Personalized Route Planning in Road Networks

Stefan Funke
Universität Stuttgart
Germany 70569 Stuttgart, Germany
funke@fmi.uni-stuttgart.de

Sabine Storandt
University of Freiburg
79110 Freiburg, Germany
storandt@cs.uni-freiburg.de

ABSTRACT

Computing shortest paths in road networks with millions of nodes and edges is challenging on its own. In the last few years, several preprocessing-based acceleration techniques have been developed to enable query answering orders of magnitudes faster than a plain Dijkstra computation. But most of these techniques work only if the metric which determines the optimal path is static or rarely changes. In contrast to that, we aim at answering *personalized route planning queries*. Here, every single query comes with a specification of its very own metric. This increases the combinatorial complexity of the problem significantly. We develop new preprocessing schemes that allow for real-time personalized route planning in huge road networks while keeping the memory footprint of the preprocessed data and subsequent queries small.

Categories and Subject Descriptors

E.1 [**Data Structures**]: Graphs and Networks

General Terms

Algorithms

Keywords

Route Planning, Personalization

1. INTRODUCTION

Common route planners based on Spatial Network Data Bases (SNDBs), as Google Maps or Bing Maps, allow to specify starting location and destination, and compute the optimal route between them very efficiently. Here, the optimal route typically means the quickest or shortest route. Such route planning engines neglect, though, that the notion of optimality differs vastly from person to person. Some users indeed only care about reaching their destination as quickly as possible, but others want to minimize fuel consumption or prefer scenic routes. Often a fair trade-off between several such preferences leads to the envisioned result. Moreover, the individual driving style differs (e.g. speeder or leisurely driver) and other aspects like the type of the car can have an effect on certain metrics as travel time, turn costs or energy consumption. Hence we define the problem of *personalized route planning* as follows:

Given a street network $G(V, E)$, we have for each edge $e \in E$ a d-dimensional non-negative cost vector $c(e) \in \mathbb{R}^d$ (e.g. c_1 corresponding to travel time, c_2 to gas price, and so on). A query consists not only of source and target $s, t \in V$ but also of non-negative weights $\alpha_1, \alpha_2, \cdots, \alpha_d$, where the α_i are either determined by factors like vehicle characteristics, fuel prices or express in general the importance of edge cost component i for the user. The goal is to compute the path p from s to t in G which minimizes $\sum_{e \in p} \alpha^T c(e)$.

Figure 1 illustrates a set of routes based on varying choices of α (for $d = 10$ natural metrics). We observe that different α lead to significantly differing routes. Hence individual user preferences should not be ignored in route planning engines.

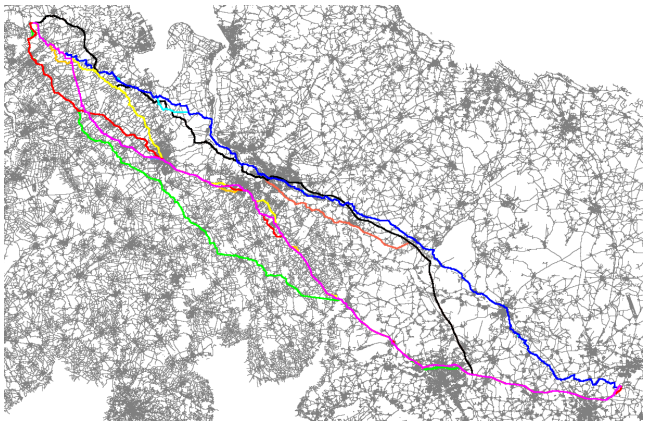

Figure 1: Example for personalized routes with fixed source and target in a road network. Every color represents a user with his indivual route preferences.

For the classical route planning problem of finding the minimum cost path considering *single static* cost values, various speed-up techniques have been developed (like *contraction hierarchies (CH)* [15], *transit nodes* [3] or *hub labels* [2]) that improve on the Dijkstra baseline significantly. In fact, they reduce query times from several seconds (on continental

SIGSPATIAL'15, November 03-06, 2015, Bellevue, WA, USA
 ISBN 978-1-4503-3967-4/15/11...$15.00
DOI: http://dx.doi.org/10.1145/2820783.2820830

sized street networks) to milliseconds or even microseconds by making use of precomputed auxiliary data.

But not all edge costs are as unshakable as distances. When considering e.g. travel times, the edge cost values vary significantly over the day (e.g. due to the morning rush hour), and unpredictable events like accidents can lead to streets being totally blocked. To take care of this, time-dependent variants of speed-up schemes have been developed (e.g. time-dependent CH [4]), which work on edge cost functions rather than scalar values. Furthermore, fast update procedures have been designed which adapt the auxiliary data to changed cost values rather than recomputing this data from scratch (see e.g. [15] or [5]). But these update methods are only applicable if few edge costs are modified at a time. If a significant fraction of all edges in the network change their costs, updates are typically too slow. Therefore a new type of speed-up frameworks has been developed, which allows for *customization*, i.e. switching the whole metric on demand. Incarnations of this idea are the *customizable route planning approach (CRP)* [6] and *customizable contraction hierarchies (CCH)* [9]. They allow to update e.g. the travel time on all edges in a large network in less than 15 seconds on a single core (with good parallelizability, though). This is surely sufficient for incorporating real-time traffic information. Subsequent queries can then be answered in the order of milliseconds. In [9] it was claimed that customization times are also in the order of milliseconds, but a later published erratum [10] corrected the unit to seconds. So the problem with those methods in context of our application is, that despite customization times being small, they are comparable to the runtimes of a plain Dijkstra computation in the network. So if every query comes with its own metric preference, one can run Dijkstra instead of customizing and answering the query subsequently. Therefore, new methods have to be developed to enable answering personalized route planning queries significantly faster than the Dijkstra baseline. As high-dimensional cost vectors increase the size of the input data already significantly, special care has to be taken to compute concise auxiliary data.

1.1 Related Work

Many SNDB-based route planning engines feature the computation of alternative routes. So besides the shortest route from A to B, two or three other routes are presented along to the user in order to provide him with some freedom of choice. But here, the goal is rather to find alternatives which are sufficiently distinguishable from the shortest route but still close-to-optimal for the considered, *static* metric [1]. Personal preferences are not incorporated.

Another form of flexibility is to forbid certain kinds of roads in a query, as roads with toll costs or interstates [14] (e.g. Google Maps and Bing Maps allow such queries). But such restrictions are yes or no decisions, no trade-offs are considered. We can easily include restrictions in our personalized model, by defining suitable edge metrics and setting the respective α_i to ∞ in a query.

