share the set of feasible solutions $\mathcal{S}$ since the graph topology remains invariant.

5.2 Related work

The following overview concentrates on non-adaptive route planning under uncertainty. A detailed survey of both adaptive and non-adaptive algorithms can be found in [31].

In the literature, there exist two main directions to handle uncertainty. *Stochastic optimization* assumes that the probability for a road to be congested is known in advance, and the goal is to optimize the expected travel time as well as the probability of taking longer than expected. If we want to ignore the risk of being late, it is sufficient to compute a shortest path in a graph where the expected travel time is assigned to each edge [58]. A straightforward approach to regard both the travel time as well as the risk is the computation of Pareto-optimal solutions for a bi-criteria function of the mean and the variance of the solutions. However, the number of such solutions may be exponential [42]. Nikolova [63, 64] applied a common approach from economics to route planning to compute a path P that minimizes the convex combination $\alpha\mu_P + (1-a)\sigma_P$ of the mean μ_P and the standard deviation σ_P. The parameter α determines the trade-off between the expected travel time and the risk of arriving later than expected. The situation where early and/or late arrivals at a given destination are penalized is studied in [65]. The authors also show that the problem to minimize the expected value of the penalty function is NP-complete in some cases [65]. A different approach is to compute a route that maximizes the probability that the travel time for this route remains below a certain threshold [33]. However, no complexity results are given.

Another direction to handle uncertainty is *robust optimization*. No assumptions about probabilities are made, and the goal is to compute a route with acceptable travel time even in the worst possible scenario. The situation where a set of N possible scenarios is given is studied in [37]. The authors provide two definitions of a robust path: the *absolute robust shortest path* minimizes the maximum path length over all instances, while the *robust deviation shortest path* minimizes the maximum distance of the computed path to the optimal path over all instances. It is proven that the computation of both path variants is strongly NP-hard if the number of instances N is not constant. Both problems can be solved in time $\mathcal{O}(n^3(\text{lp}_{\max})^N)$ where $\text{lp}_{\max}$ is the length of a longest s-t-path. Furthermore, they prove that both problems are NP-complete for so-called *k-layered graphs* where the vertex set is partitioned into $k+2$ disjoint *layers* V_i, $0 \leq i \leq k+1$, $V_0 := \{s\}$, $V_{k+1} := \{t\}$, and $E \subseteq \bigcup_{i=0}^{k} V_i \times V_{i+1}$. The problems remain NP-complete if only two instances with only four layers are given. Yaman *et al.* [81] study the problem variant where each edge $e \in E$ has a cost between l_e and u_e. The absolute robust shortest path can be efficiently found by computing a shortest path in the graph where every edge e has edge cost u_e. On the other hand, the computation of a robust deviation shortest path is NP-hard, even if one considers only planar acyclic directed graphs with a maximum vertex degree of three [83]. Both the computation of an absolute robust as well as a robust deviation shortest path remain NP-hard if a convex polytope defines an uncertainty region and the vector of edge costs is drawn from this region [62]. Liebchen *et. al.* introduced the concept of *recoverable robustness* as a compromise between the adaptive and the non-adaptive scenario [57]. In the beginning, there exists only a prediction of the edge costs. After a route has been computed, the real costs are disclosed and we are allowed to change up to k edges of the precomputed route. Let $\mathcal{F}$ be the set of real costs that can occur, i.e. the set of possible instances. The goal is to compute a path that minimizes the maximum cost when an instance from $\mathcal{F}$ is taken. Büsing [13] showed that the problem is NP-hard for various choices of $\mathcal{F}$.

Maximizing the similarity of instances The method of Buhmann *et al.* [12] is substantially different from the ones proposed in the previous section. They assume that an unknown *problem generator* $\mathfrak{PG}$ generates related instances that differ due to noise. Nothing is known about the noise or $\mathfrak{PG}$ itself, and all we are given are two instances I_1 and I_2 generated by $\mathfrak{PG}$. The goal is to compute a robust (simple) path that is likely to be good for a future (yet unknown) instance I_3 from $\mathfrak{PG}$. Since nothing is known about the underlying noise, it is a natural choice to consider only paths that are good both for I_1 and I_2. Therefore, we compute the *approximation sets* $A_\rho(I_1)$ and $A_\rho(I_2)$, where, for a given instance I and a suitable value $\rho \geq 1$ (we explain the meaning of "suitable" later on),

$$A_\rho(I) := \{s \in \mathcal{S} \mid c_I(s) \leq \rho \cdot c(s_{\text{OPT}}(I))\}, s_{\text{OPT}}(I) := \arg\min_{s\in\mathcal{S}} c_I(s). \tag{24}$$

Then we pick a path at random from the intersection $A_\rho(I_1) \cap A_\rho(I_2)$ of the two approximation sets. It should be clear that not all these paths are robust in the sense that they will be good for a future instance. Especially, the probability to pick a robust path at random depends on the choice of ρ: if the intersection size is too small, then the contained paths are too much influenced by the noise of I_1 and I_2. On the other hand, if the intersection size is too large, then the ratio of robust paths and the intersection size is too small, i.e. the probability to pick a robust path at random is small. Buhmann *et al.* propose to choose ρ as the value that maximizes

$$\frac{|A_\rho(I_1) \cap A_\rho(I_2)|}{\mathbb{E}_{B\in\mathcal{F}_{|A_\rho(I_1)|}, C\in\mathcal{F}_{|A_\rho(I_2)|}}\left(|B\cap C|\right)}, \tag{25}$$

where $\mathcal{F}_k$ is the set of all approximation sets of size k. Note that the denominator corresponds to the expected size of the intersection of two *unrelated* random instances. The idea behind the ratio (25), the so-called *similarity between instances*, is the following: intuitively, if the actual intersection (for some ρ) is larger than expected, then the instances are somehow related and the intersection contains paths that are likely to be good for other related instances. Thus, to successfully apply the above technique to a concrete optimization problem, we essentially need algorithms for the following tasks:

1) Compute the size of $|A_\rho(I_1) \cap A_\rho(I_2)|$ for a given ρ,

2) Compute the expected size of the intersection for a given ρ,

3) Pick a solution from $|A_\rho(I_1) \cap A_\rho(I_2)|$ at random.

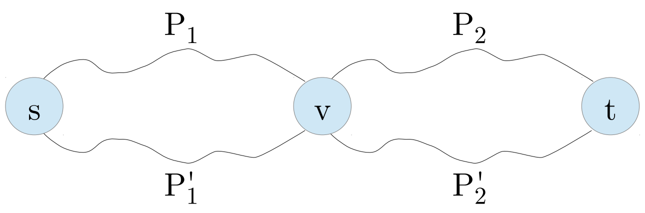
Properties and complexity issues A straightforward approach to compute the value of ρ that maximizes the ratio (25) is to sample some values for ρ and to compute the ratio of the intersection size and the expected intersection size for this concrete value of ρ. For many optimization problems it is not clear how to do this, because the numerator and the denominator cannot simply be calculated. Especially the calculation of the expected value is a crucial point. The situation actually is easy if every subset $S' \subseteq \mathcal{S}$ is an approximation set for some instance, i.e. if for every $S' \subseteq S$ there exists an instance I and a value ρ such that $S' = A_\rho(I)$. In this case, the expected size of the intersection reduces to

$$\mathbb{E}_{A\in\mathcal{F}_{|A_\rho(I_1)|}, B\in\mathcal{F}_{|A_\rho(I_2)|}}\left(|A\cap B|\right) = \frac{|A_\rho(I_1)||A_\rho(I_2)|}{|\mathcal{S}|}, \tag{26}$$

and thus we have to compute

$$\rho^* = \arg\max_\rho \frac{|A_\rho(I_1) \cap A_\rho(I_2)|}{|A_\rho(I_1)||A_\rho(I_2)|}. \tag{27}$$

As the following example shows, this condition unfortunately does not hold for the shortest path problem. Consider a network where two edge-disjoint paths P_1 and P_1' connect the source s with an intermediate vertex v, and two edge-disjoint paths P_2 and P_2' connect v with the target t.



Then, there exists no feasible approximation set that contains exactly the paths P_1P_2 and $P_1'P_2'$. Suppose there was such an approximation set. Let $c(P)$ denote the cost of the path P and c_{OPT} be the length of a shortest path from s to t. Then there exists a constant $\rho \geq 1$ such that

$$c(P_1P_2) = c(P_1) + c(P_2) \leq \rho\, c_{\text{OPT}} \tag{28}$$

$$c(P_1'P_2') = c(P_1') + c(P_2') \leq \rho\, c_{\text{OPT}}, \tag{29}$$

thus $c(P_1) + c(P_1') + c(P_2) + c(P_2') \leq 2\rho\, c_{\mathrm{OPT}}$. On the other hand, the paths P_1P_2' and $P_1'P_2$ are not contained in the approximation set, so we have

$$c(P_1'P_2) = c(P_1') + c(P_2) > \rho\, c_{\mathrm{OPT}} \tag{30}$$
$$c(P_1P_2') = c(P_1) + c(P_2') > \rho\, c_{\mathrm{OPT}}, \tag{31}$$

thus $c(P_1) + c(P_1') + c(P_2) + c(P_2') > 2\rho\, c_{\mathrm{OPT}}$, which is a contradiction. However, it can be shown [12] that

$$\frac{|A_\rho(I_1)||A_\rho(I_2)|}{|\mathcal{S}|} \leq \mathbb{E}_{A\in\mathcal{F}_{|A_\rho(I_1)|}, B\in\mathcal{F}_{|A_\rho(I_2)|}}\left(|A \cap B|\right) \leq |A_\rho(I_1)||A_\rho(I_2)|. \tag{32}$$

Although an asymptotic estimation of the expected intersection size can be derived for some optimization problems such as the *maximum subarray sum* problem, this is not the case for the shortest path problem. Unfortunately, the situation is even worse. From the results of Valiant [76], it follows that the computation of $|A_\rho(I)|$ for a given instance I and a value ρ is #P-hard. Furthermore, the computation of $|A_\rho(I_1) \cap A_\rho(I_2)|$ can be reduced to the bi-criteria shortest-path problem, which is known to be weakly NP-hard [4]. Therefore we cannot hope to develop polynomial-time algorithms to compute the size of individual approximation sets or their intersection. On the other hand, this point of view is too pessimistic: we do not need to compute the sizes exactly; rather, an estimation may be sufficient. For this reason we will also concentrate on approximative methods (see Section 5.4).

5.3 Current solutions

One of the main difficulties of the method proposed by Buhmann *et al.* is to compute the size of the approximation sets for two given instances, $|A_\rho(I_1)|$ and $|A_\rho(I_2)|$. When applied to the *shortest path problem under uncertainty*, this translates into counting the number of simple paths between two vertices of a given graph. In this section, we focus on this particular problem, and we propose a pseudo-polynomial time algorithm that computes an upper bound for this number. We also consider implementation issues for this algorithm, and we inspect whether well-known heuristics that are often used to speedup in route planning applications can be extended for this algorithm as well.

