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: 38 months

D2.2.1 - New Algorithms for Eco-friendly Vehicle Routing

Workpackage:	WP2 - Algorithms for vehicle routing
Due date of deliverable:	30 November 2014
Actual submission date:	30 November 2014
Responsible Partner:	ETHZ
Contributing Partners:	CERTH, KIT, CTI

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	06.10.2014	ToC	Sandro Montanari, ETHZ
1.0	07.11.2014	First Draft	Sandro Montanari, ETHZ
1.0	07.11.2014	Sent to internal reviewers	Sandro Montanari, ETHZ
1.1	12.11.2014	Reviewers' comments incorporated (sent to PQB)	Sandro Montanari, ETHZ
1.2	15.11.2014	PQB's comments incorporated	Sandro Montanari, ETHZ
1.3	30.11.2014	Final (approved by PQB, sent to the Project Officer)	Christos Zaroliagis, CTI

Deliverable manager

- Sandro Montanari, ETHZ

List of Contributors

- Sandro Montanari, ETHZ
- Matùš Mihálak, ETHZ
- Christos Zaroliagis, CTI
- Spyros Kontogiannis, CTI
- Dimitrios Gkortsilas, CTI
- Kalliopi Giannakopoulou, CTI
- Andreas Paraskevopoulos, CTI
- Georgia Papastavrou, CTI
- Dionisis Kehagias, CERTH
- Julian Dibbelt, KIT

List of Evaluators

- Spyros Kontogiannis, CTI
- Felix König, TomTom

Summary
The purpose of this deliverable is to present the research results obtained by the project partners in the last 20 months of the project. The focus is on extending the algorithmic solutions previously developed for problems concerning routing of private vehicles and fleet of vehicles in urban areas.

Contents

1 Introduction

This deliverable presents the research results obtained by the project's partners in the last 20 months of the project with respect to eco-aware routing for private vehicles and fleet of vehicles. It describes how the algorithmic solutions developed for the problems related to WP2 [31] were improved and extended in order to yield better solutions in a more efficient way.

1.1 Objectives and scope of D2.2.1

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 20 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 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 routes 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.

The solutions proposed in this document assume a general additive cost model that can represent travel times, monetary costs, or environmental impact (e.g., fuel consumption and therefore CO_2 emissions). In deliverable D2.4 we perform an experimental assessment of these algorithms and show that they yield high quality solutions in a very efficient way. Furthermore, even when the optimization is performed in terms of more standard costs like, for example, travel times or distance, the deviation of the eco-footprint of the computed routes with respect to the environmental optimum is very limited. Therefore, the applications developed within eCOMPASS turn out to be a very viable and environmentally friendly option even for those users that do not deem as essential the optimization of environmental impact factors. Since, according to the User Requirement Analysis of D1.1, these users seems to constitute a large portion of the potential user base of routing applications, we believe the adoption of the eCOMPASS solutions to be a key ingredient for the reduction of the environmental impact of routing in the near future.

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 deals with traffic prediction techniques. Section 3 describes answering shortest path queries in the dynamic scenario where the edge weights of the network depend on the time of the day at which the query has been asked. Section 4 explains how to compute routes when the users can select the metric that should be optimized. Section 5 considers 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.

2 Traffic Prediction

2.1 Introduction

Short-term traffic forecasting is one of the most challenging tasks of modern intelligent transport systems (ITS) as their accuracy is crucial for enabling more efficient and robust advanced traffic management and traveler information systems. In the context of WP2 a set of various traffic prediction techniques were developed focusing on improving the prediction accuracy.

In this section we the results of our latest research conducted within Tasks 2.2 and 2.3 on traffic prediction. It actually adds up to the results reported in the previous version of the deliverable in a two-fold way: a) it improves the performance of the parametric Lag-STARIMA technique that was originally introduced in D2.2, and b) by introducing a new non-parametric approach.

The first approach is based on the previous time series model, but it also adds a few enhancements. In particular, we split traffic time series into segments (that represent different traffic trends) and use different time series models on the different segments of the series. The proposed method was evaluated using historical GPS traffic data from the city of Berlin, Germany covering a total period of two weeks. We extensively pre-processed the available data before the application of the implemented algorithm.

The second approach is the introduction of a new non-parametric traffic forecasting technique. Its novelty resides on the construction of road profiles by applying clustering techniques on available traffic data, based on the dynamic features of traffic, expressed in the form of the first and second derivatives of speed. This technique is used in order to reduce the dimension of the available feature space, preserving the maximum information gain of the original data, and also to improve the time required for data processing. The outcome of the application of a data clustering algorithm on the aforementioned feature space is the exposure of road speed behaviour patterns, in the form of data clusters or speed profiles. Based on the statistical features of each cluster it becomes feasible to perform traffic forecasting for a particular road whose average speed evolves according to a particular identified speed profile. The essence of our study lies on the importance of the first and second derivative speed dynamics for capturing sufficient information on how traffic evolves. This forms the basis for building speed profiles that can be used for extracting traffic estimations that improve the quality of short-term forecasting. The accuracy of our traffic forecasting technique was evaluated using the same traffic dataset from Berlin, as above.

The rest of this section is organized as follows. Subsection 2.2 provides a brief summary of the techniques that are reported in deliverable D2.2. Subsection 2.3 describes in details the updates on the existing Lag-STARIMA model that was extensively described in D2.2, which resulted in improved accuracy. In subsection 2.5 we describe in detailed the new contributions, comprising the non-parametric approach.

2.2 Summary of previously developed technique

The primary traffic prediction algorithm that we developed in WP2 is based on the classic parametric traffic forecasting model STARIMA, which is derived as follows.

The *Auto-Regressive* (AR) part provides the current value X_t as the linear aggregate of p previous values (1):

$$X_t = \sum_{k=1}^{p} \varphi_k X_{t-k} + e_t, \tag{1}$$

where e_t is the error term and follows a Gaussian distribution of type $(0, \sigma^2)$ (white noise). The *Moving Average* (MA) part provides the current value X_t as the aggregate of q previous error terms:

$$X_t = \sum_{k=1}^{q} \theta_k e_{t-k} + e_t \tag{2}$$

Hence, according to (1) and (2) the mixed autoregressive and moving average, or $ARMA(p,q)$ model is formed as follows:

$$X_t = \sum_{k=1}^{p} \varphi_k X_{t-k} + \sum_{k=1}^{q} \theta_k e_{t-k} + e_t \tag{3}$$

or equivalently:

$$\left(1 - \sum_{k=1}^{p} \varphi_k B^k\right) X_t = \left(1 + \sum_{k=1}^{q} \theta_\kappa B^\kappa\right) e_t, \tag{4}$$

where B is the backwards shift operator $B^k X_t = X_{t-k}$. Upon differencing the series at the d-th degree, i.e., $(1-B)^d X_t$ the ARIMA model is formed:

$$\left(1 - \sum_{k=1}^{p} \varphi_k B^k\right) (1-B)^d X_t = \left(1 + \sum_{k=1}^{q} \theta_k B^k\right) e_t \tag{5}$$

or equivalently:

$$\varphi(B)(1-B)^d X_t = \theta(B) e_t \tag{6}$$

The above equation describes an $ARIMA(p,d,q)$ model.

The multivariate variation of the $ARIMA(p,d,q)$ model, the *Space-Time ARIMA* (STARIMA), takes into account the spatiotemporal relations between the time series. The model is defined by the following equation:

$$\varphi_{p,\lambda}(B)\, \Phi_{P,\Lambda}\left(B^S\right)(1-B)^d \left(1-B^S\right)^D X_t = \theta_{Q,M}\left(B^S\right) e_t, \tag{7}$$

where:

$$\varphi_{p,\lambda}(B) = 1 - \sum_{k=1}^{p} \sum_{l=0}^{\lambda^k} \varphi_{k,l} W_l B^k \tag{8}$$

$$\Phi_{P,\Lambda}\left(B^S\right) = 1 - \sum_{k=1}^{P} \sum_{l=0}^{\Lambda^k} \Phi_{k,l} W_l B^{kS} \tag{9}$$

$$\theta_{q,m}(B) = 1 - \sum_{k=l}^{q} \sum_{l=0}^{m^k} \theta_{k,l} W_l B^k \tag{10}$$

$$\Theta_{Q,M}\left(B^S\right) = 1 - \sum_{k=1}^{Q} \sum_{l=0}^{M^k} \Theta_{k,l} W_l B^{kS} \tag{11}$$

The parameters of the STARIMA model 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. The neighbouring matrix

W, is an $N \times N$ square matrix with each row summing to one that contains the weights that define the spatial relationship of the terms. In order to calculate these weights we use the Coefficient of Determination (CoD) metric, which is defined for two time series x and y at lag k, as follows:

$$CoD_{xy}(k) = 100 \left[\frac{E\left[(x_t - \mu_x)(y_{t+k} - \mu_y)\right]}{\sigma_x \sigma_y} \right]^2, \; k = 0, \pm 1, \ldots \tag{12}$$

CoD provides a generic way of determining the percentage of variance between two time series. Concerning traffic prediction the interest is mainly 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 our original method aimed at calculating CoD for the whole network in order to decide which roads to include in the prediction model provided by (7).

2.3 Existing algorithm updates

In this section we introduce an improvement of our previous time series-based traffic prediction method, by adopting the best fitting model after splitting the time period to which the prediction is applied into more concrete segments. When trying to predict the next value of a series, as in a traffic scenario, it is common that the series has different behaviour (e.g. trend) on different segments. Our motivation is to provide an adaptive model that fits the given dataset better, by taking into account distinct trends that occur in traffic during one day, e.g. free-flow or congested traffic trends.

2.3.1 Data Pre-processing

Before applying any traffic forecasting method, the aforementioned Berlin dataset is organised in a way that will ensure data tolerance and traffic forecasting technique scalability. At first, the links of the network are combined to form roads. Each road is defined as any segment between two intersections, whereas links are defined as straight lines, thus a road contains an arbitrary number of links. Furthermore, instead of instantaneous speeds, only their arithmetic and harmonic averages for 5 minutes time intervals are stored. Supposing n values of raw speed x_i recorded at interval t, their harmonic average is defined as:

$$x_{i,t} = \frac{n}{\sum_{j=1}^{n} \frac{1}{x_j}} \tag{13}$$

and is used because (according to [71]) corresponds better to travel times. Finally for every road of the network a traffic time series is constructed which represents a day of traffic data for the specific road. In particular every traffic time series consists of harmonic speeds for every 5 minutes interval of a day, so it has size of 288.

The presence of outliers on a time series can greatly affect the thresholds which determine the boundaries of the segments, resulting in the creation of segments that do not necessarily follow an actual trend. The presence of extreme values in the data is not uncommon since some drivers may travel at far lower or higher than the average speeds for various reasons. Since the harmonic speed tends strongly toward the least elements, it tends to mitigate the impact of large outliers and aggravate the impact of small ones. The outlier filter used to minimize the effects of any outliers is described in [5] and is known as one-sided median method for cleaning noisy data. In this method every new value is compared to the median and if it is above a threshold then the value is considered an outlier and is substituted with the value of the median.

The Moving Average filter which is described in 14 and 15 (where s_M is the M-th member of the time series) is applied after the outlier filter and results in a smoother time series. Then the segment limits are determined based on the filtered series while the models are trained (and tested) using the actual unfiltered values.

$$SMA = \frac{s_M + s_{M-1} + \ldots + s_{M-(n-1)}}{n} \tag{14}$$

$$SMA_t = SMA_{t-1} - \frac{s_{M-n}}{n} + \frac{s_M}{n} \tag{15}$$

As Vlahogianni [70] underlined, if the forecasting models do not have the ability to deal with false or missing values, it is up to the practitioner to select the proper data-filling technique. This stage of traffic data preparation is of outmost importance in the case of conventional statistical approaches. Smith and Demetsky [64] underlined the inability of ARIMA models to deal with missing values. Later, Chen et al. [10] commented on the effect of missing values in a comparative study between an ARIMA model and the neural network approach. The findings showed the significant sensitivity of ARIMA models in dealing with missing values and the performance of the method using various types of filling techniques. Furthermore, our dataset is based on GPS speed probes and as a result the speed information is sparse (e.g. a link may have no data for different moments in time). While there are various filling techniques, these are based on loop detector datasets that are more consistent and have missing values due to technical issues such us power outages etc., which result in gaps that are far less in numbers and different in nature (more consistent rather than random) than the GPS datasets. Overall time series techniques may not always be appropriate for GPS datasets since the low GPS sampling rate makes it very hard to construct reliable historical speed profiles for all the roads.

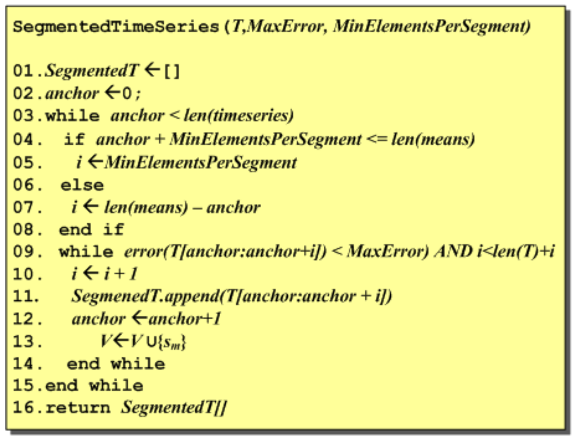
In order to impute the missing data values a method based on *K-means* clustering is employed. *K-means* is a method of vector quantization that partitions n observations into k clusters, in which each observation belongs to the cluster with the nearest mean, serving as a prototype of the cluster. The clusters are formed by applying the algorithm on the training set. After the clusters are formed, probability density functions (histograms) of speed for every cluster and time interval are constructed.

2.3.2 Segmentation Methods

In our work we use two segmentation methods as described below.

1. *Sliding windows*: The sliding windows algorithm iterates over the time series values and for each new value it uses a sliding window containing past values in order to check whether certain error criteria are met to keep the value in the current window or if it should spawn a new segment. The algorithm appears in an online mode since it processes data points one by one, every time a new data point is added to the series. For each new segment, the algorithm has an anchor which is the starting point of the segment. Every time a new value comes, variable i increases and the error of the sub-series containing the segment and the new value is checked against a threshold. The values keep adding to the segment as long as the threshold is not surpassed. If the threshold is surpassed, the segment is saved (or simply returned for the online case), and a new segment is created by setting the anchor to the next value of the series. The algorithm is presented in pseudocode form in Fig. 1.

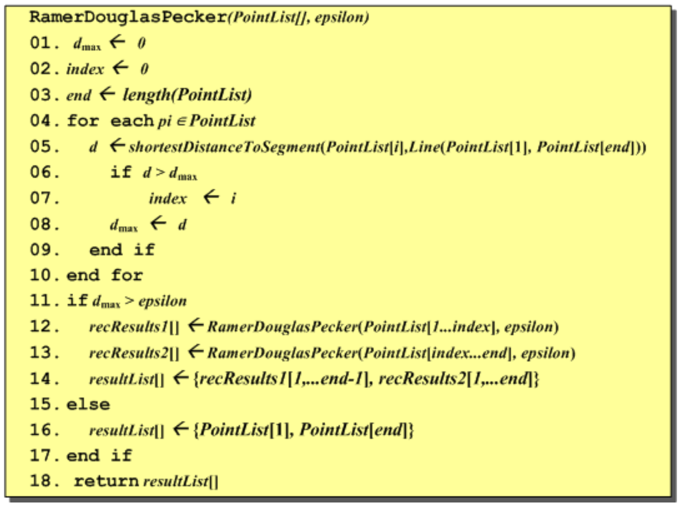
2. *Ramer-Douglas-Pecker*: This algorithm is a top-down approach which recursively divides the time series. Initially it is given all the points between the first and the last point and automatically marks the first and the last point to be kept. It then finds the point that is furthest (in terms of Euclidean distance) from the line segment with the first and last point as end points. If the point is closer than e (maximum error threshold) to the line segment then any points not currently marked to keep can be discarded without simplified curve being worse than e. If the point distance from the line segment is greater than e from the approximation, then that point must be kept as a segment limit. The algorithm is presented in pseudo code form in Fig. 2.



```
SegmentedTimeSeries (T,MaxError, MinElementsPerSegment)

SegmentedT ← []
anchor ← 0;
while anchor < len(timeseries)
  if anchor + MinElementsPerSegment <= len(means)
    i ← MinElementsPerSegment
  else
    i ← len(means) – anchor
  end if
  while error(T[anchor:anchor+i]) < MaxError AND i<len(T)+i
    i ← i + 1
    SegmenedT.append(T[anchor:anchor + i])
    anchor ← anchor+1
        $V \leftarrow V \cup \{s_m\}$
   end while
end while
return SegmentedT[]
```

Figure 1: Sliding windows segmentation algorithm



```
RamerDouglasPecker(PointList[], epsilon)
$d_{\max}$ ← 0
index ← 0
end ← length(PointList)
for each pi ∈ PointList
   d ← shortestDistanceToSegment(PointList[i],Line(PointList[1], PointList[end]))
      if $d > d_{\max}$
            index ← i
      $d_{\max}$ ← d
   end if
end for
if $d_{\max}$ > epsilon
   recResults1[] ← RamerDouglasPecker(PointList[1...index], epsilon)
   recResults2[] ← RamerDouglasPecker(PointList[index...end], epsilon)
   resultList[] ← {recResults1[1,...end-1], recResults2[1,...end]}
else
   resultList[] ← {PointList[1], PointList[end]}
end if
 return resultList[]
```

Figure 2: Ramer-Douglas-Pecker segmentation algorithm

Both approaches were implemented and tested on time series from several roads. The results were examined and the Ramer-Douglas-Pecker algorithm was chosen as it could provide much more accurate segmentation that captures the general trends of the series. An example of a segmentation of a time series using the Ramer-Douglas-Pecker algorithm is presented in Fig. 4

2.4 The enhanced prediction model

The implemented model, namely Segmented Lag-STARIMA (SLS), is based on a simplified version of the STARIMA model which is given by (16).

$$Z_{t+T} = \varphi_{00}Z_t + \varphi_{10}Z_{t-1} + \varphi_{20}Z_{t-2} + \varphi_{11}W_1Z_t + \varphi_{12}W_2Z_t + \dots, \tag{16}$$

where Z_t represent the speed(s) at time t, W_o is the neighbor matrix of order o, T is the prediction time ahead and ϕ_{to} is (are) the parameter(s) for road(s) of order o at interval t.

In the presented method the neighbour matrix W was populated using the CoD metric, given by (12).

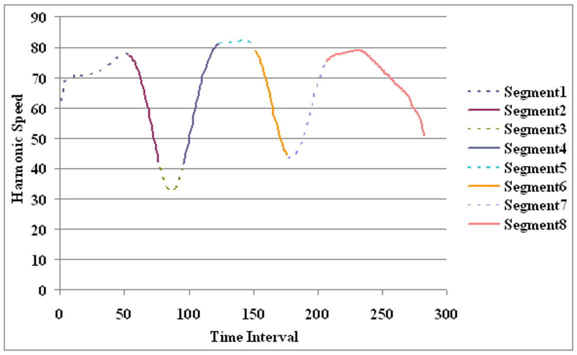


Figure 3: Time series segments using the Ramer-Douglas-Pecker algorithm

Given two roads i and j the element of the weight matrix W (CoD matrix) in the i-th row and j-th column is computed as follows:

$$w_{ij}{}^{l} = \begin{cases} \frac{1}{\sum\limits_{j=1}^{N} w_{ij}{}^{l}}, \text{ifCoD}_{\text{ij}}\left(\text{l}\right) \in M_l \\ 0, \text{otherwise} \end{cases}, \tag{17}$$

where M_l is the set with the largest CoDs for lag l. As in [26], three lag values were considered (for three time intervals before the current one, respectively) and the 10 largest for values of CoD for a specific lag are taken into account. In these terms, speeds in (16) reflect lags, W_l is the neighbour matrix of lag l and ϕ_{tl} is (are) the parameter(s) for road(s) of lag l at interval t.

The model takes into account the current speed of the road, the speeds of the past three time intervals and the average speed of the most correlated roads that have a maximum time lag of 15 minutes. Naturally, using a training set it is attempted to approximate the ϕ_{ij} parameters in (16) and then test the method by predicting the next Z_t values with a test set. As noted in [58] the best estimates of the ϕ values, from many points of view, are the maximum likelihood estimates but since without a priori knowledge of the initial values they cannot be exactly calculated, a close approximation of them is calculated via least squares. In particular, for every training sample an equation like (16) is formed where ϕ_{ij} are the only unknown parameters. This leads to an over determined system, in the form of Xβ = y, or written with the normal equations:

$$(\mathrm{X}^{\mathrm{T}}\mathrm{X})\beta = \mathrm{X}^{\mathrm{T}}y \tag{18}$$

that using the linear least squares method has a solution of the following form:

$$\beta = (\mathrm{X}^{\mathrm{T}}\mathrm{X})^{-1}\mathrm{X}^{\mathrm{T}}y \tag{19}$$

Since STARIMA models are non-linear in form, it is necessary to estimate the parameters of the model using any non-linear optimization technique. As a result the Levenberg Marquardt algorithm [52] (also known as Damped Least Squares method) was used in order to solve the equation as a non-linear least square problem. The LM method actually solves a slight variation of (18), known as the augmented normal equation described by the following equation:

$$N\beta = \mathrm{X}^{\mathrm{T}}y \tag{20}$$

where the off-diagonal elements of N are identical to the corresponding elements of X^TX and the diagonal elements are given by (21).

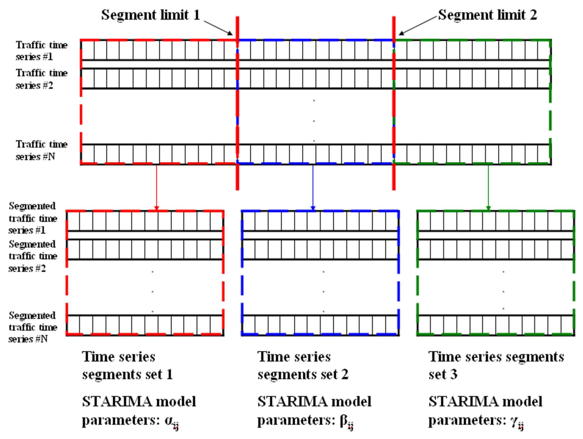

Segment limit 1
Segment limit 2
Traffic time series #1
Traffic time series #2
Traffic time series #N
Segmented traffic time series #1
Segmented traffic time series #2
Segmented traffic time series #N
Time series segments set 1
STARIMA model parameters: α_{ij}
Time series segments set 2
STARIMA model parameters: β_{ij}
Time series segments set 3
STARIMA model parameters: γ_{ij}

Figure 4: Segmented Lag- STARIMA model for 3 segments per time series

$$N_{ii} = m + \left[\mathrm{X}^{\mathrm{T}}\mathrm{X}\right]_{ii}, m > 0 \tag{21}$$

The term m is called *damping term* and the rest of the LM algorithm, which was firstly introduced in [54], remains unchanged.