In [13], the personalization problem was considered but only for $d = 2$; also α could not be chosen completely freely there. For this model a modified CH approach combined with *landmarks* [16] leads to a speed-up of three orders of magnitude over the Dijkstra baseline.

The problem of dealing with personalized queries for arbitrary dimensions d was considered first in [12]. There, a CH variant was introduced which takes care of all possible weighted metric combinations (i.e. arbitrary choices of α). For $d = 2$ a speed-up of about three orders of magnitude over the Dijkstra baseline was reported, and a speed-up about a factor of 150 to 200 for $d = 3$. While in theory the algorithms presented in this paper work for arbitrary dimension d, the experiments showed that query times increase notably with the dimension. But the main bottleneck for practical use in high dimensions (so large values of d) is the construction time of the auxiliary data. In fact, the preprocessing time was shown to grow exponentially in d, which limits its applicability.

In [11] up to 64 metrics were considered in the context of personalized route planning. The speed-up compared to the Dijkstra baseline reported there is about a factor 8, for fewer metrics up to 13. The algorithm is based upon constructing a k-Path Cover (k-PC) for G and augmenting the induced overlay graph with cost vectors. As noted in [11] this approach unfortunately has a prohibitive space consumption for practical use in very large networks. We will use the core idea behind k-PC to develop a general approach for tackling the personalized route planning problem. Moreover the algorithms developed in this paper will decrease the memory footprint of k-PC dramatically, which results in a much wider applicability.

Finally, in [8] it was stated that the highly tuned version of CRP described there can also be used for answering personalized queries. Indeed, customization times between 300 and 1200 milliseconds can be achieved with their approach (corresponding to a speed-up of 10 to 60 over Dijkstra). In a follow-up paper [7], the customization procedure was transferred to the GPU, allowing for a further speed-up. But all these results are based on heavy parallelization (12 CPU cores in [8] and an ASUS NVIDIA GTX Titan with 14 multiprocessing units, each with 192 cores, so 2688 cores in total in [7]); sequential times are still comparable to the runtime of a plain Dijkstra. Of course, having the ability to parallelize is an advantage, but especially in a client/server architecture it is not ideal if a single user query occupies that many cores (and more importantly *space* to temporarily store the result of the customization phase).

Our goal is to preprocess the graph in a way that sequential query answering is much faster than a Dijkstra computation but at the same time a query does not demand more space than a normal Dijkstra run for a single user.

1.2 Contribution

We provide the following new results and ideas on personalized route planning:

- Schemes for customizable route planning are impractical for personalized queries because customization takes too long. We design a new abstract framework that integrates customization partly in the preprocessing and partly in the query answering phase. Applying this framework, we can automatically turn customization approaches into personalization approaches.

- We identify as the key challenge of personalized route planning schemes the task of pruning sets of high-dimensional cost vectors efficiently. We develop several pruning algorithms that are used to reduce sets in an optimality preserving way. For k-PC, we can thereby decrease the space consumption of the auxiliary data

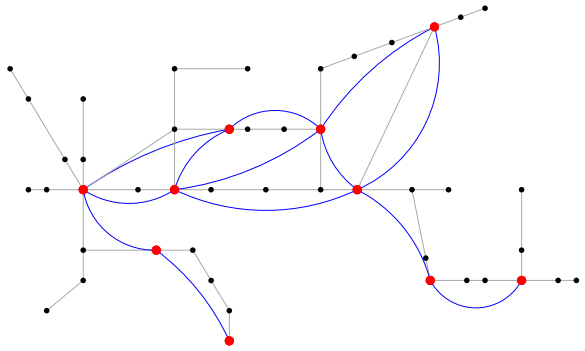

Figure 2: Example of a k-Path Cover for $k = 4$ (red nodes), and the induced overlay graph (blue edges).

to less than 5% of the numbers reported in [11].

- Our extensive experimental study proves that our new developed personalized route planning techniques are the first to achieve a speed-up of a factor of more than 50 over the Dijkstra baseline for high dimensions. At the same time, our approaches exhibit manageable preprocessing times and small memory footprints.

2. PERSONALIZATION WITH K-PC

In this section, we first review how k-Path Covers (k-PC) can be instrumented for personalized route planning. Subsequently, we design an abstract two-phase framework for personalized route planning based on the core ideas of k-PC. This framework will turn out to be very useful for developing new efficient personalization approaches, as elaborated in Section 3.

2.1 k-Path Covers

Following the definition in [11], a k-Path Cover on a graph $G(V, E)$ is a subset of the nodes $W \subseteq V$, such that for every simple path in G consisting of k nodes at least one of those nodes is contained in W, see Figure 2 for an example. Efficient computation of a concise cover is possible as described in [11] (e.g., less than two minutes for the road network of whole Germany and $k = 32$ on a single core).

k-Path Covers can be instrumented for personalized route planning as follows. First, an overlay graph is computed, which contains an edge between any two neighbors in the cover. Here, two cover nodes are neighbors if there exists a simple path between them in G not containing any other cover node (see again Figure 2 for an illustration). Then every edge (u, v) in the overlay graph is augmented with cost vectors. More concretely, for every simple path p from u to v without another cover node on it, the accumulated cost vector $\sum_{e \in p} c(e)$ is assigned to (u, v), see Figure 3 for an example. This completes the preprocessing phase.

In a query, first local Dijkstra computations are run from s and t (reversely) until all paths in the Dijkstra search tree contain at least one settled cover node. The remaining search between nodes settled in the runs from s and t is conducted in the overlay graph only. During the search edges are relaxed using α provided with the query. So every cost vector assigned to an overlay graph edge is multiplied with α and the minimum resulting scalar value defines the cost.

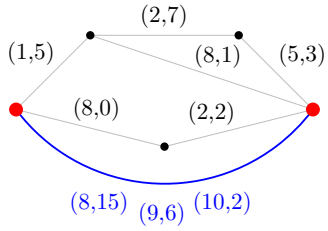

Figure 3: Example graph with two metrics ($d = 2$). Cover nodes are red, the overlay graph consisting of a single edge is blue. As there are three different simple paths from one cover node to the other, the overlay edge between them gets assigned three labels.

The query times are dominated by the search in the overlay graph, in particular by the number of edges and cost vectors assigned to these edges. These values also define the space consumption of k-PC. As the number of simple paths between two nodes can be exponential in the number of nodes in the network, assigning all cost vectors to every edge in the overlay graph becomes very space intensive already for rather small values of d and k. We will present methods to reduce the number of cost vectors without compromising optimality of the query result in Section 4.

2.2 Abstract Personalization Framework

The k-PC approach for personalized route planning consists of two phases, a preprocessing and a query phase.