In the following, we assume $G = (V, E)$ to be a directed graph with vertex set V, edge set $E \subseteq V^2$ and edge costs $c : E \to \mathbb{N}$. Given $s, t \in V$ and a value $C_{\max} \in \mathbb{N}$, we want to count the number of simple paths from s to t with cost *at most* $C_{\max}$. Given that this problem is #P-hard [76], the best we can hope to achieve without recurring to approximation is a pseudo-polynomial time algorithm. However, to the best of our knowledge, no such algorithm is known. What we propose in the following is to compute an upper bound for this value by counting the number of non-simple paths, or *walks*, from s to t with cost at most $C_{\max}$. This problem is #P-hard as well, but we provide a pseudo-polynomial algorithm for solving it exactly.

The label propagating algorithm Our algorithm is based on the following observation. Suppose that for a vertex $v \in V$ and some $c_v \in \mathbb{N}$ the number n_v of simple paths from s to v with cost equal to c_v is known. For every vertex $w \in V$ such that $(v, w) \in E$, n_v is an upper bound on the number of paths from s to w with cost equal to $c_v + c(v, w)$ having the edge (v, w) as last step. n_v is only an upper bound rather than the exact number because one or more of the n_v paths from s to v may contain w as an intermediate step. From this observation, we can design an algorithm that counts the exact number of walks from s to t with cost at most $C_{\max}$.

The algorithm maintains a set of labels and a counter n_{st}. A *label* (c_v, v, n_v) represents a lower bound n_v on the number of walks from s to v with cost c_v. The counter n_{st} represents the number of walks from s to t with cost at most $C_{\max}$. The set of labels is partitioned in two parts: For

Algorithm 1 COUNTPATHS($G = (V, E), c,\ s,\ t,\ C_{\max}$)

```
1 $n_{st} \leftarrow 0$; $Q = \emptyset$;
2 INCREASE-LABEL($Q, 0, s, 1$)
3 **while** $Q \neq \emptyset$ **do**
4     $(c_v, v, n_v) \leftarrow$ EXTRACT-MIN($Q$)
5     **if** $v = t$ **then** $n_{st} \leftarrow n_{st} + n_v$
6     **else**  **for each** $(v, w) \in E$ **do**
7               $c_w \leftarrow c_v + c(v, w)$
8               **if** $c_w \leq C_{\max}$ **then** INCREASE-LABEL($Q, c_w, w, n_v$)
9 **return** $n_{st}$
```

temporary labels the number n_v may be lower than the exact number of walks from s to v with cost c_v. For *permanent labels* the number n_v is equal to the number of walks from s to v with cost c_v.

In the beginning $n_{st} := 0$, the set of permanent labels is empty, and the set of temporary labels contains only $(0, s, 1)$ (an initialization value with the purpose of simplifying the explanation of the algorithm).

At any intermediate step, the algorithm considers the smallest temporary label (c_v, v, n_v) in lexicographical order and declares it as permanent. Then, if $v = t$, it increases n_{st} by n_v. Otherwise, for every vertex $w \in V$ such that $(v, w) \in E$ and $c_v + c(v, w) \leq C_{\max}$, it checks whether a temporary label $(c_v + c(v, w), w, n_w)$ exists for some $n_w \in \mathbb{N}$ (it will be clear from the algorithm that there will be at most one such label). If it does, it is updated by setting $n_w := n_w + n_v$. Otherwise, the algorithm generates a new temporary label (c_w, w, n_w) with $n_w := n_v$ and $c_w := c_v + c(v, w)$. The algorithm terminates when the set of temporary labels is empty and no temporary label can be declared as permanent. At this point, n_{st} is the number of walks from s to t with cost at most $C_{\max}$. The correctness of the algorithm (proved below) is based on the fact that we can always designate the smallest temporary label in lexicographical order as permanent.

For the set of temporary labels, we use a data structure Q. Such structure supports the following operations:

Extract-Min(Q) extract from Q the smallest temporary label in lexicographical order and set it as permanent;

Increase-Label(Q, c_w, w, n_w) checks whether a label (c_w, w, n_x) exists in Q, for some $n_x \in \mathbb{N}$. If it does, it updates the value of n_x by increasing it by n_w. Otherwise, the label (c_w, w, n_w) is created and inserted in Q.

Given this data structure, Algorithm 1 shows the pseudo-code of the above algorithm for counting the number of walks from s to t with cost at most $C_{\max}$. Since the main operations of this algorithm concern generating and propagating labels, we refer to it as a *label propagating algorithm.*

Correctness To prove the correctness of the algorithm we first show that at every iteration of the external loop, the cost of the label declared permanent is at least the cost of the label declared permanent in the previous iteration. That is, the costs of the labels declared permanent is not decreasing. We prove this by contradiction. Let (c_v, v, n_v) be the label declared permanent at the current iteration, and suppose toward contradiction that (c_u, u, n_u) is the permanent label of the previous iteration, for some $u, v \in V$, $n_u, n_v \in \mathbb{N}$, and $c_u > c_v$. When (c_u, u, n_u) is declared permanent, the label (c_v, v, n_v) was not present in Q because it is smaller in lexicographical order and it would have been chosen instead. Therefore, (c_v, v, n_v) must have been created after (c_u, u, n_u) was declared permanent. From the algorithm, this implies that $(u, v) \in E$ and $c_v = c_u + c(u, v)$. This is a contradiction, because $c_u > c_v$ and $c(u, v) > 0$.

Given the above, we prove correctness by showing that, for every $v \in V$ and $0 \leq c \leq C_{\max}$, if $n_v > 0$ is the number of walks from s to v with cost c, the algorithm declares the label (c, v, n_v) as permanent exactly once. This implies correctness because the value n_{st} returned in the end is the sum of all n_t, for every permanent label (c, t, n_t) with $0 \leq c \leq C_{\max}$.

The proof is by induction on the integer value $C_{\max}$. For $C_{\max} = 0$ it is trivially true, because the only label declared permanent is the dummy label $(0, s, 1)$. We now assume that the algorithm declares permanent labels correctly up to $C_{\max} - 1$ and we show that it also declares permanent labels correctly for $C_{\max}$. Let $v \in V$ be a vertex for which there exists at least one walk from s to v with cost $C_{\max}$, and $\mathcal{N}(v) = \{u | (u, v) \in E\}$ be the vertices from which there exists an edge to v. Clearly, any walk from s to v must cross one of the vertices in $\mathcal{N}(v)$ in the last step before v. For each $u \in \mathcal{N}(v)$, let $c_u = C_{\max} - c(u, v)$. If $c_u < 0$, there cannot be a path from s to v having the edge (u, v) as last step. If $C_{\max} > c_u > 0$, we know by inductive hypothesis that a label (c_u, u, n_u) is declared permanent exactly once, where n_u is the number of walks from s to u with cost c_u, and that it is declared permanent before any label with cost $C_{\max}$. When (c_u, u, n_u) is declared permanent, the operation Increase-Label$(Q, C_{\max}, v, n_u)$ is invoked, and a label with cost $C_{\max}$ and vertex v is either created or updated. Since labels are removed from Q only when they are declared permanent, and there exists a temporary label with cost $C_{\max}$ and vertex v, a label $(C_{\max}, v, n_v)$ will be declared permanent, for some $n_v \geq n_u$. When this happens, every label with node $u \in \mathcal{N}(v)$ and cost $C_{\max} - c(u, v) > 0$ has been declared permanent exactly once, and since the vertices in $\mathcal{N}(v)$ are the only ones that can reach v, n_v is equal to the sum of the number of walks from s to any node $u \in \mathcal{N}(v)$ with cost $C_{\max} - c(u, v) > 0$. Thus, n_v is the number of walks from s to v with cost $C_{\max}$.

Running Time We now study the worst-case complexity of the label propagating algorithm. In the following, we assume the operations Increase-Label and Extract-Min require both time $\mathcal{O}(\log |Q|)$ when invoked on a data structure of size $\mathcal{O}(|Q|)$. In the following sections, we show how we can implement Q in order to meet this requirement.

First, we consider the external loop. By construction, this loop is repeated as many times as the number of permanent labels. Since there can be a permanent label for every $v \in V$ and every $0 \leq c \leq C_{\max}$, the external cycle can be repeated up to $nC_{\max}$ times, where $n = |V|$.

At every iteration of the external cycle, we perform one Extract-Min operation and at most one Increase-Label operation for every vertex that can be reached from the current vertex. From above we know that the cost of the labels declared as permanent is not decreasing and that every temporary label in Q is declared permanent exactly once. Thus, for every cost $0 \leq c \leq C_{\max}$, we can have at most one Increase-Label operation per edge in E. In overall, the running time of Algorithm 1 is $\mathcal{O}((n + m)C_{\max} \cdot \log(nC_{\max}))$, where $m = |E|$.

5.3.1 Implementation details

In this section we give implementation details of the label propagating algorithm. In particular, we discuss our choice of data structures, and existing alternatives to this.

Recall that the input to the algorithm consists of a graph $G = (V, E)$ with a function c that indicates a cost for each edge, a source vertex s, a target vertex t, and a value $C_{\max}$ that specifies the upper bound on the cost of the admitted path/walk. The graph G is stored as a vector of adjacency lists, where the i-th list corresponds to the neighbors of the vertex v_i and for each of the neighbors the list also contains the weight/cost of the corresponding edge.

The label propagating algorithm, as described above, requires a data structure Q that has properties of a priority queue and additionally allows updating the values of the stored elements in the described way. In other words, we need a data structure that performs efficiently the elementary operations update, insert, and extract minimum.

A standard way to implement a priority queue in a Dijkstra-like algorithm is using a heap (for example, a Fibonacci heap that gives better asymptotic running time than other heaps). However, since in our algorithm the size of the queue is not known upfront and we need to search for specific keys in the queue efficiently, implementing the queue as a standard heap is not a right choice in our case.

For our purposes a more suitable option is to implement the priority queue using a self-balancing binary search tree, for example an AVL tree. An AVL tree is a data structure that stores elements as key-value pairs and allows to search according to its keys. To specify what we use as keys and values in this search tree, we first explore the needs of the algorithm in more detail. Recall that the algorithm works with labels (c_v, v, n_v), and the data structure Q that holds the labels needs to support mainly the operations Extract-Min and Increase-Label that in effect consist of the following operations:

- insert (c_v, v, n_v) into Q,
- update the value n_v in a triple (c_v, v, n_v) that is already present in Q,
- extract the lexicographical minimum label from Q.