Using the Ramer-Douglas-Pecker segmentation algorithm (as mentioned above) n segment limits $l_1, l_2, l_3, \ldots, l_n$ of the time series are discovered and then for every road of the network a set of $n+1$ equations are formed (one for every segment $[Begin, l_1), [l_1, l_2), \ldots, [ln, End)$). For example supposing that every time series splits in three segments, a set of equations like the following are formed for every road:

$$Z_{t+\mathrm{T}} = \alpha_{00}Z_t + \alpha_{10}Z_{t-1} + \alpha_{20}Z_{t-2} + \alpha_{11}W_1Z_t + \alpha_{12}W_2Z_t + \ldots t \in [l_1, l_2) \tag{22}$$

$$Z_{t+\mathrm{T}} = \beta_{00}Z_t + \beta_{10}Z_{t-1} + \beta_{20}Z_{t-2} + \beta_{11}W_1Z_t + \beta_{12}W_2Z_t + \ldots t \in [l_2, l_3) \tag{23}$$

$$Z_{t+\mathrm{T}} = \gamma_{00}Z_t + \gamma_{10}Z_{t-1} + \gamma_{20}Z_{t-2} + \gamma_{11}W_1Z_t + \gamma_{12}W_2Z_t + \ldots t \in [l_3, l_4] \tag{24}$$

The equations parameters α_{ij}, β_{ij}, γ_{ij} essentially define three different STARIMA models and are estimated by the method described above using the training data. After the parameters are estimated the prediction is straight forward. The Z array is constructed for the present interval t and the next value is estimated by calculating the value of $Z_t + T$. If $t + T$ is exactly on the limit of a segment then the average value of the previous and next prediction models is considered. Intuitively the trained parameters would be able to fit better the model, capturing trends and thus provide smaller prediction errors. An example of the implementation of the described model for three segments per time series is presented in Fig. 4.

An example of a trained model is given by (25) below.

$$\begin{aligned} &Z_{t+\mathrm{T}} = 0.034Z_t - 0.041Z_{t-1} + 0.317Z_{t-2} + \\ &0.465W_1Z_t + 0.27W_2Z_t + 0.013\mathrm{W_3Z_t} \end{aligned} \tag{25}$$

For an input vector of speeds $Z = [76.3, 75, 64, 83, 43, 42]$, where the first three elements correspond to the speeds of the same road and the last three to the average of speeds of the most correlated roads with lag 1,2, and 3 intervals, respectively, the predicted speed is 76.847 km/h.

2.5 New approaches

In this section we introduce a new non-parametric approach as opposed to the previous parametric approach described in Section 2.3. This consists of two main phases: (a) the training phase and (b) the forecasting phase. The training phase constructs the road profiles, based on the traffic dynamics of all roads in the network. Each profile describes similar speed behaviours that different roads in the network may have. Also, within the same profile road traffic behaviour can be different from time to time. Similarly to the previous method, we divide each 24-hours day into 288 five-minute time intervals. After the training phase is completed the created profiles are used in the forecasting phase. The goal of this approach is to perform forecasting after matching in an optimal way the road in question against one of the road profiles generated in the training stage. The best matched profile provides sufficient information for estimating the probability distribution of the average speed and therefore the travel time that we want to predict.

2.5.1 Data Pre-processing

In the first step a sort of pre-processing operations are applied on the existing dataset. In particular, the original dataset is formulated as a set of time series. Let us denote X_i the time series of speed measurements corresponding to an arbitrary road r_i , given by the following equation:

$$X_i = \{x_{i,1}, x_{i,2}, ..., x_{i,N}\} \tag{26}$$

where $x_{i,t}$ is the harmonic average of the instantaneous speeds of vehicles passing the road i (in km/h) recorded within the t-th interval, where $t = 1, ..., N$ that will be referred from now on as harmonic speed. Let n values of raw speed y_i recorded in the time interval t. The harmonic speed in this is defined as follows:

$$x_{i,t} = \frac{n}{\sum_{j=1}^{n} \frac{1}{y_j}} \tag{27}$$

After the dataset is pre-processed as described above, the next step is to create the feature space on which clustering will be applied. This is done by extracting appropriate features out of the available time series, including traffic dynamics. Our intuition relies on the fact that each profile to be generated should be characterised by unique feature in terms of data dynamics. Also the introduction of new features based on time series dynamics introduce a reduction of the original feature space dimension, resulting in shorter training times and improved model interpretability. In order to validate our assumption, we applied various feature selection techniques by combining the original features with time series dynamics. In particular, we initially extract the following features from each time series i:

- minimum harmonic speed: $x_{i,\min} = \min\{x_{i,1}, x_{i,2}, \ldots, x_{i,N}\}$
- maximum harmonic speed: $x_{i,\max} = \max\{x_{i,1}, x_{i,2}, \ldots, x_{i,N}\}$
- mean harmonic speed: $\bar{x_i} = \frac{1}{N} \times \sum\limits_{j=1}^{N} x_{i,j}$
- mean acceleration: $\bar{x}_i' = \frac{1}{N-1} \times \sum_{j=2}^{N} \frac{x_{i,j} - x_{i,j-1}}{\Delta t}$
- standard deviation: $std_i = \sqrt{\frac{1}{\mathrm{N}} \times \sum\limits_{j=1}^{N} (x_{ij} - \bar{x_i})^2}$

After these features have been extracted, we calculate their interestingness score (IS) to see how much information they carry. IS is given by the following equation:

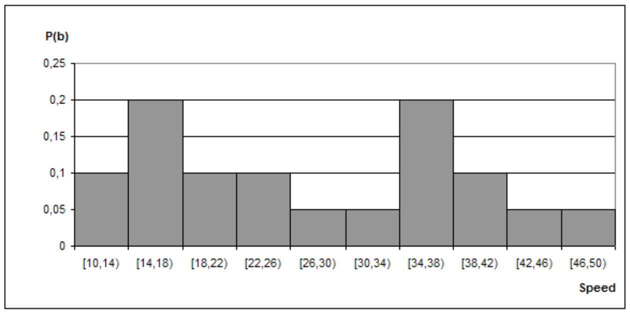


Figure 5: A sample histogram of a speed profile for a particular time interval

$$IS = (m - H(x))^2, \tag{28}$$

where $H(X)$ is the entropy of a feature X, given by:

$$H(x) = -\sum_{i=1}^{N} \left(p(x_i) \times \log_b(x_i)\right) \tag{29}$$

and m is the mean entropy of all features. As a result, we form three feature sets (FS) that include those features with the maximum scores, i.e. minimum entropy, taking into account all 288 features of the original dataset.

2.5.2 Clustering

After transforming the set of time series into vectors in a space of smaller dimension as already explained, we applied a clustering technique for generating the various road profiles. We compared various clustering techniques including partitioning (k-means, k-medoids, k-medians), hierarchical (agglomerative) and density-based (DBSCAN) ones, with respect to compactness of the resulted clusters as well as their overall performance. We finally concluded that the most appropriate choice in our case is the k-means clustering algorithm. The most important drawback of the k-means algorithm that affects its performance is the proper selection of the number of clusters K. If the value of K is not properly selected the discriminating capacity of each cluster may be poor, affecting the forecast accuracy. There are several ways to determine an ideal number of clusters for the k-means algorithm according to the form of the dataset.

2.5.3 Speed probability distribution estimation

The final step of the training phase is the estimation of the probability density function (histogram) for each one of the generated road profiles and for all future time intervals within an hour. This is done by calculating the speed histogram of each speed profile, using the values of speed for every road that belong to the same cluster. Figure 5 illustrates a sample histogram calculated for a specific profile and for a target five minute interval, where the speed measurements are divided into 10 equally sized bins.

The main stages of the forecasting include:

1. Read data
2. Create new instance in the feature space
3. Perform classification of the new instance into one of the existing clusters

4. Calculate the speed probability distribution
5. Perform speed forecasting based on the calculated distribution.

At the final forecasting step, the chosen histogram is used for the estimation of the probability distribution function of each road and target time interval. Two methods for generating a random number from an arbitrary probability distribution were implemented.

The first one is a nave approach (nave random number generator) which works as follows:

- A random number in the range of the values of chosen histogram is generated (where this number follows uniform distribution in this range)
- The bin in which the above number belongs is discovered
- A random number in this specific bin is generated (where this number follows uniform distribution in this bin)

The second approach (sophisticated random number generator) is essentially based on the Inverse Probability Integral Transform Method.

Firstly, using the chosen histogram (shown in Figure 5) the corresponding cumulative probability distribution function (CDF) is constructed. Then we generate a random probability value p with uniform distribution in the range $[0, 1]$. In order to calculate the predicted speed value we find the values of probabilities p_1, p_2, so that $p_1 < p < p_2$ that correspond to a bin with right and left bounds s_1 and s_2, respectively. We finally calculate the predicted value of speed using the following formula:

$$s = (s_2 - s_1) \times \frac{p - p_1}{p_2 - p_1} + s_1 \tag{30}$$

3 Time-Dependent Shortest Paths

3.1 Introduction

Distance oracles are succinct data structures encoding shortest path information among a carefully selected subset of pairs of vertices in a graph. The encoding is done in such a way that the oracle can *efficiently answer* shortest path queries for arbitrary origin-destination pairs, exploiting the preprocessed data and/or *local* shortest path searches. A distance oracle is exact (resp. approximate) if the returned distances by the accompanying query algorithm are exact (resp. approximate). A bulk of important work (e.g., [68, 67, 57, 60, 72, 73, 3]) is devoted to constructing distance oracles for *static* (i.e., *time-independent*), mostly undirected networks in which the arc-costs are fixed, providing trade-offs between the oracle's space and query time and, in case of approximate oracles, also of the stretch (maximum ratio, over all origin-destination pairs, between the distance returned by the oracle and the actual distance). For an overview of distance oracles for static networks, the reader is deferred to [66] and references therein.

In many real-world applications, however, the arc costs may vary as functions of time (e.g., when representing travel-times) giving rise to *time-dependent* network models. A striking example is route planning in road networks where the travel-time for traversing an arc $a = uv$ (modelling a road segment) depends on the temporal traffic conditions while traversing uv, and thus on the departure time from its tail u. Consequently, the optimal origin-destination path may vary with the departure-time from the origin. Apart from the theoretical challenge, the time-dependent model is also much more appropriate with respect to the historic traffic data that the route planning vendors have to digest, in order to provide their customers with fast route plans. For example, TomTom's *LiveTraffic* service provides real-time estimations of average travel-time values, collected by periodically sampling the average speed of each road segment in a city, using the connected cars to the service as sampling devices. The crux is how to exploit all this historic traffic information in order to provide *efficiently* route plans that will adapt to the departure-time from the origin. Towards this direction, we consider the continuous, piecewise linear (pwl) interpolants of these sample points as *arc-travel-time functions* of the corresponding instance.

Computing a time-dependent shortest path for a triple (o, d, t_o) of an origin o, a destination d and a departure-time t_o from the origin, has been studied long time ago (see e.g., [12, 30, 56]). The shape of arc-travel-time functions and the waiting policy at vertices may considerably affect the tractability of the problem [56]. A crucial property is the *FIFO property*, according to which each arc-arrival-time at the head of an arc is a *non-decreasing* function of the departure-time from the tail. If *waiting-at-vertices* is forbidden and the arc-travel-time functions may be non-FIFO, then subpath optimality and simplicity of shortest paths is not guaranteed. Thus, (even if it exists) an optimal route is not computable by well known techniques (Dijkstra or Bellman-Ford) [56]. Additionally, many variants of the problem are also $\mathbf{NP}-$hard [63]. On the other hand, if arc-travel-time functions possess the FIFO property, then the problem can be solved in polynomial time by a straightforward variant of Dijkstra's algorithm (**TDD**), which relaxes arcs by computing the arc costs "on the fly", when scanning their tails. This has been first observed in [30], where the *unrestricted waiting policy* was (implicitly) assumed for vertices, along with the non-FIFO property for arcs.

The FIFO property may seem unreasonable in some application scenarios, e.g., when travellers at the dock of a train station wonder whether to take the very next slow train towards destination, or wait for a subsequent but faster train. Our motivation in this work stems from *route planning* in urban-traffic road networks where the FIFO property seems much more natural: Cars are assumed to travel according to the same (possibly time-dependent) average speed in each road segment, and overtaking is not considered as an option. Additionally, when shortest-travel-times are well defined and optimal waiting-times at nodes always exist, a non-FIFO arc with *unrestricted-waiting-at-tail* policy is equivalent to a FIFO arc in which waiting at the tail is useless [56]. Therefore, our focus in this work is on networks with FIFO arc-travel-time functions.

Until recently, most of the previous work on the time-dependent shortest path problem concentrated on computing an optimal origin-destination path providing the earliest-arrival time at destination when departing at a *given* time from the origin, and neglected the computational complexity of providing succinct representations of the entire earliest-arrival-time *functions*, for *all* departure-times from the origin. Such representations, apart from allowing rapid answers to several queries for selected origin-destination pairs but for varying departure times, would also be valuable for the construction of *distance summaries* (a.k.a. *route planning maps*, or *search profiles*) from central vertices (e.g., *landmarks* or *hubs*) towards other vertices in the network, providing a crucial ingredient for the construction of distance oracles to support real-time responses to arbitrary queries $(o, d, t_o) \in V \times V \times \mathbb{R}$.

The complexity of succinctly representing earliest-arrival-time functions was first questioned in [14, 16, 15], but was solved only recently in [36] which, for FIFO-abiding pwl arc-travel-time functions, showed that the problem of succinctly representing such a function for a *single origin-destination pair* has space-complexity $(1 + K) \cdot n^{\Theta(\log n)}$, where n is the number of vertices and K is the total number of breakpoints (or legs) of all the arc-travel-time functions. Polynomial-time algorithms (or even PTAS) for constructing *point-to-point* approximate distance functions are provided in [36, 17]. Such approximate distance functions possess *succinct representations*, since they require only $\mathcal{O}(1 + K)$ breakpoints per origin-destination pair. It is also easy to verify that K could be substituted by the number K^* of *concavity-spoiling* breakpoints of the arc-travel-time functions (i.e., breakpoints at which the arc-travel-time slopes increase).

To the best of our knowledge, the problem of providing distance oracles for time-dependent networks with *provably* good approximation guarantees, small preprocessing-space complexity and sublinear time complexity, has not been investigated so far. Due to the hardness of providing succinct representations of exact shortest-travel-time functions, the only realistic alternative is to use approximations of these functions for the distance summaries that will be preprocessed and stored by the oracle. Exploiting a PTAS (such as that in [36]) for computing approximate distance functions, one could provide a trivial oracle with query-time complexity $Q \in \mathcal{O}(\log\log(K^*))$, at the cost of an exceedingly high space-complexity $S \in \mathcal{O}\big((1+K^*)\cdot n^2\big)$, by storing succinct representations of all the point-to-point $(1+\varepsilon)-$approximate shortest-travel-time functions. At the other extreme, one might use the minimum possible space complexity $S \in \mathcal{O}(n+m+K)$ for storing the input, at the cost of suffering a query-time complexity $Q \in \mathcal{O}(m + n\log(n)[1+\log\log(1+K_{\max})])$ (i.e., respond to each query by running **TDD** in real-time using a predecessor search structure for evaluating pwl functions)[1]. The main challenge considered in this work is to smoothly close the gap between these two extremes, i.e., to achieve a better (e.g., *sublinear*) query-time complexity, while consuming smaller space-complexity (e.g., $\mathrm{o}\big((1+K^*)\cdot n^2\big)$) for succinctly representing travel-time *functions*, and enjoying a small (e.g., close to 1) approximation guarantee.

We present the *first* approximate distance oracle for sparse directed graphs with time-dependent arc-travel-times, which achieves all these goals. Our oracle is based only on the sparsity of the network, plus two assumptions of travel-time functions which are quite natural for route planning in road networks (cf. Assumptions 1 and 2 in Section 3.2). It should be mentioned that: (i) even in static undirected networks, achieving a stretch factor below 2 using subquadratic space and sublinear query time, is possible only when $m \in \mathrm{o}\big(n^2\big)$, as it has been recently shown [60, 3]; (ii) there is important applied work [23, 6, 18, 55] to develop time-dependent shortest path *heuristics*, which however provide mainly empirical evidence on the success of the adopted approaches.

At a high level, our approach resembles the typical ones used in *static* and *undirected* graphs (e.g., [68, 60, 3]): Distance summaries from selected landmarks are precomputed and stored; fast responses to arbitrary real-time queries are provided by growing small distance balls around the origin and the destination, and then closing the gap between the prefix subpath from the origin and the suffix subpath towards the destination. However, it is not at all straightforward how this generic approach can be extended to *time-dependent* and *directed* graphs, since one is confronted with two highly non-trivial challenges: (i) handling directedness, and (ii) dealing with time-dependence, i.e., deciding the arrival-times to grow balls around vertices in the vicinity of the destination, because we simply do ***not*** know the earliest-arrival-time at destination – actually, this is what the original query to the oracle asks for. A novelty of our query algorithms, contrary to other approaches, is exactly that we achieve the approximation guarantees by growing balls only from vertices around the origin. Managing this was a necessity for our analysis since growing balls around vertices in the vicinity of the destination at the *right* arrival-time is essentially not an option.

Let U be the worst-case number of breakpoints for an $(1+\varepsilon)-$approximation of a *concave* distance function stored in our oracle, and TDP be the maximum number of time-dependent shortest path probes during their construction[2]. The following theorem summarizes our results.

Theorem 3.1. For time-dependent instances compliant with Assumptions 1 and 2, a distance oracle is provided storing $(1+\varepsilon)-$approximate distance functions from landmarks, which are uniformly and independently selected with probability ρ, to all other vertices, and uses a recursion depth (budget) r in the query algorithm, guaranteeing expected values of: (i) preprocessing space $\mathcal{O}\big(\rho n^2(1+K^*)U\big)$; (ii) preprocessing time $\mathcal{O}\big(\rho n^2(1+K^*)\log(n)\log\log(K_{\max})TDP\big)$; (iii) query time $\mathcal{O}((1/\rho)^r\log(1/\rho)\log\log(K_{\max}))$. The guaranteed stretch is $1+\varepsilon\frac{(1+\frac{\varepsilon}{\psi})^{r+1}}{(1+\frac{\varepsilon}{\psi})^{r+1}-1}$, where ψ is a fixed constant depending on the characteristics of the arc-travel-time functions, but is independent of the network size.

Note that, apart from the choice of landmarks, our algorithms are deterministic. Due to space limitations, proofs and a table with solid examples of the oracle's space/query-time/stretch tradeoffs can be found in the full version [50].

[1] $K_{\max}$ denotes the maximum number of breakpoints in an arc-travel-time function.

[2] As proved in [50], U and TDP are independent of the network size n.

3.2 Ingredients and Overview of Our Approach

Our input is provided by a directed graph $G = (V, A)$ with n vertices and m arcs. Every arc $uv \in A$ is equipped with a periodic, continuous, piecewise-linear (pwl) *arc-travel-time* (a.k.a. *arc-delay*) function $D[uv] : \mathbb{R} \to \mathbb{R}_{>0}$, such that $\forall k \in \mathbb{Z}, \forall t_u \in [0, T),\ D[uv](k \cdot T + t_u) = D[uv](t_u)$ is the arc-travel-time of uv when the departure-time from u is $k \cdot T + t_u$. $D[uv]$ is represented succinctly as a continuous pwl function, by K_{uv} breakpoints describing its projection to $[0, T)$. $K = \sum_{uv\in A} K_{uv}$ is the number of breakpoints to represent all the arc-delay functions in the network, and $K_{\max} = \max_{uv\in A} K_{uv}$. K^* is the number of *concavity-spoiling* breakpoints, i.e., the ones in which the arc-delay slopes increase. Clearly, $K^* \le K$, and $K^* = 0$ for *concave* pwl functions. The space to represent the entire network is $\mathcal{O}(n + m + K)$. The *arc-arrival* function $Arr[uv](t_u) = t_u + D[uv](t_u)$ represents arrival-times at v, depending on the departure-times t_u from u. For any $(o, d) \in V \times V$, $\mathcal{P}_{o,d}$ is the set of $od-$paths, and $\mathcal{P} = \cup_{(o,d)}\mathcal{P}_{o,d}$. For a path $p \in \mathcal{P}$, $p_{x\leadsto y}$ is its subpath from (the first appearance of) vertex x until (the subsequent first appearance of) vertex y. For any pair of paths $p \in \mathcal{P}_{o,v}$ and $q \in \mathcal{P}_{v,d}$, $p \bullet q$ is the $od-$path produced as the concatenation of p and q at v. For any path (represented as a sequence of arcs) $p = \langle a_1, a_2, \cdots, a_k\rangle \in \mathcal{P}_{o,d}$, the *path-arrival* function is the composition of the constituent arc-arrival functions: $\forall t_o \in [0, T),\ Arr[p](t_o) = Arr[a_k](Arr[a_{k-1}](\cdots(Arr[a_1](t_o))\cdots))$. The *path-travel-time* function is $D[p](t_o) = Arr[p](t_o) - t_o$. The *earliest-arrival-time* and *shortest-travel-time* functions from o to d are: $\forall t_o \in [0, T), Arr[o, d](t_o) = \min_{p\in\mathcal{P}_{o,d}} \{Arr[p](t_o)\}$ and $D[o, d](t_o) = Arr[o, d](t_o) - t_o$. Finally, $SP[o, d](t_o)$ (resp. $ASP[o, d](t_o)$) is the set of shortest (resp., with stretch-factor at most $(1 + \varepsilon)$) $od-$paths for a given departure-time t_o.

Facts of the FIFO property. We consider *networks* $(G = (V, A), (D[a])_{a\in A})$ with continuous arc-delay functions, possessing the *FIFO* (a.k.a. *non-overtaking*) property, according to which all arc-arrival-time functions are non-decreasing:

$$\forall t_u, t'_u \in \mathbb{R}, \forall uv \in A, t_u > t'_u \Rightarrow Arr[uv](t_u) \ge Arr[uv](t'_u) \tag{31}$$

The FIFO property is *strict*, if the above inequality is strict. The FIFO property implies that: (i) the slope of any arc-delay function is greater than -1; (ii) the slope of any path-delay or shortest-travel-time function is greater than -1. The *strict* FIFO property implies *subpath optimality* of shortest paths. For formal statements and proofs of these facts, see [50].

Towards a time-dependent distance oracle. Our approach for providing a time-dependent distance oracle is inspired by the generic approach for general *undirected* graphs under *static* travel-time metrics. However, we have to tackle the two main challenges of *directedness* and *time-dependence*. Notice that together these two challenges imply an *asymmetric* distance metric which also *evolves* with time. Consequently, to achieve a smooth transition from the static and undirected world towards the time-dependent and directed world, we have to quantify the *degrees of asymmetry and evolution* in our metric. Towards this direction, we make two assumptions on the kind of shortest-travel-time functions in the network. Both assumptions are quite natural and justified by a thorough investigation of historic traffic data for the city of Berlin, kindly provided to us by TomTom [33] (see [50] for a more detailed justification). The first assumption, called *Bounded Travel-Time Slopes*, asserts that the partial derivatives of the shortest-travel-time functions between any pair of origin-destination vertices are bounded in a given fixed interval $[\Lambda_{\min}, \Lambda_{\max}]$.

Assumption 1 (Bounded Travel-Time Slopes)**.** There are *constants* $\Lambda_{\min} > -1$ and $\Lambda_{\max} \ge 0$ s.t.: $\forall(o, d) \in V \times V,\ \forall t_1 < t_2,\ \frac{D[o,d](t_1)-D[o,d](t_2)}{t_1-t_2} \in [\Lambda_{\min}, \Lambda_{\max}]$.

The second assumption, called *Bounded Opposite Trips*, asserts that for any given departure time, the shortest-travel-time from o to d is not more than a *constant* $\zeta \ge 1$ times the shortest-travel-time in the opposite direction (but not necessarily along the same path).