1. In the preprocessing phase, first an overlay graph is constructed which considers only the topology of the road network. Up to this point no metric is involved. In this overlay graph, edges represent (sets of) simple paths in the original network. Then the second step of the preprocessing consists of assigning cost vectors to the edges in the overlay graph, one per corresponding simple path in the original network.

2. In the query phase, the precomputed overlay graph is used to find the optimal route, by relaxing overlay graph edges where possible instead of considering the respective complete paths in the original network.

We observe that these two phases are not tied to the specific overlay graph construction based on k-PC. Other metric-independent overlay graph construction schemes could be plugged in without invalidating the correctness. In the next section, we show that overlay graphs as used for customizable route planning serve our purpose very well.

3. FROM CUSTOMIZABLE TO PERSONALIZABLE

The two existing approaches for customization – customizable route planning (CRP) and customizable contraction hierarchies (CCH) – are both three phase approaches: In the first (preprocessing) phase, a metric-independent overlay graph is constructed. In the second phase, the actual customization phase, the graph and the auxiliary data are coated with a metric. The metric is given explicitly as one scalar value per edge. In the third phase, queries are answered by making use of the auxiliary data computed in the first two phases.

Of course, we could use customization techniques directly to solve the personalized route planning problem. For that purpose, we compute for given α the explicit edge costs for the original network and customize with those. Then the whole graph is personalized. But as already mentioned above, conventional customization takes considerably more time than query answering. As for personalized route planning customization has to be redone with every query, we cannot use customization approaches straightaway for real-time answers.

We now describe CCH and CRP in detail and point out some modifications which allow to trade customization time against query time, such that in combination for personalized queries better timings are achievable. Subsequently, we develop new personalizable variants of CCH and CRP that follow the two phase approach rather than the three phase approach – making those schemes much more suitable for personalized queries.

3.1 Customizable CH (CCH)

In the preprocessing phase of CCH [9], the original graph G is augmented with overlay edges (so called shortcuts) which span simple paths in the network. The basic operation to construct such shortcuts is node contraction. Here, a node v and its adjacent edges get removed from the graph, and edges between all former direct neighbors of v are inserted (avoiding self-loops and not inserting edges that are already present in the graph). These edges are the shortcuts. So a shortcut always spans two edges, with each of these being either an original edge or also a shortcut. After contracting all nodes in the network one-by-one, the shortcuts constructed in the process are added to the original graph G, resulting in a new CH-graph G'. The order in which the nodes are contracted is crucial for the number of shortcuts in the end. The goal is to find an ordering for which the resulting graph is sparse. We refer to the rank of a node in that order as the label of the node.

In the customization phase first all original edges are augmented with the new costs in a single sweep. To assign correct cost values to the shortcuts as well, they are sorted increasingly by the maximum node label that led to the insertion of the shortcut (or would have, if it has not already been there). Obviously, the costs of a shortcut can only be influenced by shortcuts with a lower rank in this ordering. Then the shortcuts are parsed in the respective order. For each shortcut, all node contraction steps that would have led to that particular shortcut insertion are considered. For every such contraction, the summed costs of the two edges that the shortcuts spans are the potential costs of the shortcut. Of course, in the end the minimum among all potential costs is chosen. Note, that the respective edge pairs can be stored during the contraction process to make the customization more efficient. In the original paper [9], a shortcut $\{u, w\}$ with a corresponding edge pair $\{u, v\}, \{v, w\}$ is called a lower triangle, because v has to have a smaller node label than u and w. So essentially one has to evaluate all lower triangles to get the correct cost value for the shortcut.

Queries are answered in the same manner as for conventional CH-graphs [15]. By construction, for every pair of nodes s, t there exists a shortest path in G' with the node labels only increasing at first, and then monotonously decreasing until t. Therefore, a shortest path can be found via a bi-directional Dijkstra computation; with the forward run from s considering only edges which point to higher node labels, and vice versa in the backward run from t.

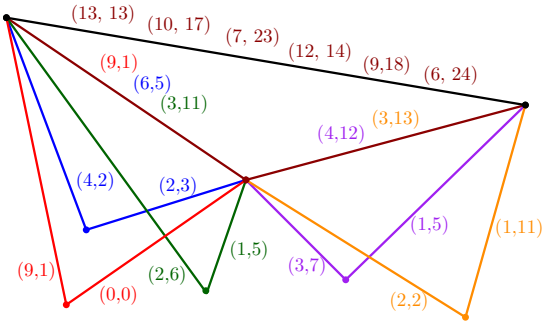

Figure 4: Assignment of cost vectors to shortcut edges for $d = 2$. Node heights in the image correspond to node labels. Each lower triangle has to be evaluated and the cost vectors have to be combined. The uppermost shortcut receives six vectors as a result of combining three cost vectors assigned to the left shortcut and two vectors assigned to the right shortcut.

Accelerating Customization. The customization times for CCH are determined by the number of shortcut edges in the overlay graph as well as the number of lower triangles which have to be evaluated to compute the correct edge costs. In a typical CH overlay graph construction, nodes that are contracted late (i.e. nodes with high labels) induce many shortcuts as due to earlier shortcut insertions they have a lot of direct neighbors. Also those lately inserted shortcuts often span many long simple paths in the original network, resulting in many lower triangles which have to be considered in customization.

Therefore, customization times can be improved by not contracting all nodes in the network but stopping the process at some point. This idea was already used for other CH-based applications where full contraction results in too high preprocessing time or space consumption (see e.g. [12]). The remaining uncontracted nodes are called the *core* of the graph. Optimal query answering can still be guaranteed when considering edges between core nodes in the bi-directional Dijkstra run as well. In the extreme scenario, where all nodes in the network are core nodes, we end up with zero shortcuts and a conventional bi-directional Dijkstra computation. So customization times are small but query times are high. With no core nodes, it is the other way around. So there should exist some point to stop the contraction process, such that customization times and query sum up to the minimal possible value. This yields the best trade-off point for answering personalized queries with CCH.

Personalizable Contraction Hierarchies (PCH). Let us now modify CCH to work in a two phase manner like the k-Path Cover based approach. That means in particular that we want to avoid the necessity of the customization phase. We will call the resulting scheme *personalizable contraction hierarchies (PCH)*.

The preprocessing phase starts just like for CCH with the construction of the overlay graph G'. But then, in addition, we augment the CH-graph with the d-dimensional cost vec-

tors, proceeding similarly to the former customization phase. So for every lower triangle corresponding to a shortcut, we combine the cost vectors of the respective edge pair to a new vector and assign this vector to the shortcut (see Figure 4 for a small example). While in the CCH setting, every shortcut receives exactly one value, the number of vectors assigned to a shortcut $\{u, w\}$ using PCH corresponds to the number of paths between u and w not containing a node with a label higher than the minimum of the labels of u and w.