These operations lead to the following principal options for key-value pairs in Q.

i) The first variant is to use the pairs (c_v, v) as keys (c_v as a primary key, and v as a secondary key), and the corresponding n_v as values.

ii) In the second variant, the keys are only the costs. The value corresponding to a cost c is then a data structure S holding key-value pairs (v, n_v) that correspond to labels (c_v, v, n_v) with $c_v = c$. Some of the options for this inner data structure S are:
   - (Self-balancing binary) search tree, AVL for example. This might be a good choice especially in case there are many labels with the same cost, since this allows for efficient search.
   - Linked list or other simple data structure with no overhead for the cases where only a couple of labels share the same cost value.

To compare the variants i) and ii), we first assume that each cost c appears in approximately the same number of labels. We then observe that the running time of an operation will be roughly the same for the two variants:

i) In the first variant, the running time is $\mathcal{O}(\log k)$, where k is the number of (c, v) pairs.

ii) In the second variant, the running time is $\mathcal{O}(\log a + \log b_c)$, where a is the number of different costs and b_c is the number of (c, v) pairs for a cost c. We get $\mathcal{O}(\log a + \log b_c) = \mathcal{O}(\log a \cdot b_c) = \mathcal{O}(\log k)$, where the last equality follows from the assumption.

In the case the assumption does not hold, the worst case running time of an operation in the second variant is at most twice as bad as in the first variant, since the size of both outer and inner tree is bounded by k. Thus, the running time is asymptotically the same for both variants.

The second variant might be more space-efficient. This will depend mostly on the number of different entries per cost value.

Note that we assume the costs of the edges in G to be positive, so a cost c appearing in an operation Increase-Label triggered by a label (c_v, v, n_v) is always strictly greater than c_v. Thus, when extracting the minimum, the algorithm always processes consecutively all the entries for one cost. Therefore, the second variant gives better amortized running time for the operation Extract-Min, since to extract all the labels with the same cost c it needs to search the outer tree only once.

We chose to implement the data structure Q as an AVL tree, where each node is keyed by the cost c and a value contains an inner AVL tree that holds key-value pairs (v, n_v) for each label (c_v, v, n_v) with $c_v = c$.

5.3.2 Exact heuristics for speedup

Early elimination of labels from Q The label propagating algorithm at each step extracts from Q a label (c_v, v, n_v) with a minimum cost. The value n_v of this label corresponds to a number of walks from s to v with cost c_v. This label is then "propagated" to the out-neighbors of v, and for each neighbor of v either a new label is added to Q, or a label that is already present in Q is updated. The only cases when a label (c_v, v, n_v) is not propagated to a neighbor w of v is either if v is the target t, or if the cost $c = c_v + c(v, w)$ of the walk extended from v to w is greater than $C_{\max}$.

However, there may be many labels (c_v, v, n_v) in Q that, in some sense, correspond to s-v walks that can never be successfully extended to the vertex t below the cost $C_{\max}$. This will be observed, however, only after they are extended enough to exceed the cost $C_{\max}$. Removing these labels from the data structure Q early leads to a noticeable speedup of the label propagating algorithm.

To identify a label (c_v, v, n_v) that cannot be extended to t, we use an admissible heuristic $h(v, t)$ to estimate the distance from v to t. In other words, we use a similar approach as the A* search algorithm [44]. As the admissible heuristic $h(v, t)$, we use the shortest path from v to t. We calculate the values of $h(v, t)$ for all the vertices v upfront. In particular, we run Dijsktra's algorithm from the vertex t to all other vertices in a graph G', where G' is the input graph G with reversed orientations of the edges. The cost of the shortest path from t to v in G' corresponds then to $h(v, t)$.

Therefore, when a label (c_v, v, n_v) is extracted from Q and propagated to the neighbors, a new label for a neighbor w of v is added to Q only if the corresponding cost c_w is smaller than $C_{\max} - h(w, t)$. Note that this approach does not affect the result in any way, it only speeds up the running time of the algorithm.

Why a straightforward bidirectional search is of no help Another standard technique to speed up Dijsktra-like algorithms is using bidirectional search. The basic idea consists of running the shortest path search simultaneously from both source s and target t. In the case of the classical shortest s-t path problem, a shortest path can be output as soon as the searches from the two directions meet in the middle. In practice, this approach often leads to reducing the branching factor of the search and to speeding up the running time notably.

However, in the case of the label propagating algorithm, this approach seems to offer too little advantage. A straightforward way to apply bidirectional search in this case is to start to propagate the labels from both s and t. When two labels meet at some vertex, this gives us a number of certain walks and we update the result accordingly. The problem here is that since we are not looking for a single path but for all the walks within some cost limit, we cannot remove the two labels that have met from the data structure Q. In fact, we need to propagate both the labels further, as there may be also other ways how to extend the walks they represent. Thus, we do not save any calculation this way; quite the opposite, the algorithm needs to perform twice as much work as before.

The situation is slightly better with the use of the admissible heuristic $h(v, t)$ to identify non-extendable labels. There, we can stop the calculation already when the search from one direction is finished, but this hardly offers any advantage.

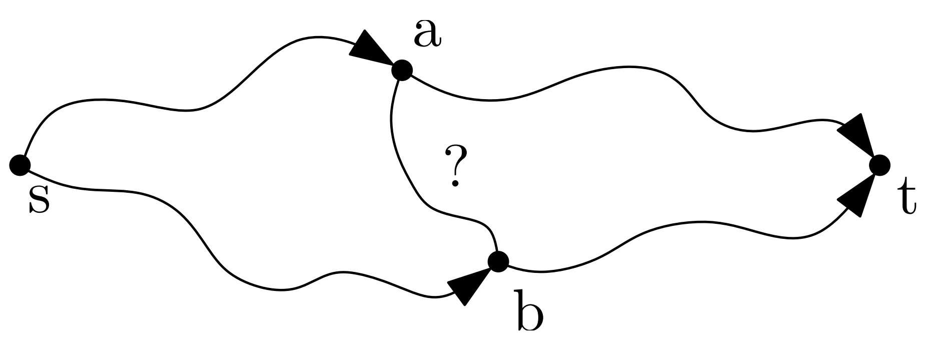


Figure 15: Some edges have implicit orientation if we know s and t, even if the edges of the transportation network are not directed.

5.4 Future Directions

The main difficulty for counting simple paths in graphs is to detect and avoid cycles. In the following, we propose two ways to deal with this issue.

The first one is to consider networks that are acyclic by construction, or that are made acyclic artificially. For this particular class of graphs, we propose an algorithm that computes the number of simple paths between two vertices of the network with arbitrary precision.

The second one is to extend the proposed algorithm in order to detect cycles and avoid the propagation of labels in case a cycle has been detected. This can be done by spending some additional effort during the computation.

What is shown in this section is ongoing work, and will be developed further during the following months of the project.

5.4.1 Shortest paths on DAGs

In the previous sections we have shown that it is difficult to count exactly shortest simple paths in a general graph. A large part of the difficulty seems to stem from the fact that the graph can contain cycles. We need to keep track of them if we are searching for simple paths, but they make it difficult to order vertices so as to facilitate a dynamic programming approach even if we try to count paths that may contain loops.

It is however possible that we are making the problem more difficult than it really is. For instance, it seems unlikely that any reasonably short path would contain a long loop. Consider the s–t path problem in Figure 15. Some of the paths have obvious implicit orientation. No reasonably optimal path would take any of the edges in direction opposite to the denoted orientation. On the other hand, there might be edges with no implicit orientation, such as the edge from the vertex a to the vertex b. If we could find a heuristic that orients a large number of edges in a intuitively reasonable way, leaving only a constant number of un-oriented edges, we could try all possible orientations of these and count short paths on a directed acyclic graph instead. Alternatively, we might be able to devise algorithms that count more effectively in parts on highly oriented parts of the graphs and less effectively elsewhere.

In this section we explore how to count approximately short paths in a directed acyclic graph more efficiently, than our existing algorithms for general graphs. Before showing how to do this, we will introduce a related problem.

Counting small sums in $X_1 + X_2 + \ldots + X_n$ Let $X_1, \ldots, X_n$ be sets of integers. Given a value S, we are interested in the number of ways one can choose $(x_1, x_2, \ldots, x_n) \in X_1 \times X_2 \times \ldots \times X_n$ such that $\sum_i x_i < S$. This is a variant of a problem that was introduced by Mizoguchi and Johnson

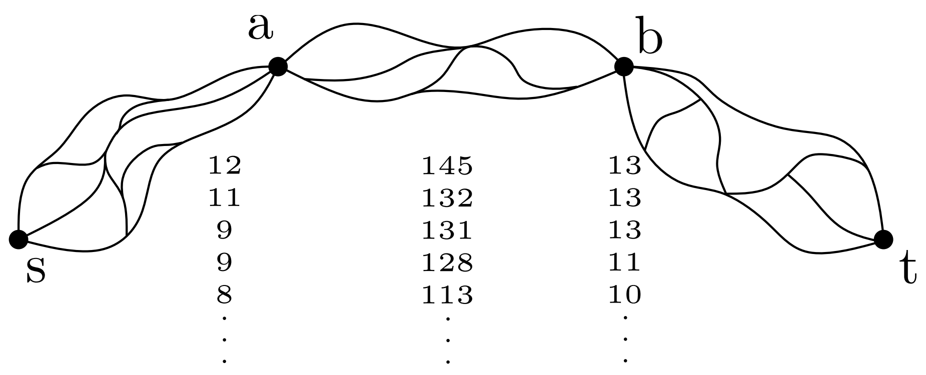


Figure 16: Precomputed lengths of paths from s to a, from a to b, and from b to t.

[49]. They asked for the element with K-th smallest sum from $X_1 + X_2 + \ldots + X_n$ and showed that this problem is NP-complete.

Having an efficient algorithm would allow us to combine pre-computed parts of the shortest path problem under uncertainty. Consider for instance the network in Figure 16. If we know that any s to t path has to go through the vertices a and b (for instance local highway exits), we can pre-compute the lengths of all paths and store the lengths of all s to a paths as the set X_1, a to b paths as the set X_2, and b to t paths as the set X_3 and calculate the number of ρ-optimal s to t paths by counting sums in $X_1 + X_2 + X_3$.

