Institute of Formal Methods in Computer Science

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Learning metrics for balancing load in street-networks

Dominic Parga Cacheiro

Course of Study:	Informatik
Examiner:	Prof. Dr. Stefan Funke
Supervisor:	Florian Barth (M.Sc.)
Commenced:	December 19, 2019
Completed:	August 14, 2020

Abstract

Especially in daily rush-hour-scenarios, a street-network requires enough capacity to support the amount of drivers. On the other hand, a street-network of too much capacity would be inefficient outside of rush-hour-scenarios. Hence, the overload during rush-hour should be spreaded over the network to reduce the impact of the overload (e. g. traffic-jams). To improve the distribution of routes, alternative routes in multicriteria settings are computed. To support multicriteria settings, Dijkstra's algorithm is combined with personalized routing. Here, these multicriteria or multiple metrics, e. g. travel-time or travel-distance, can be reduced to scalars by applying preferences to them. Resulting routes or paths are called personalized paths. However, many previous approaches for computing alternative routes need too much parameter-tuning or simply lack in their computational complexity, their needed runtime or in the diversity of their found routes. Therefore, this thesis presents a combination of enumerating personalized paths with the creation of a new penalizing metric. The goal is to compensate other popular metrics with this new metric, which then improves the spread of many routes in the network. This new metric can be processed by every routing-algorithm, that is capable of dealing with multicriteria-routes. By enumerating personalized paths with this new penalization, found routes can be distributed successfully over the network. In addition, user-provided tolerances for preferred metrics are hold. In the end, some experiments on street-networks from OpenStreetMap are presented. Here, results are compared between routes from Dijkstra (with personalized routing) and the enumeration of personalized routes. To speed the route-queries up significantly, the underlying graphs of the networks are contracted via contraction-hierarchies before the searches happen. Here, the contraction is realized by a linear program to improve the performance with multicriteria contractions.

Contents

List of Figures

List of Tables

1 Introduction

A navigation-system, asked for the fastest route, usually returns a route over larger streets, like motorways. When many drivers receive the same answer from their independent navigation-systems, the traffic tends to overload these larger streets. These larger streets probably have multiple lanes and allow a higher speed-limit than smaller streets, resulting in a high driver-capacity and hence a high throughput. In addition, due to rush-hours, the traffic is not balanced over the day. In fact, a street-network can be quite loaded during rush-hour-scenarios. Increasing the network's capacity often isn't sensible, because it reduces the overall efficiency in non-rush-hour-scenarios. So, a typical street-network is a tradeoff between fitting the rush-hour-scenario and keeping the network efficient, when there is no rush-hour. Hence these aspects lead to traffic-jams and former fast routes become inefficient due to their popularity. Besides environmental impact and circumstances to handle the daily amount of rush-hour-traffic, traffic-jams in large (capital) cities cost days of the people's time and hence much money per year ([INR19]). Taking the popularity into account seems to be a mandatory step to reduce traffic-jams. At the latest, this will be interesting with many self-driving cars, where many routes will be needed and driven automatically. It would be inefficient to send all vehicles over the same streets. So distributing this amount of traffic over the network is and will be important to save both, time and money.

This thesis explains, how a given street-network of multidimensional metrics (travel-distance, travel-time) can be augmented, such that diverse alternative paths can be found and used to spread the load (of a typical workload) while limiting the detours of individual source-target-pairs (s-t-pairs). Although the presented methods can be transferred to different kinds of networks, this thesis focusses on street-networks. A different kind of network can be wireless-networks or the Internet, where the focus lays more on nodes than edges. Here, data-packages are transmitted much faster than drivers are driving through street-networks and traffic-jams occur in nodes, not in the edges.

1.1 Related Work

In the following, dynamic and static methods are presented, that are currently used to solve these issues, having their pros and cons. This thesis picks up on some static methods.

Dynamic (or adaptive) approaches describe the ability of vehicles finding new paths based on current conditions or circumstances, while driving along another path. So in opposite to static approaches, dynamic routing processes time-dependent, volatile data and processes it (more or less) immediately. Statically in this context means calculating the routes with help of non-volatile data. So, routing-computations are not dependent on a street-network's state, but on its attributes. Paths are calculated before a vehicle drives it, and the vehicles' path (typically) doesn't change over time. That's why a static approach is less dependent on dynamic traffic and more dependent on the street-network itself, in opposite to the dynamic approaches.

1.1.1 Dynamic routing

One example for a dynamic approach is Google, that has gathered [Bar09] anonymous data from Android-users to predict current traffic-jams in the street-network. Since most smartphones use Android [Wor20], there is enough data being collectable for sufficient accuracy. On the other hand, this approach requires a global observer, including infrastructure, being able to process incoming queries in reasonable time. With the drivers' data, like their location, the global observer can compute heuristics to approximate the current workload of other network-parts. Usually, this cost additional knowledge-exchange between the global observer (e. g. Google) and the drivers. If the drivers aren't willing or able to share relevant information, the infrastructure has to be extended to collect relevant data. The need of extending the infrastructure makes the routing dependent on the infrastructure and could cost money.

1.1.2 Static routing

There are several static computation-methods to compute a selection of several routes with respect to only one metric, that are sufficiently distinct. One idea is the computation of the k shortest paths for a given s-t-pair. According to [Epp99], the computation is complex while resulting paths don't differ much. Another quite intuitive idea is the penalization of popular routes to reduce their usage before computing the shortest path again. The focus in [BDGS11] lays on measuring the quality of alternative paths via node-degrees and using heuristics for computing multidimensional shortest paths. In [CBGL15], the problem is addressed, that paths of a set of found k shortest paths for a given s-t-pair should differ from eachother, not only from the shortest path itself. Hence they build the set iteratively and discard unwished paths with a user-provided threshold. To this end, a similarity between paths is defined. However, both approaches suffer from the sensitivity of tuning the parameters and graphs are considered to have only one metric.

For multiple metrics, another approach could be to distribute the found paths by using alternative routes. Here, the drivers would not necessarily get only the option of the fastest route, but also some other reasonable ones. When it comes to graphs of multidimensional metrics, pareto-optimal solutions occur implicitly. Thus, a first idea is computing all pareto-optimal paths. As stated in [DW09], computing all pareto-optimal paths is too expensive to be practical at all. That's why [BFS19] develops an algorithm, that enumerates paths with extending the search for shortest paths by a geometric interpretation in the metrics' underlying cost-space. To handle multidimensional metrics, the algorithm uses personalized routing (introduced in [GKS10]), which takes predefined preferences to compute one scalar out of multiple metrics. To speed the queries up, a preprocessing-technique called contraction-hierarchies is applied in their paper. In addition, while the original approach of contraction-hierarchies is developed in [GSSD08], an adjusted version [FLS17] using linear programming is used for this thesis. The adjustment enables more efficient contraction-hierarchies when using personalized routing. Further, to reduce the search-space, the enumeration is restricted by a clever check from [BF19] without influencing its correctness or completeness.

1.2 Main contribution

The contribution of this thesis is the combination of penalization and personalized routing. A computation-phase called balancing is added, that analyzes the street-network with the help of the Restricted Enumerating Personalized Routing (REPR) from [BF19]. This preprocessing leads to a new metric, penalizing edges of high popularity. Applying personalized routing with all metrics, including the new one(s), leads to a more distributed set of routes. Thus, the street-network is used in a broader manner. At the same time, due to personalized routing, routing-queries can be processed as quickly as before. To guarantee a certain tolerance towards the shortest path, a filter is added to the REPR.

1.3 Outline

This thesis is structured as following.

Chapter 2 – Preliminaries: This chapter explains the methods, that are combined in this thesis to create the new metric, that balances the provided graph to spread the found routes. These methods refer to a way of storing the graph in memory, describe the speedup of multicriteria-routing by contraction-hierarchies and a modified Dijkstra, and explain an algorithm to find alternative paths in a multicriteria setting.

Chapter 3 – Optimizing network-utilization by balancing workload: After the preliminaries are explained, the balancing is presented. The balancing is the core of this thesis and describes the looped chain of procedures, that creates the new metric.

Chapter 4 – Experiments: In this chapter, street-networsk from OpenStreetMap are used to show the balancing in action.

Chapter 5 – Conclusion and Outlook: Here, a summary is depicted and future work is discussed, such as performance-improvements and variations of balancing.

2 Preliminaries

The creation of a new metric is basically done by a chain of methods, that is looped to update the metric. This chapter introduces these methods and explains them briefly.

2.1 Graphs stored as adjacency-array

Let a street-network be given as a graph $G = (V, E)$ with vertices V (also called nodes), edges E and a cost-function $c : E \rightarrow \mathbb{R}^d_+$, that returns a cost-vector for each edge $e \in E$, also called multidimensional metrics in this thesis. Metrics are also called costs depending on the context.

For the theory of this thesis, above definitions are sufficient. However, the implementation-details in Section 3.2 concretize the graph's definition to store the graph efficiently. The underlying graph's data-structure is based on the idea of adjacency-arrays, which is mentioned in [MS08]. For this reason, this brief idea is extended both in the following to demonstrate the brief idea and in Section 3.2. Before the adjacency-array can be concretized, the graph has to be slightly redefined, so it can be referred with arrays and indices.

Since V and E are finite, let $n := |V|$ be the number of vertices and $m := |E|$ be the number of edges. Then, the graph's components are interpreted and stored as follows.

Vertices: For implementation, vertices in V are just ids, which are stored in an array in ascending order. The order is needed for accessing the graph's data later. So let V be an array of node-ids and $V[i_V]$ be the node-id at node-index $0 \leq i_V < n$.

Edges: Edges are considered to be directed and E is defined as an array of tuples of two vertex-indices, so $E \subseteq \{0, 1, \ldots, n-1\}^2$. However, the sources and destinations in E are stored in separate arrays, so $E := (S, D)$ and $E[i_E] = (S[i_E], D[i_E])$ is the edge from source-index in S at edge-index i_E to destination-index in D at edge-index i_E. Since the graph is stored as adjacency-array, E is considered to be sorted by source-, then by destination-id (although vertex-indices are stored in S and D), in ascending order. Let the respective offset-array for remaining D be O_D. The creation and use of O_D is explained below. With the interpretation of the offset-array, storing S will not be necessary, because S (without duplicates) will be $[0, 1, \ldots, n-1]$ and thus match V.

After these definitions, the core of the adjacency-array is basically the creation of the offset-array O_D, which then helps accessing the graph's components in constant time. For every vertex $v \in V$, the offset-array O_D stores the edge-index i_E, where the leaving edges starting in v occur. Since V is sorted by source-id first, all these leaving edges will be grouped in O_D (and D). Here, the sorted E is needed. This is demonstrated in Table 2.1 and explained below.

Vertices V ($\sim$ sources S)	id_0	id_1			id_2			id_3			id_4		-
Destinations D	1	0	2	3	1	3	4	1	2	4	2	3	-
Offsets O_D	0	1			4			7			10		12

Table 2.1: A graph with $|V| = 5$ and $|E| = 12$ stored as adjacency-array, shown with the respective offset-array. Every edge in the graph is bidirectional, but stored directed and represents as a tuple of source- and destination-vertex. All destinations of leaving edges starting in $V[s = 3] = id_3$ are in D between positions $O_D[s] = 7$ (including) and $O_D[s + 1] = 10$ (excluding).

Let $e = E[i_E] = (S[i_E], D[i_E]) = (s, d)$ be an edge at position i_E and s be the position of the edge's source in V. Because the edges are sorted by source, than by destination, all edges starting in $V[s]$ are in D between positions $O_D[s]$ (including) and $O_D[s + 1]$ (excluding). Since every edge-data can be stored according to the edges' destinations D (like metrics), this allows quick access to all relevant edge-components by simply providing the edge-index while keeping the memory-usage low. Besides that, it can be extended quite easily and complex, as it is done in Section 3.2.

2.2 Contraction-hierarchies

This thesis uses sets of many s-t-pairs to search for optimum paths. These paths are used to optimize routing-algorithms for the underlying street-network. In fact, the largest bottleneck of the whole thesis is the search for optimum paths. This also offers the opportunity to improve the performance. One approach to do this is the reduction of the search-space and a method called contraction-hierarchies is used for this purpose. The algorithm is divided into a preprocessing and the actual user-provided query. The first is explained here, the latter in Section 2.3.

The preprocessing phase iterates over all vertices in a graph and contracts them. The contraction basically removes each vertex $v \in V$ from the graph. Every shortest path from a neighbouring vertex of v to a neighbouring vertex of v, that is the only shortest path between these respective neighbours and that needs v to exist, is replaced by a new edge between the respective neighbours. This new edge is called shortcut and represents the shortest path by adapting its weights. Additionally, levels ($\in \mathbb{N}$) are assigned to every vertex, and the levels increase with the iteration-progress. These levels are considered in the query to reduce the search-space.

The quality of the contracted graph (and the query-time with it) highly depends on the contraction-order. If too many shortcuts have to be added with a vertex-contraction, the graph gets more verbose. For this reason, different criteria for good vertices are applied, e. g. considering the number of created shortcuts and the removed edges or whether a neighbour of a vertex has just been contracted.