Assumption 2 (Bounded Opposite Trips)**.** There is a *constant* $\zeta \ge 1$ such that: $\forall(o, d) \in V \times V,\ \forall t \in [0, T),\ D[o, d](t) \le \zeta \cdot D[d, o](t)$.

As we show in Section 3.4, the parameters $\Lambda_{\max}$ and ζ allow us to quantify the degree of asymmetry and evolution in time in our distance metric and achieve the aforementioned smooth transition. Another assumption we make and which can be easily guaranteed is that the maximum out-degree is bounded by 2.

Overview of our approach. We follow (at a high level) the typical approach adopted for the construction of approximate distance oracles in the static case. In particular, we start by selecting a subset $L \subset V$ of *landmarks*, i.e., vertices which will act as reference points for our distance summaries. For our oracle to work, several ways to choose L would be acceptable. Nevertheless, for the sake of the analysis we assume that this is done by deciding for each vertex randomly and independently with probability $\rho \in (0, 1)$ whether it belongs to L. After having L fixed, our approach is deterministic. We start by constructing (concurrently, per landmark) and storing the *distance summaries*, i.e., all landmark-to-vertex $(1 + \varepsilon)-$approximate travel-time functions, in time $\mathrm{o}\big((1 + K^*)n^2\big)$ and consuming space $\mathrm{o}\big((1 + K^*)n^2\big)$ which is indeed asymptotically optimal w.r.t. the required approximation guarantee (cf. Section 3.3). Then, we provide two approximation algorithms for arbitrary queries $(o, d, t_o) \in V \times V \times [0, T)$. The first (**FCA**) is a simple *sublinear*-time constant-approximation algorithm (cf. Section 3.4.1). The second (**RQA**) is a recursive algorithm growing small **TDD** outgoing balls from vertices in the vicinity of the origin, until either a satisfactory approximation guarantee is achieved, or an upper bound r on the depth of the recursion (the *recursion budget*) has been exhausted. **RQA** finally responds with a $(1 + \sigma)-$approximate travel-time to the query in *sublinear* time, for any constant $\sigma > \varepsilon$ (cf. Section 3.4.2). As it is customary in the distance oracle literature, the query times of our algorithms concern the determination of (upper bounds on) shortest-travel-time from o to d. An actual path guaranteeing this bound can be reported in additional time that is linear in the number of its arcs.

3.3 Preprocessing Distance Summaries

We now demonstrate how to construct the preprocessed information that will comprise the *distance summaries* of the oracle, i.e., all landmark-to-vertex shortest-travel-time functions. If there exist $K^* \geq 1$ *concavity-spoiling* breakpoints among the arc-delay functions, then we do the following: For each of them (which is a departure-time t_u from the tail u of an arc $uv \in A$) we run a variant of **TDD** with root (u, t_u) on the *reverse network* $(\overleftarrow{G} = (V, A, (\overleftarrow{D}[a])_{a \in A})$, where $\overleftarrow{D}[uv]$ is the delay of arc uv, measured now as a function of the arrival-time t_v at the tail v. The algorithm proceeds backwards both along the connecting path (from the destination towards the origin) and in time. As a result, we compute all *latest-departure-times* from landmarks that allow us to determine the images (i.e., projections to appropriate departure-times from all possible origins) of concavity-spoiling breakpoints. For each landmark, we repeat the procedure described in the rest of this section for every $K^* + 1$ subinterval of $[0, T)$ determined by *consecutive* images of concavity-spoiling breakpoints. Within each subinterval all arc-travel-time functions are concave, as required in our analysis.

We must construct in polynomial time, for all $(\ell, v) \in L \times V$, succinctly represented upper-bounding $(1 + \varepsilon)-$approximations $\Delta[\ell, v] : [0, T) \to \mathbb{R}_{>0}$ of the shortest-travel-time functions $D[\ell, v] : [0, T) \to \mathbb{R}_{>0}$. An algorithm providing such functions in a *point-to-point* fashion was proposed in [36]. For each landmark $\ell \in L$, it has to be executed n times so as to construct all the required landmark-to-vertex approximate functions. The main idea of that algorithm is to keep sampling the travel-time axis of the unknown function $D[\ell, v]$ at a logarithmically growing scale, until its slope becomes less than 1. It then samples the departure-time axis via bisection, until the required approximation guarantee is achieved. All the sample points (in both phases) correspond to breakpoints of a lower-approximating function. The upper-approximating function has at most twice as many points. The number of breakpoints returned may be suboptimal, given the required approximation guarantee: even for an affine shortest-travel-time function with slope in $(1, 2]$ it would require a number of points logarithmic in the ratio of max-to-min travel-time values from ℓ

to v, despite the fact that we could avoid all intermediate breakpoints for the upper-approximating function.

Our solution is an improvement of the approach in [36] in two aspects: (i) it computes *concurrently* all the required approximate distance functions from a given landmark, at a cost equal to that of a single (worst-case with respect to the given origin and all possible destinations) point-to-point approximation of [36]; (ii) within every subinterval of consecutive images of concavity-spoiling breakpoints, it provides asymptotically optimal space per landmark, which is also independent of the network size per landmark-vertex pair, implying that the required preprocessing space per vertex is $\mathcal{O}(|L|)$. This is also claimed in [36], but it is actually true only for their second phase (the bisection). For the first phase of their algorithm, there is no such guarantee. Even for a linear arc-travel-time function, the first phase of that algorithm would still require a number of samples which is logarithmic in the max-to-min travel-time ratio.

Our algorithm, in order to achieve a concurrent one-to-all construction of upper-bounding approximations from a given landmark $\ell \in L$, is purely based on bisection. This is done because the departure-time axis is common for all these unknown functions $(D[\ell, v])_{v\in V}$. In order for this technique to work, despite the fact that the slopes may be greater than one, a crucial ingredient is an *exact closed-form estimation* of the worst-case absolute error that we provide. This helps our construction to indeed consider only the necessary sampling points as breakpoints of the corresponding (concurrently constructed) shortest travel-time functions. It is mentioned that this guarantee could also be used in the first phase of the approximation algorithm in [36], in order to discard all unnecessary sampling points from being actual breakpoints in the approximate functions.

In a nutshell, we construct two continuous pwl-approximations of the *unknown* shortest-travel-time function $D[\ell, v] : [0, T) \to \mathbb{R}_{>0}$, an upper-bounding approximate function $\overline{D}[\ell, v]$ (playing the role of $\Delta[\ell, v]$) and a lower-bounding approximate function $\underline{D}[\ell, v]$. Our construction guarantees that the exact function is always "sandwiched" between these two approximations. For a given landmark $\ell \in L$ and a subinterval $[t_s, t_f) \subseteq [0, T)$ of departure times from ℓ, in which all the (unknown) shortest-travel-time functions from ℓ are concave, the algorithm proceeds as follows (details are provided in [50]): The current subinterval $[t_s, t_f)$ is bisected in the middle $t_m = \frac{t_s+t_f}{2}$. The result of this bisection is for the lower-approximating function $\underline{D}[\ell, v]$ to be augmented by the new breakpoint t_m, for all still *active* (having not yet met their required approximation guarantee) destination vertices v w.r.t. $[t_s, t_f)$. Our next step is, for each $v \in V$, to check whether the upper-approximating function $\overline{D}[\ell, v]$, consisting of the lower-envelope of the tangents of $D[\ell, v]$ at t_s, t_m and t_f, i.e., at most five breakpoints for the subinterval $[t_s, t_f)$, is already a $(1 + \varepsilon)-$approximation of $\underline{D}[\ell, v]$ within $[t_s, t_m)$ and $[t_m, t_f)$. Each destination vertex that is already satisfied by the current approximation becomes *inactive* for the subsequent subintervals. If any of the two subintervals still has active destination nodes, it is recursively bisected.

$L[\ell, v]$ and $U[\ell, v]$ denote the numbers of breakpoints for $\underline{D}[\ell, v]$ and $\overline{D}[\ell, v]$, $U = \max_{\ell,v}\{U[\ell, v]\}$, and TDP is the number of shortest-path probes during a bisection. By construction it holds that $U[\ell, v] \leq 2 \cdot L[\ell, v]$ (for an explanation see [50]). The expected number of landmarks is $\mathbb{E}\{|L|\} = \rho n$. It is then easy to deduce the required time and space complexity of our entire preprocessing.

Theorem 3.2. The preprocessing has expected space/time complexities $\mathbb{E}\{\mathcal{S}\} \in \mathcal{O}\big(\rho n^2(1 + K^*)U\big)$ and $\mathbb{E}\{\mathcal{P}\} \in \mathcal{O}\big(\rho n^2 \log(n) \log\log(K_{\max})(1 + K^*)TDP\big)$.

U and TDP are independent of n (cf. [50]), so we treat them as constants. If all arc-travel-time functions are concave, i.e., $K^* = 0$, then we achieve subquadratic preprocessing space and time $\forall\rho \in \mathcal{O}(n^{-\alpha})$, where $0 < \alpha < 1$. Real data (e.g., TomTom's traffic data for the city of Berlin [33]) demonstrate that: (i) only a small fraction of the arc-travel-time functions exhibit non-constant behaviour; (ii) for the vast majority of these non-constant-delay arcs, their functions are either concave, or can be very tightly approximated by a typical *concave* bell-shaped pwl function. It is only a tiny subset of critical arcs (e.g., bottleneck road segments) for which it would be meaningful to consider non-concave behaviour. Therefore, $K^* \in \mathrm{o}(n)$ is the typical case. E.g., assuming

$K^* \in \mathcal{O}(\text{polylog}(n))$, we can fine-tune ρ and the parameters σ, r (cf. Section 3.4.2) so as to achieve *subquadratic* preprocessing space and time. In particular, for $K^* \in \mathcal{O}(\log(n))$ and $K_{\max} \in \mathcal{O}(1)$, $\forall \gamma > \frac{1}{2}$, $\mathbb{E}\{\mathcal{S}\} \in \mathcal{O}\big(n^{2-\varepsilon/(\gamma\psi)} \log(n)\big)$ and $\mathbb{E}\{\mathcal{P}\} \in \mathcal{O}\big(n^{2-\varepsilon/(\gamma\psi)} \log^2(n)\big)$, where $\psi = \psi(\zeta, \Lambda_{\max})$ is a constant that will be specified in Theorem 3.3. More details are provided in [50].

3.4 Query Algorithms

3.4.1 Constant-approximation query algorithm.

Our next step towards a distance oracle is to provide a fast query algorithm providing constant approximations to the actual shortest-travel-time values of arbitrary queries $(o, d, t_o) \in V \times V \times [0, T)$. Here we propose such a query algorithm, called *Forward Constant Approximation* (**FCA**), which grows an outgoing ball $B_o \equiv B[o](t_o) = \{x \in V : D[o, x](t_o) < D[o, \ell_o](t_o)\}$ around (o, t_o) by running **TDD**, until either d or the closest landmark $\ell_o \in \arg\min_{\ell \in L}\{D[o, \ell](t_o)\}$ is scanned. We call $R_o = D[o, \ell_o](t_o)$ the *radius* of B_o. **FCA** returns either the exact travel-time value, or the approximate travel-time value via ℓ_o. Figure 6 gives an overview of the whole idea. The pseudocode is provided in [50].

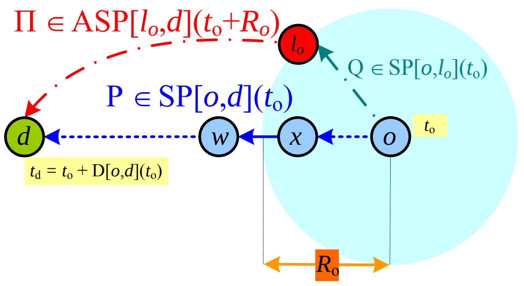


Figure 6: The rationale of **FCA**. The dashed (blue) path is a shortest $od-$path for query (o, d, t_o). The dashed-dotted (green and red) path is the via-landmark $od-$path indicated by the algorithm, if the destination vertex is out of the origin's **TDD** ball.

Correctness. The next theorem demonstrates that **FCA** returns $od-$paths whose travel-times are constant approximations to the shortest travel-times.

Theorem 3.3. $\forall (o, d, t_o) \in V \times V \times [0, T)$, **FCA** returns either an exact path $P \in SP[o, d](t_o)$, or a via-landmark $od-$path $Q \bullet \Pi$, s.t. $Q \in SP[o, \ell_o](t_o)$, $\Pi \in ASP[\ell_o, d](t_o + R_o)$, and $D[o, d](t_o) \leq R_o + \Delta[\ell_o, d](t_o + R_o) \leq (1 + \varepsilon) \cdot D[o, d](t_o) + \psi \cdot R_o \leq (1 + \varepsilon + \psi) \cdot D[o, d](t_o)$, where $\psi = 1 + \Lambda_{\max}(1 + \varepsilon)(1 + 2\zeta + \Lambda_{\max}\zeta) + (1 + \varepsilon)\zeta$.

Note that **FCA** is a generalization of the 3$-$approximation algorithm in [3] for symmetric (i.e., $\zeta = 1$) and time-independent (i.e., $\Lambda_{\min} = \Lambda_{\max} = 0$) network instances, the only difference being that the stored distance summaries we consider are $(1 + \varepsilon)-$approximations of the actual shortest-travel-times. Observe that our algorithm smoothly departs, through the parameters ζ and $\Lambda_{\max}$, towards both *asymmetry* and *time-dependence* of the travel-time metric.

Complexity. The main cost of **FCA** is to grow the ball $B_o = B[o](t_o)$ by running **TDD**. Therefore, what really matters is the number of vertices in B_o, since the maximum out-degree is 2. L is chosen randomly by selecting each vertex v to become a landmark independently of other vertices, with probability $\rho \in (0, 1)$. Clearly $\mathbb{E}\{|B_o|\} = 1/\rho$, and moreover (as a geometrically distributed random variable), $\forall k \geq 1$,$\mathbb{P}\{|B_o| > k\} = (1 - \rho)^k \leq e^{-\rho k}$. By setting $k = (1/\rho)\ln(1/\rho)$ we conclude that: $\mathbb{P}\{|B_o| > (1/\rho)\ln(1/\rho)\} \leq \rho$. Since the maximum out-degree is 2, **TDD** will relax at most $2k$ arcs. Hence, for the query-time complexity $\mathcal{Q}_{FCA}$ of **FCA** we conclude that $\mathbb{E}\{\mathcal{Q}_{FCA}\} \in \mathcal{O}((1/\rho)\ln(1/\rho)\log\log(K_{\max}))$, and $\mathbb{P}\big\{\mathcal{Q}_{FCA} \in \Omega\big((1/\rho)\ln^2(1/\rho)\log\log(K_{\max})\big)\big\} \in \mathcal{O}(\rho)$.

3.4.2 $(1+\sigma)-$approximate query algorithm.

The *Recursive Query Algorithm* (**RQA**) improves the approximation guarantee of the chosen $od-$path provided by **FCA**, by exploiting carefully a number (called the *recursion budget*) of recursive accesses to the preprocessed information, each of which produces (via a call to **FCA**) another candidate $od-$path sol_i. The crux of our approach is the following: We assure that, unless the required approximation guarantee has already been reached by a candidate solution, the recursion budget must be exhausted and the sequence of radii of the consecutive balls that we grow recursively is lower-bounded by a *geometrically increasing* sequence. We prove that this sequence can only have a *constant* number of elements, since the sum of all these radii provides a lower bound on the shortest-travel-time that we seek.

A similar approach was proposed for *undirected* and *static* sparse networks [3], in which a number of recursively growing balls (up to the recursion budget) is used in the vicinities of *both* the origin *and* the destination nodes, before eventually applying a constant-approximation algorithm to close the gap, so as to achieve improved approximation guarantees.

In our case the network is both directed and time-dependent. Due to our ignorance of the exact arrival time at the destination, it is difficult (if at all possible) to grow incoming balls in the vicinity of the destination node. Hence, our only choice is to build a recursive argument that grows outgoing balls in the vicinity of the origin, since we only know the requested departure-time from it. This is exactly what we do: So long as we have not discovered the destination node within the explored area around the origin, and there is still some remaining recursion budget, we "guess" (by exhaustively searching for it) the next node w_k along the (unknown) shortest $od-$path. We then grow a new out-ball from the new center $(w_k, t_k = t_o + D[o, w_k](t_o))$, until we reach the closest landmark-vertex ℓ_k to it, at distance $R_k = D[w_k, \ell_k](t_k)$. This new landmark offers an alternative $od-$path $sol_k = P_{o,k} \bullet Q_k \bullet \Pi_k$ by a new application of **FCA**, where $P_{o,k} \in SP[o, w_k](t_o)$, $Q_k \in SP[w_k, \ell_k](t_k)$, and $\Pi_k \in ASP[\ell_k, d](t_k + R_k)$ is the approximate suffix subpath provided by the distance oracle. Observe that sol_k uses a *longer* optimal prefix-subpath P_k which is then completed with a shorter approximate suffix-subpath $Q_k \bullet \Pi_k$. The pseudocode is provided in [50]. Figure 7 provides an overview of **RQA**'s execution.

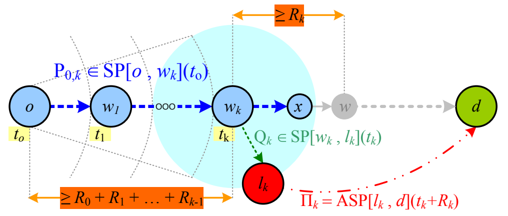

$\geq R_k$
$P_{0,k} \in SP[o , w_k](t_o)$
o
w_1
ooo
w_k
x
w
d
t_o
t_1
t_k
$Q_k \in SP[w_k , l_k](t_k)$
l_k
$\geq R_0 + R_1 + \ldots + R_{k-1}$
$\Pi_k = ASP[l_k , d](t_k+R_k)$

Figure 7: Overview of the execution of **RQA**.

Correctness & Quality. The correctness of **RQA** implies that the algorithm always returns some $od-$path. This is true due to the fact that it either discovers the destination node d as it explores new nodes in the vicinity of the origin node o, or it returns the shortest of the approximate $od-$paths $sol_0, \ldots, sol_r$ via one of the closest landmarks $\ell_o, \ldots, \ell_r$ to "guessed" nodes $w_0 = o, w_1, \ldots, w_r$ along the shortest $od-$path $P \in SP[o, d](t_o)$, where r is the recursion budget. Since the preprocessed distance summaries stored by the oracle provide approximate travel-times corresponding to actual paths from landmarks to vertices in the graph, it is clear that **RQA** always implies an $od-$path whose travel-time does not exceed the alleged upper bound on the actual distance.

Our next task is to study the quality of the provided stretch $1 + \sigma$ guaranteed by **RQA**. Let $\delta > 0$ be a parameter such that $\sigma = \varepsilon + \delta$ and recall the definition of ψ from Theorem 3.3. In [50] it is shown that the sequence of ball radii grown from vertices of the shortest $od-$path $P[o, d](t_o)$ by the recursive calls of **RQA** is lower-bounded by a *geometrically increasing* sequence. The next theorem shows that **RQA** indeed provides $(1+\sigma)-$approximate distances in response to arbitrary queries $(o, d, t_o) \in V \times V \times [0, T)$.

Theorem 3.4. For the stretch of **RQA** the following hold:

1. If $r = \left\lceil \frac{\ln\left(1+\frac{\varepsilon}{\delta}\right)}{\ln\left(1+\frac{\varepsilon}{\psi}\right)} \right\rceil - 1$ for $\delta > 0$, then, **RQA** guarantees a stretch $1+\sigma = 1+\varepsilon+\delta$.

2. For a given recursion budget $r \in \mathbb{N}$, **RQA** guarantees stretch $1+\sigma$, where $\sigma = \sigma(r) \leq \frac{\varepsilon \cdot (1+\varepsilon/\psi)^{r+1}}{(1+\varepsilon/\psi)^{r+1}-1}$.

Note that for time-independent, undirected-graphs (for which $\Lambda_{\min} = \Lambda_{\max} = 0$ and $\zeta = 1$) it holds that $\psi = 2+\varepsilon$. If we equip our oracle with *exact* rather than $(1+\varepsilon)-$approximate landmark-to-vertex distances (i.e., $\varepsilon = 0$), then in order to achieve $\sigma = \delta = \frac{2}{t+1}$ for some positive integer t, our recursion budget r is *upper bounded* by $\frac{\psi}{\delta} - 1 = t$. This is exactly the amount of recursion required by the approach in [3] to assure the same approximation guarantee. That is, at its one extreme ($\Lambda_{\min} = \Lambda_{\max} = 0,\ \zeta = 1,\ \psi = 2$) our approach matches the bounds in [3] for the same class of graphs, without the need to grow balls from both the origin and destination vertices. Moreover, our approach allows for a *smooth* transition from static and undirected-graphs to directed-graphs with FIFO arc-delay functions. The required recursion budget now depends not only on the targeted approximation guarantee, but also on the degree of asymmetry (the value of $\zeta \geq 1$) and the steepness of the shortest-travel-time functions (the value of $\Lambda_{\max}$) for the time-dependent case. It is noted that we have recently become aware of an improved bidirectional approximate distance oracle for static undirected graphs [2] which outperforms [3] in the stretch-time-space tradeoff.

Complexity. It only remains to determine the query-time complexity $\mathcal{Q}_{RQA}$ of **RQA**. This is provided by the following theorem.

Theorem 3.5. For networks having $|A|/|V| \in \mathcal{O}(1)$, the expected running time of **RQA** is $\mathbb{E}\{\mathcal{Q}_{RQA}\} \in \mathcal{O}((1/\rho)^r \cdot \ln(1/\rho) \cdot \log\log(K_{\max}))$, and it holds that:
$\mathbb{P}\left\{\mathcal{Q}_{RQA} \in \mathcal{O}\left(\left(\frac{\ln(n)}{\rho}\right)^r \cdot [\ln\ln(n) + \ln(1/\rho)] \cdot \log\log(K_{\max})\right)\right\} \in 1 - \mathcal{O}\left(\frac{1}{n}\right)$.

Continuing the discussion in the paragraph following Theorem 3.2, we can fine-tune the parameters σ, r so as to achieve, along with subquadratic space and preprocessing time, sublinear query-time complexity $\mathbb{E}\{\mathcal{Q}_{RQA}\} \in \mathcal{O}\left(n^{1/(2\gamma)} \log(n)\right)$, $\forall \gamma > \frac{1}{2}$. More details (and examples) are provided in [50].

4 Fast, Dynamic and Highly User-Configurable Route Planning

4.1 Introduction

Computing optimal routes in road networks has many applications such as navigation, logistics, traffic simulation or web-based route planning. Road networks are commonly formalized as weighted graphs and the optimal route is formalized as the shortest path in this graph. The environmental impact of a vehicle must be taken into account when determining the weights. For example roads with a lower driving speed are generally more eco-friendly as vehicles consume less energy when traversing them. However, different types of vehicles have diffenent energy consumption profiles. For example electric cars can recuperate when going downhill whereas a combustion-based vehicle can not. Yet, users have very specific and personal requirements and preferences, and solely optimizing travel time or eco-friendliness with regard to their car model will not to be favorable for them. Hence, maintaining a user-specific personal shortest path metric is required. Indeed, this can easily be achieved by adjusting the graph's weights according to preference.