Queries are also answered in the bi-directional way described above. The only difference is, that now the edge costs have to be determined during edge relaxation by multiplying α with every cost vector assigned to the edge and picking the minimum value.

Again, just like for the k-Path Cover approach, large numbers of cost vectors assigned to (shortcut) edges, increase the query time significantly and result in a huge space consumption for storing the augmented overlay graph.

3.2 Customizable Route Planning (CRP)

The preprocessing phase for CRP [8] starts by partitioning the graph into c cells of roughly the same size for some parameter c. The goal is to construct these cells such that the number of nodes at the border of the cells and the number of edges connecting different cells is small. Then between all pairs of border nodes of a single cell (with border nodes being adjacent to an edge connecting to another cell) overlay edges are inserted. To make customization and query answering more efficient later on, the partitioning approach is typically employed in a multi-layer fashion. So every cell is considered as a small graph on its own, and partitioning and overlay graph construction is recursively applied (until the cell size drops below a certain threshold).

Customization works bottom-up just like for CCH. After the original edges are augmented with new costs, the overlay edges of the cells on the lowest level are considered. Their costs can be computed via one-to-many Dijkstra runs (one for every border node) with the search being restricted to the specific cell. Cells on the next higher level use the overlay graphs of the cells below for this purpose, and so forth. In the more sophisticated variant of CRP described in [8], the authors use CH to construct the overlay graphs and compute the respective costs. In fact, they fix an ordering of the nodes in each cell a priori, and perform the contraction process on demand. With a fixed cost metric, a shortcut $\{u, w\}$ spanning $\{u, v\}$ and $\{v, w\}$ is only inserted if u, v, w is a shortest path from u to w.

Queries are answered by a bi-directional Dijkstra computation on the final overlay graph. Only cells containing s or t have to be inspected more closely, i.e. the runs start at a level where s or t are actually contained in the graph and then work their way up in the hierarchy of layers.

Accelerating Customization. In the CRP approach the number of generated overlay edges per cell is quadratic in the number of border nodes of this cell. All nodes being in a single partition, or every node forming a partition on its own, results both in zero overlay edges and no acceleration in query answering. Choosing the number of partitions c between 1 and $|V|$ leads to trade-offs between more overlay edges (i.e. higher customization times) and better query times. Again, like for CCH, there is a optimal trade-off when accumulated customization and query times are minimal.

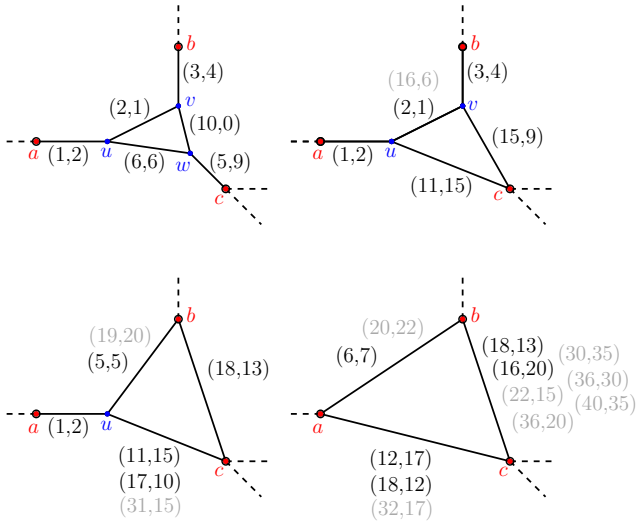

Figure 5: Overlay graph construction and cost vector assignment for a cell. The red nodes a, b, c are the border nodes of the cell. The inner nodes are contracted in the order w, v, u.

Personalizable Route Planning (PRP). In order to improve the performance of CRP for personalized queries, we perform similar changes to the construction scheme as for CCH. After the computation of the metric-independent auxiliary data (multi-level partitioning and overlay graph construction), we assign cost vectors to the overlay edges in a bottom-up fashion. Hereby, in a cell, the cost vectors for an overlay edge $\{u, v\}$ correspond to all simple paths between u and v in the cell. In congruency with [8], we could also use CH inside the cells. In fact, they use kind of a CCH scheme there (even though the paper was published earlier than [9]), so we replace CCH by our PCH variant described above. We call the resulting approach *personalizable route planning (PRP)*.

In Figure 5, the process of applying PCH to a cell is illustrated. We observe two problems when assigning the cost vectors: The first problem is that some simple paths are represented by a cost vector despite never being optimal. For example, compare the path u, w, v to u, v. The edge $\{v, w\}$ is initially augmented with the cost vector $(2, 1)$. After the contraction of w, it additionally receives the vector $(16, 6)$ resulting from summing up the vectors assigned to $\{u, w\}$ and $\{v, w\}$. As $(2, 1)$ is cheaper in both dimensions, no choice of α in a query will include the path from u to v via w. So the cost vector $(16, 6)$ is superfluous. This does not affect correctness, but increases the space consumption and the query times later on. Note, that this also happens when using k-PC or PCH directly. The second problem is that the concatenation of simple paths has not to be a simple path itself. Consider the label $(22, 15)$ assigned to the overlay edge $\{b, c\}$. It is the result of summing up the labels $(5, 5)$ and $(17, 10)$. Those two labels stem from the simple paths b, v, u and u, v, w, c respectively. So the combined path that leads to the label $(22, 15)$ is indeed b, v, u, v, w, c. Obviously this never makes sense, as b, v, w, c is a better choice no matter what cost vectors are assigned to the edge $\{u, v\}$.

All unnecessary labels (either due to problem 1 or 2 or

both) are colored gray in Figure 5. We observe that even for small examples their number can be huge. Of course, we would like to avoid the insertion of such unnecessary labels as far as possible. In the next section, we explain how to detect unnecessary labels efficiently and describe pruning procedures to get rid of those.

4. PRUNING COST VECTORS

As observed in the last section, at the heart of all of the three potential approaches for answering personalized queries (k-PC, PRP, PCH) lies the need to attach as few cost vectors as possible to edges in the overlay graph in order to keep space consumption and query times low. But as an overlay edge naively gets assigned at least one cost vector per corresponding simple path in the road network, the sets of cost vectors per edge are likely to become huge. Hence the question arises for each edge whether at least some of the assigned cost vectors can be pruned. Obviously, in order to guarantee optimality, we can only prune cost vectors that are never part of an optimal solution for sure – no matter how the individually chosen weights α look like.

In the following, we denote with S the set of vectors associated with a single edge. We explain in detail pruning oracles that reduce the size of S without compromising optimality of the query result.