This thesis deals with multidimensional metrics. Because of this, the insertion of a new shortcut needs to be redefined. If any α exists, such that the only optimum path with respect to minimum $\sum_{i=1}^{d} \alpha_i \cdot c_i(path)$ needs the contracted vertex v, then a shortcut is added. The challenge here is to explorate the set of possible α efficiently. [FLS17] provides an approach using linear programming using Dijkstra as separation-oracle. A linear program solves an optimization-problem in finite many variables, where a linear objective function is optimized under a finite number of linear equality or inequality constraints. A separation-oracle helps in discarding non-optimal solutions more efficiently.

2.3 Dijkstra's algorithm (bidirectional) in contracted graphs

An unidirectional Dijkstra (introduced in [Dij59]) computes an optimum path for a provided s-t-pair in a graph with only one metric. It assigns the weight of zero to $s \in V$, marks s as visited and goes through all neighbours $v \in V$ of s to update their path-cost, that belongs to the path of minimum cost from s to v so far. Then, every neighbour of s is pushed to a priority-queue, except for those, that have already been visited (which is only s in this first iteration). In the next step, the vertex of minimum cost in the priority-queue is pulled from the queue and marked as visited. It updates its neighbours' optimum paths' cost and all neighbours, that improve their cost by using v, are pushed into the priority-queue. Note, that a node can be enqueued multiple times with different costs. This procedure stops as soon as $t \in V$ is pulled from the priority-queue. The optimum path can be constructed by remembering predecessors during the visit-phase.

To reduce the search-space of Dijkstra, it can be modified to search from both s and from t towards each other, which is called bidirectional Dijkstra. Pushing vertices to a common priority-queue, while vertices remember their search-direction (forwards from s or backwards from t), can be used to balance both sub-queries. As soon as a vertex is visited by both sub-queries, no more neighbours need to be enqueued. This vertex, that is visited by both sub-queries, is called meeting-vertex. If the best meeting-vertex has better total costs than all vertices v remaining in the queue, the query stops. The optimum path can be constructed from the meeting-vertex, similar to unidirectional Dijkstra, by remembering predecessors and successors during the visit-phase.

When doing bidirectional Dijkstra with contracted graphs, some adjustments are required. To reduce the search-space, only neighbours of dequeued vertices are considered, that have at least the assigned level of v. Typically, neighbouring vertices don't have common levels, but the contraction may be stopped before all vertices are contracted. This speeds the query up significantly. However, it also causes the sub-queries to search in different sub-graphs. The sub-queries in bidirectional Dijkstra search in the same sub-graphs. That leads to the issue, that found optimum paths of a sub-query are not necessary optimal for the whole query. To fix this, just keep on enqueuing neighbours even when a meeting-vertex is already found. As soon, as the best meeting-vertex has better total costs than all vertices remaining in the queue, the query stops.

The optimum path can be constructed as in the case of bidirectional Dijkstra. Of course, this optimum path might contain shortcut-edges, but these can be replaced by the original edges recursively.

2.4 (Restricted) Enumerating personalized routing

Personalized routing is basically applying a preference-vector α to reduce the multidimensional costs to one scalar by computing the dot-product of α and the respective cost-vector. Dijkstra can be adjusted to compute the optimum path considering a user-provided α. Instead of computing paths' costs for one metric, they are computed for the α-dependent dot-product.

Interpreting a path's cost as point in the d-dimensional cost-space, the dot-product with a vector α can be seen as the projection of the path's cost-vector onto the chosen vector α. In other words, Dijkstra is searching for the minimum dot-product, hence the projection onto α of minimum length. Thus, Dijkstra finds extrema of the convex hull of all existing paths for a given s-t-pair in the underlying d-dimensional cost-space.

This is used in [BFS19] to develop an algorithm enumerating this convex hull's extrema. An initial convex hull in the d-dimensional cost-space contains several $(d-1)$-dimensional simplexes (e. g. triangles in 3D). In Section 3.1.2, it is explained and adjusted, how such an initial convex hull can be found. Each simplex has a normal-vector, which can be computed by a d-dimensional linear-equation-system. Since simplexes are only $(d-1)$-dimensional, the length of the resulting normal-vector is arbitrary. To fix this, the last equation in the system forces the sum of all entries in the normal-vector to be 1. It occurs, that this normal-vector can be interpreted as α-vector, which leads to identical costs for all paths being involved in the simplex. Now, if Dijkstra finds a path with this α, that is not part of the simplex, this path must be better than every path of the simplex (with respect to this particular α) and thus must be another extremum of the convex hull. Adding the new found path to the convex hull brings new simplexes up, that can be treated as already described, possibly resulting in new paths for the found convex hull.

This method stops as soon as no new paths has been found and is called .

When using initial α-vectors preferring only one metric, the resulting path is the optimum path over all α with respect to this very metric. With this, a tolerance can be defined, that describes the accepted maximum relative difference between this optimum path and any other path, that is found. [BF19] improves this enumeration in its runtime by discarding simplexes, where all paths have at least one metric, where no path can beat the tolerance. This allows to discard the simplex and reduces the search-space, thus runtime. The resulting method is a slightly changed version of Enumerating Personalized Routing (EPR) and is called Restricted Enumerating Personalized Routing (REPR). It is used in this thesis to find multiple alternative paths.

3 Optimizing network-utilization by balancing workload

The last chapter brings an overview of different notations, algorithms and techniques. In this chapter, all these instruments are combined to a chain for distributing routes more evenly. The goal is to augment a given graph, such that diverse alternative paths can be found and used to spread too much load while limiting the detours of individual s-t-pairs. After the theory is explained, implementation-details are discussed. When talking about performance, it is referred to the used testing-machine, which is an ordinary, but good home-computer. Further details about the testing-machine and experimental results can be found in Chapter 4 on page 29.

For a given s-t-pair and a street-network given as a graph $G = (V, E)$ with vertices V, edges E and a cost-function $c : E \longrightarrow \mathbb{R}_+^d$, that returns a cost-vector for each edge $e \in E$, a path should be found through an algorithm, that implicitly tends to distribute found paths over the network, while keeping the paths' costs sufficiently near every cost-dimension's optimum. Note, that $\mathbb{R}_+^d$ doesn't contain zero, which would, depending on the metric, result in bad alternative routes otherwise.

To achieve this, a computation-phase called balancing analyzes the graph G and computes a new metric, called workload-metric. With this workload-metric, routing-algorithms like REPR and Dijkstra (with personalized routing) can compute paths as usual. For comparison, multiple scenarios using different combinations of these two routing-algorithms will be used. Please note, that Dijkstra doesn't guarantee a tolerance when used with personalized routing and hence, some combinations don't provide an upper bound for path-costs. This will also be pointed out in the respective experiments in Chapter 4.

This thesis focusses only on distance and travel-time, where travel-time is the only metric with a provided tolerance.

It should be noted, that Dijkstra (bidirectional) instead of A* is used for every occuring routing-query, may it be in REPR or anywhere else. The reason is, that A* works with heuristics, which are difficult to generalize over custom, artificial metrics. The only benefit from using A* is the reduction of the search-space, which can be reduced much more using bidirectional Dijkstra on graphs contracted via contraction-hierarchies. Besides that, the graph has multidimensional metrics, for which reason Dijkstra is used with personalized routing.

3.1 Balancing to create a new metric

This balancing is depicted in Figure 3.1 on the next page and explained briefly in the caption or detailled in the following. It basically runs routing-queries, counts the resulting workload and adjusts a third metric (in addition to travel-distance and travel-time) to influence future query-executions. After the new workload-metric will have been settled, routing-algorithms for multidimensional

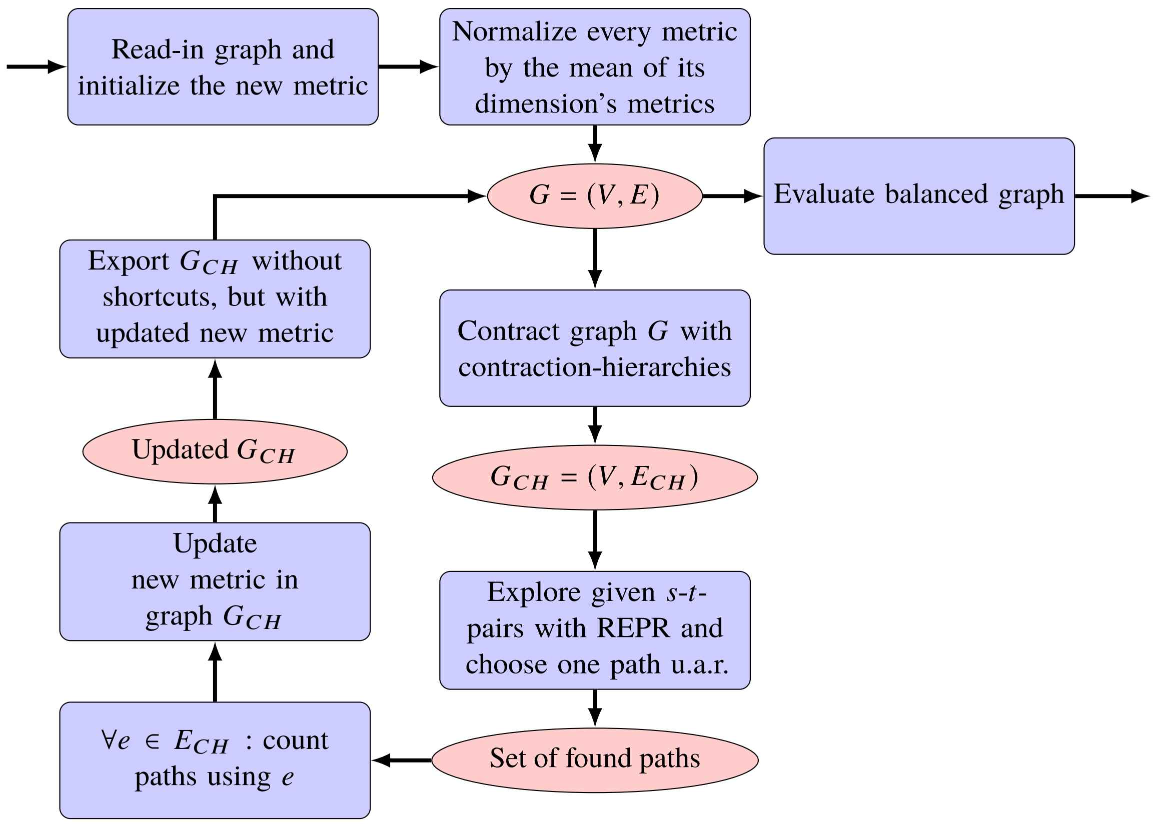

Read-in graph and initialize the new metric
Normalize every metric by the mean of its dimension's metrics
$G = (V, E)$
Evaluate balanced graph
Export G_{CH} without shortcuts, but with updated new metric
Contract graph G with contraction-hierarchies
Updated G_{CH}
$G_{CH} = (V, E_{CH})$
Update new metric in graph G_{CH}
Explore given s-t-pairs with REPR and choose one path u.a.r.
$\forall e \in E_{CH}$: count paths using e
Set of found paths

Figure 3.1: This flow shows the balancing, that analyzes a given graph G. Here, an evaluation-phase is included afterwards. Shapes for actions are rectangular and blue, shapes for data are elliptical and red. A new, artificial metric is created for G and gets updated iteratively. In the end, this new metric allows a routing-algorithm to distribute upcoming workload (like during rush-hour) more evenly over the underlying street-network in G. This more spreaded workload can be seen in the evaluation-step, where paths for a set of s-t-pairs are searched, once using and once ignoring the new metric.
Note, that the action doing the path-searches ignores the new metric in the first balancing-iteration.

metrics can run queries on the balanced graph G as before, now considering the new workload-metric. Therefore, queries will be run in the experiments (Chapter 4) to evaluate the new spread of the street-network. The set of s-t-pairs will be chosen differently than the set used in balancing, to reduce the dependence on it.

3.1.1 Selection of route-pairs

Let a street-network be given as a graph $G = (V, E)$ as defined in Section 2.1. Before such a graph can be balanced, a set of s-t-pairs has to be chosen. Here, different strategies can be applied. For simplicity, the set of s-t-pairs, which are used for balancing G, might consist of s and t chosen uniformly at random (u. a. r.) from V. This has the disadvantage, that a sufficiently large set of

s-t-pairs is needed to overload G. This clearly becomes a bottleneck for performance on larger graphs (beginning with small German counties like Saarland with almost $|V| \approx 580{,}000$ and $|E| \approx 1{,}160{,}000$ after parsing), especially when using REPR, that computes multiple Dijkstra-queries per run. To alleviate the computational burden of a large number of s-t-pairs, knowledge of the typical workloads in this graph could be used. Instead of picking u. a. r. from $|V|$, other distributions could be chosen by weighting $v \in V$ differently or by using some s-t-pairs multiple times (on purpose, not only by random). For example, real, daily, critical s-t-pairs are not distributed u. a. r. over V. The rush-hour in the morning, when everybody travels from their home to work, is a good counterexample here. The set of chosen t might be much smaller than the set of chosen s, and less people live near industry or motorways. Heuristics as described in [BLM+14] can return s-t-pairs based on approximating population-data from street-networks. So considering more specific sets of s-t-pairs might be interesting and helpful to boost the balancing up, but is not covered in this thesis. The reason for keeping the choice u. a. r. is the easy implementation, while results are sufficiently good enough, as long as G shows differently (over-) loaded streets.