Unfortunately, road graphs tend to be huge in practice with vertex counts in the tens of millions, rendering Dijkstra's algorithm [27] impracticable for interactive use: It needs several seconds of running time for a single path query. For practical performance on large road networks, preprocessing techniques that augment the network with auxiliary data in an (expensive) offline phase have proven useful. See [4] for an overview. Among the most successful techniques are Contraction Hierarchies (CH) by [39], which have been utilized in many scenarios. However, their preprocessing is in generally metric-dependent, e. g., edge weights (also called the graph metric) need to be known apriori. Substantial changes to the metric, e. g., due to the varying environmental impact of the vehicles, may require expensive recomputation. For this reason a Customizable Route Planning (CRP) approach was proposed in [19], extending the multi-level-overlay MLD techniques of [62, 46]. It works in three phases: In a first expensive phase, auxiliary data is computed that solely exploits the topological structure of the network, disregarding its metric. In a second much less expensive phase, this auxiliary data is *customized* to the specific metric, enabling fast queries in the third phase. In this work we extend CH to support such a three-phase approach, achieving similar robustness to metric changes at higher query speeds.

Nested Dissection Order One of the central building blocks of this paper is to use metric-independent nested dissection orders (ND-orders) for CH precomputation instead of the metric-dependent order of [39]. This approach was proposed by [7], and a preliminary case study can be found in [74]. A similar idea was followed by [24], where the authors employ partial CHs to engineer subroutines of their customization phase (they also had preliminary experiments on full CH). Worth mentioning are also the works of [59]. They consider small graphs of low treewidth and leverage this property to compute good orders and CHs (without explicitly using the term CH). Interestingly, our experiments show that also large road networks have relatively low treewidth. Real world road graphs with vertex counts in the 10^7 have treewidths in the 10^2.

Our Contribution The main contribution of our work is to show that Customizable Contraction Hierarchies (CCH) solely based on the ND-principle are feasible and practical. Compared to CRP [19] we achieve a similar preprocessing–query tradeoff, albeit with slightly better query performance at slightly slower customization speed (and somewhat more space). Interestingly, for less well-behaved metrics such as travel distance, we achieve query times below the original metric-dependent CH of [39]. Besides this main results there are number of side results. We show that given a fixed contraction order a metric-independent CH can be constructed in time essentially linear in the Contraction Hierarchy with working space memory linear in the input graph. Our specialized algorithm has better theoretic worst case running times and performs significantly better in experiments than the dynamic adjacency arrays used in [39]. Another contribution of our work are perfect witness searches. We show that for ND-orders it is possible to construct CHs with a minimum number of arcs in about a minute on continental road graphs. Our construction algorithm has a running time performance completely independent of the weights used. We further show that an order based on nested dissection results in a constant factor approximation for metric-independent CHs on a class of graphs with very regular recursive vertex separators. Experimentally we show that road graphs have such a recursive separator structure.

Outline This work is organized as follows. Section 4.2 sets necessary notation, while Section 4.3 discusses metric-dependent orders traditionally used in Contraction Hierarchies (highlighting specifics of our implementation). Next, we discuss metric-independent orders in Section 4.4, construction of the corresponding CH in Section 4.5, and a preprocessing step for efficient enumeration of lower arc triangles in Section 4.6. In terms of the three-phase model, these steps correspond to the first phase: They only depend on the topology and can be preprocessed once–before considering metrics. Then, Section 4.7 considers the second phase, i. e., the customization of the datastructures w. r. t. to a given metric, while Section 4.8 describes the third phase: distance queries and path unpacking. Section 4.9 discusses extensions of the approach for enabling turn restrictions and costs.

4.2 Basics

We denote by $G = (V, E)$ an *undirected* n-vertex *graph* where V is the set of *vertices* and E the set of *edges*. Furthermore, $G = (V, A)$ denotes a *directed graph* where A is the set of *arcs*. A graph is *simple* if it has no loops or multi-edges. Graphs in this paper are always simple unless noted otherwise (e. g., in parts of Section 4.5). We denote by $N(v)$ the set of adjacent vertices of v in an undirected graph.

A *vertex separator* is a vertex subset $S \subseteq V$ whose removal separates G into two disconnected subgraphs induced by the vertex sets A and B. The sets S, A and B are disjoint and their union forms V. Note that the subgraphs induced by A and B are not necessarily connected and may be empty. A separator S is *balanced* if $|A|, |B| \leq 2n/3$.

A *vertex order* $\pi : \{1 \ldots n\} \rightarrow V$ is a bijection. Its inverse π^{-1} assigns each vertex a *rank*. Every undirected graph can be transformed into a *directed upward graph* with respect to a vertex order π, i. e., every edge $\{\pi(i), \pi(j)\}$ with $i < j$ is replace by an arc $(\pi(i), \pi(j))$. Note that all upward directed graphs are acyclic. We denote by $N_u(v)$ the neighbors of v with a higher rank than v and by $N_d(v)$ those with a lower rank than v. We denote by $d_u(v) = |N_u(v)|$ the *upward degree* and by $d_d(v) = |N_d(v)|$ the *downward degree* of a vertex.

Undirected edge weights are denoted using $w : E \rightarrow \mathbb{R}^+$. With respect to a vertex order π we define an *upward weight* $w_u : E \rightarrow \mathbb{R}^+$ and a *downward weight* $w_d : E \rightarrow \mathbb{R}^+$. One-way streets are modeled by setting w_u or w_d to ∞.

A path p is a sequence of adjacent vertices and incident edges. Its *hop-length* is the number of edges in p. Its *weight-length* with respect to w is the sum over all edges' weights. Unless noted otherwise length always refers to weight-length in this paper. A shortest st-path is a path of minimum length between vertices s and t. The minimum length in G between two vertices is denoted by $\text{dist}_G(s, t)$. (We set $\text{dist}_G(s, t) = \infty$ if no path exists.) An *up-down path* p with respect to π is a path that can be split into an upward path p_u and a downward path p_d. The vertices in the upward path p_u must occur by increasing rank π^{-1} and the vertices in the downward path p_d must occur by decreasing rank π^{-1}.

The vertices of every acyclic directed graph (DAG) can be partitioned into *levels* $\ell : V \rightarrow \mathbb{N}$ such that for every arc (x, y) it holds that $\ell(x) < \ell(y)$. We only consider levels such that each vertex has the lowest possible level. Note that such levels can be computed in linear time given a directed acyclic graph.

The (unweighted) *vertex contraction* of v in G consists of removing v and all incident edges and inserting edges between all neighbors $N(v)$ if not already present. The inserted edges are refereed to as *shortcuts* and the other edges are *original edges*. Given an order π the *core graph* $G_{\pi,i}$ is obtained by contracting all vertices $\pi(1) \ldots \pi(i-1)$ in order of their rank. We call the original graph G augmented by the set of shortcuts a *contraction hierarchy* $G^*_\pi = \bigcup_i G_{\pi,i}$. Furthermore, we denote by $G^\wedge_\pi$ the corresponding upward directed graph.

Given a fixed weight w one can exploit that in many applications it is sufficient to (only) preserve all shortest path distances [39]. We define the *weighted vertex contraction* of a vertex v in the graph G as the operation of removing v and inserting the minimum number of shortcuts among the neighbors of v to obtain a graph G' such that $\text{dist}_G(x, y) = \text{dist}'_G(x, y)$ for all vertices $x \neq v$ and $y \neq v$. To compute G', we iterate over all pairs of neighbors x, y of v increasing by $\text{dist}_G(x, y)$. For each pair we check whether a xy-path of length $\text{dist}_G(x, y)$ exists in $G \backslash \{v\}$, i. e., we check whether removing v destroys the xy-shortest path. This check is called *witness search* [39] and the xy-path is called *witness* (if it exists). If a witness is found then we skip the pair and do

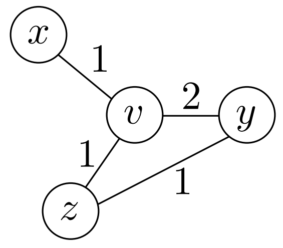


Figure 8: Contraction of v. If the pair x, y is considered first then a shortcut $\{x, y\}$ with weight 3 is inserted. If the pair x, z is considered first then an edge $\{x, z\}$ with weight 2 is inserted. This shortcut is part of a witness $x \to y \to z$ for the pair x, y. The shortcut $\{x, y\}$ is *not* added.

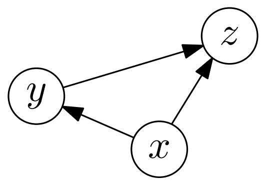


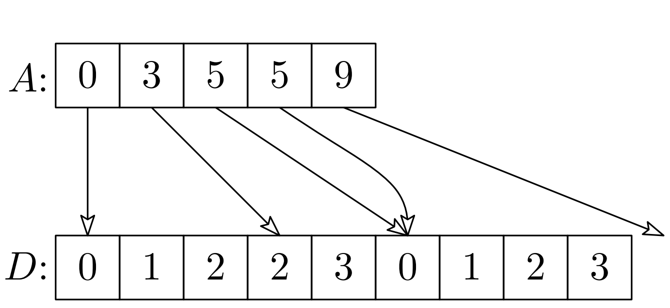
Figure 9: A triangle in $G^{\wedge}_{\pi}$. The triple (y, z, x) is a lower triangle of the arc (y, z). The triple (x, z, y) is an intermediate triangle of the arc (x, z). The triple (x, y, z) is an upper triangle of the arc (x, y).

nothing. Otherwise depending on whether an edge $\{x, y\}$ already exists we either decrease its weight to $\mathrm{dist}_G(x, y)$ or insert a shortcut edge with that weight to G. This new shortcut edge is considered in witness searches for subsequent neighbor pairs as part of G. It is important to iterate over the pairs increasing by $\mathrm{dist}_G(x, y)$ because otherwise more edges than strictly necessary can be inserted: Shorter shortcuts can make longer shortcuts superfluous. However, if we insert the shorter shortcut after the longer ones then the witness search will not consider them. See Figure 8 for an example. Note that the witness searches are expensive and therefore usually the witness search is aborted after a certain number of steps [39]. If no witness was found until then, we assume that none exists and add a shortcut. This does not affect the correctness of the technique but might result in slightly more shortcuts than necessary.

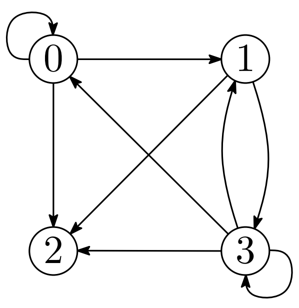
We call a witness search *without* such a one-sided error *perfect*. For an order π and a weight w the *weighted core graph* $G_{w,\pi,i}$ is obtained by contracting all vertices $\pi(1) \ldots \pi(i-1)$. The original graph G augmented by the set of weighted shortcuts is called a *weighted contraction hierarchy* $G^*_{w,\pi}$. The corresponding upward directed graph is denoted by $G^{\wedge}_{w,\pi}$.

The search space $\mathrm{SS}(v)$ of a vertex v is the subgraph of $G^{\wedge}_{\pi}$ (respectively $G^{\wedge}_{w,\pi}$) reachable from v. For every vertex pair s and t, it has been shown that a shortest up-down path must exist. This up-down path can be found by running a bidirectional search from s restricted to $\mathrm{SS}(s)$ and from t restricted to $\mathrm{SS}(t)$ [39]. A graph is *chordal* if for every cycle of at least four vertices there exists a pair of vertices that are non-adjacent in the cycle but are connected by an edge. An alternative characterization is that a vertex order π exists such that for every i the neighbors of $\pi(i)$ in the $G_{\pi,i}$ form a clique [37]. Such an order is called a *perfect elimination order*.

The elimination tree $T_{G,\pi}$ is a tree directed towards its root $\pi(n)$. The parent of vertex $\pi(i)$ is its upward neighbor $v \in N_u(\pi(i))$ of minimal rank $\pi^{-1}(v)$. Note that this definition already yields a straightforward algorithm for constructing the elimination tree. As shown in [7] the set of vertices on the path from v to $\pi(n)$ is the set of vertices in $\mathrm{SS}(v)$. Computing a contraction hierarchy (without witness search) of graph G consists of computing a chordal supergraph G^*_{π} with perfect elimination order π. The height of the elimination tree corresponds to the maximum number of



(a) Adjacency array



(b) Represented graph

Figure 10: The left figure depicts two arrays: The top is the *index array* I and the bottom the *data array* D. The index array has $|V|+1$ entries. The data array has $|A|$ entires. Each entry in I is an index into the data array D. The neighbors of a vertex with ID x have the IDs $D[I[x]], D[I[x]+1], \ldots, D[I[x+1]-1]$.

vertices in the search space. Note that the elimination tree is only defined for undirected unweighted graphs.

A *lower triangle* of an arc (x, y) in $G_\pi^\wedge$ is a triple (x, y, z) such that arcs (z, x) and (z, y) exist. Similarly an *intermediate triangle* is a triple such that (x, z) and (z, y) exist and an *upper triangle* is a triple such that (x, z) and (y, z) exist. The situation is illustrated in Figure 9. Recall that arcs are directed according to rank and do not necessarily reflect travel direction.

4.2.1 Metrics

In the following, we denote weights on $G_\pi^\wedge$ as *metrics*. We say that a metric m *respects* a weight w of G if $\text{dist}_G(x, y) = \text{dist}_{G_\pi^*}(x, y)$ for all vertices x and y. Every weight on G can trivially be extended to a w-respecting metric by assigning the weights of w to the original arcs and ∞ to all shortcuts. We refer to this metric as the *w-initial metric*. A metric is called *customized* if for all lower triangles (x, y, z) the *lower triangle inequality* holds, i. e., $m(x, y) \leq m(z, x) + m(z, y)$. Note that the w-initial is w-respecting but it is not customized.

Lemma. Let m be a customized metric on $G_\pi^\wedge$ respecting a weight w on a graph G. For all pairs s and t with $\text{dist}_G(s, t) \neq \infty$ a shortest up-down st-path exists in $G_\pi^\wedge$.

Proof. As $\text{dist}_G(s, t) \neq \infty$ a shortest st-path in G must exist. If on $G_\pi^\wedge$ this is not an up-down path, then it must contain a subpath $x \to y \to z$ with $\pi^{-1}(x) > \pi^{-1}(y)$ and $\pi^{-1}(y) < \pi^{-1}(z)$. As y is contracted before x and z an arc (x, y) must exist. As (x, y, z) is a lower triangle and m is customized, we know that removing the vertex y from the path cannot make the path longer. As the path has only finitely many vertices iteratively replacing these lower triangles yields a shortest up-down path after finitely many steps. □

Denote by M_w the set of metrics that respect a weight w and are customized. A metric $m \in M_w$ is *w-maximum* if no other metric exists in M_w that has a higher weight on some arc. Analogously a *w-minimum* metric is one where no arc weight can be decreased. Note that both of these metrics are unique. Furthermore, a w-minimum metric can be characterized as one where every arc (x, y) has the weight of a shortest xy-path.

4.2.2 Adjacency Array

An *adjacency array* is a data structure that is used to map IDs onto other objects. As depicted in Figure 10, it consists of two arrays and can be used to store graphs by mapping a vertex ID onto the neighboring vertices' IDs. Note that adjacency arrays also have other applications: For example instead of mapping vertex IDs on the neighboring vertices' IDs, it could map onto the incident arc IDs. Another example would be to map the ID of an arc (x, y) onto vertex IDs z_i such that each (x, y, z_i) forms a lower triangle of (x, y). Adjacency arrays are an omnipresent basic building block in efficient graph algorithms and as such they have many different names. Other names include *compressed row* and *forward star*.

4.3 Metric-Dependent Orders

Most papers using Contraction Hierarchies use greedy orders in the spirit of [39]. As the exact details vary from paper to paper, we describe our precise variant in this section. Our witness search aborts once it finds some path shorter than the shortcut—or when both forward and backward search each have settled at most p vertices. For most experiments we choose $p = 50$. The only exception is the distance metric on road graphs, where we set $p = 1500$. We found that a higher value of p increases the time per witness-search but leads to sparser cores. For the distance metric we needed a high value because otherwise our cores get too dense. This effect did not occur for the other weights considered in the experiments. Our weighting heuristic is similar to the one of [1]. We denote by $L(x)$ a value that approximates the level of vertex x. Initially all $L(x)$ are 0. If x is contracted then for every incident edge $\{x, y\}$ we perform $\ell(y) \leftarrow \max\{\ell(y), \ell(x) + 1\}$. We further store for every arc a a hop length $h(a)$. This is the number of arcs that the shortcut represents if fully unpacked. Denote by $D(x)$ the set of arcs removed if x is contracted and by $A(x)$ the set of arcs that are inserted. Note that $A(x)$ is not necessarily a full clique because of the witness search and because some edges may already exist. We greedily contract a vertex x that minimizes its *importance* $I(x)$ defined by

$$I(x) = L(x) + \frac{|A(x)|}{|D(x)|} + \frac{\sum_{a \in A(x)} h(a)}{\sum_{a \in D(x)} h(a)}$$

We maintain a priority queue that contains all vertices weighted by I. Initially all vertices are inserted with their exact importance. As long as the queue is not empty, we remove a vertex x with minimum importance $I(x)$ and contract it. This modifies the importance of other vertices. However, our weighting function is chosen such that only the importance of adjacent vertices is influenced (if the witness search was perfect). We therefore only update the importance values of all vertices y in the queue that are adjacent to x. In practice (with limited witness search), we sometimes choose a vertex x with a sightly suboptimal $I(x)$. However, preliminary experiments have shown that this effect can be safely ignored.

4.4 Metric-Independent Order

The metric-dependent orders presented in the previous section lead to very good results on road graphs with travel time metric. However, the results for the distance metric are not as good and the orders are completely impracticable to compute Contraction Hierarchies without witness search.To support metric-independence, we use *nested dissection* orders as suggested in [7] (or ND-orders for short). An order π for G is computed recursively by determining a balanced separator S of minimum cardinality that splits G into two parts induced by the vertex sets A and B. The vertices of S are assigned to $\pi(n-|S|) \ldots \pi(n)$ in an arbitrary order. Orders π_A and π_B are computed recursively and assigned to $\pi(1) \ldots \pi(|A|)$ and $\pi(|A|+1) \ldots \pi(|A|+|B|)$, respectively. The base case of the recursion is reached when the graphs are empty. Computing ND-orders requires good graph bisectors, which in theory is NP-hard. However, recent years have seen heuristics that solve the problem very well even for continental road graphs [61, 21, 20]. This justifies assuming in our particular context that an efficient bisection oracle exists. Note that graph bisectors usually compute edge cuts and not vertex separators. On our instances, a vertex separator is derived by arbitrarily picking for every edge one of its incident vertices. After having obtained the nested dissection order we reorder the in-memory vertex IDs of the input graph accordingly, i. e., the contraction order of the reordered graph is the identity. This improves cache locality and we have seen a resulting acceleration of a factor 2 to 3 in query times. In the remainder of this section we prepare and provide a theoretical approximation result.

For $\alpha \in (0, 1)$, let K_α, be a class of graphs that is closed under subgraph construction and admits balanced separators S of cardinality $O(n^\alpha)$.

Lemma. For every $G \in K_\alpha$ a ND-order results in $O(n^\alpha)$ vertices in the maximum search space.

The proof of this lemma is a straightforward argument using a geometric series as described in [7]. As a direct consequence, the average number of vertices is also in $O(n^\alpha)$ and the number of arcs in $O(n^{2\alpha})$.

Lemma. For every connected graph G with minimum balanced separator S and every order π, the chordal supergraph G^*_π contains a clique of $|S|$ vertices. Furthermore, there are at least $n/3$ vertices such that this clique is a subgraph of their search space in $G^\wedge_\pi$.

This lemma is a minor adaptation and extension of [51]. We provide the full proof for self-containedness.

Proof. Consider the subgraphs G_i of G^*_π induced by the vertices $\pi(1) \dots \pi(i)$ (not to be confused with the core graphs $G_{\pi,i}$). Choose the smallest i such that a connected component A exists in G_i such that $|A| \geq n/3$. As G is connected, such an A must exist. We distinguish two cases:

1. $|A| \leq 2n/3$: Consider the set of vertices S' adjacent to A in G^*_π. Let B be the set of all remaining vertices. S' is by definition a separator. It is balanced because $|A| \leq 2n/3$ and $|B| = n - \underbrace{|A|}_{} - \underbrace{|S'|}_{} \leq 2n/3$. As S is minimum we have that $|S'| \geq |S|$. For every pair of vertices $u, v \in S'$ there exists a path through A as A is connected. As u and v have the highest ranks on this path (the vertices in A have rank $1 \dots i$), there must be and edge $\{u, v\}$ in G^*. S' is therefore a clique. Furthermore, from every $u \in A$ to every $v \in S'$ there exists a path such that v has the highest rank. Hence, v is in the search space of u, i.e, there are at least$|A| \geq n/3$ vertices whose search space contains the full S'-clique.

2. $|A| > 2n/3$: As i is minimum, we know that $\pi(i) \in A$ and that removing it disconnects A into connected subgraphs $C_1 \dots C_k$. We know that $|C_j| < n/3$ for all j because i is minimum. We further know that $|A| = 1 + \sum |C_j| > 2n/3$. We can therefore select a subset of components C_k such that the number of their vertices is at most $2n/3$ but at least $n/3$. Denote by A' their union. (Note that A' does not contain $\pi(i)$.) Consider the vertices S' adjacent to A' in G^*_π. (The set S' contains $\pi(i)$.) Using an argument similar to Case 1, one can show that $|S'| \geq |S|$. But since A' is not connected, we cannot directly use the same argument to show that S' forms a clique in G^*. Observe that $A' \cup \{\pi(i)\}$ is connected and thus the argument can be applied to $S' \backslash \{\pi(i)\}$ showing that it forms a clique. This clique can be enlarged by adding $\pi(i)$ as for every $v \in S' \backslash \{\pi(i)\}$ a path through one of the components C_k exists where v and $\pi(i)$ have the highest ranks and thus an edge $\{v, \pi(i)\}$ must exist. The vertex set S' therefore forms a clique of at least the required size. It remains to show that enough vertices exist whose search space contains the S' clique. As $\pi(i)$ has the lowest rank in the S' clique the whole clique is contained within the search space of $\pi(i)$. It is thus sufficient to show that $\pi(i)$ is contained in enough search spaces. As $\pi(i)$ is adjacent to each component C_k a path from each vertex $v \in A'$ to $\pi(i)$ exists such that $\pi(i)$ has maximum rank showing that S' is contained in the search space of v. This completes the proof as $|A'| \geq n/3$.

□

Theorem. Let G be a graph from K_α with a minimum balanced separator with $\Theta(n^\alpha)$ vertices then a ND-order gives an $O(1)$-approximation of the average and maximum search spaces of an optimal metric-independent contraction hierarchy in terms of vertices and arcs.

Proof. This key observation of this proof is that the top level separator solely dominates the performance. Denote by π the ND-order and by π_{opt} the optimal order. First we show a lower bound on the performance of π_{opt} and then show that π achieves this lower bound showing that π is an $O(1)$-approximation. As the minimum balanced separator has cardinality $\Theta(n^\alpha)$ we know by Lemma 4.4 that at least $n/3$ vertices exist, whose search space in $G^\wedge_{\pi_{opt}}$ contains a clique with $\Theta(n^\alpha)$ vertices. Thus the maximum number of vertices in a search space is $\Omega(n^\alpha)$ as it must contain this clique and as the clique is dense the maximum number of arcs is in $\Omega(n^{2\alpha})$. The average number of vertices is $2/3 \cdot \Omega(0) + 1/3 \cdot \Omega(n) = \Omega(n)$ and as the clique is dense the average number of arcs is in $\Omega(n^{2\alpha})$. From Lemma 4.4 we know that the number of vertices in the maximum search space of $G^\wedge_\pi$ is in $O(n^\alpha)$. A direct consequence is that the average number of vertices is also in $O(n^\alpha)$. In the worst case the search space is dense resulting in $O(n^{2\alpha})$ arcs in the average and the maximum search space. As the derived bounds are tight this shows that π is an $O(1)$-approximation. □

4.5 Constructing the Contraction Hierarchy

In this section, we describe how to efficiently compute the hierarchy $G^\wedge_\pi$ for a given graph G and order π. Weighted contraction hierarchies are commonly constructed using a dynamic adjacency array representation of the core graph. Our experiments show that this approach also works for the unweighted case, however, requiring more computational and memory resources because of the higher growth in shortcuts. It has been proposed [74] to use hash-tables on top of the dynamic graph structure to improve speed but at the cost of significantly increased memory consumption. In this section, we show that the contraction hierarchy construction can be done significantly faster on unweighted and undirected graphs. (Note that graph weights and directed arcs are handled during customization.)