NP-completeness reduction It is perhaps unsurprising that counting the number of approximate solutions for both problems is NP-complete. We can reduce the decision version of the partition problem to the problem of counting sums in $X_1 + X_2 + \ldots + X_n$. Given a set of positive integers $S = s_1, \ldots, s_n$, the partition problem asks if there is a partition of S into sets S_1 and S_2 such that the sums of numbers in both sets are equal. We let the sets X_i for the problem of counting sums in $X_1, X_2, \ldots, X_n$ be $X_i = \{-s_i, s_i\}$. The partition problem on the set S has a solution, if we can achieve a choice of $x_i \in X_i$ (each x_i being equal to s_i or $-s_i$), such that the sum of x_i is equal to 0. If we can efficiently count sums in $X_1 + X_2 + \ldots + X_n$, then we can count the number of solutions when the sum of $x_i \in X_i$ is 0 and when it is 1. If these two counts are not equal, there must exist a solution with the sum of 0 and the answer to the partition problem is "yes". Otherwise the answer is "no". Since the partition problem is known to be weakly NP-complete, the $X_1 + X_2 + \ldots + X_n$ problem is weakly NP-hard. It is easy to see that we can reduce the partition problem to the problem of counting paths in a DAG in the same manner, if the graph is a chain of vertices $v_1, \ldots, v_n$ with multiple edges between vertices v_i and v_{i+1}.

FPTAS for directed acyclic graphs We showed that we cannot count ρ-approximate shortest paths in directed acyclic graphs exactly in polynomial time. We can, however, approximate this number with arbitrary precision in polynomial time. We will sketch the method and algorithms here. A more complete exposure along with proofs of theorems can be found in the technical reports series of eCOMPASS [61].

We first show a recurrence that can be used to exactly count the number of paths that approximate the shortest path in a graph with single set of edge weights within some multiplicative threshold ρ, i.e. the size of the approximation set $A_\rho(I)$. Evaluating the recurrence takes exponential time, but we will later show how to group partial solutions together in such way that we trade accuracy for the number of recursive calls. We adapt the approach of Stefankovic et al. [74], which they used to approximate the number of all feasible solutions to the knapsack problem.

Let G be a directed acyclic graph with n vertices. We will label the vertices $v_1, \ldots, v_n$ in such order that there is no path from v_i to v_j unless $i < j$, i.e. $v_1, \ldots, v_n$ defines a topological ordering. We suppose that $v_1 = s$ and $v_n = t$, otherwise the graph can be pruned by discarding all vertices that appear before s and after t in the topological order, since no path from s to t ever visits these.

For a concrete vertex v_i with in-degree d, let us denote its d neighbors that precede it in the topological order by $p_1, \ldots, p_d$ and let us denote the corresponding incoming edge lengths by $l_1, \ldots, l_d$. Instead of asking for the number of s-t paths that are shorter than L for a given L, we indirectly ask for smallest threshold L, such that there are at least a paths from s to t, shorter than L. Let $\tau(v_i, a)$ denote the minimum length L such that there are at least a paths from v_1 to v_i of length at most L. $\tau(v_i, a)$ can be computed by the recurrence

$$\begin{aligned}
&\tau(v_1, 0) = -\infty \\
&\tau(v_1, a) = 0, \forall a : 0 < a \leq 1 \\
&\tau(v_1, a) = \infty, \forall a : a > 1 \\
&\tau(v_i, a) = \min_{\substack{\alpha_1, \ldots, \alpha_d \\ \sum \alpha_j = 1}} \max \begin{cases} \tau(p_1, \alpha_1 a) + l_1 \\ \vdots \\ \tau(p_d, \alpha_d a) + l_d \end{cases} .
\end{aligned}$$

Intuitively, the at least a paths starting at v_1 and arriving at v_i must split in some way among incoming edges. The values α_j define this split. We look for a set of $\alpha_1, \ldots, \alpha_d$ that minimizes the maximum allowed path length needed such that the incoming paths can be distributed according to α_j, $j = 1, \ldots, d$.

To find the number of paths of length at most L, we search for a such that $\tau(v_n, a) \leq L < \tau(v_n, a + 1)$. In particular, if the length of the shortest s-t path is OPT, we can find the number of ρ-approximate s-t paths by setting $L := \rho OPT$.

Calculating τ using the given recurrence will not result in a polynomial time algorithm since we might need to consider an exponential number of values for a, namely 2^{n-2} on a DAG with maximal number of edges. To overcome this, we will consider only a polynomial number of possible values for a, and always round down to the closest one in the evaluation. If we are looking for an algorithm that counts with $1 + \varepsilon$ precision, the ratio between two successive considered values of a must be at most $1 + \varepsilon$.

For this purpose, we introduce a new function τ'. In order to achieve precision of $1 + \varepsilon$, we will only consider values of τ' for minimum path numbers in the form of q^k for all positive integers k such that $q^k < 2^{n-2}$, where $q = \sqrt[n+1]{1 + \varepsilon}$. The values of τ' for other numbers of paths will be undefined. The function τ' is defined by the following recurrence.

$$\begin{aligned}
&\tau'(v_1, 0) = -\infty \\
&\tau'(v_1, a) = 0, \forall a : 0 < a \leq 1 \\
&\tau'(v_1, a) = \infty, \forall a : a > 1 \\
&\tau'(v_i, q^j) = \min_{\substack{\alpha_1, \ldots, \alpha_d \\ \sum \alpha_j = 1}} \max \begin{cases} \tau'(p_1, q^{\lfloor j + \log_q \alpha_1 \rfloor}) + l_1 \\ \vdots \\ \tau'(p_i^d, q^{\lfloor j + \log_q \alpha_d \rfloor}) + l_d \end{cases}
\end{aligned} \tag{33}$$

To give a meaning to the expression $q^{\lfloor j + \log_q \alpha_i \rfloor}$ when $\alpha_i = 0$, we define it, for our purposes, to be equal to 0, which is consistent with its limit when α_i goes to 0. We can show that the rounding does not make the values of τ' too different from the values of τ, namely that $\tau(v_i, q^{j-i}) \leq \tau'(v_i, q^j) \leq \tau(v_i, q^j)$.

We will then evaluate the function τ' and find k such that $\tau'(v_n, q^k) \leq L < \tau'(v_n, q^{k+1})$. We can show that the value q^k will be a $(1 + \varepsilon)$-approximation of the "correct" value a, for which $\tau(v_n, a) \leq L < \tau(v_n, a + 1)$.

The time necessary to do this is $O(mn^2\varepsilon^{-1}\log n)$. There are only $O(n^2\varepsilon^{-1})$ many different values of q^k that we need to consider. The rough idea is that we calculate, for each vertex v_i, all the values of $\tau'(v_i, q^k)$ for different k in one go, by progressively increasing the values $\alpha_1, \ldots, \alpha_d$ in the recurrence.

FPTAS for solutions that approximate two instances We can extend the result to an algorithm that counts solutions that are ρ-approximate for two instances at the same time. We will define a function τ_2 similar to τ that adds one of the edge lengths in a form of a "budget". $\tau_2(v_i, a, L_1)$ will be equal to the shortest length L_2 with respect to the edge lengths in the second instance such that there are at least a paths from v_1 to v_i, no longer than L_1 with respect to the edge lengths in the first instance. We will denote the edge lengths of the d incoming edges of vertex v_i in the second instance by l'_i. τ_2 can then be evaluated by the following recursion.

$$\begin{aligned}
&\tau_2(v_1, 0, x) = -\infty, \forall x \in \mathbb{R}^+\\
&\tau_2(v_1, a, x) = 0, \forall a : 0 < a \leq 1, \forall x \in \mathbb{R}^+\\
&\tau_2(v_1, a, x) = \infty, \forall a : a > 1, \forall x \in \mathbb{R}^+\\
&\tau_2(v_i, a, L_1) = \min_{\substack{\alpha_1,\ldots,\alpha_d\\ \sum \alpha_j = 1}} \max \begin{cases} \tau_2(p_1, \alpha_1 a, L_1 - l_1) + l'_1\\ \vdots\\ \tau_2(p_d, \alpha_d a, L_1 - l_d) + l'_d \end{cases}
\end{aligned}$$

If used to solve the problem, the function τ_2 would have to be evaluated not only for an exponential number of path counts a but also for a possibly exponential number of values of L_1. To end up with polynomial runtime, we need to consider only a polynomial number of values for both. We will introduce a function τ'_2 that does this by considering only path lengths in the form of r^k, where $r = \sqrt[n]{1+\delta}$, and path numbers a in the form of q^j, where $q = \sqrt[n]{1+\varepsilon}$, for positive ε and δ.

$$\begin{aligned}
&\tau'_2(v_1, 0, x) = -\infty, \forall x \in \mathbb{R}^+\\
&\tau'_2(v_1, a, x) = 0, \forall a : 0 < a \leq 1, \forall x \in \mathbb{R}^+\\
&\tau'_2(v_1, a, x) = \infty, \forall a : a > 1, \forall x \in \mathbb{R}^+\\
&\tau'_2(v_i, q^j, r^k) = \min_{\substack{\alpha_1,\ldots,\alpha_d\\ \sum \alpha_j = 1}} \max \begin{cases} \tau'_2(p_1, q^{\lfloor j + \log_q \alpha_1 \rfloor}, r^{\lfloor \log_r (r^k - l_1) \rfloor}) + l'_1\\ \vdots\\ \tau'_2(p_d, q^{\lfloor j + \log_q \alpha_d \rfloor}, r^{\lfloor \log_r (r^k - l_d) \rfloor}) + l'_d \end{cases}
\end{aligned}$$

We can again show that τ'_2 approximates τ_2, this time in both variables. Compared to the single variable version, we cannot get a $(1+\varepsilon)$ approximation to the optimal value, because we need to round the value of L_1 to a power of r in order for it to be legal parameter of τ'_2 and we further round it during the evaluation of τ'_2. We will therefore relate the result of τ'_2 to the results of τ_2 we would have gotten if we considered the value of L_1 when rounded up towards the nearest number that can be represented as r^k for integer k and the value r^{k-n}. Due to the choice of r, the ratio of these two values is $1+\delta$.

We will look for a q^k such that $\tau'_2(v_n, q^k, r^{\lceil \log_r L_1 \rceil}) \leq L < \tau'_2(v_n, q^{k+1}, r^{\lceil \log_r L_1 \rceil})$, and this q^k will be by a factor of at most $(1+\varepsilon)$ different from the number that is correct for some L'_1, which is different from the "true" value L_1 by a factor of at most $1+\delta$. By an analysis similar to the one for a single variable, we can show that the running time of our algorithm is $O(mn^3\varepsilon^{-1}\delta^{-1}\log n \log L_1)$, where m denotes the number of edges in the graph.

FPTAS for counting small sums in $X_1 + X_2 + \ldots + X_n$ The modification of the approximation schemes for paths on directed acyclic graphs into approximation schemes for counting sums in

$X_1 + X_2 + \ldots + X_n$ is analogous to the NP-hardness reduction from this section. We transform the set of sets X_i into a directed acyclic graph with $n+1$ vertices and $m := \sum_i |X_i|$ edges. We can therefore calculate the ε-approximation to the number of sums smaller than some value v in time $O(mn^2\varepsilon^{-1}\log n)$ for a single instance, and in time $O(mn^3\varepsilon^{-1}\delta^{-1}\log n \log L_1)$ for two instances, if we allow multiplicative error of $(1+\delta)$ for v.