4.1 Pruning Dominated Vectors

The most straightforward strategy to prune vectors from S is by domination. A vector $v \in \mathbb{R}^d$ dominates another vector $v' \in \mathbb{R}^d$ if $v_i \leq v'_i \ \forall i = 1, \ldots, d$ and $v_i < v'_i$ for at least one $i \in \{1, \ldots, d\}$.

One can compare each vector v to every other vector v' in the set, checking component-wise if $v_i \leq v'_i$ for $i = 1, \cdots, d$ in order to prune out all dominated vectors. This leads to a runtime of $\mathcal{O}(d|S|^2)$.

For large sets S a quadratic runtime might be prohibitive for using the dominance pruner, though. Alternatively, we could select a good set of candidates from S and check only for those if they dominate other vectors in S. A natural choice for a promising set is to pick for each dimension the vector with minimum value in the respective component. This provides us with d candidates and accordingly a runtime of $\mathcal{O}(d|S|)$. This approach can easily be extended to select d^2 many candidates by picking the minimum vector not for every dimension but the combination of every two dimensions. This results in a runtime of $\mathcal{O}(d^2|S|)$.

4.2 Pruning Spanned Vectors

Of course, pruning by domination is not sufficient on its own, as not every non-dominated vector is optimal for some choice of α. The following Lemma characterizes superfluous, non-dominated cost vectors that can never be optimal no matter what α has been chosen.

LEMMA 1. *A vector $v \in \mathbb{R}^d$ can be pruned from a set of vectors $S \subset \mathbb{R}^d$ if a convex combination v' of at most d other vectors from S dominates v.*

PROOF. Assume for contradiction that for given α vector v uniquely defines the minimum cost $\alpha^T v = z$, that is, v cannot be pruned. Let $w_1, w_2, \cdots, w_d$ be the d vectors that span v' which dominates v. So we can represent v' as $\gamma_1 w_1 + \gamma_2 w_2 + \cdots + \gamma_d w_d$ with $\sum_{i=1}^{d} \gamma_i = 1$, $\gamma_i \geq 0$. By the domination property we know that $\alpha^T(\gamma_1 w_1 + \gamma_2 w_2 + \cdots + \gamma_d w_d) \leq z$ as well. This formula can be rearranged as follows:

$$\gamma_1 \alpha^T w_1 + \gamma_2 \alpha^T w_2 + \cdots + \gamma_d \alpha^T w_d \leq z$$

As $\alpha^T v = z$ is the minimum among all possible vectors, we conclude that $\alpha^T w_i > z$ for $i = 1, \cdots, d$. Plugging this observation in the formula above, we get:

$$\sum_{i=1}^{d} \gamma_i \alpha^T w_i > \sum_{i=1}^{d} \gamma_i z = z \sum_{i=1}^{d} \gamma_i = z$$

This obviously contradicts the fact that the left hand side is less or equal to z. Hence the initial assumption that v is necessary to get the minimum cost for some α is wrong, and v can be pruned. □

Lemma 1 tells us that a vector v can be pruned if it is dominated by a vector v' spanned by d other vectors. Given a set of d vectors $w_1, w_2, \cdots, w_d$, the test whether a vector v is pruned by them boils down to checking whether the following system of linear inequalities has at least one feasible solution:

$$\begin{aligned} \gamma_1 w_{11} + \gamma_2 w_{21} + \cdots + \gamma_d w_{d1} &\leq v_1 \\ \gamma_1 w_{12} + \gamma_2 w_{22} + \cdots + \gamma_d w_{d2} &\leq v_2 \\ &\cdots \\ \gamma_1 w_{1d} + \gamma_2 w_{2d} + \cdots + \gamma_d w_{dd} &\leq v_d \\ \gamma_1 + \gamma_2 + \cdots + \gamma_d &= 1 \end{aligned}$$

So one way of constructing a pruning oracle is to select d vectors from S and prune other vectors using them. But looking at all possible selections, namely $\binom{|S|}{d}$ many, this quickly becomes too expensive. Selecting b bases, and checking for each vector if it is pruned by the base, the runtime is $\mathcal{O}(b \cdot |S| \cdot poly(d))$. Checking everything would lead to $b = \mathcal{O}(|S|^d)$. Hence the goal is to keep b small while pruning as many vectors as possible.

Just like for dominance pruning, we can select a base greedily by picking the minimum cost vector for every dimension, or the best cost vectors for each two dimensions combined. In practice, the resulting set of vectors quickly prunes a large fraction of superfluous vectors out of S. We therefore call this approach the *quick pruning oracle*.

4.3 Triangle Pruning

So far, we considered the set of cost vectors assigned to a single edge as input for our pruning oracles. But taking the structure of the overlay graph into account further pruning is possible.

In Figure 6, an example is provided where not only cost vectors but a whole overlay edge can be pruned. The basic idea is to consider an overlay edge $\{u, w\}$ and all induced triangles, i.e. all pairs of edges $\{u, v\}, \{v, w\}$. For such a pair, we compute the set of cost vectors induced by their concatenation. Then we augment the set with the cost vectors directly assigned to $\{u, w\}$ and run our pruning oracles on the resulting set. Cost vectors originally belonging to $\{u, w\}$ that are pruned from this set are never needed for optimal query answering. If all vectors belonging to $\{u, w\}$ are pruned, we can even delete the edge. Note, that this is a valid approach for all our personalization schemes. The red nodes in the example illustration could either be cover nodes resulting from the k-PC approach, or border nodes

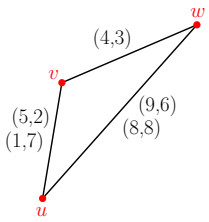

Figure 6: Triangle structure in the overlay graph. The overlay edge $\{u, w\}$ is superfluous as the cost vectors induced by the path u, v, w, namely $(9, 5)$ and $(5, 10)$ dominate or span the vectors $(9, 6)$ and $(8, 8)$ assigned to $\{u, w\}$.

		# nodes	# edges
Stuttgart	ST	1,024,885	2,079,185
Southern Germany	SG	6,603,683	13,466,194
Germany	GER	21,721,456	44,108,723

Table 1: Benchmark graphs. The graphs are extracted from OSM and are cut-outs of the road network of Germany.

from the PRP approach, or nodes in the PCH overlay graph if the rank order of the nodes is either u, v, w or w, v, u.

As accessing the triangles simply consists of comparing the outgoing edges of u to the ingoing edges of w to retrieve suitable nodes v, the performance of the triangle pruning mainly depends on the efficiency of the pruning oracles.