3.1.2 First steps of the balancing-process

At first, the graph G, that should be balanced, has to be read in. Before continuing, every metric is divided by the respective metric-mean resulting in $G = (V, E)$ of normalized metrics. Non-normalized metrics of different scales would cause REPR to find α-values of different scale, making some metrics more important than others just based on their different scales. With metrics of different importance, the chosen paths tend to the respective metrics and distort the spread. So normalization doesn't just allow to compare different graphs (of different metrics), but also improves the spread in REPR.

This graph G is contracted via contraction-hierarchies to G_{CH}. The step of contracting is not needed for correctness, but reduces the needed runtime (even if G is not being fully contracted). For this reason, G instead of G_{CH} is used in the following, to emphasize, that contraction is not necessary from a theoretical point of view.

Given the s-t-pairs and a graph G, a path in G is searched for every s-t-pair. The search may use either Dijkstra or REPR. Multiple paths are found by REPR, so one of them is chosen u. a. r. after filtering all found paths by a given tolerance. Note, that Dijkstra with personalized routing doesn't consider any tolerance. But since REPR calls Dijkstra multiple times, just Dijkstra is faster. On the other hand, when using REPR with choosing a path u. a. r., the resulting paths are more spreaded over the network. This results in a better coverage of E when updating the workload-metric, which allows the REPR to find more pareto-optimal personalized routes in the convex hull. Both methods are compared in the experiments in Chapter 4 on page 29.

When using REPR, the quality of the balanced graph highly depends on the choice of initial paths for REPR. The basic approach in [BFS19] suggests to use $(d + 1)$ initial alphas to define one initial convex hull for a graph of d-dimensional metrics.

$$\begin{aligned} \left(\alpha^{(1)}, \alpha^{(2)}, \ldots, \alpha^{(d)}\right) &= \mathbb{I}^d \\ \alpha^{(d+1)} &= \left(\frac{1}{d}, \ldots, \frac{1}{d}\right)^T \end{aligned} \tag{3.1}$$

This suggestion is nice to get a tolerance, because every metric's optimal path is computed. However, in the balancing, this suggestion needs some adjustment. When the new workload-metric is added (hence let the graph's metrics' dimension be $d+1$), $\alpha^{(d+1)}$ becomes $\alpha^{(d+2)}$, which is a different direction in the $(d+1)$-cost-space than the previous $\alpha^{(d+1)}$ points to in the $(d+1)$-cost-space. For example (considering only α's direction, not its length), let $d = 2$ become $d = 3$, hence $\alpha^3 = (1,1) \in \mathbb{R}_+^2$ becomes $\alpha^4 = (1,1,1) \in \mathbb{R}_+^3$ and respectively $(1,1,0) \in \mathbb{R}_+^3$ from the previous case $d = 2$ is missing now. In the case of Saarland in the experiments, this lead to the issue, that the initially found paths (as suggested in [BFS19]) weren't enough to get an initial convex hull. Because of this, less paths have been found on Saarland in the experiments. Therefore, the balancing extends the original suggestion by [BFS19] and takes the initial $\alpha^{(j)}$ from $\{0,1\}^{(d+1)}$ excluding $(0,\ldots,0)^T$. Only the direction of α is relevant, so the previous notation in Equation (3.1) can be simplified.

$$\alpha \in \{0,1\}^{(d+1)} \setminus \left\{(0,\ldots,0)^T\right\} \Rightarrow \alpha \text{ is an initial } \alpha^{(j)} \tag{3.2}$$

After finding a set of paths for the given s-t-pairs, every edge $e \in E$ counts the number of found paths, that contain this edge e. This workload of balancing-iteration i is normalized by all workloads' mean and the result is used to update the graph's workload-metric.

$$\begin{aligned} workloads_{e,i} &= |\{(s,t) \in s\text{-}t\text{-pairs} : \text{found path from } s \text{ to } t \text{ uses } e \in E\}| \\ \Rightarrow new_{e,i} &= \frac{workloads_{e,i}}{mean(workloads_i)} \end{aligned} \tag{3.3}$$

3.1.3 Updating the new metric with the collected workload

Updating the old workload-metric is a very small step in the balancing with a large impact on the ongoing balancing's performance and the quality of the balanced graph. Two approaches have been tested and are depicted in the following, after talking about the interpretation of the new artificial workload-metric.

The workload-metric can be associated with popularity, because it refers to high usage. A path of high popularity should be chosen less often than without the workload-metric. So, hopefully, updating the workload-metric dependent on the current workload-metric reveals the final, balanced state of the network eventually. You could suggest, that a street's capacity should be considered as well, because large streets like motorways can support more vehicles than a street in a city and thus should tolerate a higher popularity than smaller streets. A street-capacity could be approximated using a street's number of lanes or the respective street-type (e. g. motorways) and its length. Herewith, the workload-metric, penalizing popularity, would be corrected in favor of larger streets with (usually) higher speed-limit. That is exactly contraproductive, because the travel-time does already consider the capacity of streets with the speed-limit. In other words, the popularity-penalization should compensate the popularity of larger streets, that have a better travel-time. This wished effect would be disturbed by the consideration of capacity.

The first approach is inspired by a simpler numerical method for solving initial-value-problems called explicit Euler. Let old_i be the workload-metric (already normalized after G has been initialized) already stored in the graph and new_i be the new workload-metric (normalized by their mean, see Equation (3.3)).

$$old_{e,i+1} = old_{e,i} + (new_{e,i} - old_{e,i}) \cdot correction \tag{3.4}$$

In each balancing-iteration, after the workload-metric is updated (as in Equation (3.4)), every value of the workload-metric smaller than a predefined minimum value ϵ (e. g. 0.1) is set to ϵ. This removes the metrics of value 0.0, which improves the set of computed shortcuts when applying contraction-hierarchies and thus performance. In addition, the result isn't normalized by its mean in general, for which reason it is finally normalized.

$$\begin{aligned} old_{e,i+1} &\leftarrow max\left(\epsilon, old_{e,i+1}\right) \\ old_{e,i+1} &\leftarrow \frac{old_{e,i+1}}{mean(old_{i+1})} \end{aligned} \tag{3.5}$$

In fact, this approach works okay in smaller, overloaded networks like Isle of Man ($|V| \approx$ 50,000 and $|E| \approx$ 100,000 after parsing), but lacks in convergence. In the first update-iteration, popular paths are ignored as much as possible, before they are highly preferred in the second iteration, ignored in the third iteration and so on. This behaviour could be reduced by adding the *correction*, but larger networks (like the small German county Saarland, $|V| \approx$ 580,000 and $|E| \approx$ 1,160,000 after parsing) worsen the convergence. Further, adding *correction* leads to the need of tuning it, for which reason only the second approach is shown in the experiments in Chapter 4 on page 29.

Therefore, another approach, replacing the explicit Euler from Equation (3.4) on the preceding page, is presented in Equation (3.6), that does spread the workload well and within only two iterations.

$$old_{e,i+1} = \frac{i \cdot old_{e,i} + new_{e,i}}{i+1} \tag{3.6}$$

This result is treated as before, meaning Equations 3.5 are also applied. From Equation (3.6), both $old_{e,1} = new_{e,0}$ and $old_{e,2} = \frac{old_{e,1}+new_{e,1}}{2}$ hold before normalization. The interpretation of these is quite intuitive. The first iteration favors popular routes. After the first update, the workload-metric is just the normalized workload from the original, unbalanced graph. Hence the following iteration has an aversion to popular routes and tends to avoid them. This second iteration favors alternative routes and the outcoming workload completes the previously generated workload-metric. A third iteration has already a workload-metric with a balance between popular and alternative routes, so further iterations appear disruptive.

3.2 Details on implementation

The implementation has the goal to be efficient, easy to use and maintainable. For this reason, some important decisions were made and the most important ones are explained (from a theoretical point of view) in the following.

3.2.1 Configuration-files

Main concept of the implementation is the use of configuration-files. The whole parser, simulation and computation can be adjusted with these files. One big advantage are the parser-related settings to define the graphs nodes and edges. Both meta-data and metric-data is defined in the configuration-file. This allows the graph to be static, meaning its allocated memory doesn't change during runtime after the graph is built and shrinked to fit its needs. However, its values can be changed, which is necessary, since the balancing updates a metric.

3.2.2 Storing and accessing graphs

The underlying graph-structure is implemented as adjacency-array with several extensions. The adjacency-array is explained in Section 2.1. This data-structure fits well, because it reduces redundancy by optimizing the graph's memory-consumption. At the same time, accessing the graph can be done in constant or logarithmic time (depending on access via indices or via ids) through indexing-logic. The indexing-logic from adjacency-arrays has been extended by a mapping to support forward-, backward- and shortcut-edges in the graph, which is necessary for the bidirectional-Dijkstra.

First, the graph from preliminaries is extended by metrics. For every edge in E, a vector of floats is stored. To formalize this, the cost-function is slightly redefined (and duplicated edges are being removed) to have edge-costs being associated as metrics. Let $C \subset E \times \mathbb{R}_+^d$ be the set of metrics and c be the bijective function $c : E \to C$. With this definition, $C[i_E]$ can be associated as the cost-vector of the edge E at edge-index i_E, where C is stored as two-dimensional array of floats.

The vertices V are an array of vertex-ids sorted in ascending order. This is required by the adjacency-array, but also allows the search for nodes by their ids in logarithmic time. However, this is only needed for users outside the graph, because inside, the graph uses only vertex-indices.

A backward-graph is a graph, whose edges are reversed according to the original graph. So a backward-graph maps edges (s, d) to the reversed edge (d, s). Since this graph is used for bidirectional routing-queries, it has to support the access to both forward-edges and backward-edges. The backward-edges could be stored accordingly to the forward-edges, meaning storing the source-indices S in addition to the destination-indices D. The occuring issue with this is the sort-order of vertices V. All vertices are sorted by their source-id to match the offset-array O_D for the forward-edges and for D. Respectively, all vertices should be sorted by their destination-id to match the offset-array O_S for the backward-edges and for S. This would need a copy of all vertices and an edge-index i_E wouldn't point to the same edge in S and D, leading to two access-implementations doing basically the same offset-mapping. The metric-array C could be accessed via i_E only by Dijkstra's forward-search, not the backward-search, unless a mapping from backward-edge-indices to forward-edge-indices is built (or C is also stored twice). Therefore, this issue is solved by using this very mapping, such that an edge-index i_E belongs to $S[i_E]$, $D[i_E]$, $C[i_E]$ independent of the edge's orientation and the graph can be stored without data-redundancy. The decision towards no data-redundancy implies consistency, which is important for the metric-update. Further, it improves maintainability and usage, because accessing the forward-edges and backward-edges can be done with the same accessor simply using different arrays. The indices-mappings are illustrated in Table 3.1 and explained afterwards with two example-queries. The graph from Section 2.1 contains only bidirectional edges, thus the edge (s, d) with $(V[s], V[d]) = (1, 2)$ is removed to make the graph containing unidirectional edges.

$i_E = i_{fwd}$	Storing data with respect to forward-edges											
V	id_0	id_1		id_2			id_3			id_4		-
S	0	1	1	2	2	2	3	3	3	4	4	-
D	1	0	3	1	3	4	1	2	4	2	3	-
C	a	b	c	d	e	f	g	h	i	j	k	-
i_{fwd}	0	1	2	3	4	5	6	7	8	9	10	-
$map_{bwd} : i_{fwd} \mapsto i_{bwd}$	1	0	6	2	7	9	3	4	10	5	8	-
$O_D : i_V \mapsto i_{fwd}$	0	1		3			6			9		11

$i_E = i_{bwd}$	Storing data with respect to backward-edges											
V	id_0	id_1			id_2		id_3			id_4		-
D_{bwd}	0	1	1	1	2	2	3	3	3	4	4	-
S_{bwd}	1	0	2	3	3	4	1	2	4	2	3	-
C_{bwd}	b	a	d	g	h	j	c	e	k	f	i	-
$map_{fwd} : i_{bwd} \mapsto i_{fwd}$	1	0	3	6	7	9	2	4	10	5	8	-
i_{bwd}	0	1	2	3	4	5	6	7	8	9	10	-
$O_S : i_V \mapsto i_{bwd}$	0	1			4		6			9		11

Table 3.1: This table shows one graph stored with respect to forward-edges above and with respect to backward-edges below. Note that a backward-graph of a graph has just the edges being reversed, so $(s, d) \mapsto (d, s)$. The more intuitive way is storing the graph and necessary indices forwardly. That's why the forward graph's notation is used as default (e. g. S is used instead of S_{fwd} according to S_{bwd}). However, both offset-arrays O_D and O_S and map_{fwd} are needed and hence each table is needed.
The tables highlight important values of the two example-queries from the text.