5.4.2 Cycle avoidance

The label propagating algorithm proposed above also considers walks that contain cycles, and we would like to avoid such walks. In this section we describe a method to extend the algorithm to avoid walks containing cycles up to a given length (i.e., the number of edges). The algorithm is especially efficient when the maximum degree of the input graph is bounded by some small constant d and when the cycles to be avoided are short. Assuming a bounded degree graph as an input is reasonable in practice, where the input graphs, representing road networks for example, usually have this property.

When we propagate a label (c_v, v, n_v) to an out-neighbor w of v, we create a new label (c_w, w, n_w). However, the label (c_v, v, n_v) aggregates several s-v walks, and by extending them to w we may create a cycle in some of these extended walks. We would like to identify every such walk and to exclude it from the label (c_w, w, n_w). In other words, we would like to decrease the number n_w by 1 for each such s-w walk that ends with a cycle.

Unfortunately, counting the number of simple s-t paths is #P-hard, thus avoiding all such walks with cycles is a difficult task. We focus on avoiding walks that contain short cycles, whose length is at most k. When k is a constant, the running time of the whole algorithm increases only by a constant factor. From a practical point of view, we believe that this would already make a difference, since in typical road networks the number of big cycles is usually much smaller than the number of small cycles.

For avoiding cycles of length at most k in a single s-v walk, we could remember the last k vertices and then, when extending to a vertex w, we would only check if w appears in the set of the currently stored vertices. However, applying this approach to the labels would cause increasing the number of labels, as each label typically aggregates several walks. Instead, we propose to decompose the label that we want to propagate into groups of walks, where each group corresponds to walks that for the last k steps followed the same path. The decomposition of the label is not stored but made on demand at the time of propagating the label. Then, by checking the last k vertices of each such group, we can identify those groups of walks for which extending to the vertex w creates a cycle.

More formally, let (c_v, v, n_v) be a label we want to propagate, let w be an out-neighbor of v that is currently considered to extend the walks aggregated in (c_v, v, n_v), and let n'_w denote the number of these walks that after extending to w end with a cycle of length at most k. Initially, we set $n'_w = 0$. Assume that we have stored all the labels previously extracted from the data structure Q in some data structure S, with a time $T(S)$ necessary to access an entry of S. In other words, Q contains temporary labels, and S contains permanent labels. The data structure S contains each label (c_v, v, n_v) as a key-value pair, (v, c_v) being the key and n_v the corresponding value.

In order to calculate how many of the walks aggregated in (c_v, v, n_v) and extended to w contain a newly created cycle, we decompose (c_v, v, n_v) iteratively as follows. We determine the set of labels that are direct predecessors of (c_v, v, n_v)—we say that a label (c_u, u, n_u) is a direct predecessor of (c_v, v, n_v) if u is in-neighbor of v and $c_u = c_v - c(u, v)$. This can be done by iterating over the in-neighbors of v: for each in-neighbor u of v we calculate the cost $c_u = c_v - c(u, v)$ and obtain the corresponding number of walks n_u by querying the data structure S, thus we obtain (c_u, u, n_u), a direct predecessor of v (or not, if the label is not in S). Afterwards, for each of the predecessors we determine its predecessors. We continue up to depth k. Then, for each label (c_u, u, n_u) visited this way we check whether the vertex u is the same as w. If this is the case, then all the walks aggregated in (c_v, v, n_v) that correspond to those extended from (c_u, u, n_u) will contain a cycle if they are further extended to w, so we add n_u to n'_w. Thus, at the end of this process n'_w will correspond to the number of walks extended from (c_v, v, n_v) to w that contain a cycle shorter than k ending in w.

We modify the label propagating algorithm as follows. At each step, after we extract a label (c_v, v, n_v) from Q, we determine the set P of label predecessors of (c_v, v, n_v) up to depth k. Then, for each out-neighbor w of v, for which $c_v + c(v, w)$ is at most $C_{\max}$, we determine the number n'_w of walks aggregated in (c_v, v, n_v) whose extending to w creates a short cycle: $n'_w = \sum_{(c_u,u,n_u)\in P, \text{ with } u=w} n_u$. Next, if the label for the vertex w and the cost $c_w = c_v + c(v, w)$ is already in Q, we update the corresponding n_w by adding $n_v - n'_w$ to it. Otherwise, we create a new label (c_w, w, n_w) with $c_w = c_v + c(v, w)$, and $n_w = n_v - n'_w$ and insert it to Q. After all the out-neighbors of v are processed, the label (c_v, v, n_v) is inserted to the data structure S.

In terms of running time, we need additional $O(d^k \cdot T(S))$ time per label to perform the described modification. Since in the basic label propagating algorithm we only needed to store the temporary labels in Q and we could forget the temporary labels as soon as they were processed, we need additional space to store the data structure S.

The algorithm is not yet implemented, but a hash table is a reasonable candidate for the data structure S.

Remembering the short cycles One can observe that some cycles will appear repeatedly during the described process. In the case of short cycles, this is likely to happen relatively often. Thus, a natural effort is to try to store and reuse these cycles.

Having in mind the above described modification of the label propagating algorithm, we observe the following. When we are at the situation of propagating a label (c_v, v, n_v) to an out-neighbor w and calculating the number n'_w of those walks in (c_v, v, n_v) that will create a short cycle, we considered the set P of all the label predecessors of (c_v, v, n_v) up to depth k. However, next time when we are in a similar situation of propagating a label (c^*_v, v, n^*_v) from the vertex v to the out-neighbor w, the set P^* of label predecessors will be in a certain sense similar to the previous set P. In particular, there is the following one to one mapping between the two sets: $(c_u, u, n_u) \in P \mapsto (c^*_u, u, n^*_u) \in P^*$ if and only if $c_v - c_u = c^*_v - c^*_u$. This comes from the fact that the short paths passing through the vertices u and v do not change in time, i.e., the length and the cost of such a path is the same both times. Similarly, also the set of cycles passing through u and v is always the same.

Next, we realize that in order to calculate the number n'_w, we only need to query the data structure S for those label predecessors (up to depth k) where the vertex w appears, that is those that by extending via v to w give a cycle. Therefore, together with the previous observation, to calculate n'_w the next time we propagate from a label that contains v, we only need to know the relative cost $c = c_v - c_u$ for each label predecessor (c_u, u, n_u) of (c_v, v, n_v), where $u = w$. From that we can reconstruct the corresponding label predecessors directly, query the data structure S for them, and calculate n'_w in the described way.

Thus, for a pair of vertices v, w we can store a list of all relevant relative costs, and we will never need to explore the label predecessors again, when propagating from a label with v to the vertex w. The shorter the list is, the less additional space is needed to store it, and the more computational time is saved. In particular, depending on the length of the list, we need at most $O(d^k)$ additional space to store one such list; and one such list saves up to $O(d^k)$ operations every time we propagate from v to w.

Further tweaks Obviously, the above explained approach provides us with a lot of trade-off options to explore. One possibility is to determine these lists of relative costs for each pair of vertices. This could be done even upfront, before we start the process of counting the number of s-t paths/walks. However, this may not be the best option, as a lot of additional space would be needed. Moreover, there might be vertices that will never be considered in the computation, since they are too far apart from all the reasonable s-t paths, and thus the information stored for them is never used. Another possibility is then to determine these lists only on demand, whenever necessary, but then store them for a later use. Also, we may consider storing only those lists that are reasonably short, since they take less space and at the same time offer bigger advantage in terms of saved running time.

5.4.3 Other Directions

Future work will be focused mainly on improving the time needed to answer a query for a robust route. For this goal, a promising direction is to develop a good preprocessing techniques for the *shortest path problem under uncertainty.*

Query structures Since a main bottleneck for the computation of robust routes is to count approximate paths, we can inspect whether well-known preprocessing techniques for the shortest path problem can be extended for counting. However, preliminary investigations suggest that this cannot be done efficiently. The main difficulty is that typical preprocessing techniques for shortest path queries involve the partition of the vertices of the graph into a hierarchy. Queries are then answered by performing a search from the source and a reverse search from the target upward or downward in the hierarchy until the two searches meet. This is true both for techniques that are good theoretically, and for those used in practice. When it comes to counting approximate paths, though, the search criteria that are used for shortest paths do not apply. In this case, a hierarchical structure built on top of a road network may not be useful at all.

Alternative routes The second approach to speed-up the computation of robust routes is to consider only a fixed set of feasible s-t paths, and to count and use only approximate solutions that lie in this set. Since one of the goals of Task 2.2 of eCOMPASS is to compute meaningful alternatives routes for users, we could use these alternatives as the starting set of routes in which we look for the most robust one. At the moment of writing it is not clear if this could be done efficiently, and whether we could expect competitive results when compared to the solution obtained by solving the *shortest path problem under uncertainty* from scratch.

Approximation and Randomization The third direction for speeding-up the computation of robust routes is to consider approximation algorithms and randomized algorithms. For some problems involving counting, there exist techniques that can approximate the correct answer, or that can find the correct answer with good probability. It is interesting to inspect whether this techniques can be applied to the *shortest path problem under uncertainty* as well.

6 Fleets-of-Vehiles Route Planning

6.1 Problem statement and Preliminaries

One important aspect of the eCOMPASS project is route planning for fleets of vehicles. In this problem there are given: a set of customers and the demand of each customer, a time window associated with each customer, a depot, a fleet of vehicles and a cost measure (in our case distance and time) for traveling from customer i to customer j. Each customer is also associated with a quantity of goods that needs to be delivered. A time window is a time interval with an earliest arrival time that a vehicle can begin serving the customer and a latest arrival time after which serving is no longer possible. For a formal definition of time windows see Section 6.3.1. A *cluster* is a group of customers with compatible time windows. This means that if a vehicle serves a customer i in a cluster, it can also serve a customer j that belongs to the same cluster. The goal is to create routes (tours) which start and end at the depot, serve all customers and minimize the total traveling distance (or time) of the vehicles.