Denote by n the number of vertices, m the number of edges in G, by m' the number of edges in $G^\wedge_\pi$, and by $\alpha(n)$ the inverse $A(n, n)$ Ackermann function. For simplicity we assume that G is connected. Our algorithm enumerates all arcs of $G^\wedge_\pi$ in $O(m'\alpha(n))$ running time and has a memory consumption in $O(m)$ (to store the arcs of $G^\wedge_\pi$, additional space in $O(m')$ is needed). The approach is heavily based upon the method of the quotient graph [41]. To the best of our knowledge it has not yet been applied in the context of route planning. We also were not able to find an complexity analysis for the specific variant employed by us. Therefore, in the remainder of this section, we both discuss the approach and present a running time analysis. As a first step, we describe a complex datastructure that supports efficient *edge* contraction and neighborhood enumeration. Then, we show how this datastructure is used to realize a datastructure that supports efficient *vertex* contraction and neighborhood enumeration.

4.5.1 Technicalities

In the following, we identify vertices with an ID from the range $1 \ldots n$. For edges we do not store any IDs. To avoid problems with ID-relabeling, we never remove vertices. That is, contracting a vertex v consists of removing all incident edges and connecting all adjacent vertices, but we do not remove v. After the vertex contraction v has degree 0. Contracting an edge $\{u, v\}$ consists of removing all edges $\{v, w\}$ and adding edges $\{u, w\}$ if necessary. After edge contraction, again, v has degree 0. Note that this makes edge contraction strictly speaking a non-symmetric operation. Enumerating the neighborhood of a vertex v (given by its unique ID) consists of enumerating the IDs of all adjacent vertices exactly once.

4.5.2 Efficient Edge Contraction

The core idea is to organize contracted vertices in a linked list. Even if G is simple, edge contraction can create unwanted multi edges or loops. We remove these unwanted edges during the enumeration. As an edge contraction does not create news edges we can remove at most as many as there were in G. To efficiently rewire the edges we further need a union find datastructure that introduces some $\alpha(n)$ terms. Our datastructure has an edge contraction in $O(1)$. The enumeration of the neighbors of v needs $O(d(v)\alpha(n))$ amortized running time. Finally there are global edge removal costs of at most $O(m\alpha(n))$ that do not depend on the operations applied to the datastructure.

We combine an adjacency array, a doubly linked list, a union-find datastructure and a boolean array. The adjacency array initially stores for every vertex in v the IDs of the adjacent vertices in G. The doubly linked list links together the vertices of G that have been contracted. We say that two vertices that are linked together are on the same *ring*. Initially no edges were contracted and therefore all rings only contain a single vertex. The union find datastructure is used to efficiently determine a representative vertex ID for every ring given a vertex of that ring. The boolean array is used to mark vertices and is needed to assure that the neighborhood iteration outputs no vertex twice and that v is not a neighbor of v. Initially all entries are false. After each neighborhood enumeration the entries are reset to false. All vertices in a ring are regarded as having degree 0 with the exception of the representative, which is regarded as incident to all edges incident to the ring.

Contracting an edge is the easy operation. During the enumeration most of the work occurs. To contract an edge $\{u, v\}$ we first check whether u and v are the representatives of their ring. If u or v is not then they have degree 0 and there is nothing to do. We merge the rings of u and v and unite u and v in the union find datastructure and choose either u or v as representative. To enumerate the neighbors of a vertex a we first check whether it is its own representative in the union find datastructure. If it is not then a has degree 0 as the edges have been contracted away. Otherwise we mark a in the boolean array. Next we iterate over all vertex IDs b in the linked ring of a. For every b we iterate over the vertex IDs c in the adjacency array for b. For every c we lookup its representative d in the union find datastructure. If d is not marked in the boolean array we found a new neighbor of a. We output it and mark it. Otherwise we do not output d but remove c from b's adjacency in the array. If this empties the adjacency of b we remove b from a's ring but keep b in the same union as a. After the enumeration we iterate a second time over it to reset the boolean array.

4.5.3 Analysis

We first analyze the memory consumption. There is no memory allocation during the algorithm and the sizes of the initial datastructure are dominated by the adjacency array that needs $O(m)$ space. The running time of an edge contraction is in $O(1)$ as all its operations are in $O(1)$. (Note that we are only checking whether u is the representative, we do not actually compute the representative if it is not u.) Analyzing the neighborhood enumeration is more complex. Three key insights are needed: First there are only m initial edges and therefore at most m entries can be deleted. The costs are accounted for in the global $O(m\alpha(n))$ term. The second insight is that as we remove empty adjacencies from the rings a ring never contains more pointers than vertices and therefore the time needed to follow the pointers is dominated by the time spend visiting the vertices. The third insight is that a second enumeration of v cannot find duplicates as they have been removed in the first iterations. Therefore reseting the boolean array is in $O(d(v))$.

4.5.4 Efficient Vertex Contraction

Based on the efficient edge contraction datastructure described above we design an efficient vertex contraction datastructure. The allowed operations are slightly more restrictive. We require that each enumeration of the neighborhood of v is followed by v's vertex contraction.

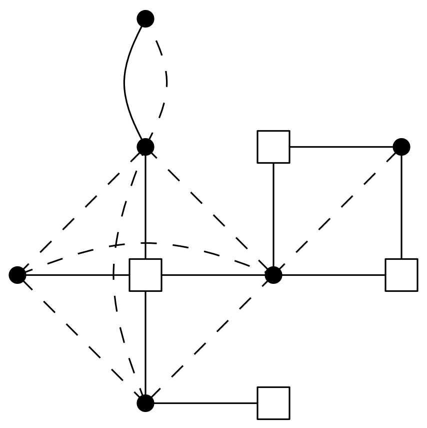


Figure 11: The dots represent vertices that have non-zero degree in $G_{\pi,i}$ and in $G'_{\pi,i}$. The squares are the additional super vertices in $G'_{\pi,i}$. The solid edges are in $G'_{\pi,i}$ and the dashed ones in $G_{\pi,i}$. Notice how the neighbors of each super vertex in $G'_{\pi,i}$ forms a clique in $G_{\pi,i}$. Furthermore, there are no two adjacent super vertices in $G'_{\pi,i}$, i. e., they form an independent set.

Instead of storing the graphs $G_{\pi,i}$ explicitly we store a different graph $G'_{\pi,i}$. We do not replace a contracted vertex v by a clique among its neighbors. Instead we replace it by a star with a virtual dummy vertex at its center. If a vertex is adjacent to a star center then it is recognized as being adjacent to all vertices in the star. If two star centers become adjacent we merge the stars by contracting the edge between the centers. The complexity of the resulting star is the sum of both original stars. This contrasts with explicitly representing the induced cliques whose complexity would grow super linearly. The idea is illustrated in Figure 11.

Formally the vertices that have non-zero degree in both $G_{\pi,i}$ and $G'_{\pi,i}$ are called *regular* vertices. The additional dummy vertices that have a non-zero degree in $G'_{\pi,i}$ are called *super* vertices. We maintain the invariant that $G'_{\pi,i}$ does not contain two adjacent super vertices. Furthermore, for every edge $\{x, y\}$ in $G_{\pi,i}$ there exists an edge $\{x, y\}$ in $G'_{\pi,i}$ or a path $x \to z \to y$ in $G'_{\pi,i}$ where z is a super vertex and x and y regular vertices.

An alternative way to describe the datastructure is to say that we maintain a graph on which we only perform edge contractions and we maintain an independent set of virtually contracted super vertices.

The datastructure only needs to support a single operation: Enumerating the neighbors of an arbitrary vertex x in $G_{\pi,i}$ followed by x's contraction. We actually do it in reversed order : We first contract v and then enumerate the neighbors of the new super vertex v. To contract x we first mark it as super vertex. We then enumerate its neighbor in $G'_{\pi,i}$ to determine all adjacent super vertices y_i. We then contract all edges $\{x, y_i\}$ to assure that x is no longer adjacent to any super vertex. Afterwards all neighbors of x in $G'_{\pi,i}$ are regular and therefore they coincide with those in $G_{\pi,i}$ making the enumeration straightforward.

4.5.5 Analysis

In addition to the edge contraction datastructure we need a boolean array to mark vertices as super nodes. The additional memory consumption is therefore in $O(n)$ and is negligible. The running time is shown using an amortized analysis. Denote by x the contracted vertex. There are three cost factors: The enumeration of the neighbors $y_1 \dots y_p$ before the contraction, the enumeration of the neighbors $z_1 \dots z_q$ after the contraction, and the contraction of the arcs. The enumeration of each y_i and z_i carries a cost of $\alpha(n)$ resulting from the underlying edge contraction datastructure. Note that while the y_i contain super and regular vertices the z_i contain only regular vertices. As there are at most as many regular y_i vertices as there are z_i vertices we can account for the regular y_i vertices in the costs of the z_i vertices. The remaining y_i are super vertices. Their enumeration is always followed by an edge contraction and therefore we account for their cost in the edge contraction costs. The enumeration costs of the y_i are therefore accounted for. As at most m edges can be contracted their total costs result in a global $O(m\alpha(n))$ term in the running time. As the number of z_i coincides with the out-degree of x in $G_\pi^\wedge$ we can account the costs of the z_i to the arcs of $G_\pi^\wedge$ resulting in $O(m'\alpha(n))$ total costs.

4.5.6 Obtaining statistics for badly conditioned hierarchies

For every graph G and order π yielding a small m' we efficiently construct and store G_π^* (and use it for route planning applications). However, even for orders yielding large m', we are interested in the characteristic numbers of G_π^*(e. g., to exactly quantify the quality (or badness) of an order). We obviously cannot store all arcs. But using the contraction graph datastructure, given enough time, we can count them (recall that our datastructure only requires $O(m)$ space). Furthermore, we can construct the elimination tree of $G_\pi^\wedge$ and compute the out-degree of all vertices. From these we derive the size of $G_\pi^\wedge$ (i. e., m') as well as the average and maximum search space size in $G_\pi^\wedge$.

4.6 Enumerating Triangles

Efficiently enumerating all lower triangles of an arc is an important base operation of the customization and path unpacking algorithms (see Section 4.7 and Section 4.8). It can be implemented using adjacency arrays or accelerated using extra preprocessing. Note that in addition to the vertices of a triangle we are interested in the IDs of the participating arcs.

4.6.1 Basic Triangle Enumeration

Construct an upward and a downward adjacency array for $G_\pi^\wedge$, where incident arcs are ordered by their head vertex ID. Unlike common practice, we also assign and store arc IDs. (By lexicographically assigning arc IDs we eliminate the need for arc IDs in the upward adjacency array.) Denote by $N_u(v)$ the upward neighborhood of v and by $N_d(v)$ the downward neighborhood. All lower triangles of an arc (x, y) are enumerated by simultaneously scanning $N_d(x)$ and $N_d(y)$ by increasing vertex ID to determine their intersection $N_d(x) \cap N_d(y) = \{z_1 \dots z_k\}$. The lower triangles are all triples (x, y, z_i). The corresponding arc IDs are stored in the adjacency arrays. Similarly intersecting $N_u(x)$ and $N_u(y)$ yields all upper triangles, and intersecting $N_u(x)$ and $N_d(y)$ yields all intermediate triangles. This approach requires space proportional to the number of arcs in $G_\pi^\wedge$.

4.6.2 Triangle Preprocessing

Instead of merging the neighborhoods on demand to find all lower triangles, we propose to create a *triangle adjacency array* structure that maps the arc ID of (x, y) onto the pair of arc ids of (z, x) and (z, y) for every lower triangle (x, y, z). This requires space proportional to the number of triangles t in $G_\pi^\wedge$ but allows for a very fast access. Analogous structures allow efficient access all upper triangles and all intermediate triangles.

4.6.3 Hybrid Approach

For less well-behaved graphs the number of triangles t can significantly outgrow the number of arcs in $G_\pi^\wedge$. In the worst case G is the complete graph and the number of triangles t is in $\Theta(n^3)$ whereas the number of arcs is in $\Theta(n^2)$. It can therefore be prohibitive to store a list of all triangles. We therefore propose a hybrid approach. We only precompute the triangles for the arcs (u, v) where the level of u is below a certain threshold. The threshold is a tuning parameter that trades space for time.

4.6.4 Comparison with CRP

Our triangle preprocessing has similarities with micro and macro code [24]. The micro code approach is basically a huge array containing triples of arc IDs that participate in a triangle. The macro code stores for each vertex v a block that contains an array of incident arc IDs and a matrix of the arcs in the clique that replaces v after its contraction. We compare the space consumption only against a triangle adjacency array that enumerates only lower triangles as this is sufficient for an efficient customization.

The authors of [24] operate on directed graphs graphs but we operate on undirected graphs. Let t denote the number of undirected triangles and m the number of arcs in $G_\pi^\wedge$. Furthermore, denote by t' the number of directed triangles and by m' the number of arcs used in [24]. If one-way streets are rare then $m' \approx 2m$ and $t' \approx 2t$.

The micro code approach requires storing $3t' \approx 6t$ arc IDs. Our approach needs to store $2t+m+1$ arc IDs. Estimating the space requirement for the macro code approach is more complex. A lower triangle (x, y, z) is stored in the block of z. Denote by $d(z)$ the degree of z. The block of z needs to store $d^2(z) + d(z) + O(1)$ arc ids (the $O(1)$ data is needed to mark the end of a block). As z participates in $d^2(z)$ many triangles as lowest ranked vertex and every triangle has exactly one lowest ranked vertex we know that $\sum_{z \in V} d^2(z) = t$. Summing over all vertices therefore yields a space requirement of $t' + m' + O(n) = 2t + 2m + O(n)$.

Our approach always outperforms micro code. Furthermore, our approach is slightly more compact than macro code under the assumption that one-way streets are rare. If one-way streets are common then our approach needs at most twice as much data. However, the main advantage of our approach over macro code is that it allows for random access, which is crucial in the algorithms presented in the following sections.

4.7 Customization

In this section, we describe how to transform a w-initial metric m_0 into a w-maximum metric m_1. In a second step we transform m_1 into a w-minimum metric m_2. Based on m_2, it is possible to construct a weighted contraction hierarchy with perfect witness search. We also discuss how to apply multi-threading and single instruction multiple data (SIMD) instructions. Furthermore, we show how to update a metric if only the weights of a few edges change.

4.7.1 Maximum Metric

We want to turn an initial metric m_0 into a customized one m_1. For this we first copy m_0 into m_1 and then modify m_1 as following: Our algorithm iterates over all vertex levels $\ell(x)$ in $G_\pi^\wedge$ from the lowest level upward. On level i, it iterates (using multiple threads) over all arcs (x, y) with $\ell(x) = i$. Between each level all threads must be synchronized. For each such arc (x, y), the algorithm enumerates all lower triangles (x, y, z) and performs $m_1(x, y) \leftarrow \min\{m_1(x, y), m_1(z, x)+m_1(z, y)\}$, i. e., it makes sure that the lower triangle inequality holds. The resulting metric still respects w as we only set weights $m_1(x, y)$ to the distances of xy-paths. Note that this xy-path is not necessarily the shortest and thus the resulting metric is not necessarily minimum. Furthermore, by definition m_1 is customized. The metric is w-maximum, because increasing the weight of a shortcut (x, y) would violate the lower triangle equality of some lower triangle of (x, y). As all threads only write to the arc they are assigned to and only read from arcs processed in a strictly lower level we can guarantee that no read/write conflicts occurs. Hence, no locks or atomic operations are needed.

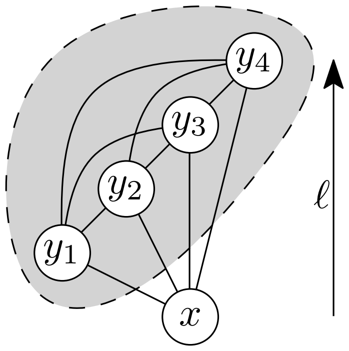


Figure 12: The vertices $y_1 \dots y_4$ denote the upper neighborhood $N_u(x)$ of x. They form a clique (the gray area) because x was contracted first. As $\ell(x) < \ell(y_j)$ for every j we know by the induction hypothesis that the arcs in this clique are weighted by shortest path distances. We therefore have an all-pair shortest path distance table among all y_j. We have to show that using this information we can compute shortest path distances for all arcs outgoing of x.

4.7.2 Minimum Metric and Perfect Witness Search

Suppose m_1 is already customized. We want to turn it into a w-minimum metric m_2. Recall that a w-minimum metric is a metric where every arc (x, y) has the weight of a shortest xy-path. As a side-product our algorithm marks all arcs that a perfect witness-search would remove. We first describe what our algorithm does and afterwards why it is correct. We first copy m_1 over into m_2 and then modify m_2. The algorithm iterates over all levels downward starting at the top-most level. It then iterates over all arcs (x, y) with $\ell(x) = i$. On most processor architectures the algorithm can iterate over the arcs of a level in parallel as long as it synchronizes between levels. However, this depends on the exact details of how write-conflicts are resolved. In some cases a different strategy is needed to enable parallelization. We postpone the details to the end of this subsection. For every arc (x, y) our algorithm enumerates all upper triangles (x, y, z) and if $m_2(x, z) + m_2(y, z) \leq m_2(x, y)$ it sets $m_2(x, y) \leftarrow m_2(x, z) + m_2(y, z)$ and marks (x, y) for removal. Analogously it iterates over all intermediate triangles (x, y, z) and if $m_2(x, z) + m_2(z, y) \leq m_2(x, y)$ it sets $m_2(x, y) \leftarrow m_2(x, z) + m_2(z, y)$ and marks (x, y) for removal. Notice that we mark the arcs for removal even if both sides are equal. The order in which the intermediate and upper triangles for one specific arc are enumerated does not matter. The resulting metric is w-minimum. The arcs marked for removal are exactly those that a perfect witness search would prune.

It remains to show that the algorithm is correct. We have to show that after the algorithm has finished processing a vertex x all of its outgoing arcs are weighted by the shortest path distance. We prove this by induction of the levels over the processed vertices. The top-most vertex is the only vertex in the top level. It does not have any outgoing arcs and thus the algorithm does not have to do anything. This forms the base case of the induction. In the inductive step we assume that all vertices with a strictly higher level have already been processed. As detailed in Figure 12 we know that vertices in $N_u(x)$ form a clique weighted by shortest paths. Pick some arbitrary outgoing arc (x, y_j). Either it already has the shortest path weight and there is nothing left to do or a shortest path through some vertex y_k in $N_u(x)$ must exist. As we know that (y_j, y_k) is a shortest path we know that $x \to y_k \to y_j$ is also a shortest path. What our algorithm does is enumerate the paths for every possible y_k. The upper triangles correspond to paths with $\ell(y_k) > \ell(y_j)$ and the intermediate triangles to paths with $\ell(y_k) < \ell(y_j)$. Our algorithm marks an arc (x, y) for removal if an xy-up-down-path exists that has the same length or is shorter and does not use (x, y). As only the existence of a shortest xy-up-down-path is needed for correctness we can not remove additional arcs. Further for all st-pairs a shortest up-down st-path exists and thus the shortest path queries are correct. The the witness search is thus perfect.

As already hinted it is less clear how to parallelize the operations within a level than it is for a plain customization. Consider the following situation: Thread A processes arc (x, y_A) at the same time as thread B processes the arc (x, y_B). Notice that (x, y_A) and (x, y_B) are both outgoing arcs of the same vertex x. Suppose that thread A updates the weight at (x, y_A) at the same moment as thread B enumerates the (x, y_B, y_A) triangle. In this situation it unclear what value thread B will see. However our algorithm is correct as long it is guaranteed that thread B will either see the old value or the new value. The new value must be smaller than the old one and therefore only an additional shortest path can have been created by thread A. If thread B sees the new value then it will see an additional shortest path. If it does not then it sees the old shortest path that has the same length and goes through some different y_j. Which shortest path thread B sees does not matter as all of them have the same length and seeing one is enough. Further seeing multiple shortest paths is not harmful. The algorithm is non-deterministic but the results is always correct. On most processor architectures (including x86) it is guaranteed that 32-bit-aligned 32-bit writes have the required property. However, if the weights have 64-bits then this property might not be given as the compiler might generate two consecutive 32-bit writes to memory. If the processor used does not have the necessary write-conflict resolution then the algorithm should iterate in parallel over all vertices in a level in parallel and each thread iterates sequentially over all outgoing arcs. This approach guarantees that all operations that might conflict are performed sequentially and does not need locks or atomic operations.

4.7.3 Directed Graphs and Single Instruction Multiple Data

A metric can be replaced by an interleaved set of k metrics by replacing every $m(x, y)$ by a vector of k elements. This allows us to customize all k metrics in one go, amortizing triangle enumeration time. A further advantage is that the customization can be accelerated using single instruction multiple data (SIMD) operations to combine the metric vectors. The processor needs to support component-wise minimum and saturated addition (i.e. $a + b = \text{int}_{\max}$ in case of overflow).

Up to now we have focused on customizing undirected graphs. If the graph is directed then we use two metrics: an upward metric m_u and a downward metric m_d. It is natural to store these two metrics interleaved. For correctness it is important to customize both metrics simultaneously because the data they convey must be interweaved. For every lower triangle (x, y, z) we set $m_u(x, y) \leftarrow \min\{m_u(x, y), m_d(z, x) + m_u(z, y)\}$ and $m_d(x, y) \leftarrow \min\{m_d(x, y), m_u(z, x) + m_d(z, y)\}$. The perfect customization can be adapted analogously. We can use single SIMD-operations to process the upward and downward metrics in parallel given that the processor is capable of permuting vector components efficiently.

A current SSE-enabled processor supports all the necessary operations for 16-bit integer components. For 32-bit integer saturated addition is missing. There are two possibilities to work around this limitation: The first is to emulate saturated-add using a combination of regular addition, comparison and blend/if-then-else instruction. The second consists of using 31-bit weights and use $2^{31} - 1$ as value for ∞ instead of $2^{32} - 1$. The algorithm only computes the saturated addition of two weights followed by taking the minimum of the result and some other weight, i. e., if computing $\min(a + b, c)$ for all weights a, b and c is unproblematic then the algorithms works correctly. We know that a and b are at most $2^{31} - 1$ and thus their sum is at most $2^{32} - 2$ which fits into a 32-bit integer. In the next step we know that c is at most $2^{31} - 1$ and thus the resulting minimum is also at most $2^{31} - 1$.

