Personal Routes with High-Dimensional Costs and Dynamic Approximation Guarantees*

Stefan Funke[1], **Sören Laue**[2], **and Sabine Storandt**[3]

1 **University of Stuttgart, Stuttgart, Germany**
`funke@fmi.uni-stuttgart.de`

2 **Friedrich-Schiller-Universität Jena, Jena, Germany**
`soeren.laue@uni-jena.de`

3 **JMU Würzburg, Würzburg, Germany**
`storandt@informatik.uni-wuerzburg.de`

Abstract

In a personalized route planning query, a user can specify how relevant different criteria as travel time, gas consumption, scenicness, etc. are for his individual definition of an optimal route. Recently developed acceleration schemes for personalized route planning, which rely on preprocessing, achieve a significant speed-up over the Dijkstra baseline for a small number of criteria. But for more than five criteria, either the preprocessing becomes too complicated or the query answering is slow. In this paper, we first present a new LP-based preprocessing technique which allows to deal with many criteria efficiently. In addition, we show how to further reduce query times for all known personalized route planning acceleration schemes by considering approximate queries. We design a data structure which allows not only to have personalized costs but also individual approximation guarantees per query, allowing to trade solution quality against query time at the user's discretion. This data structure is the first to enable a speed-up of more than 100 for ten criteria while accepting only 0.01% increased costs.

1998 ACM Subject Classification E.1 [Data Structures] Graphs and Networks

Keywords and phrases personalized route planning, contraction hierarchies, linear program, separation oracle, approximate queries

Digital Object Identifier 10.4230/LIPIcs.SEA.2017.18

1 Introduction

While conventional route planning engines usually compute the shortest or quickest path between a given source and a target, individual preferences of users might differ. For example, a user might accept a slightly later arrival time at the target in exchange for a less crowded route, or reduced gas consumption, or fewer traffic lights or left turns on the way. To provide full flexibility, all possible trade-offs between all criteria should be valid definitions of an optimal route – and each query should be allowed its own definition. The personalized route planning problem captures this idea. Formally, it can be phrased as follows: Given a street network $G(V, E)$, with a d-dimensional non-negative cost vector $c(e) \in \mathbb{R}^d$ for each edge $e \in E$ (where each entry reflects one criterion, e.g. c_1 travel time, c_2 number of traffic lights, c_3 gas price, and so on); a query consists of source and target $s, t \in V$ and non-negative

* This work was supported by a Google Research Award on 'Personalized Route Planning'. Sören Laue acknowledges the support of Deutsche Forschungsgemeinschaft (DFG) under grant LA-2971/1-1.

weights $\alpha_1, \alpha_2, \cdots, \alpha_d$ (expressing the importance of each cost component for the user). The goal is to compute the path p from s to t in G which minimizes $\sum_{e \in p} \alpha^T c(e)$.

Dijkstra's algorithm which computes individual edge costs $\alpha^T c(e)$ on demand can solve the personalized route planning problem in $\mathcal{O}(n \log n + dm)$ with $|V| = n, |E| = m$. But for applications where efficiency is important – as in a server-client scenario with multiple queries per second – Dijkstra's algorithm does not allow for sufficient throughput. Therefore, like for the conventional route planning problem, preprocessing based speed-up techniques were investigated in previous work. But they either suffer from a complex preprocessing routine (based on contraction hierarchies [14]) which limits the practical applicability to $d \leq 3$ [13], [11], or with lighter preprocessing the speed-up for $d \geq 5$ is less than 100 compared to the Dijkstra baseline [10], [8], [12].

In this paper, we show how to instrument contraction hierarchies for the personalized route planning problem with the help of linear programming. Our method is conceptually simpler than previous attempts to use contraction hierarchies for this scenario. As a consequence, we can perform the preprocessing also for $d > 3$ which was impractical before. Furthermore, we describe a framework which allows to compute personalized queries with an approximation guarantee. Our respective data structure allows to trade query time against solution quality. Thereby the quality is a parameter to be set by the user, and can vary with every query. In our experiments, we show that accepting only slightly suboptimal results enables query answering two orders of magnitude faster than the Dijkstra baseline even for high-dimensional costs.

1.1 Related Work

The personalized route planning problem was introduced in [13]. There, a contraction hierarchy augmented with landmarks [15] was shown to provide a speed-up of three orders of magnitude over the Dijkstra baseline; but only for $d = 2$, and with some restrictions on the weights that can be chosen. In [11], a contraction hierarchy construction scheme was designed that maintains all optimal paths for arbitrary dimensions d and arbitrary weights α. But the CH construction relies on a rather complicated subroutine which involves the computation of a dynamically changing convex hull of points in $\mathbb{N}^d$. The speed-ups presented there were about three orders of magnitude for $d = 2$ and two orders of magnitude for $d = 3$. For larger d, no practical results could be provided.

To handle a larger number of metrics, the preprocessing paradigm for customizable route planning schemes was applied to personalized route planning [10], [8], [12]. Here the idea is to first construct an overlay graph independently of the metric(s). Then, in a second phase, the edges of the overlay graphs are augmented with suitable costs or cost vectors. In [10], the overlay graph was based on a k-path cover of the graph. This allowed to consider up to 64 metrics for the first time. But the speed-up was rather moderate: about 12 for $d = 2$ and 8 for $d = 64$. In [8], a slightly more general personalized route planning model was used, where also restrictions to binary weights and non-additive costs are possible. Here, the overlay graph is obtained by some simple rules which only rely on the topology of the graph. A speed-up of about 30 was reported for $d = 8$ and about 15 for $d = 64$. In [12], customizable contraction hierarchies [7] and customizable route planning [4] were turned into personalizable schemes. As the number of cost vectors of an edge in the overlay graph were shown to become huge, optimality-preserving pruning strategies were described. With that, the speed-up for $d = 2$ was reported to be about 100, for $d = 10$ about 70 and for $d = 64$ about 20.