Two example-queries are explained in the following, assuming the graph being stored with respect to forward edges. In general, leaving edges of vertex $V[s]$ or incoming edges of a vertex $V[d]$ can be determined by following mappings (referring to Table 3.1):

$$\begin{aligned}
\mathit{offset} &\in \{O_D[s], O_D[s]+1, \ldots, O_D[s+1]-1\} \\
d_{\mathit{offset}} &= D\left[\mathit{identity}_{fwd}[\mathit{offset}]\right] \\
s &\mapsto \left\{\left(s, d_{\mathit{offset}}\right)\right\} \\
\\
\mathit{offset} &\in \{O_S[d], O_S[d]+1, \ldots, O_S[d+1]-1\} \\
s_{\mathit{offset}} &= S\left[\mathit{map}_{fwd}[\mathit{offset}]\right] \\
d &\mapsto \left\{\left(s_{\mathit{offset}}, d\right)\right\}
\end{aligned} \tag{3.7}$$

The mapping map_{fwd} can be computed by sorting all edges according to the forward-graph (first source-id, than destination-id) before sorting all edges again according to the backward-graph (first destination-id, than source-id). In between, the edge-positions from the first sorted edges are linked to the edges, such that they get reordered by sorting a second time. After sorting the second time, the remembered positions become the mapping map_{fwd}.

In the examples, one query is asking for all leaving edges starting in $V[s = 4] = id_4$ and one is asking for all incoming edges ending in $V[d = 1] = id_1$. The choice of $d = 1$ is handy, because $(2, 1) \in E \wedge (1, 2) \notin E$ holds, which might help with understanding the examples. Important values in these examples are highlighted in the tables.

To get all leaving edges starting in $V[s = 4] = id_4$, the offset-array O_D is used. All forward-indices within $\{O_D[s], O_D[s] + 1, \ldots, O_D[s + 1] - 1\}$ refer to the leaving edges. Since $O_D[4] = 9$ and $O_D[5] = 11$, all leaving-edges are at positions $\{9, 10\}$ with respect to forward-edges. Because the graph is stored with respect to forward-edges, these resulting positions are already the edge-indices i_E of the leaving-edges. The Table 3.1 still marks the respective values 9 and 10 in the forward-table. Thus, all leaving-edges are $\{(4, D[9] = 2), (4, D[10] = 3)\} = \{(4, 2), (4, 3)\}$.

Getting all incoming edges ending in $V[d = 1] = id_1$ works similar and uses the offset-array O_S respectively. All backward-indices within $\{O_S[d], O_S[d] + 1, \ldots, O_S[d + 1] - 1\}$ refer to the incoming edges. The difference here is the outcome from O_S. Due to $O_S[1] = 1$ and $O_S[2] = 4$, all incoming-edges are at positions $\{1, 2, 3\}$ with respect to backward-edges, but the graph is stored with respect to forward-edges. Therefore, the indices are not matching and the resulting positions $\{1, 2, 3\}$ need to be mapped to the respective (forward-) edge-indices i_E. For this, map_{fwd} is used.

$$\begin{aligned} map_{fwd}[1] &= 0 \Rightarrow S[0] = 0 \\ map_{fwd}[2] &= 3 \Rightarrow S[3] = 2 \\ map_{fwd}[3] &= 6 \Rightarrow S[6] = 3 \end{aligned} \tag{3.8}$$

Thus, all incoming-edges are $\{(0, 1), (2, 1), (3, 1)\}$.

3.2.3 Storing and accessing shortcuts in graphs

Shortcuts have metrics and source-/destination-vertices as well, so they are full edges and can be added to the current data-structure. However, due to construction of a contracted graph, a shortcut replaces two edges, which may be shortcuts. This functionality is missing in the graph's data-structure yet. A first intuition may be extending edges by a tuple of two edge-indices. Implementing this causes much redundancy, because normal edges would need a placeholder for their tuple. The street-networks from the experiments in Chapter 4 have less than $|E|$ shortcuts added. So after adding all shortcuts, more than half of all edges will have unused allocated memory of two indices. For example, Saarland from the experiments contains over 1,000,000 edges. If these placeholders would be used, each of these million edges would have two unused indices. Assuming one index needs 64 b, this sums up to 16 MB of unused memory. This number scales with the graph's size, so Germany with over 100,000,000 edges would have already 1.6 GB of completely unused, but allocated memory.

Therefore, since edge-indices i_E can be used for all edge-data (S, D, C), this whole indexing-procedure from Section 3.2.2 can be and is used in this thesis for storing and accessing shortcuts without such placeholders. Like the offsets O_D and O_S, another offset-array is created. Let this new offset-array be called O_L (L for link, since S is already used). The significant difference between O_D and O_S on one side and O_L on the other side is, that O_L contains an offset-value per edge, whereas O_D and O_S contain offset-values per vertex. Lining all shortcut-tuples of edge-indices up according to the forward-edges' order in the graph, this results in the data-array L. This array L behaves exactly similar as the array S or D. So, despite the indexing-mapping, the shortcut-indices of an edge-index i_E can be found in L between $O_L\,[i_E]$ (including) and $O_L\,[i_E + 1]$ (excluding). A shortcut can be determined by comparing neighbouring values in O_L. If $O_L\,[i_E] = O_L\,[i_E + 1]$ holds, the edge at i_E is no shortcut.

When analyzing the memory-usage, this method has improved the other approach of storing tuples in the experiments. Let m be the number of edges without shortcuts and m_L the number of shortcuts in the graph. Previously, the number of needed indices was $2 \cdot (m + m_L)$. The offset-procedure stores $2 \cdot m_L$ replaced edge-indices and $(m + m_L)$ offset-indices, resulting in $m + m_L + 2 \cdot m_L$. As long as $m_L < m$ holds, the offset-procedure saves memory, which was the case in all experimental graphs.

3.2.4 Multithreading

The implementation of searching for paths in parallel has brought a huge performance-boost. Optimal paths of different s-t-pairs may have different costs and thus need a different amount of runtime. This holds especially for REPR, where longer paths lead to more alternative paths. Therefore, the number of s-t-pairs is not distributed evenly over all threads. In fact, there is one single master-thread pulling some s-t-pairs from the user-provided set of s-t-pairs and sends this work-package to one of several worker-threads. Every worker-thread has its own instance of the routing-algorithm, searches for the optimum paths, and returns a simple list of all occurred edge-indices i_E to the master-thread. The master-thread counts the workloads for each edge and pulls the next work-package until all s-t-pairs are processed.

With this procedure, the workload and not the number of s-t-pairs is distributed evenly. The communication-overhead when using smaller work-package-sizes is negligible compared to idle worker-threads, that wait for the last few worker-threads to finish their larger work-package.

4 Experiments

All past chapters refer to the theory and name some theoretical implementation-details, all to verify the developed procedure. However, this chapter presents some results to validate, that the spread of found paths really improves on maps of different scale.

4.1 The testing-machine and simulation-setup

Firstly, the used machine runs an AMD Ryzen 7 3700X 8-Core Processor (3.6 GHz base and 4.4 GHz max-boost) and has 32 GB of RAM built in, although only a few GB are needed (highly dependent on the map and number of threads). The operation-system is Arch Linux [x86_64] with the kernel 5.7.9-arch1-1 installed. The used code for this thesis can be found in [Par20] and is implemented in Rust. Python is used for the visualization. Internally, this repository uses [Bar18] (written in C++) as git-submodule for the contraction-hierarchies and [Bar19] for memory-management of the convex hull. The latter is basically a Rust-wrapper for CGAL (C++).

Graphs for street-networks are downloaded from [Ope17]. Experiments run on the graph for Isle of Man (from March 2020) and on the graph for the German county Saarland (from July 2020). Considered metrics are travel-distance (all paths tolerated) and travel-time (40 % tolerance on Isle of Man, 25 % on Saarland). A tolerance of 25 % refers to a worst-case-scenario of 15 min more travel-time for originally 60 min, which seems to be a good tradeoff. However, a tolerance of 40 %, which is used for Isle of Man, refers to a worst-case of additional 24 min for originally 60 min. According to a simple Google-maps-query, one of the paths from one end of the island to the other takes $\approx$ 45 min. Hence, typical queries won't suffer that much. For routing, two sets of 10,000 s-t-pairs are chosen u. a. r. from each map. One set is for balancing on the particular map, the other set is for evaluating the balanced graph afterwards. This reduces the evaluation-results' dependence on the balancing. Both sets are created in a way, such that a path can be found for every s-t-pair in the set.

Two routing-algorithms, namely REPR on one side and Dijkstra with personalized routing on the other side, are used for the simulations. Due to balancing and evaluation each use one routing-algorithm independent from each other, four simulation-scenarios occur:

- Both, balancing and evaluating are done with Dijkstra. This scenario has the best performance of all scenarios, because REPR calls Dijkstra internally multiple times. On the other hand, REPR is already constructed to find a subset of pareto-optimal paths for same s-t-pairs and leads to a better spread. Most important, this case with using only Dijkstra doesn't guarantee a user-provided tolerance for found alternative paths. In general, Dijkstra finds the optimum path, but here, personalized routing is used. Hence, paths, that are optimal for some α, can be arbitrary bad for a certain metric, when compared to an optimal path of an α favoring only this metric (e. g. $(0, 1, 0, 0)^T$).

- Both balancing and evaluating use REPR. This combination is expected to deliver the highest variety of alternative paths, but suffers from the highest runtime compared to simply running a Dijkstra-query. Besides that, REPR guarantees the user-provided tolerance for found paths due to its construction (see Equation (3.2) on page 22).
- Two mixes of the previous cases are remaining, where balancing is done with one routing-algorithm and evaluation with the other. Especially the case is interesting, where Dijkstra is used for balancing and REPR for evaluation. The evaluation can be seen as representing a user-perspective. Hence, using Dijkstra with personalized routing for balancing and REPR for evaluation keeps the guarantee for user-provided tolerances while keeping the balancing-runtime reduced. As shown in the experiments, the results are not better than using only REPR, but still satisfactory.

4.2 Balancing Isle of Man

Isle of Man is a small island next to Great Britain. The graph consists of around 50,000 nodes and 100,000 edges after parsing. The REPR uses a tolerance of 40 % for travel-time and 99.8 % of the graph's nodes are contracted.

4.2.1 Meta-data from balancing

The simulation is not benchmarked in detail, but the coarse results in Table 4.1 on the next page and Table 4.3 on page 36 might help for getting an impression. Four threads are used. Please note, that the git-submodule for contracting graphs is rebuild in the first iteration without the new workload-metric and build again in the following iteration with the new workload-metric being considered. For Isle of Man, these build-times take much longer (still just $\approx$1 min) than the actual contraction, for which reason below numbers discard the build-time.

Besides a first impression of the needed runtime, Table 4.1 on the next page already explains the differences in performance. The needed time for the balancing, basically just path-searches, is much smaller with Dijkstra than with REPR, because REPR calls Dijkstra multiple times. The number of found paths increases with iteration-number when using REPR, which is desirable. Because the standard-deviations are greater than the respective means (and the number of found paths is at least one for every s-t-pair due to selection of s-t-pairs), any small number of found paths requires a high number (or several respective high numbers) on the other side of the mean. This indicates, that the means are probably not caused just by some outliers. Further, the number of covered edges in $|E|$ is higher after balancing and improves more with REPR than with Dijkstra.

4.2.2 Actual results from balancing

The balancing consists of two workload-metric-updates, but all together, workload-plots of three iterations can be found on the following pages. Every workload leads to a new metric-update, but the last iteration doesn't update the workload-metric, so the last workloads show the final spread of paths over the street-network, after the final workload-metric-update is adapted. Despite randomness and different sets of s-t-pairs, these final workloads should be identical to those from evaluation in

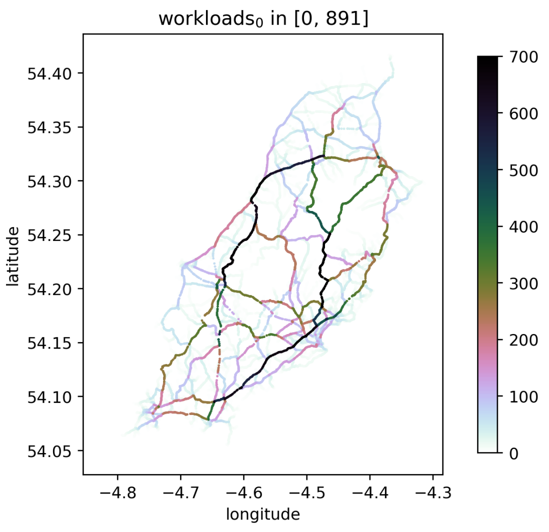
Isle of Man 4 threads	Balanced with Dijkstra Iteration 0	 Iteration 1	 Iteration 2	Balanced with REPR Iteration 0	 Iteration 1	 Iteration 2
Average query-time before contraction	$\approx$ 7 ms	$\approx$ 7 ms	$\approx$ 7 ms	$\approx$ 7 ms	$\approx$ 7 ms	$\approx$ 7 ms
Time for contracting (99.8 % of $\|V\|$)	$\approx$ 5 s	$\approx$ 5 s	$\approx$ 5 s	$\approx$ 5 s	$\approx$ 5 s	$\approx$ 5 s
Average speed-up through contraction	∞	∞	∞	∞	∞	∞
Time for balancing	$\approx$ 1 s	$\approx$ 1 s	$\approx$ 1 s	$\approx$ 2 s	$\approx$ 25 s	$\approx$ 30 s
Number of found paths ($\mu \pm \sigma$)	1 ± 0	1 ± 0	1 ± 0	$\approx 3 \pm 2$	$\approx 15 \pm 24$	$\approx 19 \pm 24$
Maximum workload	891	1,620	783	852	795	722
Number of unique edges (in 1,000)	$\approx$ 82.1	$\approx$ 81.7	$\approx$ 82.2	$\approx$ 82.8	$\approx$ 84.2	$\approx$ 83.9

Table 4.1: Isle of Man. An overview (but no detailled benchmarks) of balancing-performance with four threads on Isle of Man. Here, 99.8 % of all nodes are contracted. In Table 4.3 describing the performance during evaluation, the graph isn't contracted to show the contraction's impact on performance. The query-times before contraction refer to Dijkstra-queries from the contraction-tool, so are independent of the balancing's routing-algorithm. The maximum workloads are just copied from the plots. The number of found paths (≥ 1) is provided with a standard-deviation to indicate, that the mean is not caused by some outliers. The number of unique edges stands for the actual number of edges in $|E|$ with a workload greater than zero. The set of s-t-pairs contains 10,000 s-t-pairs.