4.7.4 Partial Updates

Until now we have only considered computing metrics from scratch. However, in many scenarios this is overkill, as we know that only a few edge weights of the input graph were changed. It is unnecessary to redo all computations in this case. The ideas employed by our algorithm are somewhat similar to those presented in [39] but our situation differs as we know that we do not have to insert or remove arcs. Denote by $U = \{((x_i, y_i), n_i)\}$ the set of arcs whose weights should be updated where (x_i, y_i) is the arc ID and n_i the new weight. Note that modifying the weight of one arc can trigger new changes. However, these new changes have to be at higher levels. We therefore organize U as a priority queue ordered by the level of x_i. We iteratively remove arcs from the queue and apply the change. If new changes are triggered we insert these into the queue. The algorithm terminates once the queue is empty.

Denote by (x, y) the arc that was removed from the queue and by n its new weight and by o its old weight. We first have to check whether n can be bypassed using a lower triangle. For this reason we iterate over all lower triangles (x, y, z) a perform $n \leftarrow \min\{n, m(z, x) + m(z, y)\}$. Furthermore, if $\{x, y\}$ was an edge in the original graph G we have to make sure that n is not larger than the original weight. If after both checks $n = m(x, y)$ holds then no change is necessary and no further changes are triggered. If o and n differ we iterate over all upper triangles (x, y, z) and test whether $m(x, z) + o = m(y, z)$ holds and if so the weight of the arc (y, z) must be set to $m(x, z) + n$. We add this change to the queue. Analogously we iterate over all intermediate triangles (x, y, z) and queue up a change to (z, y) if $m(x, z) + o = m(z, y)$ holds.

How many subsequent changes a single change triggers heavily depends on the metric and can significantly vary. Slightly changing the weight of a dirt road has near to no impact whereas changing a heavily used highway segment will trigger many changes.

4.8 Distance Query

In this section we describe how to compute a shortest up-down path in $G^{\wedge}_{\pi}$ between two vertices s and t given a customized metric and how to unpack into a shortest path edge sequence in G.

4.8.1 Basic

The basic query runs two instances of Dijkstra's algorithm on $G^{\wedge}_{\pi}$ from s and from t. If G is undirected then both searches use the same metric. Otherwise if G is directed the search from s uses the upward metric m_u and the search from t the downward metric m_d. In either case in contrast to [39] they operate on the same upward search graph $G^{\wedge}_{\pi}$. In [39] different search graphs are used for the upward and downward search. Once the radius of one of the two searches is larger than the shortest path found so far we stop the search because we know that no shorter path can exist. We alternate between processing vertices in the forward search and processing vertices in the backward search.

4.8.2 Stalling

We implemented a basic version of an optimization presented in [39] called stall-on-demand. The optimization exploits that the shortest strictly upward sv-path in $G^{\wedge}_{\pi}$ can be bigger than the shortest sv-path (which can also go down). The search from s only finds upward paths and if we observe that a shorter up-down path exists then we can prune the search. Denote by x the vertex removed from the queue. We iterate over all outgoing arcs (x, y) and test whether $d(x) \geq m(x, y) + d(y)$ holds. If it holds for some arc then an up-down path $s \rightsquigarrow y \to x$ exists that is no longer than the shortest strictly upward sx path. This allows us to prune x by not relaxing its outgoing arcs.

4.8.3 Elimination Tree

We precompute for every vertex its parent's vertex ID in the elimination tree in a preprocessing step. This allows us to efficiently enumerate all vertices in SS(s) and SS(t) at query time. The vertices are enumerated increasing by rank.

We store two tentative distance arrays $d_f(v)$ and $d_b(v)$. Initially these are all set to ∞. In a first step we compute the lowest common ancestor (LCA) x of s and t in the elimination tree. We

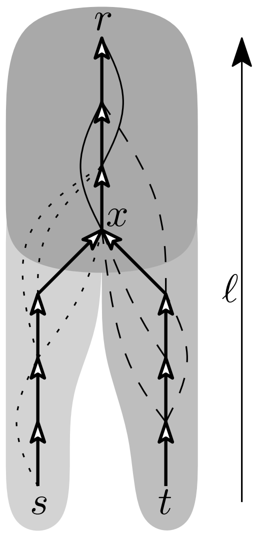


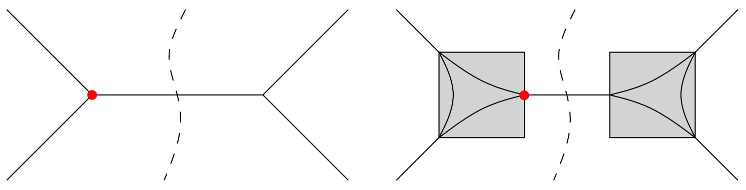
Figure 13: The union of the darkgray and lightgray areas is the search space of s. Analogously the union of the darkgray and middlegray areas is the search space of t. The darkgray area is the intersection of both search spaces. The dotted arcs start in the search space of s but not in the search space of t. Analogously the dashed arcs start in the search space of t but not in the search space of s. The solid arcs start in the intersection of the two search spaces. The vertex x is the LCA of s and t.

do this by simultaneously enumerating all ancestors of s and t by increasing rank until a common ancestor is found. In a second step we iterate over all vertices y on the tree-path from s to x and relax all forward arcs of such y. In a third step we do the same for all vertices y from t to x in the backward search. In a final fourth step we iterate over all vertices y from x to the root r and relax all forward and backward arcs. Further in the fourth step we also determine the vertex z that minimizes $d_f(z) + d_b(z)$. A shortest up-down path must exist that goes through z. Knowing z is necessary to determine the shortest path distance and to compute the sequence of arcs that compose the shortest path. In a fifth cleanup step we iterate over all vertices from s and t to the root r to reset all d_f and d_b to ∞. This fifth step avoids having to spend $O(n)$ running time to initialize all tentative distances to ∞ for each query. Consider the situation depicted in Figure 13. In the first step the algorithm determines x. In the second step it relaxes all dotted arcs and the tree arcs departing in the lightgray area. In the third step all dashed arcs and the tree arcs departing in the middlegray area and in the fourth step the solid arcs and the remaining tree arcs follow.

Contrary to the approaches based upon Dijkstra's algorithm the elimination tree query approach does not need a priority queue. This leads to significantly less work per processed vertex. Unfortunately the query must always process all vertices in the search space. Luckily, our experiments show that that for random queries with s and t sampled uniformly at random the query time ends up being lower for the elimination tree query. If s and t are close in the original graph (i.e. not sampled uniformly at random), then Dijkstra-based approaches win.

4.8.4 Path Unpacking

All shortest path queries presented only compute shortest up-down paths. This in enough to determine the distance of a shortest path in the original graph. However, if the sequence of edges that form a shortest path should be computed then the up-down path must be unpacked. The original CH of [39] unpacks an up-down path by storing for every arc (x, y) the vertex z of the


(a) undirected, no turn costs (b) undirected, with turn costs

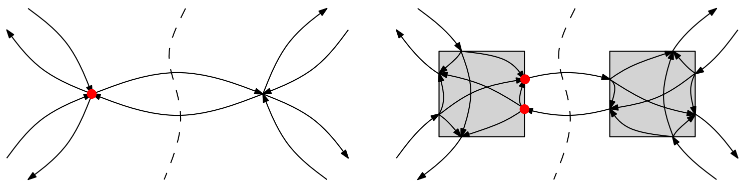

(c) directed, no turn costs (d) directed, with turn costs

Figure 14: Expanded turn models for combinations of directed and undirected, with and without turn costs. The dashed line represents the edge cut found by the bisector. The red dots represent the vertices in the derived vertex separator. The gray rectangle marks the boundaries of the turn gadgets.

lower triangle (x, y, z) that caused the weight at $m(x, y)$. This information depends on the metric and we want to avoid storing additional metric-dependent information. We therefore resort to a different strategy: Denote by $p_1 \dots p_k$ the up-down path found by the query. As long as a lower triangle (p_i, p_{i+1}, x) exists with $m(p_i, p_{i+1}) = m(x, p_i) + m(x, p_{i+1})$ insert the vertex x between p_i and p_{i+1}. For minimum metrics also intermediate and upper triangles have to be considered.

4.9 Turn Costs

In practical road route planners it is important to be able to penalize or forbid turns.

A straightforward implementation expands the graph by inserting turn clique gadgets as depicted in Figure 14. Note that many of these cliques will have the same weights and therefore a more compact representation that shares this information between cliques might be preferable in practice as described in [24, 40].

If the graph is undirected then turn costs can be added by replacing each vertex of degree d with a complete graph K_d as depicted in the Figures 14(a) and 14(b). If the graph is directed then the situation is slightly more complex as depicted in the difference between the Figures 14(c) and 14(d). A vertex of degree d is replaced by a directed $K_{d,d}$ complete bipartite graph. We refer to the vertices that only have incoming arcs inside the gadget as *exit* vertices and to the other vertices are *entry* vertices.

Recall that we determine our order by first computing an edge cut and then deriving a vertex separator from it. The first important observation is that a balanced edge cut in the unexpanded graph induces a balanced edge cut in the expanded graph. The second central observation needed is the same as the one used in the proof of Theorem 4.4: The performance is dominated by the size of the top level vertex separator. Suppose that the dashed cut represented in Figure 14 is the cut from which the top level vertex separator is derived. Denote by n the size of the vertex separator in the graph without turn costs. In the undirected case the size of this vertex separator does not increase as can be seen by comparing 14(a) to 14(b). We therefore expect the query running time performance of the CH to be mostly independent of whether turn costs are used or not. In the directed case the size of the derived top level vertex separator is doubled as can be seen by comparing 14(c) to 14(d). The top level clique in the CH is thus a K_{2n}. The number of arcs in the search spaces therefore increases by a factor of

$$\frac{|K_{2n}|}{|K_n|} = \frac{1/2(2n-1)2n}{1/2(n-1)n} \to 4$$

for n tending towards ∞. If you implement Customizable Contraction Hierarchies precisely as described so far then the search space sizes will indeed increase by this factor 4 in terms of arcs. However, one can do better. As can be seen in Figure 14(d) it is possible to assure that half of the separator vertices in the turn clique are entry vertices while the other vertices are exit vertices. Arcs between two entry vertices or two exit vertices are guaranteed to have a weight of ∞ in both directions for any metric (that respects the direction of the directed input graph). These arcs may therefore be removed from the CH. Instead of a top level K_{2n} complete clique, a complete bipartite clique $K_{n,n}$ is thus sufficient. The number of arcs is therefore only expected to increase by a factor of

$$\frac{|K_{n,n}|}{|K_n|} = \frac{n^2}{1/2(n-1)n} \to 2$$

for n tending towards ∞. To exploit this observation, we propose the following approach: First construct the CH without removing any arcs. Then, still during the metric-independent phase, "customize" it with a metric where all arcs going the wrong way through a one-way street have weight ∞ (and all others have finite weight, e. g., 0). Finally, remove from the CH all arcs that have both an upward and a downward weight of ∞ in the CH. The customization works without modifications. If the elimination-tree query should be used then it is important to construct the elimination tree before removing the arcs.

5 Robust Route Planning

Given two places in a road network, the standard goal in route planning is to compute a quickest 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 weights 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. Some of these factors can be taken into account by replacing static edge weights with time-dependent ones. The problem becomes then to find a time-dependent shortest path, usually referred to as the *quickest path* problem. Unfortunately, not everything can be modeled easily using time-dependency. A typical example is given by factors that appear often but not regularly, like traffic congestions. In the presence of such factors, one often seeks *robust* routes instead of just fast ones. In loose terms, the quality of a robust route is given by both the average and the variance of its travel time in the typical traffic situations. A slower road through the countryside that hardly sees a car per day might in this sense be considered more robust than a fast highway that is often congested.

We consider a novel approach introduced by Buhmann *et al.* [8] for finding robust solutions of general optimization problems and apply it to the quickest path problem. The only requirement of the approach 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 unknown instance (e.g., tomorrow).

Within the scope of the project eCOMPASS, we applied this approach to the quickest path problem and performed an extensive evaluation to assess its suitability for real-world applications [32]. In particular, we were interested in observing the quality of the robust routes and the time required to compute them on real-world instances. It turned out that, in terms of quality, the routes computed by Buhmann's approach are usually better than those computed by the considered competitors. However, the runtime of the algorithm used to compute those routes is in the worst-case exponential in the input size and quite impractical for real-world applications.

In this section, we investigate possibilities to make the computation of robust routes more efficient from a theoretical point of view. In particular, we propose alternative algorithms with better worst-case running times than the exponential one mentioned above. We also consider a known speed-up technique used in standard routing algorithms and show how to apply it to the situation we have in hand.

The quickest path problem under uncertainty. Let $G = (V, E)$ be a directed graph with edge weights $w : E \times T \to \mathbb{N}$ defined for a given time horizon T. A *path* P is a sequence $\langle v_1, ..., v_k \rangle$ of vertices $v_i \in V$, $1 \le i \le k$, where $(v_i, v_{i+1}) \in E$ for $i = 1, ..., k-1$, and P is called a *simple path* iff $v_i \ne v_j$ for each $i \ne j$. We overload the weight function w to express the travel time of a path $P = \langle v_1, ..., v_k \rangle$ departing at time $\tau \in T$ as

$$w(P, \tau) = \begin{cases} 0 & \text{if } k = 1 \\ w((v_1, v_2), \tau) & \text{if } k = 2 \\ \tau' + w((v_{k-1}, v_k), \tau + \tau') & \text{otherwise,} \end{cases}$$

where $\tau' = w(\langle v_1, ..., v_{k-1} \rangle, \tau)$ is the travel time of P without the last hop. Note that in the above definition we do not allow waiting at vertices even though it could be beneficial if the weight of an edge decreases dramatically over time. However, in road networks this is typically not the case (for example, in the data considered in our evaluation). The *quickest path* problem asks, for a given *source* $s \in V$, *target* $t \in V$, and time $\tau \in T$ to compute an s-t-path with minimum travel time departing at τ.

For the *quickest path problem under uncertainty*, we assume that the underlying graph (i.e., the topology of the road network) is fixed, but the edge weights (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 weight function $w_I : E \times T \to \mathbb{N}$.

The approach by Buhmann *et al.* [8] assumes 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 solution that is likely to be good for a future (yet unknown) instance I_3 from $\mathfrak{PG}$. This model fits quite naturally with the quickest path problem under uncertainty where instances represent the traffic situation on different days. For example, we could be given the travel times for last Monday and Monday two weeks ago, and our wish is to plan a robust route for next Monday.

Maximizing the similarity of instances. Since nothing is known about the underlying noise, it is a natural choice to consider only paths that are good for both I_1 and I_2. From the set of all s-t paths $\mathcal{P}$ we compute the *approximation sets* $A_\rho(I_1)$ and $A_\rho(I_2)$ where, for a given instance I with departure time $\tau \in T$ and a suitable value $\rho \ge 1$ (we explain the meaning of "suitable" later on),

$$\begin{aligned} A_\rho(I) &:= \{P \in \mathcal{P} \mid w_I(P, \tau) \le \rho \cdot OPT_I\}, \\ OPT_I &:= \min_{P \in \mathcal{P}} w_I(P, \tau). \end{aligned}$$

We then 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 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 is too large, then the ratio of the number 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 the *similarity* of I_1 and I_2, defined as

$$\frac{|A_\rho(I_1) \cap A_\rho(I_2)|}{\mathbb{E}_{B \in \mathcal{F}_{|A_\rho(I_1)|}, C \in \mathcal{F}_{|A_\rho(I_2)|}} (|B \cap C|)}, \tag{32}$$

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 picked uniformly at random. The authors also show that in many situations the same ρ maximizing (32) also maximizes

$$\frac{|A_\rho(I_1) \cap A_\rho(I_2)|}{|A_\rho(I_1)| \cdot |A_\rho(I_2)|}. \tag{33}$$

Even in those cases where (32) and (33) are not equivalent, the second formula gives a close approximation of the similarity.

During an extensive evaluation of this approach [32], we observed that the value of ρ maximizing (32) most often corresponds, at least on the data provided by TomTom for the project eCOMPASS, to the first value for which the intersection $A_\rho(I_1) \cap A_\rho(I_2)$ is not empty, the so-called *first intersection*. In light of this observation, the idea of heuristically approximating a robust path with a path belonging to the first intersection of two given instances comes naturally. As an additional benefit, there is theoretical and practical indication that the first intersection problem is computationally easier than the problem of maximizing the similarity. In particular, the former is known to be in general NP-hard while the latter is #P-hard. Furthermore, there exist algorithms that can be used to solve the former problem that perform quite well in practice, while the same cannot be said for the latter.

5.1 First intersection as a bi-criteria problem

Before explaining the algorithms for computing a path in the first intersection, it is useful to express the first intersection problem as a bi-criteria shortest path problem. In particular, given two instances I_1 and I_2 with edge weights $w_1, w_2 : E \times T \to \mathbb{N}$, we can define a new bi-criteria weight function $w : E \times T \to \mathbb{N}^2$ such that

$$w(e, \tau) = \begin{pmatrix} w_1(e, \tau) \\ w_2(e, \tau) \end{pmatrix}. \tag{34}$$

We can in a similar way extend the definition of the weight of an s-t path P as

$$w(P, \tau) = w(P', \tau) + \begin{pmatrix} w_1(e, \tau + w_1(P', \tau)) \\ w_2(e, \tau + w_2(P', \tau)) \end{pmatrix}, \tag{35}$$

where P' is the path obtained from P without the last hop e.

Remark 1. The above definition might look rather unusual to a reader already familiar with bi-criteria quickest path problems. In the field it is usually assumed that, given a path P, one of the two criteria of the weight function of P is its travel time while the other one is some sort of cost depending on the travel time of P (for example, fuel consumption). Such a weight function could be written as

$$w(P, \tau) = w(P', \tau) + \begin{pmatrix} w_1(e, \tau + w_1(P', \tau)) \\ w_2(e, \tau + w_1(P', \tau)) \end{pmatrix}. \tag{36}$$

However, for our purposes we need to consider different travel times for the same path, hence we will use the definition in (35). Under assumptions similar to those that we will make in the following and with some extra care, our results can be easily generalized for the definition in (36) as well.

We can use the definition in (35) to introduce a notion of domination between paths. We say that P *dominates* P' if

$$\begin{array}{rcl} \forall i \in \{1,2\} \quad w_i(P,\tau) & \leq & w_i(P',\tau) \\ \exists j \in \{1,2\} \quad w_j(P,\tau) & < & w_j(P',\tau). \end{array}$$

If the weight of two paths is the same for both components, we say that the two paths are *equivalent*. Given a set of paths $\mathcal{P}$, the *Pareto front* $\mathcal{F}_{\mathcal{P}}$ is the set of all paths of $\mathcal{P}$ that are not dominated by another path in $\mathcal{P}$. In the following, we use $\mathcal{F}_{st}$ to denote the Pareto front of all the s-t paths for a fixed departure time $\tau \in T$.

We now characterize a path in the first intersection of I_1 and I_2 in terms of $\mathcal{F}_{st}$. Let for this purpose OPT_1 and OPT_2 be the travel times of quickest paths for respectively I_1 and I_2. The following theorem shows that a path on the Pareto front of the first intersection also belongs to $\mathcal{F}_{st}$.

Theorem 5.1. Let $\rho^* \in [1,+\infty)$ be the smallest value for which $A_\rho(I_1) \cap A_\rho(I_2)$ is not empty and $\mathcal{F}_\cap$ be the Pareto front of such an intersection. If $P \in \mathcal{F}_\cap$ then $P \in \mathcal{F}_{st}$.

Proof. Assume towards contradiction the claim not to be true. There exists then a path $P \in \mathcal{F}_\cap$ such that $P \notin \mathcal{F}_{st}$, that is, there is a path $P' \notin \mathcal{F}_\cap$ dominating P. From the definition, this means that the travel time of P' is at most the travel time of P both in I_1 and I_2, and strictly smaller in at least one of the two instances. We assume without loss of generality that this happens for I_1. Therefore,

$$\begin{array}{rcl} \rho'_1 = \dfrac{w_1(P',\tau)}{OPT_1} & < & \dfrac{w_1(P,\tau)}{OPT_1} = \rho_1 \\ \rho'_2 = \dfrac{w_2(P',\tau)}{OPT_2} & \leq & \dfrac{w_2(P,\tau)}{OPT_2} = \rho_2. \end{array}$$

Note that it holds $\rho^* = \max\{\rho_1, \rho_2\}$.

Clearly, it cannot be that $\max\{\rho'_1, \rho'_2\} < \rho^*$, because we would have found a path that belongs to $A_\rho(I_1) \cap A_\rho(I_2)$ for some ρ smaller than ρ^*. This contradicts the assumption that ρ^* is the smallest value for which the intersection is not empty. Furthermore, also $\max\{\rho'_1, \rho'_2\} = \rho^*$ results in a contradiction, because it implies that P' belongs to the first intersection and, since P' dominates P, then $P \notin \mathcal{F}_\cap$. Therefore, it must hold $\max\{\rho'_1, \rho'_2\} > \rho^*$. We can now distinguish two cases: $\max\{\rho'_1, \rho'_2\} = \rho'_1$ and $\max\{\rho'_1, \rho'_2\} = \rho'_2$. In both cases we get a contradiction, because

$$\rho_1 \leq \rho^* < \max\{\rho'_1, \rho'_2\} = \rho'_1 < \rho_1,$$

while

$$\rho_2 \leq \rho^* < \max\{\rho'_1, \rho'_2\} = \rho'_2 \leq \rho_2.$$

Therefore, such a path P' does not exist. □

Note that the opposite result is not true in general. There may exist paths that belong to $\mathcal{F}_{st}$ but not to $\mathcal{F}_\cap$. This is typically the case for a quickest path in either of the two instances.

If we wish to compute a path in the first intersection, we can do so by enumerating all paths in $\mathcal{F}_{st}$ and pick one among them for which the maximum ratio of its weight over OPT_1 and OPT_2 is minimum. The problem of computing the Pareto front of a graph with bi-criteria edge weights is well studied at least for the static case (without time-dependency). On the other hand, very little is known about its time-dependent variant.

It should be observed that we are shifting the focus from computing a path in the first intersection to the problem of computing the Pareto front of a graph with time-dependent bi-criteria edge weights. A solution to the former problem is then extracted from the latter as a "by-product". It is an interesting open question to compute the first intersection directly. A different perspective on the problem might lead to different and maybe more efficient solutions.

Algorithm 1 Time-dependent bi-criteria Martins' algorithm

```
INPUT: $G = (V, E)$,   $w : E \times T \to \mathbb{N}^2$,   $s, t \in V$, $\tau \in T$
{Initialization}
for $v \in V$ do $\pi_v := \emptyset$
$Q$.insert($s, \binom{\tau}{\tau}$)
{Compute front}
while $Q \neq \emptyset$ do
 $(u, \boldsymbol{\omega}) := Q$.extract_min()
  for $e = (u, v) \in E$ do
   $newl := (v, \boldsymbol{\omega} + \binom{w_1(e,\tau+\boldsymbol{\omega}_1)}{w_2(e,\tau+\boldsymbol{\omega}_2)})$
    if $\neg\pi_v$.dominates($newl$) and $\neg\pi_t$.dominates($newl$) then
     $Q$.insert($newl$)
```

5.2 Computing the Pareto front

In the static case, the problem of computing the Pareto front is well studied. Hansen [45] introduced several variants of the bi-criteria shortest path problem. He also showed that in general the number of paths in the Pareto front can be exponential in the input size. Furthermore, he presented a generalization of Dijkstra's algorithm that computes a *minimal complete set* of the Pareto front of a graph with non-negative bi-criteria edge weights in pseudo-polynomial time. A *complete set* of paths $\mathcal{L}$ is a set such that any path $P \notin \mathcal{L}$ is either dominated by or equivalent to a path in $\mathcal{L}$. A complete set is *minimal* if no two paths in it are equivalent. For the sake of legibility, in the following we will assume that no two paths are equivalent; it is trivial to extend our results to allow for equivalent paths.