For eCOMPASS there is an additional objective: the routes created have to be environmentally friendly (e.g. minimizing fuel used, CO_2 emissions etc . . .). In order to do so, compact and balanced clusters need to be created which lead to eco-friendly routes. A cluster C is called *compact* if for every pair of customers $i, j \in C$ there is a way (through the road network) to reach customer j from customer i and respect customer's j time window. In other words, a vehicle that visits cluster C can reach all customers that belong to this cluster. Recall that each customer expects a quantity of goods to be delivered. So, the *capacity of a cluster* is defined as the sum of all customers' goods that belong to this cluster. Moreover, two clusters C_i, C_j are called *balanced* if $T_i \approx T_j$ where T_i, T_j is the total capacity of cluster i and j, respectively. If the goals of compactness and balance are met, then they lead to eco-friendly routes in an implicit way. Eco-friendliness is achieved due to the fact that all created routes are similar in terms of the load of each vehicle (all vehicles' load is even). Each vehicle has a maximum capacity Q and a vehicle's *load* ld is a number in $[0, Q]$. Furthermore, each vehicle that serves a cluster C can reach all customers that belong to this cluster due to its construction. Thus, a vehicle will not spend additional resources (fuel, time) traveling back and forth to the depot because some customers were unreachable.

6.2 Related Work

The problem of finding routes (starting and ending at a depot) that serve a set of customers and minimize costs is known in the literature as the Vehicle Routing Problem (VRP). In its simplest form, there are given: a depot, a fleet of vehicles and a set of customers. The goal is to find routes (tours) that start and end at the depot, service all customers and minimize the total cost of the route. The cost of the route could be: total traveling distance, total traveling time or a combination of distance/time. These are the most common measures of cost studied in the literature and in real-life examples. The VRP is an important problem in the fields of transportation, distribution and logistics with many applications.

Since the introduction of VRP many variants have been introduced such as the Capacitated VRP (CVRP) in which a homogeneous fleet of vehicles is available and the only constraint is the vehicle capacity, or the VRP with Time Windows (VRPTW) in which each customer must be served within a specific time interval. Recently, much attention has been devoted to more complex variants of VRP known as "rich" VRPs (RVRPs) that are closer to real-life problems. In particular, rich VRPs take into account one or more depots, (multiple) time windows for each customer, multiple vehicle types, loading constraints, multiple tours for each vehicle and capacity constraints for each vehicle. Although rich VRPs capture real life scenarios, they are more complicated than other variants (such as CVRP), hence, are more challenging to solve.

VRP and its many variants have been studied since the problem was first introduced by Dantzig and Ramser [19] in 1959. The Traveling Salesman Problem (TSP) is a subproblem of VRP, known to be NP-Hard. This means it is unlikely that exact solutions to real life instances of the VRP can be computed quickly. The most common ways of overcoming this hurdle is by using heuristics, metaheuristics, and approximation algorithms. We refer the reader to the book edited by Toth and Vigo [75] for a comprehensive overview of many techniques used for solving VRPs.

Many heuristics and metaheuristics have been used to solve variants of the VRP. The heuristics can be roughly classified into *construction* heuristics and *improvement* heuristics. As the name suggests, a construction heuristic is used to construct initial or candidate tours. These tours are then improved by an improvement heuristics. The classical construction heuristics are the savings based method of Clarke and Wright [17] and the insertion heuristic [48]. Other methods like the two phase method of Fisher and Jaikumar [35] are also widely used. Among the improvement heuristics, the methods of [52] and [54] are well known and used.

Since almost a decade now the emphasis of research has been gradually shifting towards real life VRPs (RVRPs). For those we refer the interested reader to the survey article of Drexl [29].

For the approach adopted in this project, we studied the literature in depth, e.g., [14],[28],[73]. A general comment is that in related work, many researchers focus on the creation of clusters, due to the complexity of the VRP problem. In [14] the authors develop a clustering method, creating balanced clusters. They use the k-means algorithm in order to create clusters and suggested an improved version of the k-means algorithm. In [28] the authors also create clusters and then solve a mixed integer linear program (MILP) in order to calculate the actual routes. For the clustering phase they use a heuristic approach. Finally, in [73] the authors describe a variety of heuristics, and conduct an extensive computational study of their performance.

6.3 The eCOMPASS 3-Phase Approach

The model adopted in eCOMPASS is inspired by the "rich" VRP since we are dealing with a set of customers with (multiple) time windows, one depot, a homogeneous fleet of vehicles and further objectives to be met like compactness, balanced and eco-friendly routes.

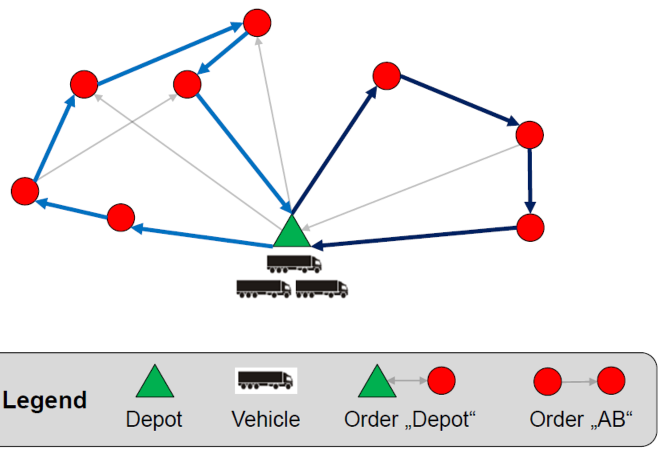


Figure 17: Instance of a VRP Problem

A directed graph $G = (V, E)$ is given where V represents the set of nodes (customers) and E the set of edges. Usually, node v_0 represents the depot and nodes $v_i \in \{1, \ldots, n-1\}$ represent each customer. Every customer $v_i \in V$ requires q_{v_i} units to be served. There is a fleet of m vehicles each associated with a maximum capacity Q. For each edge $(i, j) \in E$ a non negative routing cost c_{ij} is given which represents the cost to travel from customer i to customer j. An example of a

VRP instance is shown in Figure 17. There is one depot (green triangle) and a set of customers represented as red dots. There are two routes represented with bold lines; the grey thin lines were not chosen for any of the two routes.

More specifically, the eCOMPASS approach comprises 3 Phases. Phase I is the Clustering with Time Windows Phase, where the customers are divided into clusters. The goal of Phase I is to create clusters with the following property: a vehicle serving a customer within a cluster can also serve all the other customers in the same cluster. In other words, each cluster forms a strongly connected component not in the real life instance but in a modified graph. The construction of the modified graph is explained in Section 6.3.1. Phase II is the Partition Phase, where the original graph is partitioned into cells. A *cell* is a group of customers that are geographically close. The main idea is that customers that belong to the same cell are geographically close to each other and they may belong to the same final cluster if their time windows are compatible. Phase III is the Merge & Split Phase, where the previously created clusters and cells are merged together or split in order to form the final clusters.

6.3.1 Phase I - Clustering with Time Windows

In this phase a graph $G = (V, E)$ is created. Every customer i is represented by a node and is associated with a time window $[e_i, l_i]$ where e_i is the earliest arrival time at customer i and l_i is the latest departure time from customer i. For two customers (nodes) v_i, v_j variable t_{ij} denotes the traveling time needed to travel from customer i to customer j in seconds and variable d_{ij} denotes the distance between nodes i and j in meters. An edge e_{ij} connects nodes i, j if $l_i + t_{ij} < l_j$. The inequality shows that when a vehicle serves a customer i and leaves at the customer's latest departure time, it can reach customer j taking into account the time needed to travel from i to j respecting customer's j latest departure time.

After all edges have been created for all customers the process of creating the clusters can begin. The main idea is to find Strongly Connected Components (SCC) inside the graph G. A *Strongly Connected Component* is a maximal subgraph H of G with the following property: for any two nodes $v_i, v_j \in H$ there is a path from v_i to v_j and also there is a path from v_j to v_i. Each strongly connected component k is then considered a cluster C_k. For every strongly connected component the following property holds: node $v_i \in C_k$ is reachable from any other node v_j that belongs to the same cluster C_k

6.3.2 Phase II - Geographic Partition

The second phase is responsible for the geographical partition of the area. Since we are dealing with instances where each customer is associated with coordinates (longitude,latitude) an instance can be represented on a map by its coordinates. Hence, given an area (usually urban) the main idea is to create a partition of M cells where customers that belong to the same cell are geographically close to each other. The algorithm that performs the partition is the following: given the four outermost points and some parameters describing the height h, width w of each cell and depth d of the partition, the area is partitioned into $l = h * w$ cells. This creates the first level of partition Level 0. Then, the process is repeated d times where d denotes the number of levels that need to be created. The challenge is to experiment with the values of h, w, d because we would like to avoid creating a few cells, because all nodes will be gathered there, and also avoid creating too many small cells as this will lead to many empty cells or cells that have 1 or 2 customers in them. This is a preprocessing step thus it can be executed off-line and not create extra burden for the actual calculation of the routes. An example of the Partition Phase can be seen in Figures 18,19. For simplicity the initial area is represented by a square although this may not be the general case. To conclude, a cell corresponds to a geographical area and its size depends on its depth. For example, cells that belong to Level 0 correspond to a wider geographical area than cells that belong to Level 1. This can be seen on Figures 18,19 where cell 0 is divided into cells 00 through 08.

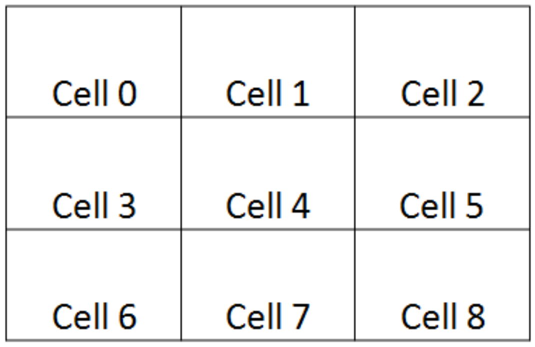


Figure 18: First level of Geographic Partition. An area is divided into 9 cells.

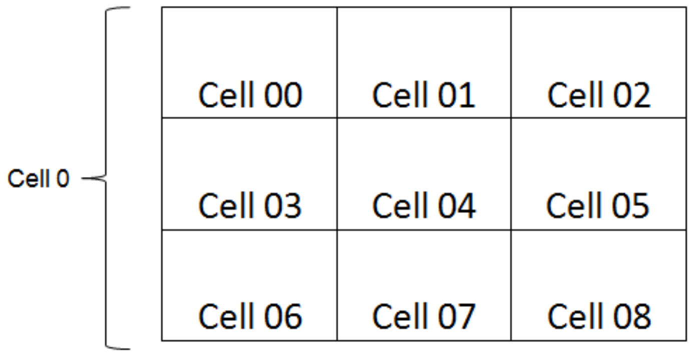


Figure 19: Second level of Geographic Partition. Each cell from the first level is further divided.