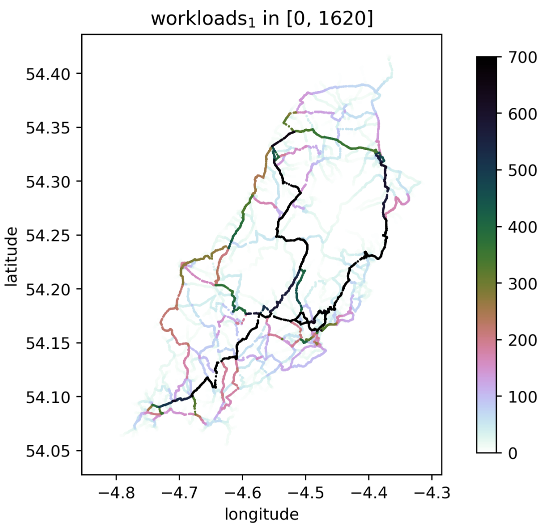
Section 4.2.3. The workloads in Figure 4.1a and Figure 4.1c on the following page lead to the first metric-update. In fact, due to construction (see Equation (3.6) on page 23), the first workloads are the actual new workload-metric after the update (after normalization by their mean).

The initial plots Figure 4.1a and Figure 4.1c are quite identical. This is not an issue of REPR, but of the considered metrics. Both, travel-distance and travel-time, don't bring that different paths without any artificial metric. Although, some more paths occur on the graph showing workloads from REPR, leading to an insignificantly smaller maximum workload here.

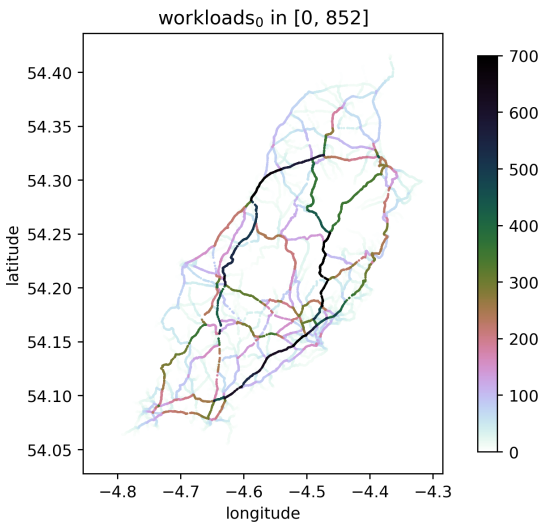
The plots of the second iteration, which is the first iteration including an artificial metric, can be seen in Figure 4.1b and Figure 4.1d on the following page. As mentioned in Section 3.1.3, the first update doesn't lead to a spreaded workload yet. Especially Dijkstra results in a maximum workload, that is twice as high as before. Nonetheless, in Figure 4.1b (after first update balancing with Dijkstra), there are still some new found routes in between (e. g. in the north of the map). In this second iteration, the previously updated workload-metric makes Dijkstra avoid popular routes more or less completely. But on such a small map, quite many of these popular routes use the fastest streets in the street-network. The same holds for REPR, but REPR does still consider the tolerance of 40 %, thus the found paths tend to stick with previously found routes, as you can see in Figure 4.1d. The maximum workload of paths from REPR is slightly better than in the initial iteration. Even some



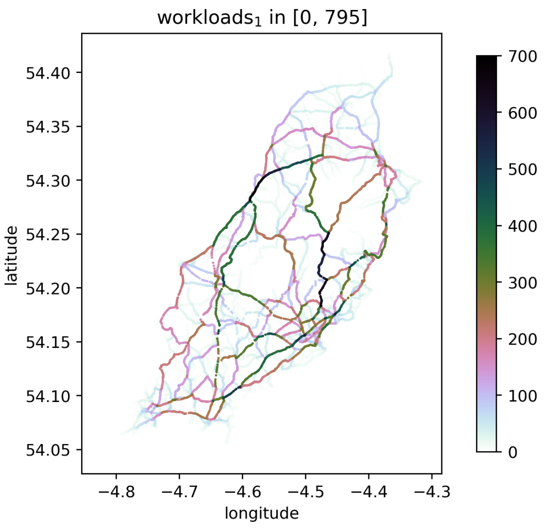
(a) Initial workloads with Dijkstra



(b) Workloads after first update with Dijkstra



(c) Initial workloads with REPR



(d) Workloads after first update with REPR

Figure 4.1: Isle of Man. The first row shows the balancing with Dijkstra, the second row shows the balancing with REPR.
The left side is showing the workloads in the first iteration, right before the first workload-metric-update. Thus, only the existing metrics are used for routing.
The right side is showing the workloads after the first and in the second iteration, right before the final workload-metric-update. Dijkstra has a much worse maximum workload, but all color-bars are kept identical.

more spread is visible in Figure 4.1d. However, despite some exceptions, the spread is still quite identical to the initial one, when compared to the globally best spreaded workloads after balancing, shown in Figure 4.2 on the next page.

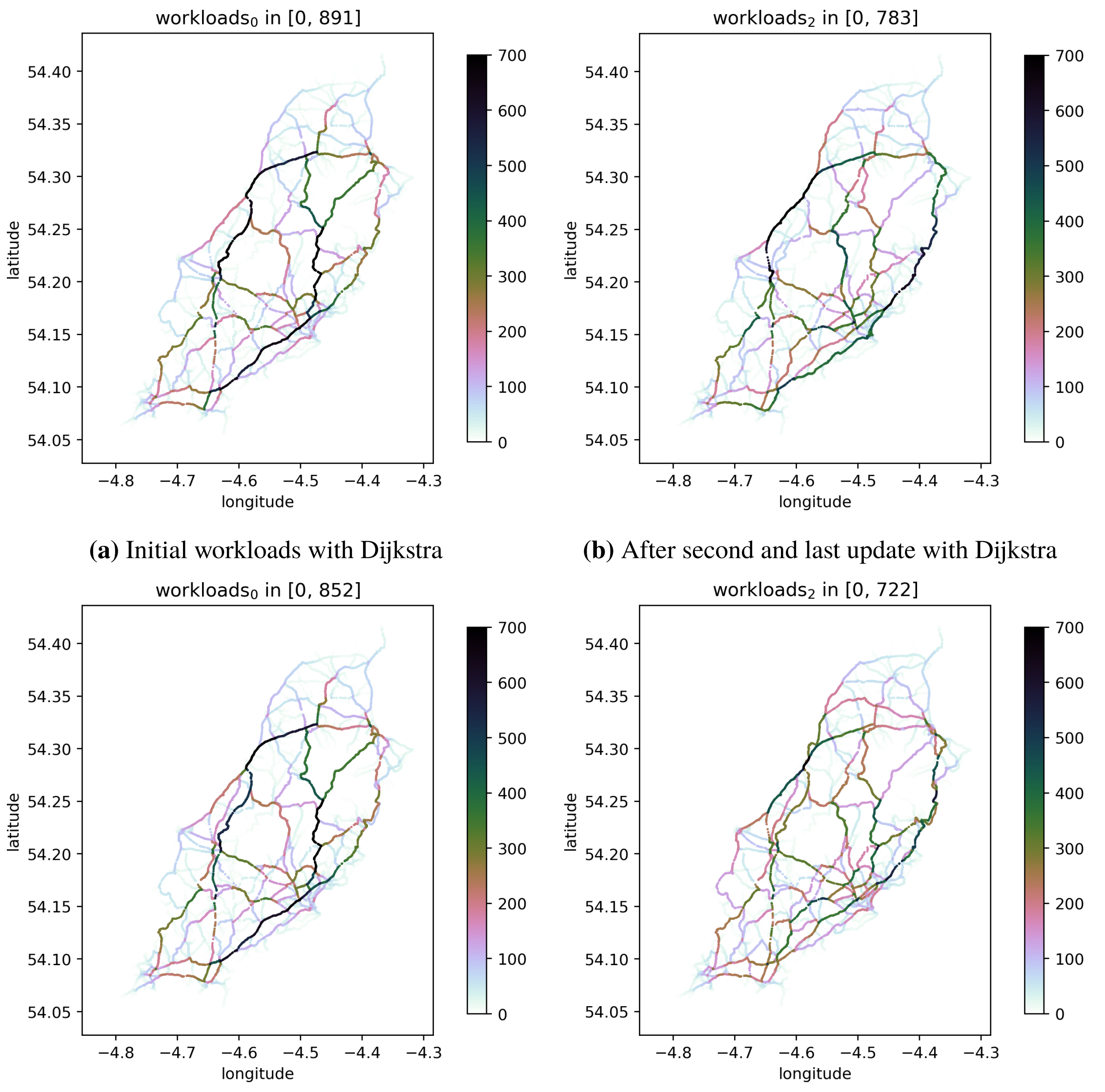
With the final workloads after balancing with REPR, shown in Figure 4.2d on the following page, it is noticable, how the clear hotspots from the initial plots flew into multiple new paths in between. At the same time, these hotspots don't vanish at all, so most popular paths are still popular, while the overall situation improved. This is good, because most popular means more optimal with respect to travel-time. In opposite to that, after balancing with Dijkstra, shown in Figure 4.2b, the workload looks less distributed than in Figure 4.2d (final workload after balancing with REPR). This doesn't necessarily imply, that the graph's balance is that much worse. In fact, when evaluating the Dijkstra-balanced graph with REPR (see Figure 4.5b on page 38), the resulting workloads actually look more spreaded than the final workloads from balancing with Dijkstra (see Figure 4.2b), noticable by the different colors in the middle of the respective plots. A comparison of the number of found paths confirms on that in Table 4.3 on page 36 (evaluating-performance).

Lastly, Chapter 3 claims and explains, that two iterations are intuitively a good choice, because popular and alternative routes are balanced here. Running Isle of Man for more than two iterations supports this statement. The relevant information about the iterations are provided in Table 4.2. After the second metric-update, the maximum-workloads are worse or equal (slightly better here and there) and number of unique edges are quite similar. But most important, the number of found paths is much worse than before, so indeed less paths are found with more iterations. Besides that, the balancing takes, of course, much more time with more iterations, while the workloads after two and after ten metric-updates (see Figure 4.3b on page 35) are more or less identical.

Isle of Man	Balanced with REPR					
4 threads	Iteration 2	Iteration 3	Iteration 4	Iteration 5	Iteration 6	Iteration 7
Number of found paths ($\mu \pm \sigma$)	$\approx 19 \pm 24$	13 ± 18	$\approx 13 \pm 17$	$\approx 15 \pm 20$	14 ± 16	$\approx 15 \pm 19$
Maximum workload	722	789	746	729	760	712
Number of unique edges (in 1,000)	≈ 83.9	≈ 84.1	≈ 84.1	≈ 84.0	≈ 84.1	≈ 83.8

	Iteration 2	Iteration 8	Iteration 9	Iteration 10
Number of found paths ($\mu \pm \sigma$)	19 ± 24	14 ± 18	$\approx 15 \pm 19$	$\approx 16 \pm 22$
Maximum workload	722	838	710	796
Number of unique edges (in 1,000)	≈ 83.9	≈ 83.9	≈ 83.9	≈ 83.9

Table 4.2: Isle of Man. A comparison of iterations to show, that two metric-updates are indeed a good choice. Here, iteration 2 refers to the respective iteration 2 in Table 4.1. The number of found paths (≥ 1) is provided with a standard-deviation to indicate, that the mean is not caused by some outliers. The number of unique edges stands for the actual number of edges in $|E|$ with a workload greater than zero.