Martins [53] extended the algorithm from Hansen for static edge weights with more than two criteria. The algorithm keeps a set of temporary labels Q and a set of permanent labels π_u for every vertex $u \in V$. Each label $(u, \boldsymbol{\omega})$ represents a path from s to u with weight $\boldsymbol{\omega} \in \mathbb{N}^k$ (for k criteria); we write $P \in \pi_u$ to indicate that the label representing P is in π_u. At the beginning every π_u is empty, and a label $(s, \mathbf{0})$ is created and put into Q. In each iteration, the algorithm extracts from Q the label $(u, \boldsymbol{\omega})$ with smallest weight in lexicographical order and puts it into π_u. Then, new labels $(v, \boldsymbol{\nu})$ are generated for each vertex v that can be reached from u, with $\boldsymbol{\nu} = \boldsymbol{\omega} + w(u, v)$. If no label in π_v or π_t dominate the new label it is inserted into Q. In this case, all labels in Q that correspond to an s-v path and are dominated by $(v, \boldsymbol{\nu})$ are removed. The algorithm ends when Q is empty; when this happens, π_t contains labels representing all paths in the Pareto front.

It is not hard to extend Martins' algorithm to time-dependent edge weights. Gräbener *et al.* [43] provide an experimental evaluation of the generalized algorithm on some publicly accessible networks. The pseudo-code for the bi-criteria case is shown by Algorithm 1. We must be careful when using this algorithm; its correctness depends on the following property.

Definition 1. Given a graph $G = (V, E)$ with edge weights $w : E \times T \to \mathbb{N}$ (single criterion), we say that w satisfies the *FIFO property* if $\tau + w(e, \tau) \leq \tau' + w(e, \tau')$ for every $\tau, \tau' \in T$ such that $\tau \leq \tau'$.

Theorem 5.2. Let $G = (V, E)$ be a graph with edge weights $w : E \times T \to \mathbb{N}^2$ as in (34). If $w_i : E \times T \to \mathbb{N}$ satisfies the FIFO property for every $i \in \{1, 2\}$, Algorithm 1 computes the Pareto front $\mathcal{F}_{st}$.

Proof. Assume towards contradiction that the front π_t computed by Algorithm 1 is not correct. This means that either there is a path in $\mathcal{F}_{st}$ that is not in π_t, or there is a path in π_t that is not in $\mathcal{F}_{st}$, or both. We consider only the first case, since the remaining two follow trivially from the fact that if π_t contains at least the paths in $\mathcal{F}_{st}$ then all other paths are dominated by those.

Let π_t be the front computed by the algorithm and suppose towards contradiction that there exists $P \in \mathcal{F}_{st}$ such that $P \notin \pi_t$. Consider the subpath P_{sv} of P from s to the first vertex v such that $P_{sv} \notin \pi_v$, and the subpath P_{vt} of P from v to t. We can express the weight of P as

$$w(P, \tau) = w(P_{sv}, \tau) + \begin{pmatrix} w_1(P_{vt}, \tau + w_1(P_{sv}, \tau)) \\ w_2(P_{vt}, \tau + w_2(P_{sv}, \tau)) \end{pmatrix}.$$

Since $P_{sv} \notin \pi_v$, there exists another path P'_{sv} dominating it, and we can obtain an s-t path P' (not necessarily simple) by concatenating P'_{sv} and P_{vt}. The weight of P' can be written as

$$w(P', \tau) = w(P'_{sv}, \tau) + \begin{pmatrix} w_1(P_{vt}, \tau + w_1(P'_{sv}, \tau)) \\ w_2(P_{vt}, \tau + w_2(P'_{sv}, \tau)) \end{pmatrix}.$$

Since P'_{sv} dominates P_{sv}, we know that, for every $i \in \{1, 2\}$, it holds that

$$w_i(P'_{sv}, \tau) \leq w_i(P_{sv}, \tau).$$

Since both w_1 and w_2 satisfy the FIFO property, we get that $w(P', \tau)$ dominates $w(P, \tau)$. This contradicts the assumption that $P \in \mathcal{F}$. □

The analysis of the runtime of Algorithm 1 follows trivially from the one by Hansen [45]. We use n and m to denote the number of vertices and edges of G, respectively.

Corollary 1. Algorithm 1 runs in time $\mathcal{O}(nmW \cdot \log(nW))$, where

$$W = \min_{i \in \{1,2\}} \{ \max_{e \in E, \tau \in T} w_i(e, \tau) \}.$$

If the FIFO property is not satisfied for all the edge weights, the correctness of Algorithm 1 is not guaranteed. Hamacher *et al.* [44] consider this more general setting and provide algorithms computing the Pareto front for a given s-t pair as well as for the all-to-all variant. However, their algorithm is slower than Algorithm 1. Since in our data the FIFO property is indeed satisfied for all edge weight functions, we restrict our attention to Algorithm 1.

5.3 Bidirectional Martins' Algorithm

Methods for speeding-up the computation of shortest paths have been researched intensively and several techniques are known that allow us to answer queries within milliseconds on continental-sized road networks [22, 38]. Some of these techniques have also been applied to quickest and bi-criteria path problems, but never on the combination of the two. In the following, we briefly review one of the most fundamental techniques, namely bidirectional search [42], and show how it can be applied to Algorithm 1.

Before going deeper into the heuristics we recall some basic terminology commonly adopted when discussing Dijkstra's algorithm. At any step of the algorithm, any vertex is in one of the following states: UNREACHED, SETTLED, and DISCOVERED. A vertex is UNREACHED if its distance from s is not known, it is SETTLED if its distance from s is known exactly, and it is DISCOVERED if only an upper bound of the distance is known. At the beginning all vertices are UNREACHED, while at the end they can only be SETTLED or UNREACHED. If a vertex is UNREACHED after the end of the computation, it means that it cannot be reached from s.

For static graphs, the idea behind bidirectional search is fairly simple: we let Dijkstra's algorithm run both from s and from t alternatively. The execution from t, usually called *backward search*, uses the edges of the reverse graph, i.e., the graph containing the edges of the original one in reverse direction. As soon as a vertex v becomes SETTLED both in the forward and in the backward search, we can be sure that a shortest s-t path has been found (such a path might however not pass through v). Any alternation strategy works correctly; a typical choice is to balance the number of iterations of the two searches. For road networks, this usually results in a considerable speed-up with respect to unidirectional search and, at the same time, it gives a guarantee that the overall number of iterations in the worst-case cannot be more than twice the number of iterations of the unidirectional search.

When it comes to graphs with static bi-criteria edge weights, the situation is not much more complicated than the single criterion case: just replace Dijkstra's algorithm with Martins'. The only issue is that, in Martins' algorithm, vertices are not SETTLED anymore, because we do not know whether the whole Pareto front of a vertex has been computed until the queue containing the temporary labels becomes empty (note however that the definitions of UNREACHED and DISCOVERED do carry over). Demeyer *et al.* [25] propose to overcome this issue by terminating both searches when the sum of the *pointwise minima* of the forward and backward queues is dominated by the Pareto front computed so far. The pointwise minimum of a queue Q, denoted as $Q.\mathrm{p\_min}()$, is the vector where each component is equal to the minimum among all labels in Q for the corresponding criterion.

While bidirectional search can be implemented quite easily for static edge weights, in the dynamic case the situation is not as simple. A major difficulty when edges have time-dependent weights is that we do not know in advance the arrival time at t. This means that we cannot implement the backward search straightforwardly by running the time-dependent Dijkstra's algorithm on the reverse graph. Nannicini *et al.* [55] proposed to overcome this issue by using static edge weights on the reverse graph. In particular, if (u, v) is an edge of the forward graph $G = (V, E)$, the static weight of (v, u) in the backward graph $\overleftarrow{G} = (V, \overleftarrow{E})$ is defined as

$$\overleftarrow{w}((v,u)) = \min_{\tau \in T} \{w((u,v), \tau)\}. \tag{37}$$

In this way, the weight of a path in the backward graph is a lower bound of its real weight in the forward direction. The authors also introduced a three phases algorithm to compute a quickest path in a bidirectional fashion. The phases of the algorithm in detail are as follows:

1. A bidirectional search is executed, where the forward search is run using the time-dependent edge weights while the backward search uses the weights in (37). This phase terminates as soon as a vertex becomes DISCOVERED in both directions.

2. Let v be the vertex that is DISCOVERED in both directions and μ be the *time-dependent* weight of the corresponding path going from s to t through v. In the second phase, both searches continue until the distances of all DISCOVERED vertices in the backward search are at least μ.

3. Only the forward search continues, with the constraint that only vertices that were SETTLED by the backward search are considered. This phase terminates when t is SETTLED by the forward search.

The intuition behind the algorithm is that the backward search, since it cannot be used to compute the path itself, serves to identify a set of promising vertices along which a quickest path may go through.

For edge weights that are time-dependent and bi-criteria both the aforementioned issues arise. Fortunately, also do the solutions. We could therefore design a bidirectional algorithm by straightforwardly combine the above methods. This would result in a three phases search that uses Martins' algorithm both from s and from t, where in the backward direction edge weights are defined as in (37) for both criteria. The termination condition of the backward search corresponds to the stopping condition proposed by Demeyer *et al.* As it turns out, however, we can do much better than that.

A critical observation to improve the straightforward algorithm is to note that in the backward direction we are not interested in computing Pareto fronts. Our only interest is to identify vertices that might be on a Pareto optimal path. In other words, whether or not the Pareto front of a given vertex contains at least one "promising" label. However, since a label that is good for one criterion might not be good for the other one, we cannot know in advance which labels are promising. Our solution is to consider only the pointwise minima of the Pareto fronts of vertices. If not even the pointwise minimum of the Pareto front of a vertex is interesting, then none of its labels are. We can improve the bidirectional algorithm in light of this observation.

If the purpose of the backward search is only to compute pointwise minima, then Martins' algorithm is more than what is necessary. Indeed, if we implement the backward search as two independent searches on the reverse graph for each criterion, then Dijkstra's algorithm suffices. We must also modify the three phases accordingly.

In phase 2, suppose we have found (in some way during phase 1) a number of (non-necessarily Pareto optimal) s-t paths, and let M denote the Pareto front of these paths. Now the forward search is continuing, as well as the two backward searches. Let $\overrightarrow{Q}$ be the forward queue and $\overleftarrow{Q_1}$ and $\overleftarrow{Q_2}$ be the backward queues. Suppose further that at some point during the computation the weight of a path in M dominates or is equivalent to

$$\overrightarrow{Q}.\text{p\_min}() + \begin{pmatrix} \overleftarrow{Q_1}.\min() \\ \overleftarrow{Q_2}.\min() \end{pmatrix}. \tag{38}$$

At this point, if a vertex was not SETTLED by both backward searches we can be sure that it does not have to be considered. The intuitive reason for this is straightforward: if a vertex v has not been SETTLED by both searches, then the weight of any s-t path passing through v will be dominated by (38) and also by a path in M (we prove the correctness of this argument more formally in the following). We can therefore terminate phase 2 as soon as a path in M dominates or is equivalent to (38).

Consider now the situation where, in phase 3, the forward search created a label $(v, \boldsymbol{\omega})$ to be inserted in $\overrightarrow{Q}$. Let $\boldsymbol{d}_v$ be the vector of distances computed by the backward searches for v. By the same argument above, if a path in M dominates

$$\boldsymbol{\omega} + \boldsymbol{d}_v, \tag{39}$$

then no path having as prefix the s-v path corresponding to $\boldsymbol{\omega}$ can be Pareto optimal. Therefore, in phase 3 we can ignore those labels for which (39) is dominated by a path in M.

We now consider phase 1, whose purpose is to compute a suitable tentative front M. In principle, a tentative front is good if the domination of (38) happens as early as possible, because less labels carry over to phase 3. It is easy to come up with different strategies to compute such a front. The one we adopted in our experiments (explained in the following) has the advantage of being simple but still working reasonably well for our purposes. Note however that several alternative strategies are possible and it is an interesting open question to identify one that works best.

In detail, the phases of the bidirectional algorithm are as follows:

1. A bidirectional search is executed. The forward search uses Algorithm 1 on the time-dependent bi-criteria edge weights. The backward search uses two independent runs of Dijkstra's algorithm for each criterion. We alternate between an iteration of the forward search and one iteration for each criterion for the backward search. This phase terminates as soon as a vertex v that is DISCOVERED by the forward search and such that $\pi_v \neq \emptyset$ is SETTLED by both backward searches. Figures 15(a) and 15(b) illustrate this phase.

2. Let v be the vertex that terminated phase 1. For each s-v path in π_v, we consider both v-t paths that the backward searches have found and insert the corresponding s-t paths in M. In the second phase, illustrated by 15(c), all searches continue until a path in M dominates or is equivalent to (38).

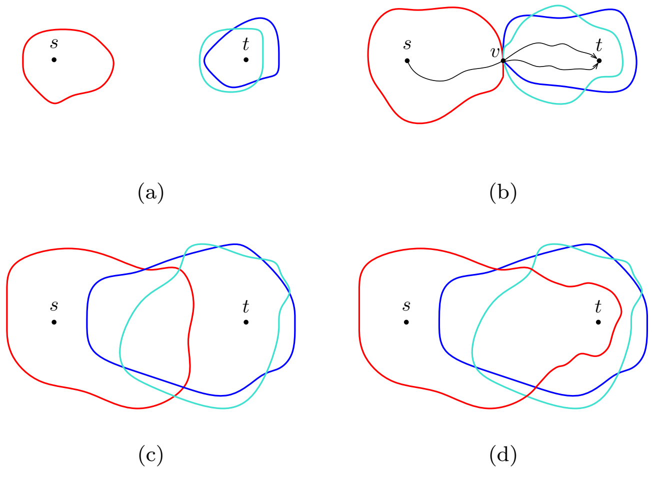


Figure 15: A schematic representation of the phases of Algorithm 2

3. Only the forward search continues, with the constraint that labels for which (39) is dominated by a path in M are ignored. This phase is shown by Figure 15(d) and terminates when the forward queue becomes empty.

Algorithm 2 shows the pseudo-code of the above algorithm.

Theorem 5.3. Algorithm 2 computes the Pareto front $\mathcal{F}_{st}$.

Proof. From Theorem 5.2 it follows that the forward search would compute $\mathcal{F}_{st}$ correctly, unless the restriction applied during phase 3 do not prevent a path in $\mathcal{F}_{st}$ to be found.

Towards contradiction, assume that at the end of an execution there exists a path $P \in \mathcal{F}_{st}$ such that $P \notin \pi_t$. Consider the subpath P_{sv} of P from s to the first vertex v such that $P_{sv} \notin \pi_v$, and the subpath P_{vt} of P from v to t. Since $P_{sv} \notin \pi_v$, there exists a path P' in M that dominates

$$w(P_{sv}, \tau) + \overleftarrow{w}(P_{vt}).$$

Note that either $P' \in \pi_t$, or there exists a path in π_t dominating it. Since $P \in \mathcal{F}_{st}$ then P' does not dominate P, that is, there exists $i \in \{1, 2\}$ such that the ith component of the time-dependent weight of P is strictly smaller than that of P'. Without loss of generality, we assume that this happens for $i = 1$. We now get a contradiction, because

$$w_1(P, \tau) < w_1(P', \tau) \leq w_1(P_{sv}, \tau) + \overleftarrow{w_1}(P_{vt}) \leq w_1(P, \tau).$$

□

In the algorithm of Nannicini *et al.* [55], the switch to phase 3 happens if the upper bound μ on the weight of a quickest path computed in phase 1 is at most the current minimum of the backward queue. The authors also proved that replacing this condition with one that, for a fixed parameter K, checks whether μ is at most K times the minimum of the backward queue results in an algorithm that computes a K-approximate quickest path (i.e., a path with weight at most K times the weight of a quickest path). This algorithm is faster than the original version.

We can show that Algorithm 2 satisfies a property similar to the one above. However, we first have to introduce a notion of an approximation of Pareto fronts.

Algorithm 2 Bi-directional time-dependent Martins' algorithm

```
{Initialization}
for $v \in V$ do
 $\pi_v := \emptyset$, $v.d_1 := \infty$, $v.d_2 := \infty$
$\overrightarrow{Q}$.insert($s, \boldsymbol{\tau}$), $\overleftarrow{Q_1}$.insert($t, 0$), $\overleftarrow{Q_2}$.insert($t, 0$)
$M := \emptyset$, $\phi := 1$, $dir := BWD$
{Compute front}
while $\overrightarrow{Q} \neq \emptyset$ do
 $dir := \phi = 3 ? FWD : \neg dir$
 if $dir = FWD$ then
  $(u, \boldsymbol{\omega})$ := $Q$.extract_min()
 else
  $u_1 := \overleftarrow{Q_1}$.extract_min()
  $u_2 := \overleftarrow{Q_2}$.extract_min()
 {terminate phase 1}
 if $\phi = 1$ then
  if $\exists v \in V$ s.t. $\pi_v \neq \emptyset$ and SETTLED_BWD($v$) then
   $M$.insert(< $s$–$t$ paths through $v$ >)
   $\phi := 2$
 {terminate phase 2}
 if $\phi = 2$ and $M$.dominates($\overrightarrow{Q}$.p_min() + $\binom{\overleftarrow{Q_1}.\min()}{\overleftarrow{Q_2}.\min()}$) then $\phi := 3$
 {relax edges}
 if $dir = FWD$ then
  for $e = (u, v) \in \overrightarrow{E}$ do
   $\boldsymbol{\nu} := \boldsymbol{\omega} + \binom{w_1(e, \tau + \boldsymbol{\omega}_1)}{w_2(e, \tau + \boldsymbol{\omega}_2)}$
   if $\phi = 3$ and $M$.dominates($\boldsymbol{\nu} + v.\boldsymbol{d}$) then continue
   if $\neg\pi_v$.dominates($\boldsymbol{\nu}$) and $\neg\pi_t$.dominates($\boldsymbol{\nu}$) then
    $\overrightarrow{Q}$.insert($v, \boldsymbol{\nu}$)
 else
  for $i \in \{1, 2\}$ do
   for $e = (u_i, v) \in \overleftarrow{E}$ do
    if $u_i.d_i + \overleftarrow{w_i}(e) < v.d_i$ then
     $\overleftarrow{Q_i}$.insert($v, u_i.d_i + \overleftarrow{w_i}(e)$)
```

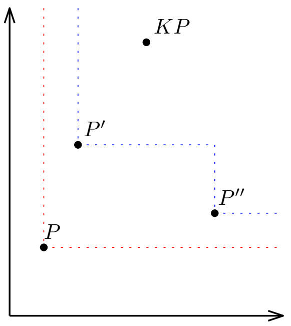


Figure 16: The blue and red lines are respectively the Pareto front and the approximate front computed by the algorithm. Path P' approximates P but P'' does not.

Definition 2. Given $\tau \in T$, a fixed parameter $K \geq 1$ and two paths P and P', we say that P K-*approximates* P' if $w(P, \tau)$ dominates or is equivalent to $K \cdot w(P', \tau)$. Furthermore, given two sets of paths $\mathcal{P}$ and $\mathcal{P}'$, we say that $\mathcal{P}$ is a K-*approximation* of $\mathcal{P}'$ if every path in $\mathcal{P}'$ is K-approximated by a path in $\mathcal{P}$.

Theorem 5.4. Let K be a fixed parameter and β be the vector corresponding to (38). If we replace the condition to terminate phase 2 with

$$M.\text{dominates}(K \cdot \beta),$$

then Algorithm 2 computes a K-approximation of $\mathcal{F}_{st}$.

Proof. Assume towards contradiction that there exists a path $P \in \mathcal{F}_{st}$ that is not K-approximated by a path in π_t. That is, for every $P' \in \pi_t$ there is $i \in \{1, 2\}$ such that

$$K \cdot w_i(P, \tau) < w_i(P', \tau). \tag{40}$$

Let P_{sv} be the subpath of P from s to the first vertex v such that $P_{sv} \notin \pi_v$, and P_{vt} be the subpath of P from v to t. Since $P_{sv} \notin \pi_v$, there is a path in M dominating

$$w(P_{sv}, \tau) + \overleftarrow{w}(P_{vt}).$$

Consider as P' either this path, if it belongs to π_t, or a path in π_t dominating it otherwise. Suppose without loss of generality that (40) holds for $i = 1$. We now get a contradiction, because

$$K \cdot w_1(P, \tau) < w_1(P', \tau) \leq w_1(P_{sv}, \tau) + \overleftarrow{w_1}(P_{vt}) \leq w_1(P, \tau).$$

□

Note that the converse of Theorem 5.4 in general does not hold. That is, there might be paths in π_t that do not approximate a Pareto optimal path. Figure 16 shows an example of such a situation.

5.4 Further improvements

As explained in Remark 1, the edge weights used by the above algorithms are quite peculiar for the application considered, that is, robust routes. In particular, each criterion in an edge weight corresponds to a travel time for a different day. If the considered days are somehow related like, for example, two working days as opposed to a working day and a Sunday, we can expect this correlation to somehow appear in the edge weights values as well. In the proposed algorithms we do not make explicit use of this property. One might ask if their efficiency improves if these features are considered more explicitly.

For example, consider the backward search of Algorithm 2 on the static edge weights of the reverse graph. Assume that, for each backward edge $e \in \overleftarrow{E}$, it holds that

$$\overleftarrow{w_1}(e) = \overleftarrow{w_2}(e),$$

where $\overleftarrow{w_i}(e)$ is the weight of e in instance I_i. This assumption might be not too unreasonable in the situation discussed above. In this case, the two backward searches used by Algorithm 2 appear quite redundant since they will settle the same vertices at the same time and with the same values. We might then save a lot of computation time by replacing the two backward searches with a single one that settles the same value two times.

Even if the quite strong assumption above does not hold, we might still get some improvement by using a single backward search that consider edge weights of the kind

$$\overleftarrow{w}(e) = \min_{i \in \{1,2\}} \{\overleftarrow{w_i}(e)\}. \tag{41}$$

The correctness of this algorithm and its K-approximation variants follow trivially from the previous proofs and hold under the same assumptions as for Algorithm 2.

The benefits of this new algorithm over the original one, however, are not trivial to estimate. On the one hand, if

$$\max_{e \in \overleftarrow{E}} \{|\overleftarrow{w_1}(e) - \overleftarrow{w_2}(e)|\} \tag{42}$$

is small, then the modified backward search will settle almost the same vertices as before, with almost the same values, at the price of one execution of Dijkstra's algorithm instead of two. Therefore, phases 1 and 2 will terminate earlier. On the other hand, the lower bounds on the distances computed in the reverse graph are less accurate. As a result, the set of labels that survive the pruning of the forward search in phase 3 might be larger than the original algorithm. Note however that less vertices might be SETTLED by the modified backward search; the penalization of phase 3 might be somehow mitigated by this fact.

In some sense, we can see the modified algorithm as one that tries to speed-up phases 1 and 2 by penalizing phase 3. The benefit of this penalization is inversely proportional to (42).

6 Fleets-of-Vehicles 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 [13] 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 [69] 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 [11] and the insertion heuristic [47]. Other methods like the two phase method of Fisher and Jaikumar [35] are also widely used. Among the improvement heuristics, the methods of [48] and [49] 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., [9],[28],[65]. 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 [9] 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 [65] 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.