6.3.3 Phase III - Cluster Refinement: Merge & Split

The third phase deals with the clusters and cells created from Phases I and II respectively. Recall that Phase I created clusters that achieved a first level of compactness and Phase II created cells in order to get balanced routes. The main idea of Phase III is the refinement of the previous two phases in order to eliminate possible problems. For example, there may exist a cluster C where there is a path connecting any two customers but their time windows are incompatible, some have to be served in the morning and others in the afternoon. This cluster must be split into two (or more, if necessary) sub-clusters that will satisfy compactness (connectivity) and balance (geographical proximity). Another case is that two cells created from Phase II can contain customers that are geographically close and they may have compatible time windows. In this case the two cells have to be merged to create a bigger cell that satisfies the properties of compactness and balance. Also, if there are empty cells from Phase II, they can be merged with their neighbour cells. Then for each final cluster any heuristic or metaheuristic algorithm can be executed in order to calculate the actual routes of the vehicles. The situation is depicted in Figures 20,21. In Figure 20, clusters C_1, C_2 are merged because they are both connected and geographically close, whereas in Figure 21 cluster C_3 is split into two sub-clusters because it contains customers that lay in different geographical areas.

6.4 Future Directions

6.4.1 Cluster Balancing

One issue that is worth studying, is the concept of balanced clusters. Recall that two clusters C_i, C_j are called *balanced* if $T_i \approx T_j$ where T_i, T_j is the total capacity of cluster i and j respectively. Balanced clusters will lead to balanced tours since the average load for each vehicle is the same. Furthermore, balanced tours have a positive impact on environmental issues due to the fact that every vehicle's load is approximately the same. This leads to less CO_2 emissions and less fuel used since there are no cases where one or more vehicles are heavily loaded and others are empty.

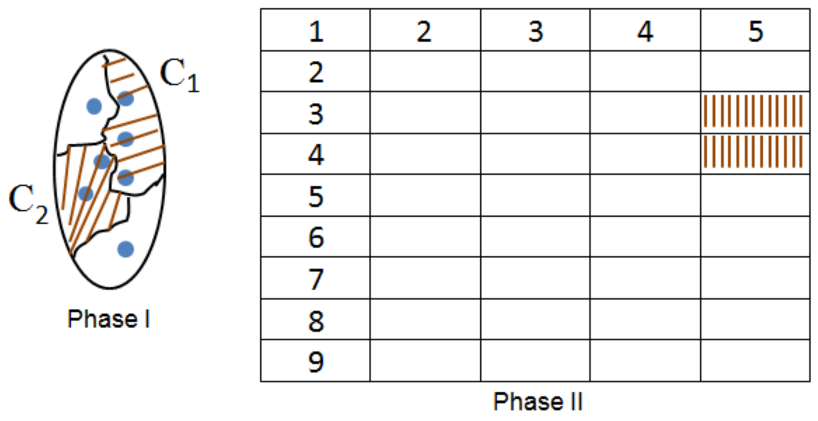


Figure 20: Phase III: Cluster Refinement - Merge Operation. Examine phases I and II and perform a merge operation.

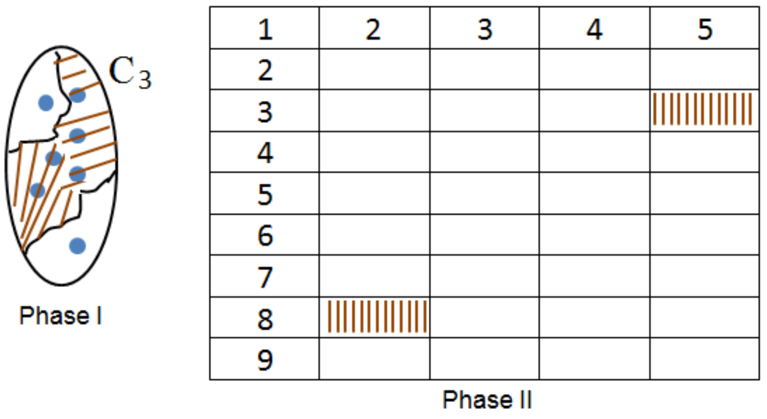


Figure 21: Phase III: Cluster Refinement - Split Operation. Examine phases I and II and perform a split operation.

7 Conlusions and Work for Next Period

In this document we presented the research results obtained by the eCOMPASS project partners in the first 18 months of the project with respect to routing problems for private vehicles and fleet of vehicles in urban areas. What is shown are the algorithms and algorithmic solutions developed for these problems. Starting from the next period the testing stage will begin, and we will experiment on real-world data in order to fine-tune the proposed algorithms for practical use. We will also implement these algorithms on prototypes, and during the pilot test stage we will assess their validity for every-day use.

However, research for even more efficient and precise algorithm will not stop. In parallel with the experimental stage we will continue to research for new solutions and algorithm, and our belief is that research and experiments will benefit one from the other. In the following, we summarize some of the most prominent future plans for each research topic presented in this document.

Traffic Prediction Concerning traffic prediction, the effectiveness of the techniques given for travel time forecasting shall be benchmarked in future deliverables. In any case, forecasting travel times is a quite challenging task, to be explored further by both Machine Learning and Time Series Analysis perspectives. Future scope could lie in clustering approaches (aimed towards regression), that may be applied along with proper feature selection in order to fully exploit the data. Different dimensions of the data can be used to improve on the application of the techniques analyzed in this deliverable, while new methods may also be designed and tested in terms of effectiveness and, possibly, efficiency. Apart from the well-specified problem of travel time forecasting, several tasks, such as traffic congestion and incident detection, shall be considered, since their effect is generally significant, when it comes to routing, either for individuals or fleets. Literature on these topics shall be analyzed and techniques for predicting and visualizing congestion/incidents will be explored so as to draw useful conclusions.

Alternative Route Planning The experiments show that both the plateau and the penalty method provide good results. We plan to evaluate the computed alternatives with respect to the quality indicators. The plateau method forms the alternatives by overlapping the shortest path tree from s with the one from t. The penalty method yields the alternatives iteratively. At each step it finds the shortest path and then penalizes its edges by increasing their weight. In the penalty method, a new technique is developed to prevent the increase of the weight of non-decision edges. Also, we propose to set the value of the penalization factors depending on the length of the shortest path, instead of a constant factor.

Robust Route Planning For the computation of robust routes it turned out that avoiding paths containing cycles is much harder than expected. In the next period we will inspect how to avoid these cycles as well as how to speed-up the proposed algorithms for counting simple paths. Our focus will be on special graph classes that do not contain cycles (DAGs), extending the label propagating algorithm as to avoid cycles, and developing preprocessing techniques for robust routes.

Other possible directions involve computing robust routes among a set of meaningful alternatives. This solution has the potential of decreasing the time required to compute a route significantly, but we will have to inspect the trade-off between the speed-up and the quality of the computed solution. Furthermore, we will consider the possibility to design approximation algorithms or randomized algorithm for counting simple paths.

Fleet-of-Vehicles Route Planning Regarding route planning for fleet-of-vehicles we have completed the 3-phase approach. Early experiments show that the creation of clusters and cells does not require high computation time. In any case, this step can be done off-line and the actual computation of the routes of each cluster will be done by using a heuristic algorithm. In this direction, we plan to use PTV's heuristic algorithms which perform performing really well in practice. As regards next steps, our focus will be towards creating balanced clusters, an element that can make of approach more concrete. In parallel, we will also focus on extending and improving the work conducted so far. As for the experiments, we already use real world instances of VRP problems and we are planning to test our algorithms in more real world instances in order to evaluate their behaviour.

References

[1] Camvit: Choice routing, 2009.

[2] Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. Hierarchical Hub Labelings for Shortest Paths. In Leah Epstein and Paolo Ferragina, editors, *Proceedings of the 20th Annual European Symposium on Algorithms (ESA'12)*, volume 7501 of *Lecture Notes in Computer Science*, pages 24–35. Springer, 2012.

[3] Rachit Agarwal and Philip Godfrey. Distance oracles for stretch less than 2. In *Proceedings of the 24th Annual ACM–SIAM Symposium on Discrete Algorithms (SODA'13)*, pages 526–538, 2013.

[4] Y. P. Aneja and K. P. K. Nair. The constrained shortest path problem. *Naval Research Logistics Quarterly*, 25(3):549–555, 1978.

[5] David A. Bader, Henning Meyerhenke, Peter Sanders, and Dorothea Wagner, editors. *Graph Partitioning and Graph Clustering: Tenth DIMACS Implementation Challenge* . DIMACS Book. American Mathematical Society, 2013. To appear.

[6] Roland Bader, Jonathan Dees, Robert Geisberger, and Peter Sanders. Alternative route graphs in road networks. In *TAPAS*, pages 21–32, 2011.

[7] Gernot Veit Batz, Daniel Delling, Peter Sanders, and Christian Vetter. Time-Dependent Contraction Hierarchies. In *Proceedings of the 11th Workshop on Algorithm Engineering and Experiments (ALENEX'09)*, pages 97–105. SIAM, April 2009.

[8] Gernot Veit Batz, Robert Geisberger, Sabine Neubauer, and Peter Sanders. Time-Dependent Contraction Hierarchies and Approximation. In Festa [34], pages 166–177.

[9] George Edward Pelham Box and Gwilym Jenkins. *Time Series Analysis, Forecasting and Control.* Holden-Day, Incorporated, 1990.

[10] Leo Breiman. Random forests. *Mach. Learn.*, 45(1):5–32, October 2001.

[11] Peter J. Brockwell and Richard A. Davis. *Introduction to Time Series and Forecasting.* Springer, 2nd edition, March 2002.

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

[13] Christina Büsing. Recoverable robust shortest path problems. *Networks*, 59(1):181–189, 2012.

[14] Buyang Cao and Fred Glover. Creating balanced and connected clusters to improve service delivery routes in logistics planning. *ISSN: 1004-3756*, DOI: 10.1007/s11518-010-5150-x.

[15] Yanyan Chen, Michael G. H. Bell, and Klaus Bogenberger. Reliable pretrip multipath planning and dynamic adaptation for a centralized road navigation system. *Trans. Intell. Transport. Sys.*, 8(1):14–20, March 2007.

[16] Tao Cheng, James Haworth, and Jiaqiu Wang. Spatio-temporal autocorrelation of road network data. *Journal of Geographical Systems*, 14(4):389–413, 2012.

[17] G. Clarke and J. V. Wright. Scheduling of vehicles from a central depot to a number of delivery points. *Operations Research*, 12:568–581, 1964.

[18] K. Cooke and E. Halsey. The shortest route through a network with time-dependent intermodal transit times. *Journal of Mathematical Analysis and Applications*, 14(3):493–498, 1966.

[19] G. B. Dantzig and J. H. Ramser. The truck dispatching problem. *Management Science*, 6:80, 1959.

[20] Daniel Delling. Time-Dependent SHARC-Routing. In *Proceedings of the 16th Annual European Symposium on Algorithms (ESA'08)*, volume 5193 of *Lecture Notes in Computer Science*, pages 332–343. Springer, September 2008. Best Student Paper Award - ESA Track B.