(a) Initial workloads with Dijkstra

(b) After second and last update with Dijkstra

(c) Initial workloads with REPR

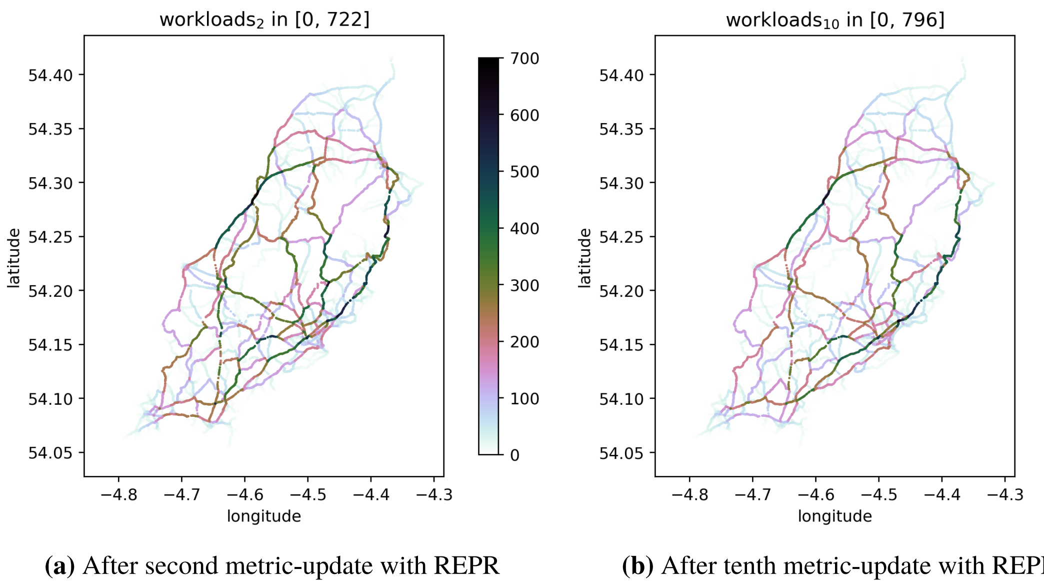
(d) After second and last update with REPR

Figure 4.2: Isle of Man. These plots show the workloads during balancing. The graph is balanced with Dijkstra and REPR. The first row shows the balancing with Dijkstra, whereas the second row shows the balancing with REPR. The left side shows the workloads before the first workload-metric-update, whereas the right side shows the workloads after two workload-metric-updates. Furthermore, despite randomness and the different s-t-pairs from evaluation, the plots on the right side correspond to the evaluation-plots in Figure 4.5 on page 38. The used metrics besides the new workload-metric are travel-distance and travel-time. When using REPR, each path's travel-time tolerates a maximum of 40 % worse than its optimum.

4.2.3 Comparison of evaluating different scenarios

The evaluation is basically recreating the workloads from last balancing-iteration with switching routing-algorithms and another set of s-t-pairs. So paths for all s-t-pairs are searched and the street-network's workload is counted again, which is shown in the evaluation-related plots in the following. A new set of u. a. r. chosen 10,000 s-t-pairs is determined to reduce the evaluation-results' dependence on the balancing.

In Table 4.3 on the next page (evaluation-related performance), similar information as in Table 4.1 on page 31 (balancing-related performance) is noted and relevant parts from previous tables are copied. To show the huge performance-impact of using contraction-hierarchies, the evaluation is done without contracting the balanced graph. The initial number of unique edges with the new set for Dijkstra is $\approx$ 82,000, for REPR it is $\approx$ 82,800. Besides the performance, especially the average number of found paths is interesting. When balancing and evaluating with REPR, the average number of found paths is $\approx 19 \pm 25$. Evaluating with REPR, but balancing with Dijkstra still brings out $\approx 12 \pm 17$ paths on average. Note, that the actual number of found paths is at least one due to selection of s-t-pairs, and that this indicates again, that the means are probably not caused by some outliers. These two means show, that doing balancing with Dijkstra is quite good, and confirm, that the similar looking initial workloads from balancing (in Figure 4.1 on page 32) and the initial workloads from evaluating (in Figure 4.4 on page 37) are indeed an issue of the metrics, not of REPR.



(a) After second metric-update with REPR **(b)** After tenth metric-update with REPR

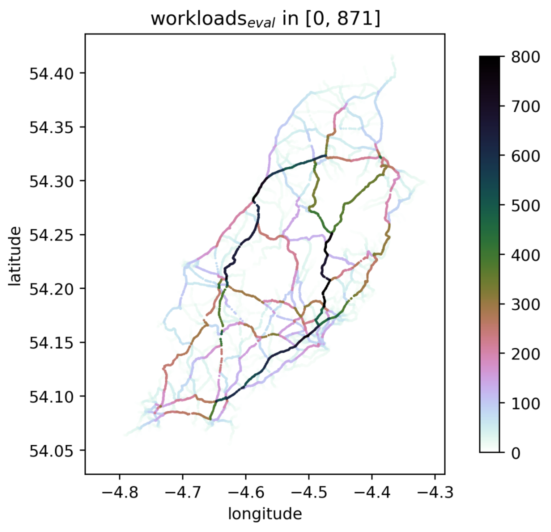
Figure 4.3: Isle of Man. These plots show the workloads during balancing. The graph is balanced with REPR. The left plot shows the workloads after two metric-updates, whereas the right plot shows the workloads after ten metric-updates. The plots are more or less identical, but Table 4.2 shows clear differences. The used metrics besides the new workload-metric are travel-distance and travel-time. When using REPR, each path's travel-time tolerates a maximum of 40 % worse than its optimum.

Isle of Man	Balanced with Dijkstra			Balanced with REPR		
4 threads	Iteration 2	Evaluated with Dijkstra	Evaluated with REPR	Iteration 2	Evaluated with Dijkstra	Evaluated with REPR
Time for searching all paths	≈ 1 s	≈ 13 s	≈ 8 min	≈ 30 s	≈ 12 s	≈ 14 min
Number of found paths ($\mu \pm \sigma$)	1 ± 0	1 ± 0	$\approx 12 \pm 17$	≈ 18	1 ± 0	$\approx 19 \pm 25$
Maximum workload	783	789	789	722	689	750
Number of unique edges (in 1,000)	≈ 82.2	≈ 82.1	≈ 83.9	≈ 83.9	≈ 82.4	≈ 83.9

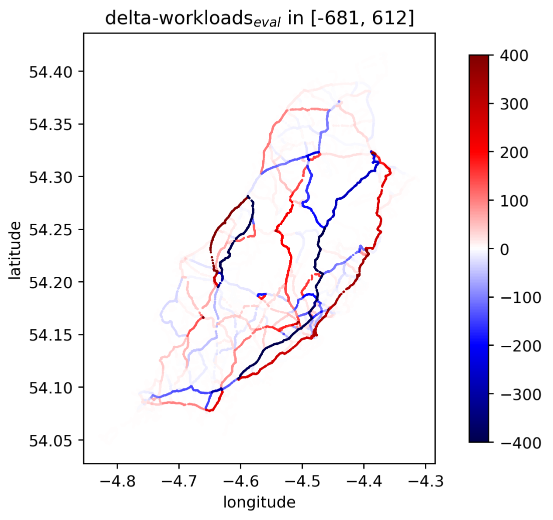
Table 4.3: Isle of Man. A comparison (but no detailled benchmarks) of evaluating-performance with the balancing-performance (from Table 4.1 on page 31), using again four threads on Isle of Man. Here, iteration 2 refers to the respective iteration 2 in Table 4.1. This time, when evaluating, the graph isn't contracted (for comparison) and hence the runtime is much longer. The number of found paths (≥ 1) is provided with a standard-deviation to indicate, that the mean is not caused by some outliers. The number of unique edges stands for the actual number of edges in $|E|$ with a workload greater than zero. The initial number of unique edges with the new evaluation-set (also 10,000 s-t-pairs) for Dijkstra is ≈ 82,000, for REPR it is ≈ 82,800.

The workloads, that are generated by ignoring the new workload-metric and that use the new evaluation-set of s-t-pairs, are shown in Figure 4.4 on the next page. They are looking more or less identical to the initial workloads from balancing in Figure 4.1 on page 32, but are still provided for completeness. Additionally, Figure 4.4 shows the differences between evaluated workloads without and with the new workload-metric. Only the differences of balancing and evaluating Dijkstra and balancing and evaluating REPR are presented, to show the most extreme distinctions. Both plots reduce the workload on previously popular routes and increase the workload on less popular routes. However, the REPR-plot showing the workload differences through evaluating (in Figure 4.4d) contains more spots for little increases than the respective Dijkstra-plot (in Figure 4.4b) does, which concentrates more on the previously popular streets. This consolidates in the maximum changes, which are clearly higher (around ±600) for Dijkstra than for REPR (around ±400).

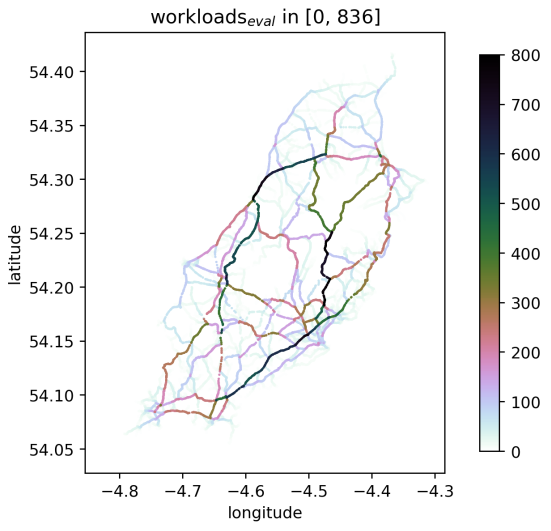
The probably most interesting plots are depicted in Figure 4.5 on page 38. Note, that the previous plots from Figure 4.2 on page 34 (showing the balancing's last iterations' workloads) are the same cases than the evaluation-related plots in Figure 4.5a and Figure 4.5d, just with a different set of s-t-pairs. Both plots, that show the workloads of the balanced graph evaluated, are clearly more spreaded than the initial evaluation-plots from Figure 4.4, because the maximum workloads are lower and the most popular routes are flown into other routes according to the color-shifts in the plots. Just using Dijkstra (Figure 4.5a) still seems to be a tradeoff towards performance, because popular routes are basically just moved to another place. On the other hand, using Dijkstra for queries after balancing with REPR surprisingly reduces the maximum workload the most. Especially in the north of the map, this combination (Figure 4.5c) shows a few more routes than using just Dijkstra (Figure 4.5a). Still an issue, when using Dijkstra with personalized routing, is the not guaranteed tolerance for found paths' cost. That's why the most important combinations are those, where REPR is used as evaluation-query (or user-query respectively), because these two combinations



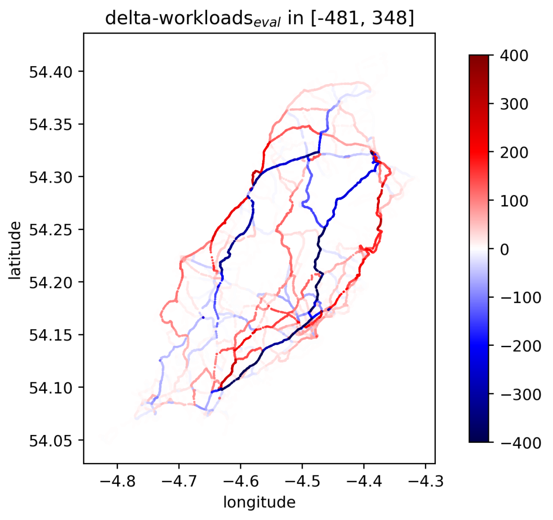
(a) Initial workloads with Dijkstra



(b) Difference between initial workloads (left) and final workloads (Figure 4.5a) using Dijkstra for both balancing and evaluating



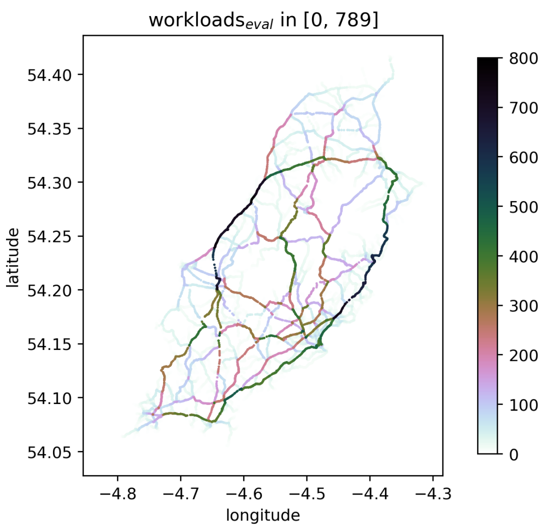
(c) Initial workloads with REPR



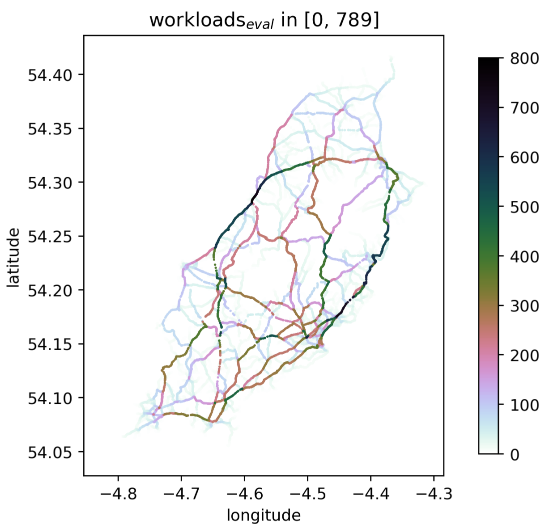
(d) Difference between initial workloads (left) and final workloads (Figure 4.5d) using REPR for both balancing and evaluating

Figure 4.4: Isle of Man. These plots show the balanced graph evaluated with Dijkstra and REPR. The first row shows the evaluation with Dijkstra after balancing with Dijkstra, whereas the second row shows the evaluation with REPR after balancing with REPR. The left side shows the evaluation without the new workload-metric. These plots look quite identical to Figure 4.1 on page 32 (initial plots of balancing), but since the evaluation uses a different set of s-t-pairs, these evaluation-plots here are required for the sake of completeness. The right side shows the workload-changes from the initial workloads here to the respective final workloads in Figure 4.5a and Figure 4.5d. The used metrics are travel-distance and travel-time, and additionally the new workload-metric for the differences. When using REPR, each path's travel-time tolerates a maximum of 40 % worse than its optimum.