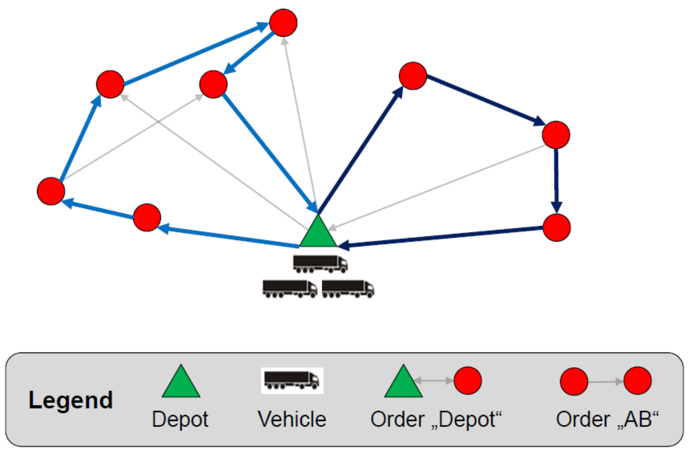


Figure 17: Instance of a VRP Problem

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 $l_i + t_{ij} < l_j$ as shown in Figure 18. 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.

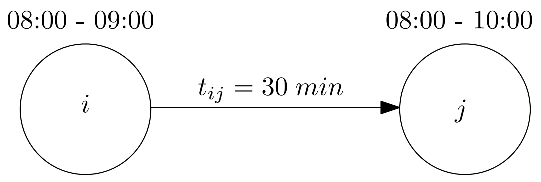


Figure 18: Two customers with compatible time windows. If a vehicle leaves customer i it can then serve customer j.

6.3.2 Phase II - Geographic Partition

The second phase makes a geographical partition of the area. We use three different techniques to achieve geographic partition: Quad Trees [34], KaHIP [61] and Natural Cuts [20]. Each technique tries to partition a given geographical area into smaller parts. The way of each technique is different and we examine all three of them in order to see which technique suites better for the VRPTW.

Quad Trees. The first technique used is the Quad Trees [34]. 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 19,20. 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 19,20 where cell 0 is divided into cells 00 through 08.

KaHIP (Karlsruhe HIgh Quality Partitioning). The second technique used is the KaHIP [61] partitioning software. KaHIP is a family of graph partitioning programs. It includes a multilevel

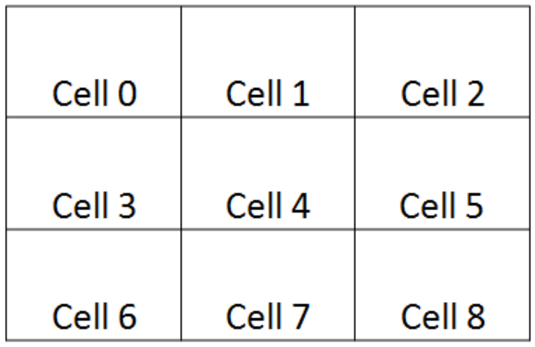


Figure 19: First level of Geographic Partition. An area is divided into 9 cells.

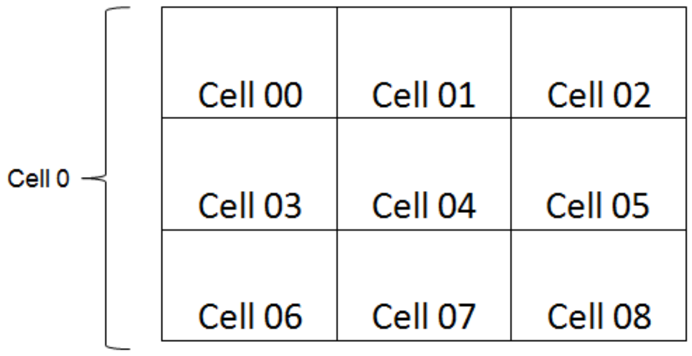


Figure 20: Second level of Geographic Partition. Each cell from the first level is further divided.

partitioning algorithm called KaFFPa along with its variants Strong, Eco and Fast depending on what type of partition one is interested in. KaHIP uses max-flow/min-cut algorithms to create the desired partition. KaHIP needs as input a graph G to be partitioned (in a special form called DIMACS 10) and the number of blocks into which the graph will be partitioned. The user can also provide more arguments such as partition type (strong, eco, fast) time limit and balance. KaHIP provides more algorithms that perform partition such as KaFFPaE (KaFFPaEvolutionary) which is a parallel evolutionary algorithm that uses KaFFPa to provide combine and mutation operations, as well as KaBaPE which extends the evolutionary algorithm.

Natural Cuts. The third technique used was natural cuts [20]. This method consists of two phases. Firstly, it identifies and contracts dense regions of the graph by using a series of minimum-cut computations. Secondly, it uses a combination of greedy and local search heuristics to create the final partition of the graph. The technique performs well on road networks, which have plenty natural cuts such as bridges, mountain passes, ferries, rivers etc.

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 21,22. In Figure 21, clusters C_1, C_2 are merged because they are both connected and geographically close, whereas in Figure 22 cluster C_3 is split into two sub-clusters because it contains customers that lay in different geographical areas.

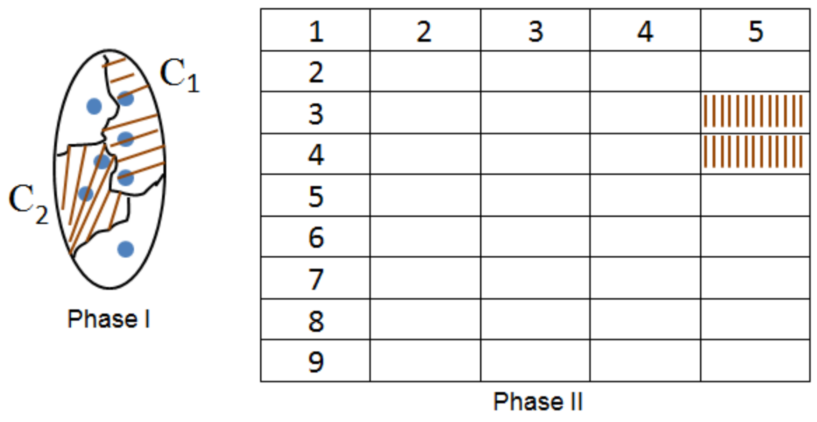


Figure 21: Phase III: Cluster Refinement - Merge Operation. Examine phases I and II and perform a merge operation.

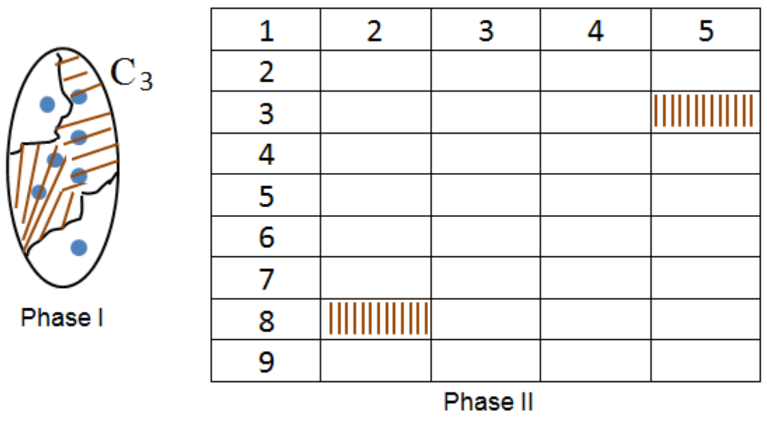


Figure 22: Phase III: Cluster Refinement - Split Operation. Examine phases I and II and perform a split operation.

7 Conclusions

In this document we presented the algorithmic solutions developed by the eCOMPASS project partners in the last 20 months of the project. We illustrated how the algorithms developed in [31] were extended in order to yield better solutions in a more efficient way. In the experimental counterpart of this deliverable, namely, D2.4, the algorithms presented here have been assessed both in terms of efficiency and environmental impact of the computed routes.

7.1 Traffic Prediction

With respect to the results and analysis given in the previous sections, the Segmented Lag-STARIMA is proven to be a satisfactory traffic forecasting approach when it comes to real datasets, such as the one of Berlin. Our analysis indicated that using different models for different time series segments is reasonable. The intuition of the segmentation step is generally well supported as to the different nature of each segment. When taking into account the statistics of each segment, including its trend, one can train a better model (in the sense that it should fit better). Furthermore the thorough pre-processing procedure that was described overcomes many of the problems that occur due to the sparse nature of GPS datasets, thus one can apply time series models that would otherwise be ineffective. Our goal is to show that fast changing urban traffic, can benefit the use of such techniques in order to build a robust system that can cope with the high standards of modern Intelligent Transportation Systems. The problem of traffic forecasting under these real-world scenario circumstances has many aspects thus the application of the outlier detection filter, the moving average filter, the missing values imputation based on k-means algorithm, and the time series segmentation is an important step towards a generic methodology for traffic prediction.

On the other hand, a new non-parametric short-term forecasting technique that we presented in Section 2.5 based on speed dynamics seems to overcome the large computational demands of the parametric method. The proposed non-parametric method suggests lowering the dimension of the available dataset for enabling more efficient deployment of clustering techniques in terms of computational resource efficiency and forecasting accuracy. This method ends up using five features extracted from the available time series. This comprises a significant improvement compared to the dimension of the original space, which is 288 (for one day data). This reduction of the feature space dimension is based on the extraction of the dynamic characteristics of traffic measured by the average values of the first and second derivates. The entropy-based interestingness measure (IS score) is used for selecting the features bearing the largest information share. As a result, three different feature sets are formed and tested. By applying the k-means clustering algorithm on the new feature space, a set of road profiles is generated and then for each profile a set of probabilistic distributions (histograms) of the harmonic speed average is constructed for each target interval. In order to perform forecasting we classify every new road instance (after it is transformed into our feature space in use) into one of the existing road profiles, choosing the histogram for the target interval and generating a random number based on the probability distribution function of this profile.

7.2 Time-Dependent Shortest Paths

We have presented the first time-dependent distance oracle for sparse networks, compliant with Assumptions 1 and 2, that achieves subquadratic preprocessing space and time, sublinear query time, and stretch factor arbitrarily close to 1. Our approach is based on a new algorithm, built upon the bisection method, that computes one-to-all approximate distance summaries from a set of selected landmarks to all other vertices of the network as well as on a new recursive query algorithm. Our assumptions, justified by an experimental analysis of real-world and benchmark data, allow us to achieve a smooth transition, from the undirected (symmetric) and static world to the directed (asymmetric) and time-dependent world, through two parameters that quantify the degree of asymmetry (ζ) and its evolution over time (expressing the steepness of the shortest travel-time functions via $\Lambda_{\min}$ and $\Lambda_{\max}$).

It would be quite interesting to come up with a new method for computing approximate distance summaries, that avoids the dependence of the preprocessing complexities on the number K^* of concavity-spoiling breakpoints.

Finally, almost all distance oracles with provable approximation guarantees in the literature, even for the static case, target at sublinearity in query times with respect to the network size. A very important aspect would be to propose query algorithms that are indeed sublinear not only in worst-case, but also for almost all possible queries in the network.

7.3 Fast, Dynamic and Highly User-Configurable Route Planning

We have extended Contraction Hierarchies to a three phase customization approach and demonstrated that the approach is practicable and efficient on real world road graphs. It promises to be fast enough to enable user-specific trade-offs between travel time, user preferences, and eco-friendliness while maintaining fast responsive query times and a light-weight preprocessing that enables consideration of current traffic conditions.

Better nested dissection orders directly increase the performance of the introduced Customizable Contraction Hierarchies. Research aiming at providing better vertex orders or proving that the existing orders are close to optimal seems useful.

Revisiting all of the existing Contraction Hierarchy extensions to see which can profit form a metric-independent vertex order or can be made customizable seems worthwhile. An interesting candidate are Time-Dependent Contraction Hierarchies [6] where computing a good metric-dependent order has proven relatively expensive. Here, we could build on some of the techniques developed in Section 3 for the computation of Time-Dependent Shortest Paths.

7.4 Robust Route Planning

We considered the computation of robust routes in road networks as defined by the framework of Buhmann et al. [8]. We showed that, in many cases, the problem of computing a robust route can be approximated by computing a path in the first intersection which, in turn, can be obtained from a Pareto front in a road network with time-dependent edge weights. For this reason, we considered an algorithm for the computation of such a front, and showed different ways to apply a most standard speed-up technique, namely bidirectional search, to it.

For the future, many interesting questions remain open. In particular, it might be interesting to assess the effectiveness of bidirectional search for bi-criteria quickest path problems in a setting more general than the one considered here. Furthermore, the three phases bidirectional search proposed allows the tuning of some implementation details, like the termination condition of phase 1. It might be interesting to consider implementation alternatives to the one proposed. Finally, there exist other speed-up techniques that are typically paired with bidirectional search like, for example, A^* search. It is one of our tasks for the future to investigate how to apply such techniques to the proposed algorithms, and whether this would result in more efficient solutions.

References

[1] Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. Hierarchical hub labelings for shortest paths. In *Proceedings of the 20th Annual European Symposium on Algorithms (ESA'12)*, volume 7501 of *Lecture Notes in Computer Science*, pages 24–35. Springer, 2012.

[2] Rachit Agarwal. The space-stretch-time trade-off in distance oracles. Manuscript, July 2013.

[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. ACM-SIAM, 2013.

[4] Hannah Bast, Daniel Delling, Andrew V. Goldberg, Matthias Müller–Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F. Werneck. Route planning in transportation networks. Technical Report MSR-TR-2014-4, Microsoft Research, 2014.

[5] Sabyasachi Basu and Martin Meckesheimer. Automatic outlier detection for time series: an application to sensor data. *Knowledge and Information Systems*, 11(2):137–154, 2007.

[6] Gernot Veit Batz, Robert Geisberger, Peter Sanders, and Christian Vetter. Minimum time-dependent travel times with contraction hierarchies. *ACM Journal of Experimental Algorithmics*, 18(1.4):1–43, April 2013.

[7] Reinhard Bauer, Tobias Columbus, Ignaz Rutter, and Dorothea Wagner. Search-space size in contraction hierarchies. In *Proceedings of the 40th International Colloquium on Automata, Languages, and Programming (ICALP'13)*, volume 7965 of *Lecture Notes in Computer Science*, pages 93–104. Springer, 2013.

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

[9] 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.

[10] Haibo Chen, Susan Grant-Muller, Lorenzo Mussone, and Frank Montgomery. A study of hybrid neural network approaches and the effects of missing data on traffic forecasting. *Neural Computing & Applications*, 10(3):277–286, 2001.

[11] 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.

[12] 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.

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

[14] Brian C. Dean. Continuous-time dynamic shortest path algorithms. Master's thesis, Massachusetts Institute of Technology, 1999.

[15] Brian C. Dean. Algorithms for minimum-cost paths in time-dependent networks with waiting policies. *Networks*, 44(1):41–46, 2004.

[16] Brian C. Dean. Shortest paths in fifo time-dependent networks: Theory and algorithms. Technical report, MIT, 2004.

[17] Frank Dehne, Omran T. Masoud, and Jörg-Rüdiger Sack. Shortest paths in time-dependent FIFO networks. *ALGORITHMICA*, 62(1-2):416–435, 2012.

[18] Daniel Delling. Time-dependent SHARC-routing. *Algorithmica*, 60(1):60–94, May 2011.

[19] Daniel Delling, Andrew V. Goldberg, Thomas Pajor, and Renato F. Werneck. Customizable route planning. In *Proceedings of the 10th International Symposium on Experimental Algorithms (SEA'11)*, volume 6630 of *Lecture Notes in Computer Science*, pages 376–387. Springer, 2011.

[20] 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.

[21] Daniel Delling, Andrew V. Goldberg, Ilya Razenshteyn, and Renato F. Werneck. Exact combinatorial branch-and-bound for graph bisection. In *Proceedings of the 14th Meeting on Algorithm Engineering and Experiments (ALENEX'12)*, pages 30–44. SIAM, 2012.

[22] Daniel Delling, Peter Sanders, Dominik Schultes, and Dorothea Wagner. Engineering route planning algorithms. In *Algorithmics of Large and Complex Networks - Design, Analysis, and Simulation [DFG priority program 1126]*, pages 117–139, 2009.

[23] Daniel Delling and Dorothea Wagner. Time-dependent route planning. In *Robust and Online Large-Scale Optimization*, volume 5868 of *Lecture Notes in Computer Science*, pages 207–230. Springer, 2009.

[24] Daniel Delling and Renato F. Werneck. Faster customization of road networks. In *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13)*, volume 7933 of *Lecture Notes in Computer Science*, pages 30–42. Springer, 2013.

[25] Sofie Demeyer, Jan Goedgebeur, Pieter Audenaert, Mario Pickavet, and Piet Demeester. Speeding up martins' algorithm for multiple objective shortest path problems. *4OR*, 11(4):323–348, 2013.

[26] Themistoklis Diamantopoulos, Dionysios Kehagias, Felix G König, and Dimitrios Tzovaras. Investigating the effect of global metrics in travel time forecasting. In *Proceedings of 16th International IEEE Conference on Intelligent Transportation Systems (ITSC)*, 2013.

[27] Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

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

[32] eCOMPASS. D2.3 – 2 validation and empirical assessment of algorithms for eco-friendly vehicle routing. Technical report, The eCOMPASS Consortium, 2013.

[33] eCOMPASS Project (2011-2014). `http://www.ecompass-project.eu`.

[34] Raphael A. Finkel and Jon Louis Bentley. Quad trees: A data structure for retrieval on composite keys. *Acta Inf.*, 4:1–9, 1974.

[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. *Algorithmica*, 68(4):1075–1097, 2014. Preliminary version in ACM-SIAM SODA 2011.

[37] Delbert R. Fulkerson and O. A. Gross. Incidence matrices and interval graphs. *Pacific Journal of Mathematics*, 15(3):835–855, 1965.

[38] Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *Experimental Algorithms, 7th International Workshop, WEA 2008, Provincetown, MA, USA, May 30-June 1, 2008, Proceedings*, pages 319–333, 2008.

[39] 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.

[40] Robert Geisberger and Christian Vetter. Efficient routing in road networks with turn costs. In *Proceedings of the 10th International Symposium on Experimental Algorithms (SEA'11)*, volume 6630 of *Lecture Notes in Computer Science*, pages 100–111. Springer, 2011.

[41] Alan George and Joseph W. Liu. A quotient graph model for symmetric factorization. In *Sparse Matrix Proceedings*. SIAM, 1978.

[42] Andrew V. Goldberg and Chris Harrelson. Computing the shortest path: *A* search meets graph theory. In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2005, Vancouver, British Columbia, Canada, January 23-25, 2005*, pages 156–165, 2005.

[43] Tristram Gräbener, Alain Berro, and Yves Duthen. Time dependent multiobjective best path for multimodal urban routing. *Electronic Notes in Discrete Mathematics*, 36:487–494, 2010.

[44] Horst W. Hamacher, Stefan Ruzika, and Stevanus A. Tjandra. Algorithms for time-dependent bicriteria shortest path problems. *Discrete Optimization*, 3(3):238–254, 2006.

[45] Pierre Hansen. Bicriterion path problems. In Günter Fandel and Tomas Gal, editors, *Multiple Criteria Decision Making Theory and Application*, volume 177 of *Lecture Notes in Economics and Mathematical Systems*, pages 109–127. Springer Berlin Heidelberg, 1980.

[46] Martin Holzer, Frank Schulz, and Dorothea Wagner. Engineering multilevel overlay graphs for shortest-path queries. *ACM Journal of Experimental Algorithmics*, 13(2.5):1–26, December 2008.

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

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

[49] 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*, pages 337 – 360, 1997.

[50] Spyros Kontogiannis and Christos Zaroliagis. Distance oracles for time-dependent networks. In *J. Esparza et al. (eds.), ICALP 2014, Part I*, volume 8572 of *LNCS*, pages 713–725. Springer-Verlag Berlin Heidelberg, 2014. Full version as eCOMPASS Technical Report (eCOMPASS-TR-025) / ArXiv Report (arXiv.org>cs>arXiv:1309.4973), September 2013.

[51] Richard J. Lipton, Donald J. Rose, and Robert Tarjan. Generalized nested dissection. *SIAM Journal on Numerical Analysis*, 16(2):346–358, April 1979.

[52] Manolis IA Lourakis. A brief description of the levenberg-marquardt algorithm implemented by levmar. *Foundation of Research and Technology*, 4:1–6, 2005.

[53] Ernesto Queirós Vieira Martins. On a multicriteria shortest path problem. *European Journal of Operational Research*, 16(2):236 – 245, 1984.

[54] Jorge J Moré. The levenberg-marquardt algorithm: implementation and theory. In *Numerical analysis*, pages 105–116. Springer, 1978.

[55] Giacomo Nannicini, Daniel Delling, Leo Liberti, and Dominik Schultes. Bidirectional A* search on time-dependent road networks. *Networks*, 59:240–251, 2012. Best Paper Award.

[56] 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.

[57] Mihai Patrascu and Liam Roditty. Distance oracles beyond the Thorup–Zwick bound. In *Proc. of 51th IEEE Symp. on Found. of Comp. Sci. (FOCS '10)*, pages 815–823, 2010.

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

[59] Léon Planken, Mathijs de Weerdt, and Roman van Krogt. Computing all-pairs shortest paths by leveraging low treewidth. *Journal of Artificial Intelligence Research*, 2012.

[60] Ely Porat and Liam Roditty. Preprocess, set, query! In *Proc. of 19th Eur. Symp. on Alg. (ESA '11)*, LNCS 6942, pages 603–614. Springer, 2011.

[61] Peter Sanders and Christian Schulz. Think locally, act globally: Highly balanced graph partitioning. In *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13)*, volume 7933 of *Lecture Notes in Computer Science*, pages 164–175. Springer, 2013.

[62] Frank Schulz, Dorothea Wagner, and Karsten Weihe. Dijkstra's algorithm on-line: An empirical case study from public railroad transport. *ACM Journal of Experimental Algorithmics*, 5(12):1–23, 2000.

[63] Hanif D. Sherali, Kaan Ozbay, and Sairam Subramanian. The time-dependent shortest pair of disjoint paths problem: Complexity, models, and algorithms. *Networks*, 31(4):259–272, 1998.

[64] Brian L Smith and Michael J Demetsky. Traffic flow forecasting: comparison of modeling approaches. *Journal of transportation engineering*, 123(4):261–266, 1997.

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

[66] Christian Sommer. Shortest-path queries in static networks. *ACM Computing Surveys*, 46, 2014.

[67] Christian Sommer, Elad Verbin, and Wei Yu. Distance oracles for sparse graphs. In *Proc. of 50th IEEE Symp. on Found. of Comp. Sci. (FOCS '09)*, pages 703–712, 2009.

[68] Mikkel Thorup and Uri Zwick. Approximate distance oracles. *J. of ACM*, 52:1–24, 2005.

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

[70] Eleni I Vlahogianni, John C Golias, and Matthew G Karlaftis. Short-term traffic forecasting: Overview of objectives and methods. *Transport reviews*, 24(5):533–557, 2004.

[71] 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. In *Data Mining Workshops (ICDMW), 2010 IEEE International Conference on*, pages 1372–1376. IEEE, 2010.

[72] C. Wulff-Nilsen. Approximate distance oracles with improved preprocessing time. In *Proc. of 23rd ACM-SIAM Symp. on Discr. Alg. (SODA '12)*, 2012.

[73] C. Wulff-Nilsen. Approximate distance oracles with improved query time. arXiv abs/1202.2336., 2012.

[74] Tim Zeitz. Weak contraction hierarchies work! Bachelor thesis, Karlsruhe Institute of Technology, 2013.