[21] Daniel Delling, Andrew V. Goldberg, Thomas Pajor, and Renato F. Werneck. Customizable Route Planning. In Panos M. Pardalos and Steffen Rebennack, editors, *Proceedings of the 10th International Symposium on Experimental Algorithms (SEA'11)*, volume 6630 of *Lecture Notes in Computer Science*, pages 376–387. Springer, 2011.

[22] 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.

[23] Daniel Delling, Peter Sanders, Dominik Schultes, and Dorothea Wagner. Engineering Route Planning Algorithms. In Jürgen Lerner, Dorothea Wagner, and Katharina A. Zweig, editors, *Algorithmics of Large and Complex Networks*, volume 5515 of *Lecture Notes in Computer Science*, pages 117–139. Springer, 2009.

[24] Daniel Delling and Dorothea Wagner. Landmark-Based Routing in Dynamic Graphs. In Demetrescu [27], pages 52–65.

[25] Daniel Delling and Dorothea Wagner. Pareto paths with sharc. In *Experimental Algorithms*, pages 125–136. Springer, 2009.

[26] 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.

[27] Camil Demetrescu, editor. *Proceedings of the 6th Workshop on Experimental Algorithms (WEA'07)*, volume 4525 of *Lecture Notes in Computer Science*. Springer, June 2007.

[28] Rodolfo Dondo and Jaime Cerda. A cluster-based optimization approach for the multi-depot heterogeneous feet vehicle routing problem with time windows. *European Journal of Operational Research*, pages 1478–1507, 2007.

[29] M. Drexl. Rich vehicle routing in theory and practice. *Technical Report LM-2011-04*, page 58 pages, 2011.

[30] Stuart E. Dreyfus. An appraisal of some shortest-path algorithms. *Operations Research*, 17(3):395–412, 1969.

[31] eCOMPASS. D2.1 – new prospects in eco-friendly vehicle routing. Technical report, The eCOMPASS Consortium, 2012.

[32] David Eppstein. Finding the k shortest paths. In *FOCS*, pages 154–165, 1994.

[33] Y. Fan, R. Kalaba, and J. Moore. Arriving on time. *Journal of Optimization Theory and Applications*, 127:497–513, 2005.

[34] Paola Festa, editor. *Proceedings of the 9th International Symposium on Experimental Algorithms (SEA'10)*, volume 6049 of *Lecture Notes in Computer Science*. Springer, May 2010.

[35] M. L. Fischer and R. Jaikumar. A generalized assignment heuristic for the vehicle routing problem. *Networks*, 11:109 – 124, 1981.

[36] Luca Foschini, John Hershberger, and Subhash Suri. On the complexity of time-dependent shortest paths. In *Proc. of 22nd ACM-SIAM Symp. on Discr. Alg. (SODA '11)*, pages 327–341. ACM-SIAM, 2011.

[37] Yu Gang and Yang Jian. On the robust shortest path problem. *Computers & OR*, 25(6):457–468, 1998.

[38] Robert Geisberger, Peter Sanders, Dominik Schultes, and Christian Vetter. Exact Routing in Large Road Networks Using Contraction Hierarchies. *Transportation Science*, 46(3):388–404, August 2012.

[39] Andrew V. Goldberg and Chris Harrelson. Computing the shortest path: *A* search meets graph theory. In *SODA*, pages 156–165, 2005.

[40] 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.

[41] 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.

[42] Pierre Hansen. Bicriterion path problems. In *Lecture Notes in Economics and Mathematical Systems*, volume 177, pages 109–127. Springer, 1979.

[43] Pierre Hansen. Bicriterion path problems. In *Multiple criteria decision making theory and application*, pages 109–127. Springer, 1980.

[44] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Trans. Systems Science and Cybernetics*, 4(2):100–107, 1968.

[45] Simon Haykin. *Neural Networks: A Comprehensive Foundation.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd edition, 1998.

[46] Martin Holzer, Frank Schulz, Dorothea Wagner, Grigorios Prasinos, and Christos Zaroliagis. Engineering planar separator algorithms. *ACM Journal of Experimental Algorithmics*, 14(1):1–31, 2009.

[47] Satu Innamaa. Short-term prediction of travel time using neural networks on an interurban highway. *Transportation*, 32(6):649–669, 2005.

[48] S. R. Jameson and R. H. Mole. A sequential route building algorithm employing a generalized saving criteria. *Operational Research Quarterly*, 27:503–511, 1976.

[49] Donald B. Johnson and Tetsuo Mizoguchi. Selecting the kth element in $x + y$ and $x_1 + x_2 + ... + x_m$. *SIAM J. Comput.*, 7(2):147–153, 1978.

[50] R. E. Kalman. A New Approach to Linear Filtering and Prediction Problems. *Journal of Basic Engineering*, 82(1):35–45, 1960.

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

[52] B. W. Kernighan and S. Lin. An effective heuristic algorithm for the traveling salesman problem. *Operations Research*, 21:498 – 516, 1973.

[53] Tim Kieritz, Dennis Luxen, Peter Sanders, and Christian Vetter. Distributed Time-Dependent Contraction Hierarchies. In Festa [34], pages 83–93.

[54] G. A. P. Kindervater and M. W. P. Savelsbergh. Vehicle routing: Handling edge exchanges. *In E. H. L. Aarts and J. K. Lenstra, editors, Local Search in Combinatorial Optimization.*

[55] Spyros Kontogiannis and Christos Zaroliagis. Approximation algorithms for time-dependent shortest paths. Technical Report eCOMPASS-TR-017, Computer Technology Institute & Press "Diophantus", 2013.

[56] Spyros Kontogiannis and Christos Zaroliagis. Time-dependent approximate distance oracles. Technical Report eCOMPASS-TR-018, Computer Technology Institute & Press "Diophantus", 2013.

[57] Christian Liebchen, Marco E. Lübbecke, Rolf H. Möhring, and Sebastian Stiller. The concept of recoverable robustness, linear programming recovery, and railway applications. In Ravindra K. Ahuja, Rolf H. Möhring, and Christos D. Zaroliagis, editors, *Robust and Online Large-Scale Optimization*, volume 5868 of *Lecture Notes in Computer Science*, pages 1–27. Springer, 2009.

[58] Ronald Prescott Loui. Optimal paths in graphs with stochastic or multidimensional weights. *Commun. ACM*, 26(9):670–676, 1983.

[59] Michail P. Paraskevopoulos A. Zaroliagis C. Mali, G. A new dynamic graph structure for large-scale transportation networks. Technical report, The eCOMPASS Consortium, 2012.

[60] E.Q. Martins. On a multicriteria shortest path problem. *European Journal of Operational Research*, 26(3):236–245, 1984.

[61] Matúš Mihalák, Rastislav Šrámek, and Peter Widmayer. Approximate counting of approximate solutions. Technical report, eCOMPASS-TR-016, 2013.

[62] Roberto Montemanni and Luca M. Gambardella. Robust shortest path problems with uncertain costs. Technical Report IDSIA-03-08, Istituto Dalle Molle di studi sullintelligenza artificiale (IDSIA / USI-SUPSI), Galleria 2, 6928 Manno, Switzerland, 2008.

[63] Evdokia Nikolova. High-performance heuristics for optimization in stochastic traffic engineering problems. In Ivan Lirkov, Svetozar Margenov, and Jerzy Wasniewski, editors, *LSSC*, volume 5910 of *Lecture Notes in Computer Science*, pages 352–360. Springer, 2009.

[64] Evdokia Nikolova. *Strategic Algorithms.* PhD thesis, MIT, 2009.

[65] Evdokia Nikolova, Matthew Brand, and David R. Karger. Optimal route planning under uncertainty. In Derek Long, Stephen F. Smith, Daniel Borrajo, and Lee McCluskey, editors, *ICAPS*, pages 131–141. AAAI, 2006.

[66] Iwao Okutani and Yorgos J. Stephanedes. Dynamic prediction of traffic volume through kalman filtering theory. *Transportation Research Part B: Methodological*, 18(1):1 – 11, 1984.

[67] Ariel Orda and Raphael Rom. Shortest-Path and Minimum Delay Algorithms in Networks with Time-Dependent Edge-Length. *Journal of the ACM*, 37(3):607–625, 1990.

[68] Phillip E. Pfeifer and Stuart Jay Deutsch. A three-stage iterative procedure for space-time modeling phillip. *Technometrics*, 22(1):35–47, 1980.

[69] 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.

[70] Dominik Schultes and Peter Sanders. Dynamic Highway-Node Routing. In Demetrescu [27], pages 66–79.

[71] Wei Shen, Y. Kamarianakis, L. Wynter, Jingrui He, Qing He, R. Lawrence, and G. Swirszcz. Traffic velocity prediction using gps data: Ieee icdm contest task 3 report. In *Data Mining Workshops (ICDMW), 2010 IEEE International Conference on*, pages 1369–1371, 2010.

[72] Brian L Smith, Billy M Williams, and R Keith Oswald. Comparison of parametric and nonparametric models for traffic flow forecasting. *Transportation Research Part C: Emerging Technologies*, 10(4):303 – 321, 2002.

[73] Marius Solomon. Algorithms for the vehicle routing and scheduling problems with time window constraints. *Operations Research*, 35:254–265, 1987.

[74] Daniel Stefankovic, Santosh Vempala, and Eric Vigoda. A deterministic polynomial-time approximation scheme for counting knapsack solutions. *SIAM J. Comput.*, 41(2):356–366, 2012.

[75] P. Toth and D. Vigo (editors). *The Vehicle Routing Problem.* SIAM Monographs on Discrete Mathematics and Applications, 2002.

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

[77] Vladimir N. Vapnik. *The nature of statistical learning theory.* Springer-Verlag New York, Inc., New York, NY, USA, 1995.

[78] Eleni I. Vlahogianni, Matthew G. Karlaftis, and John C. Golias. Optimized and meta-optimized neural networks for short-term traffic flow prediction: A genetic approach. *Transportation Research Part C: Emerging Technologies*, 13(3):211 – 234, 2005.

[79] 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.

[80] Chun-Hsin Wu, Jan-Ming Ho, and D. T. Lee. Travel-time prediction with support vector regression. *Trans. Intell. Transport. Sys.*, 5(4):276–281, December 2004.

[81] Hande Yaman, Oya E. Karasan, and Mustafa Ç. Pinar. The robust shortest path problem with interval data. Unpublished Manuscript, 2001.

[82] Jin Y. Yen. Finding the k shortest loopless paths in a network. *Management Science*, 17(11):pp. 712–716, 1971.

[83] Pawel Zielinski. The computational complexity of the relative robust shortest path problem with interval data. *European Journal of Operational Research*, 158(3):570–576, 2004.