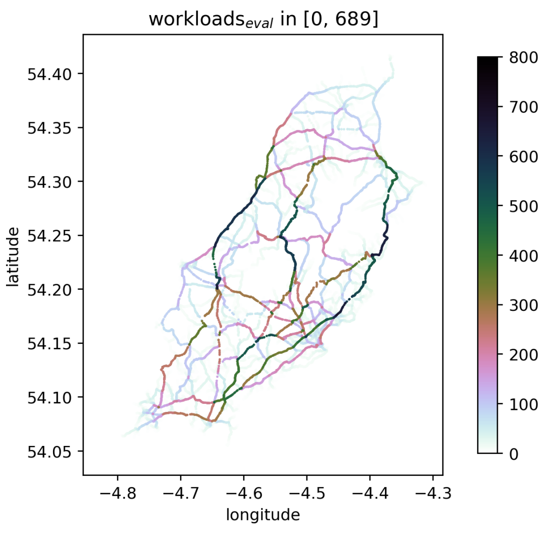
consider the optimum path with respect to travel-time. Both balancing with Dijkstra (Figure 4.5b) and balancing with REPR (Figure 4.5d) show more distribution in the middle of the map, noticable at respectively colored spots (e. g. around longitude of −4.6). Though, only using REPR clearly delivers the best spreaded results, again noticable by more reduced workloads overall, but at a much higher performance-cost during balancing.



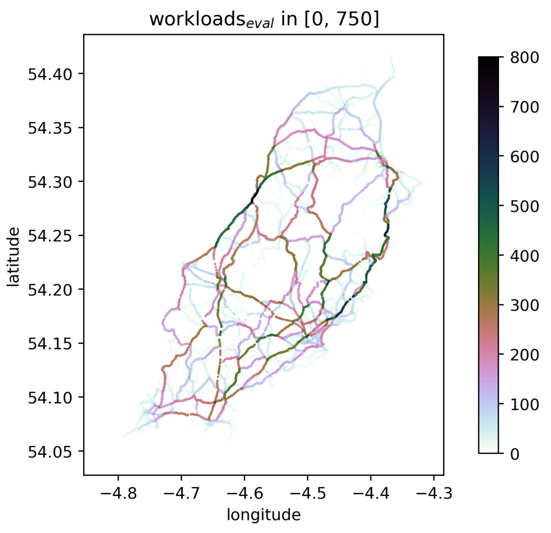
(a) Balanced and evaluated with Dijkstra



(b) Balanced with Dijkstra and evaluated with REPR



(c) Balanced with REPR and evaluated with Dijkstra



(d) Balanced and evaluated with REPR

Figure 4.5: Isle of Man. These plots show the balanced graph evaluated with Dijkstra and REPR. The first row shows the evaluation after balancing with Dijkstra, whereas the second row shows the evaluation after balancing with REPR. The left side shows the evaluation with Dijkstra, whereas the right side shows the evaluation with REPR. The used metrics besides the new workload-metric are travel-distance and travel-time. When using REPR, each path's travel-time tolerates a maximum of 40 % worse than its optimum.

4.3 Balancing Saarland

Saarland is a small German county. The graph consists of around 580,000 nodes and 1,160,000 edges after parsing, which is around ten times larger than Isle of Man. The REPR uses a tolerance of 25 % for travel-time (in contrary to 40 % with Isle of Man) and also 99.8 % of the graph's nodes are contracted.

With Isle of Man, every aspect is talked about, but trying out a larger map has revealed the lack of convergence with the first Euler-approach (see Equation (3.4)) and lead to the second and working averaging-approach (see Equation (3.6)). Therefore, it is helpful and necessary to run larger maps, for which reason Saarland is also presented shortly in the following. Besides that, the new plots show some new behaviour on less loaded maps.

First of, the balancing-performance is shown in Table 4.4 on the next page and the evaluating-performance (contracted this time) in Table 4.5 on the following page, similar to Isle of Man before. As described earlier, the performance-advantage of Dijkstra over REPR shows up here, but with much more impact in a larger map. Doing the whole balancing with Dijkstra takes a few minutes, while using REPR a little more than an hour. The reason is clear, since REPR does find much more alternative paths, which are all not worse than 25 % of the optimum path with respect to travel-time. This opens the question to reduce the tolerance (e. g. 20 % or even 15 %). Much more remarkable is the number of found paths from the evaluation-performance (Table 4.5) for balancing with Dijkstra and evaluating (executing user-queries) with REPR. This combination finds even more alternative paths than using just REPR.

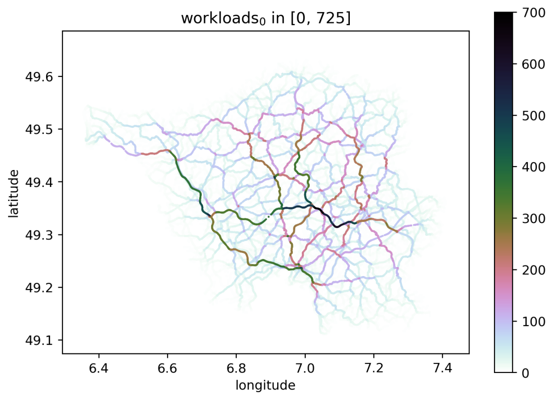
When looking at the evaluation-plots in Figure 4.7 on page 42, it is obvious, that the Dijkstra-plots (Figure 4.7a and Figure 4.7c) have much lower maximum-workload (under 500) than the REPR-plots (more than 700), which can be seen in the final balancing-plot of Dijkstra (Figure 4.6b) as well. Here should be noted, that Dijkstra doesn't consider the tolerance at all and such a barely loaded street-network offers plenty of space to spread. However, the two results using REPR for evaluation-queries (or user-queries respectively) are quite identical, but the one balancing with Dijkstra (Figure 4.7b) needs a fraction of the runtime of balancing with REPR (Figure 4.7d). Even similar improvements and new routes can be seen on both plots, especially between longitude of 7.0 and 7.2 around the latitude of 49.35. So, in addition to Isle of Man, using Dijkstra for balancing and REPR for user-queries seems to be a good compromise.

Saarland 15 threads	Balanced with Dijkstra Iteration 0	Iteration 1	Iteration 2	Balanced with REPR Iteration 0	Iteration 1	Iteration 2
Average query-time before contraction	≈ 115 ms	≈ 130 ms	≈ 125 ms	≈ 124 ms	≈ 120 ms	≈ 121 ms
Time for contracting (99.8 % of $\lvert V \rvert$)	< 30 s	≈ 1 min	≈ 1 min	< 30 s	≈ 1 min	≈ 1 min
Average speed-up through contraction	≈ 135	≈ 144	≈ 37	≈ 136	≈ 36	≈ 34
Time for balancing	≈ 22 s	≈ 24 s	≈ 25 s	≈ 5 min	≈ 21 min	≈ 47 min
Number of found paths ($\mu \pm \sigma$)	1 ± 0	1 ± 0	1 ± 0	6 ± 3	12 ± 25	28 ± 33
Maximum workload	725	658	442	914	971	774
Number of unique edges (in 1,000)	≈ 474	≈ 509	≈ 481	≈ 488	≈ 493	≈ 502

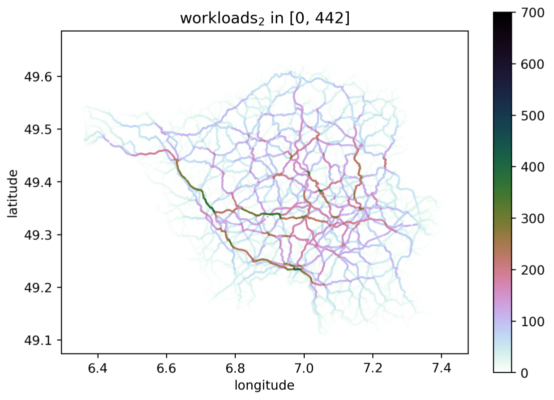
Table 4.4: Saarland. An overview (but no detailled benchmarks) of balancing-performance with 15 threads on Saarland. Here, also 99.8 % of all nodes are contracted. The query-times before contraction refer to Dijkstra-queries from the contraction-tool, so independent of the balancing's routing-algorithm. The maximum workloads are just copied from the plots. The number of found paths (≥ 1) is provided with a standard-deviation to indicate, that the mean is not caused by some outliers. The number of unique edges stands for the actual number of edges in $|E|$ with a workload greater than zero. The set of s-t-pairs contains 10,000 s-t-pairs.

Saarland 15 threads	Balanced with Dijkstra Iteration 2	Evaluated with Dijkstra	Evaluated with REPR	Balanced with REPR Iteration 2	Evaluated with Dijkstra	Evaluated with REPR
Time for searching all paths	≈ 25 s	< 2 min	≈ 54 min	≈ 47 min	≈ 28 s	≈ 47 min
Number of found paths ($\mu \pm \sigma$)	1 ± 0	1 ± 0	≈ 32 ± 36	≈ 28 ± 33	1 ± 0	≈ 28 ± 33
Maximum workload	442	485	770	774	469	789
Number of unique edges (in 1,000)	≈ 481	≈ 479	≈ 501	≈ 502	≈ 487	≈ 502

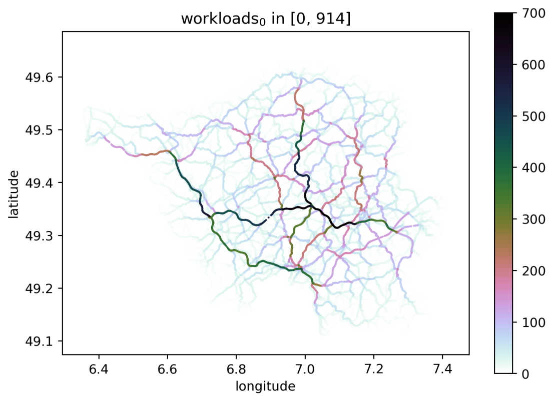
Table 4.5: Saarland. A comparison (but no detailled benchmarks) of evaluating-performance with the balancing-performance (from Table 4.4), using again 15 threads on Saarland. Here, iteration 2 refers to the respective iteration 2 in Table 4.4. This time, when evaluating, the graph is contracted (in opposite to previously with Isle of Man). The number of found paths (≥ 1) is provided with a standard-deviation to indicate, that the mean is not caused by some outliers. The number of unique edges stands for the actual number of edges in $|E|$ with a workload greater than zero. The initial number of unique edges with the new evaluation-set (also 10,000 s-t-pairs) for Dijkstra is ≈ 471,000, for REPR it is ≈ 487,000.