5. EXPERIMENTAL EVALUATION

We implemented all described approaches to answer personalized route planning queries, namely Dijkstra, k-Path Cover (k-PC), customizable contraction hierarchy (CCH), personalizable contraction hierarchy (PCH), customizable route planning (CRP) and personalizable route planning (PRP). Timings were measured on a single core of an Intel(R) i7-3770K CPU with 3.40GHz and 32GB RAM. The respective implementations are written in C++. Furthermore, we implemented the routine for pruning sets of multi-dimensional cost vectors, which is used as a subroutine for k-Path Cover, PCH and PRP. For that part of the implementation we used the Computational Geometry Algorithms Library (CGAL), [17], in particular their *exact* linear programming solver. Unless stated otherwise, query times are averaged over 1000 queries and the same source-target pairs are used when comparing different approaches.

5.1 Data and Settings

As benchmark data we used real-world street networks of varying size extracted from OSM[1]. The main features of our benchmark instances are collected in Table 1. For every edge (v, w) in the road network we defined ten cost values: *Distance* (euclidean distance with a precision of 1 meter), *travel time* (in seconds), *positive height difference* with the elevations of nodes in the road network being computed using SRTM data[2] (precision of one meter), *distance on large/medium/small roads* using OSM road categories as

[1] `openstreetmap.org`

[2] `http://srtm.csi.cgiar.org/`

d	# polls	# vectors	time ER	time total
1	$9.6 \cdot 10^6$	$1.9 \cdot 10^7$	2,638 ms	7,908 ms
2	$1.0 \cdot 10^7$	$2.1 \cdot 10^7$	2,852 ms	8,385 ms
5	$1.1 \cdot 10^7$	$2.2 \cdot 10^7$	2,996 ms	8,564 ms
10	$1.1 \cdot 10^7$	$2.3 \cdot 10^7$	3,101 ms	9,097 ms
32	$1.1 \cdot 10^7$	$2.2 \cdot 10^7$	4,262 ms	10,119 ms
64	$1.1 \cdot 10^7$	$2.3 \cdot 10^7$	6,147 ms	12,370 ms

Table 2: Experimental results on the GER network using personalizable Dijkstra. Cost values were created randomly. Timings are given in milliseconds, ER denotes the time for edge relaxation i.e. for evaluating the edge costs according to α. All values are averaged over 1000 random queries.

basis, *gas price* according to the formula in [13] (with one-tenth of a cent as basic unit), *energy consumption* for electric vehicles (in Watt) , *unit* (uniformly 1 per edge) allowing to distinguish between curvy and rather straight routes as in OSM curves are typically modeled by many small edges while long straight roads consist of few edges only, and *quietness* penalizing large roads and roads in dense road clusters (indicating city centers), with the penalty being proportional to the length of such a road, and zero for all others.

To test the performance of our algorithms for $d > 10$, we additionally created random cost metrics. We sticked to OSM data for benchmarking, since it is publicly available and allows for easy generation of many sensible metrics. For other data sets like DIMACS/TIGER/PTV which are sometimes used, this is far more difficult. The latter contain only two metrics (travel time and distance) or are not available to everyone due to licensing issues.

5.2 Experimental Results

We first evaluate the Dijkstra baseline and then proceed to demonstrate how the k-PC approach benefits from our developed pruning oracles. Finally, we investigate our new designed PCH and PRP approaches, showing their suitability for personalized queries in huge road networks.

Dijkstra Baseline. We evaluated the performance of Dijkstra for personalized queries on the GER network for dimension d between 1 and 64 (see Table 2), and for all benchmark graphs for $d = 10$ (see Table 3). We provide the number of polls (i.e. extractions of nodes from the priority queue), the number of evaluated cost vectors (which for Dijkstra equals the number of considered edges), the time to evaluate the cost vectors and the total query time. We observe that the time for answering personalized queries with Dijkstra for moderate dimensions d is not dominated by the time for evaluating the edge cost vectors, and evaluation times as well as the total time for answering queries increase only slowly with growing d. The search space, i.e. the number of considered edges/vectors is almost unaffected by d. We furthermore observe by comparing the rows for $d = 10$ and GER in both tables, that real-world metrics lead to slightly better query times than randomly created ones.

Improving k-PC. Next we want to measure the impact of our pruning oracles on the k-PC based approach. To compare the runtime and the pruning quality of our oracles, we created benchmark input sets of varying size by running the k-PC approach on the ST network and collecting the cost

$d = 10$	# polls	# vectors	time ER	time total
ST	$5.4 \cdot 10^5$	$1.1 \cdot 10^6$	112 ms	312 ms
SG	$2.9 \cdot 10^6$	$6.1 \cdot 10^6$	685 ms	1,957 ms
GER	$1.1 \cdot 10^7$	$2.2 \cdot 10^7$	2,815 ms	8,275 ms

Table 3: Experimental results on all benchmark graphs using Dijkstra and $d = 10$ real metrics.

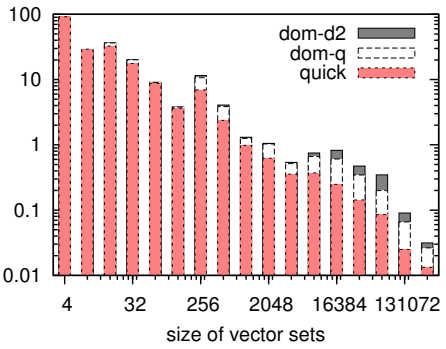

Figure 7: Percent of remaining vectors after pruning in dependency of the original vector set size for our three pruning oracles. Both axes are in logscale.

vectors for every edge naively. Then we applied individually the naive dominance pruner with quadratic runtime (dom-q), the dominance pruner which greedily selects $d + d^2$ vectors and compares to those (dom-d2) and the quick pruner which searches for spanned vectors (quick). The solution quality is depicted in Figure 7, the respective runtimes in Figure 8. We observe that for all our pruning oracles the size of the vector sets is reduced dramatically, especially for large vector sets. So for example for vector sets of size around 2^{18}, less than 0.3% of the vectors remain after dominance pruning, for our quick pruning oracle only about 0.1%, i.e. less than 2^5 vectors. Considering the runtime, we see that dom-q exhibits the expected quadratic runtime and therefore is impractical for larger inputs. The runtime of dom-d2 converges to the runtime of the quick pruning oracle; both are considerably faster than dom-q on large sets. Based on these observations, we use a pipeline of first dom-d2, then quick and finally (on assumingly small remaining sets) dom-q for the experiments to come.

We applied the described pipeline to the k-PC based overlay graphs for all our benchmark instances. In Table 4, we present timings for the overlay graph construction and the pruning process along with numbers that indicate the global reduction of vectors. We used different values of k depending on the size of the graph. We observe that for larger values of k our pruning oracles reduce the number of vectors to less than 5% of the original vectors (and therefore also the space consumption of the overlay graph). And even more important, especially very large sets of cost vectors assigned to single edges were reduced significantly. These are typically the bottlenecks for query answering. Accordingly, we observe improved runtimes for the k-PC approach. In Table 5, we see that the speed-up for GER compared to the Dijkstra baseline is over 20. Using unpruned vector sets, the speed-up was only about 12. For ST and $k = 40$ we even have a speed-up of over 30. We expect that the larger graphs would

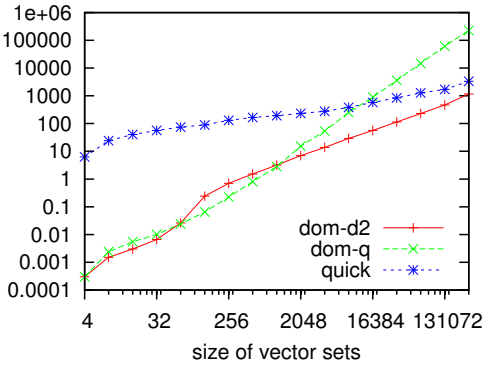

Figure 8: Average pruning time in ms in dependency of the vector set size for our three pruning oracles. Again, both axes are in logscale.

$d = 10$	time overlay	# vectors original	# vectors pruned	time pruning
ST, $k = 40$	22.8 s	6,295,791	4.8%	471.5 s
SG, $k = 30$	113.0 s	41,306,510	52.1%	3420.1 s
GER, $k = 24$	143.6 s	52,739,551	64.3%	7134.5 s

Table 4: Preprocessing step for k-PC including overlay graph construction and assignment of cost vectors.

also benefit from greater k values. Future work should focus on accelerating the computation of k-PC overlay graphs in order to make this applicable.

New Personalization Schemes. Next, we are going to present results for using our newly developed personalization schemes, namely PCH and PRP.

For comparison, we first evaluate the respective customization approaches CCH and CRP on personalized queries. We claimed that for both CCH and CRP, there is an optimal way to preprocess the graph such that the sum of customization and query time is minimized. This sum equals the query time for personalized queries. In Figures 9 query times in dependency of the overlay graph construction are presented for CCH on the example of the ST graph. We observe as postulated that stopping the contraction process for CCH earlier (for ST after contracting 99% of all nodes) provides better query times than full contraction (as customization time and total time increase significantly in the end). Experiments with varying cell sizes for CRP revealed that the best query times for CCH and CRP are comparable. We conducted respective experiments on all our test graphs and observed that the speed-up compared to the Dijkstra baseline is at most 2 for conventional CCH/CRP and increases

$d = 10$	# polls	# vectors	time	speed-up
ST	12,545	72,497	10 ms	31
SG	85,610	386,334	73 ms	27
GER	343,498	1,287,112	378 ms	22

Table 5: Experimental results on query answering with k-PC.

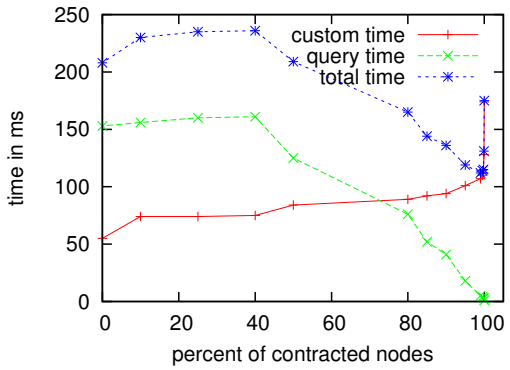

Figure 9: Performance of CCH (on the ST graph, $d = 10$) for personalized queries when varying the overlay graph construction.

	time	# shortcuts	lower triangles avg.	lower triangles max.
ST	4.48 s	1,889,036	0.6	16
SG	32.15 s	12,146,778	0.6	15
GER	118.79 s	40,290,730	0.6	18

Table 6: Overlay graph construction for PCH.

up to 4 with our modifications. Nevertheless real-time query answers for huge road networks are not within reach using those techniques.

Next we studied our new personalization schemes PCH and PRP. Let us have a closer look at the preprocessing phase of PCH: The first step is the metric-independent overlay construction. We contracted about 99% of the nodes in our test graphs and let the remaining nodes be core nodes. Contracting more nodes led to worse space consumption and query time, contracting less nodes resulted also in slower queries. The results are summarized in Table 6. We observe that the number of shortcuts is smaller than the number of original edges for all graphs, so the overlay graph has a size comparable to the original graph. The number of lower triangles that have to be considered in the personalization step is also rather small and moreover seems to be independent of the graph size. The outcome of the personalization step is investigated in Table 7. Our pruning pipeline as well as the triangle pruning approach turned out to be crucial for the preprocessing to complete. Applying only dominance pruning, for example, led to thousands of vectors assigned to overlay edges even in the ST graph. Using our described pruning strategies, the average number of vectors per edge becomes very small. But there are some overlay edges which exhibit quite large vector sets. These are typically shortcuts between core nodes. In Table 8 we report the query times for PCH. While the number of polls is reduced by over two orders of magnitude compared to the Dijkstra baseline, the speed-up is more than a factor of 50 as we have to consider several vectors per relaxed edge (three on average) and the node degree in the overlay graph is larger than in the original graph. But as the number of polls also is an indicator for how much space is needed in a query, we conclude that PCH significantly improves on the Dijkstra baseline in terms of query times and space consumption.

$d = 10$	time	vectors total	vectors avg.	vectors max.
ST	240.2 s	4,521,560	1.145	174
SG	1812.1 s	28,201,283	1.105	512
GER	8982.7 s	90,982,610	1.098	1,498

Table 7: Personalization step for PCH.

$d = 10$	# polls	# vectors	time	speed-up
ST	3,148	82,106	5 ms	62
SG	23,077	480,826	34 ms	57
GER	212,459	1,587,009	164 ms	50

Table 8: Query answering statistics for PCH.

Finally, we consider PRP which uses PCH as a subroutine. We used a three-layer variant for the overlay graph construction, with the lowest level exhibiting cells of roughly 4,000 nodes. The next two layers combine four cells of the layer below into a single cell. In Table 9, the number of border nodes and overlay edges induced by the partitioning of the graph on the lowest level are provided. Only very few nodes in the graph become border nodes which is beneficial as it leads to sparse overlay graphs. Each cell is then considered as graph on its own. We applied PCH to all those cells by manifesting the border nodes as core nodes. Then we ran queries on the resulting graph. The experimental results for all test graphs are collected in Table 10. In addition, Table 11 shows the dependency of the performance on the dimension d on the example of the SG graph. Query answering with PRP slightly outperforms query answering with PCH because less vectors are evaluated in a query. The reason might be that in PRP we have indirectly more control about how many simple paths an overlay edge spans by fixing cell sizes and inserting overlay edges only inside cells. Also queries with source and target in the same cell are answered by only considering edges inside this cell. For those queries, runtimes are significantly better for PRP than for PCH which affects the average value.