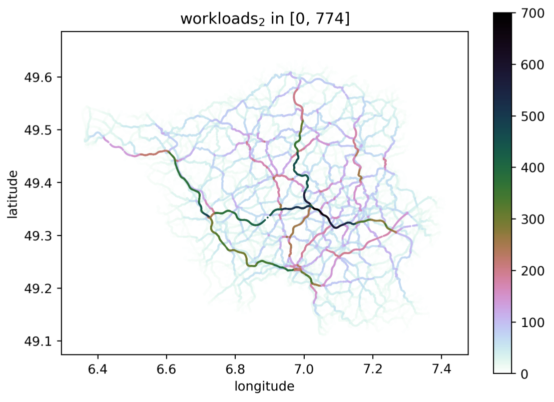
(a) Initial workloads with Dijkstra



(b) After second and last update with Dijkstra

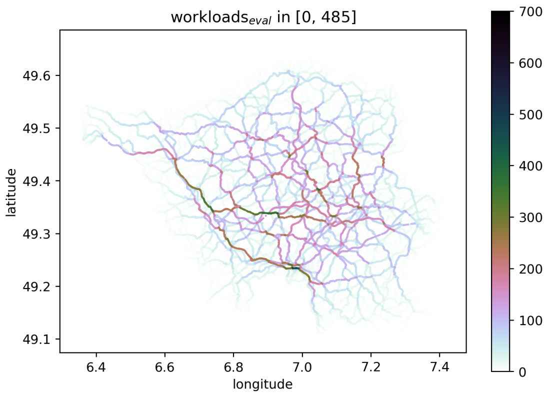


(c) Initial workloads with REPR

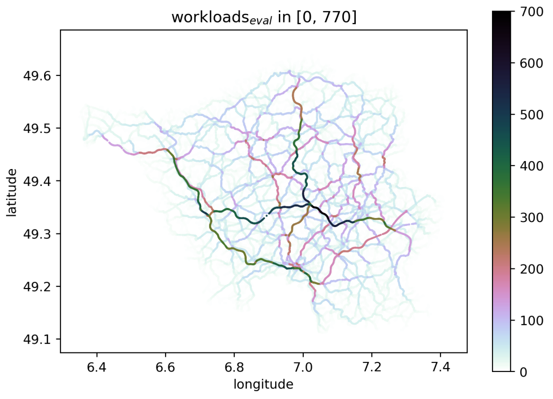


(d) After second and last update with REPR

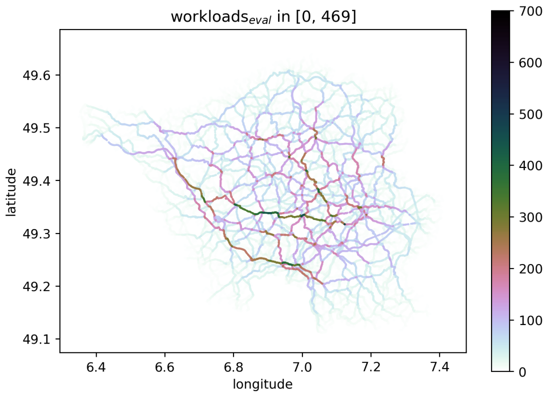
Figure 4.6: Saarland. These plots show the workloads during balancing. The graph is balanced with Dijkstra and REPR. The first row shows the balancing with Dijkstra, whereas the second row shows the balancing with REPR. The left side shows the workloads before the first workload-metric-update, whereas the right side shows the workloads after two workload-metric-updates. Furthermore, despite randomness and the different s-t-pairs from evaluation, the plots on the right side correspond to the evaluation-plots in Figure 4.7 on the next page. The used metrics besides the new workload-metric are travel-distance and travel-time. When using REPR, each path's travel-time tolerates a maximum of 25 % worse than its optimum.



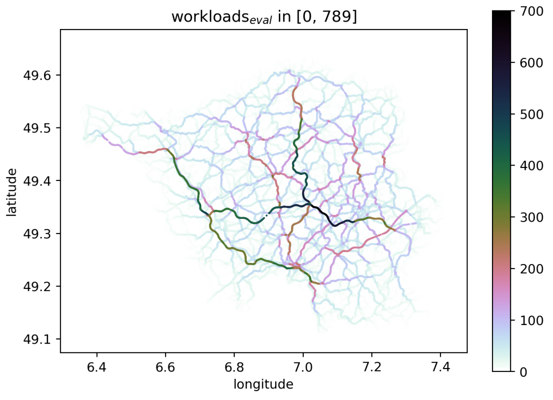
(a) Balanced and evaluated with Dijkstra



(b) Balanced with Dijkstra and evaluated with REPR



(c) Balanced with REPR and evaluated with Dijkstra



(d) Balanced and evaluated with REPR

Figure 4.7: Saarland. These plots show the balanced graph evaluated with Dijkstra and REPR. The first row shows the evaluation after balancing with Dijkstra, whereas the second row shows the evaluation after balancing with REPR. The left side shows the evaluation with Dijkstra, whereas the right side shows the evaluation with REPR. The used metrics besides the new workload-metric are travel-distance and travel-time. When using REPR, each path's travel-time tolerates a maximum of 25 % worse than its optimum.

5 Conclusion and Outlook

5.1 Conclusion

In this thesis, existing procedures have been combined, such that a new penalizing metric is computed for provided graphs. This new metric compensates the popularity of routes' edges in the graph, because these higher popularities causes to overload a graph's underlying network, when many routes are computed. This is achieved with the help of an existing procedure from [BFS19], that interpretes optimum paths found by Dijkstra in the underlying cost-space. With help of this procedure, several alternative routes for a provided pair of source and destination are enumerated. Because of the new metric, the number of found alternative routes is increased significantly while a user-provided tolerance (with respect to travel-time) holds.

Advantages of the presented process to create the new metric is the intuitive idea behind it. While other approaches for finding alternative routes lack in performance, parameter-tuning, complexity or diversity of found alternative routes, the method in this thesis can be summarized in two iterations. The first iteration updates the graph's new metric by favoring popular routes. The second and last iteration considers this preference negatively and updates the graph's new metric, respectively trying to avoid these popular routes. In the end, the average of both updates results in a penalizing metric, that can be used in existing routing-algorithms for graphs of multidimensional metrics. The new metric leads to more spreaded routes, as shown in this thesis with street-networks from OpenStreetMap.

5.2 Future work

Remaining issues with the procedure are mainly performance-related, although preprocessing-methods as contraction-hierarchies are already used to speed the route-queries significantly up.

One performance-improvement would bring a better choice of sources and destinations for the user-provided set of s-t-pairs. Yet, sources and destinations are chosen u. a. r. from the graph's vertices. This requires a high number of s-t-pairs to actually overload the graph. Heuristics, as one described in [BLM+14] to approximate the population based on OpenStreetMap-data, may help to weight the vertices in a more realistic way, which is usually less u. a. r. when thinking of rush-hour-scenarios.

One slighter performance-improvement might be the flattening of shortcuts, that are created by the contraction-hierarchies, right after all found routes are collected, not right after a route is found. Another improvement refers the implemented graph-parser, that assumes, that drivers are okay with choosing even dirt tracks for their paths, leading to more edges in the graph. As the tolerance for travel-time holds, this shouldn't affect the spread a lot, but the routing-algorithms (especially the contraction-hierarchies) are much affected by the number of edges. Further, the contraction-hierarchies contracts just almost all vertices, which takes much less time than actually creating the new metric. A higher contraction-rate to reduce the creation-time might improve the total runtime here.

Although the results from the shown experiments are satisfying, only one metric is created. Maybe, the results would be even better with several artificial penalization-metrics being created, since the averaging-approach behaves accordingly to this idea. In addition, once a graph is balanced, the graph's balancing-progress is more or less fixed. Here, it would be more practical to have the graph being balanced right after each s-t-pair is processed. Despite the metric-update, that relies on the mean over all workloads, a remaining issue would be the adaption of contraction-hierarchies. This could be done by a replacement-strategy, where the last contracted graph is used for answering incoming queries and a copy of it is further balanced in the meanwhile. At some point, the contracted graph, that is answering incoming queries, is replaced by the new graph, after this is contracted. This would fit better for use-cases in the real world, especially with large capacities, as Google can provide. Google additionally has enough user-queries to produce valid workloads.

Another aspect is the influence of crossroads on travel-time. The tolerance holds for travel-time, which is based solely on speed-limits and distances, not on behaviour or circumstances at crossroads. Here, the hop-distance would be important, too. On the other hand, the idea behind all of this is the reduction of occuring traffic-jams. Probably, circumstances at crossroads might be the better option than standing in a traffic-jam.

Bibliography

[Bar09] D. Barth. *The bright side of sitting in traffic: Crowdsourcing road congestion data*. Website. 2009. URL: https://googleblog.blogspot.com/2009/08/bright-side-of-sitting-in-traffic.html (cit. on p. 12).

[Bar18] F. Barth. *GitHub: multi-ch-constructor*. 2018. URL: https://github.com/lesstat/multi-ch-constructor (cit. on p. 29).

[Bar19] F. Barth. *GitHub: nd-triangulation*. 2019. URL: https://github.com/lesstat/nd-triangulation (cit. on p. 29).

[BDGS11] R. Bader, J. Dees, R. Geisberger, P. Sanders. "Alternative Route Graphs in Road Networks". In: *Theory and Practice of Algorithms in (Computer) Systems*. Ed. by A. Marchetti-Spaccamela, M. Segal. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 21–32. ISBN: 978-3-642-19754-3. DOI: 10.1007/978-3-642-19754-3_5 (cit. on p. 12).

[BF19] F. Barth, S. Funke. "Alternative Routes for Next Generation Traffic Shaping". In: *Proceedings of the 12th ACM SIGSPATIAL International Workshop on Computational Transportation Science - IWCTS'19* (2019). DOI: 10.1145/3357000.3366141 (cit. on pp. 12, 13, 18).

[BFS19] F. Barth, S. Funke, S. S. Storandt. "Alternative Multicriteria Routes". In: *2019 Proceedings of the Twenty-First Workshop on Algorithm Engineering and Experiments (ALENEX)*. 2019 Proceedings of the Twenty-First Workshop on Algorithm Engineering and Experiments (ALENEX). Society for Industrial and Applied Mathematics, 2019, pp. 66–80. DOI: 10.1137/1.9781611975499.6 (cit. on pp. 12, 18, 21, 22, 43).

[BLM+14] M. Bakillah, B. Liang, A. Mobasheri, J. Jokar Arsanjani, A. Zipf. "Fine resolution population mapping using OpenStreetMap points-of-interest". In: *International Journal of Geographical Information Science* 28 (2014). DOI: 10.1080/13658816.2014.909045 (cit. on pp. 21, 43).

[CBGL15] T. Chondrogiannis, P. Bouros, J. Gamper, U. Leser. "Alternative Routing: k-Shortest Paths with Limited Overlap". In: Nov. 2015. DOI: 10.1145/2820783.2820858 (cit. on p. 12).

[Dij59] E. W. Dijkstra. "A note on two problems in connexion with graphs". In: *Numerische Mathematik* 1.1 (1959), pp. 269–271. DOI: 10.1007/BF01386390 (cit. on p. 17).

[DW09] D. Delling, D. Wagner. "Pareto Paths with SHARC". In: *Experimental Algorithms*. Ed. by J. Vahrenhold. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 125–136. ISBN: 978-3-642-02011-7. DOI: 10.1007/978-3-642-02011-7_13 (cit. on p. 12).

[Epp99] D. Eppstein. "Finding the k Shortest Paths". In: *SIAM J. Comput.* 28.2 (1999), pp. 652–673. ISSN: 0097-5397. DOI: 10.1137/S0097539795290477 (cit. on p. 12).

[FLS17] S. Funke, S. Laue, S. Storandt. “Personal Routes with High-Dimensional Costs and Dynamic Approximation Guarantees”. In: *16th International Symposium on Experimental Algorithms (SEA 2017)*. Ed. by C. S. Iliopoulos, S. P. Pissis, S. J. Puglisi, R. Raman. Vol. 75. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, 18:1–18:13. ISBN: 978-3-95977-036-1. DOI: `10.4230/LIPIcs.SEA.2017.18` (cit. on pp. 12, 16).

[GKS10] R. Geisberger, M. Kobitzsch, P. Sanders. “Route Planning with Flexible Objective Functions”. In: *Proceedings of the Twelfth Workshop on Algorithm Engineering and Experiments, ALENEX 2010, Austin, Texas, USA, January 16, 2010. Ed.: G. E. Blelloch*. SIAM, Philadelphia (PA), 2010, pp. 124–137. ISBN: 978-0-898719-31-4 (cit. on p. 12).

[GSSD08] R. Geisberger, P. Sanders, D. Schultes, D. Delling. “Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks”. In: *Experimental Algorithms*. Ed. by C. C. McGeoch. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 319–333. ISBN: 978-3-540-68552-4. DOI: `10.1007/978-3-540-68552-4_24` (cit. on p. 12).

[INR19] INRIX. *INRIX 2019 Global Traffic Scoreboard*. 2019. URL: `https://inrix.com/scorecard/` (cit. on p. 11).

[MS08] K. Mehlhorn, P. Sanders. “Algorithms and Data Structures: The Basic Toolbox”. In: *Algorithms and Data Structures: The Basic Toolbox* (2008), pp. 168–169. DOI: `10.1007/978-3-540-77978-0` (cit. on p. 15).

[Ope17] OpenStreetMap contributors. *Planet dump retrieved from https://planet.osm.org*. 2017. URL: `https://www.openstreetmap.org` (cit. on p. 29).

[Par20] D. Parga Cacheiro. *GitHub: osmgraphing*. 2020. URL: `https://github.com/dominicparga/osmgraphing` (cit. on p. 29).

[Wor20] K. Worldpanel. *Android vs. iOS - Smartphone OS sales market share evolution*. 2020. URL: `https://www.kantarworldpanel.com/global/smartphone-os-market-share/` (cit. on p. 12).

All links were last followed on August 11, 2020.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

place, date, signature