Inspecting the results for GER and varying dimensions, we see that the speed-up naturally decreases when adding more metrics, as potentially more necessary cost vectors are assigned to overlay edges in the graph. Nevertheless, even for $d = 64$ PRP leads to a speed-up of about factor 20, resulting in query times clearly below a second.

5.3 Comparison of Personalization Schemes

We showed that all three discussed schemes for personalization, k-PC, PCH and PRP, provide significant speed-up over the Dijkstra baseline. k-PC exhibits the smallest speed-up, but at the same time the most concise auxiliary data.

	# cells	# border nodes	# overlay edges
ST	253	6,922	1,877,894
SG	1,636	46,526	12,051,865
GER	5,430	165,447	39,945,430

Table 9: Partitioning and cell contraction for PRP on the lowest level.

$d = 10$	# polls	# vectors	time	speed-up
ST	2,985	45,771	4 ms	78
SG	24,975	293,412	29 ms	67
GER	179,301	1,283,488	128 ms	65

Table 10: Experimental results on all benchmark graphs for PRP.

d	# polls	# vectors	time	speed-up
1	$1.5 \cdot 10^5$	$6.9 \cdot 10^5$	71 ms	111
2	$1.7 \cdot 10^5$	$7.3 \cdot 10^5$	78 ms	107
5	$1.7 \cdot 10^5$	$7.7 \cdot 10^5$	82 ms	104
10	$1.6 \cdot 10^5$	$1.3 \cdot 10^6$	135 ms	67
32	$1.6 \cdot 10^5$	$2.5 \cdot 10^6$	287 ms	35
64	$1.8 \cdot 10^5$	$4.9 \cdot 10^6$	612 ms	20

Table 11: PRP results on GER with randomly created metrics.

PCH and PRP both exhibit a large speed-up with PRP being the most efficient approach in our experiments. But preprocessing and query answering is easier for PCH. In fact, PCH could be interpreted as PRP on a single cell. Also PCH is a necessary subroutine for PRP. We conclude that all three schemes have their justifications. We also want to note that our schemes are not competitive for $d = 2$ compared to previous approaches custom-tailored for that scenario. But the ability for personalization on the level we desire demands a significantly higher dimension, and for those values of d our schemes are the first to allow for real-time personalized queries in huge road networks.

6. CONCLUSIONS AND FUTURE WORK

We argued and experimentally proved that customization approaches are unsuitable to answer personalized route planning queries efficiently. Including customization times for each personalized query they are at most a factor of 4 faster than plain Dijkstra in a single thread setting. Our newly designed personalization schemes demand only very little space on query time and achieve speed-ups by a factor of 20 to 100 over Dijkstra even for high dimensions. The performance of our personalization schemes heavily relies on the careful assignment of cost vectors to the constructed overlay graph. We investigated how to prune sets of such vectors efficiently. Apart from accelerating the pruning process further, future work includes the investigation of other speed-up techniques for conventional route planning or customization which can be combined with our methods for further speed-up. Acceleration schemes for the static/unpersonalized case achieve speed-ups of a factor of $1,000$ or even more over the Dijkstra baseline. Hence the question arises whether the cost of personalization can be further reduced.

Acknowledgement. This work was partially supported by a Google Faculty Research Award.

7. REFERENCES

[1] I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck. Alternative routes in road networks. In *Proceedings of the 9th international conference on Experimental Algorithms*, SEA'10, pages 23–34, Berlin, Heidelberg, 2010. Springer-Verlag.

[2] I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck. A hub-based labeling algorithm for shortest paths in road networks. In *Experimental Algorithms*, pages 230–241. Springer, 2011.

[3] H. Bast, S. Funke, P. Sanders, and D. Schultes. Fast routing in road networks with transit nodes. *Science*, 316(5824):566–566, 2007.

[4] G. V. Batz, D. Delling, P. Sanders, and C. Vetter. Time-dependent contraction hierarchies. In *ALENEX*, volume 9. SIAM, 2009.

[5] G. D'Angelo, D. Frigioni, and C. Vitale. Dynamic arc-flags in road networks. In *Experimental Algorithms*, pages 88–99. Springer, 2011.

[6] D. Delling, A. V. Goldberg, T. Pajor, and R. F. Werneck. Customizable route planning. In *Experimental Algorithms*, pages 376–387. Springer, 2011.

[7] D. Delling, M. Kobitzsch, and R. F. Werneck. Customizing driving directions with gpus. In *Euro-Par 2014 Parallel Processing*, pages 728–739. Springer, 2014.

[8] D. Delling and R. F. Werneck. Faster customization of road networks. *SEA*, 13:30–42, 2013.

[9] J. Dibbelt, B. Strasser, and D. Wagner. Customizable contraction hierarchies. In *Experimental Algorithms - 13th International Symposium, SEA 2014, Copenhagen, Denmark, June 29 - July 1, 2014. Proceedings*, pages 271–282, 2014.

[10] J. Dibbelt, B. Strasser, and D. Wagner. Erratum: Customizable contraction hierarchies. In *Experimental Algorithms - 13th International Symposium, SEA 2014, Copenhagen, Denmark, June 29 - July 1, 2014. Proceedings*, 2014.

[11] S. Funke, A. Nusser, and S. Storandt. On k-path covers and their applications. *Proceedings of the VLDB Endowment*, 7(10), 2014.

[12] S. Funke and S. Storandt. Polynomial-time construction of contraction hierarchies for multi-criteria objectives. In *Proceedings of the 15th Meeting on Algorithm Engineering and Experiments, ALENEX 2013*, pages 41–54, 2013.

[13] R. Geisberger, M. Kobitzsch, and P. Sanders. Route planning with flexible objective functions. In *ALENEX'10*, pages 124–137, 2010.

[14] R. Geisberger, M. N. Rice, P. Sanders, and V. J. Tsotras. Route planning with flexible edge restrictions. *J. Exp. Algorithmics*, 17(1):1.2:1.1–1.2:1.20, Mar. 2012.

[15] R. Geisberger, P. Sanders, D. Schultes, and C. Vetter. Exact routing in large road networks using contraction hierarchies. *Transportation Science*, 46(3):388–404, 2012.

[16] A. V. Goldberg and C. Harrelson. Computing the shortest path: A search meets graph theory. In *Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*, SODA '05, pages 156–165, Philadelphia, PA, USA, 2005. Society for Industrial and Applied Mathematics.

[17] The CGAL Project. *CGAL User and Reference Manual*. CGAL Editorial Board, 4.4 edition, 2014.