Further speed-ups were achieved by making use of parallelization either utilizing multiple cores [6] or even the GPU [5]. Here, the whole graph is personalized for every query by setting all edge costs to $\alpha^T c(e)$. But in a client/server scenario, these approaches require too much machinery and space per user to allow for high throughput. Hence, in the remainder of the paper, we only consider sequential approaches.

In some way the dual problem of personalized routing was considered in [3] and [9]. Here, given a set of paths a user has traveled, the goal is to infer his weight vector α (also called his preference) that explains why exactly these paths were chosen. While in [3] a method based on stochastic coordinate descent was introduced, the approach in [9] relies on linear programming. We will use similar LP-based techniques as proposed in [9] to allow for efficient computation of personalized routes.

Suboptimal query answering for acceleration and more concise result representation was already considered in related domains such as public transit route planning. There, the multi-criteria setting is naturally induced, as besides travel time also the number of transfers, the walking time and the ticket prices (among other criteria) are relevant when planning a journey. As already with three criteria there can be hundreds or even thousands of Pareto-optimal route options between a source and a target, fuzzy dominance [2] and rule-based filters [1] were applied to reduce the solution set. But neither of those comes with a guarantee that assures the output in the end to be close to the optimal solution. In contrast, our method for approximate personalized routes will allow to fix an arbitrary approximation factor $\delta \geq 1$ and the returned solution will never be more than δ times as expensive as the optimal route.

1.2 Contribution

We provide the following new insights and algorithms for personalized route planning:

- For contraction hierarchies, the preprocessing in previous work [11] required a complicated binary-search like algorithm which involved the computation of a dynamically changing convex hull of d-dimensional points. We present a conceptually simpler, more efficient and robust LP-based contraction hierarchy (LP-CH). This approach also considerably reduces the theoretical runtime dependency on d for a crucial operation in the construction from $\mathcal{O}(d^2 \log(nMd)(n \log n + dm + d^{4+d} \log^{d-1}(nMd)))$ to $\mathcal{O}(d^2 \log(nMd)(n \log n + dm))$ where M is the maximum cost entry of any edge.
- We develop a new data structure which allows to answer personalized queries in an approximate fashion. Thereby, the approximation factor has not to be fixed a priori but can be chosen on query time. At the heart of our data structure lies an algorithm which sorts a list of d-dimensional cost vectors, such that the first i vectors approximate the complete set of vectors provably well for every i.
- We experimentally prove that our new contraction hierarchy preprocessing allows to deal with more than three metrics efficiently. Our framework for approximate query answering combines well with contraction hierarchies and also with approaches based on customizable route planning presented in previous work. This allows us to achieve further speed-ups when accepting slightly suboptimal results. For example, for $d = 10$ and an approximation factor of 1.001, query answering with our methods is two orders of magnitude faster than with bidirectional Dijkstra.

Comparisons between results from different papers are especially challenging for personalized route planning, as not only benchmark graphs might differ but more importantly also the criteria used. The selected criteria can affect the performance significantly: Well-correlated metrics as travel time and distance are easier to manage than conflicting objectives as travel time and quietness (as illustrated already in [11]). We therefore make our largest benchmark instance with 10 meaningful metrics available[1] to alleviate future comparison of algorithms.

2 Contraction Hierarchies (CH) with an LP-Oracle

In this section, we describe how to construct contraction hierarchies for multi-dimensional costs. The main novelty compared to previous work is the usage of an LP-oracle in the preprocessing.

2.1 Conventional Contraction Hierarchies

The contraction hierarchies approach [13] computes an overlay graph in which so called shortcut edges span large sections of the shortest path. This reduces the hop length of optimal paths and therefore allows Dijkstra's algorithm to answer queries more efficiently.

The preprocessing is based on the so-called node contraction operation. Here, a node v as well as its adjacent edges are removed from the graph. In order not to affect shortest path distances between the remaining nodes, shortcut edges are inserted between all neighbors u, w of v, if and only if uvw was a shortest path (which can easily be checked via a Dijkstra run). The cost of the new shortcut edge (u, w) is set to the summed costs of (u, v) and (v, w). In the preprocessing phase all nodes are contracted one-by-one in some order. The rank of the node in this contraction order is also called the *level* of the node.

After having contracted all nodes, a new graph $G^+(V, E^+)$ is constructed, containing all original edges of G as well as all shortcuts that were inserted in the contraction process. An edge $e = (v, w)$ – original or shortcut – is called upwards, if the level of v is smaller than the level of w, and downwards otherwise. By construction, the following property holds: For every pair of nodes $s, t \in V$, there exists a shortest path in G^+, which first only consist of upwards edges, and then exclusively of downwards edges. This property allows to search for the optimal path with a bidirectional Dijkstra only considering upwards edges in the search starting at s, and only downwards edges in the reverse search starting in t. This reduces the search space significantly.

For personalized route planning, we are not dealing with scalar edge costs but cost vectors. Here, deciding the necessity of a shortcut in the CH graph is not trivial anymore. A shortcut (u, w) with costs $c(u, v) + c(v, w)$ has to be inserted upon contraction of v if and only if it encodes an optimal path p from u to w for some choice of α. But checking for all possible $\alpha \in [0, 1]^d$ whether p is the respective optimal path is obviously not possible. Therefore, we need a more efficient way to decide the necessity of a shortcut.

2.2 Shortcut Insertion Oracles

To keep the CH graph as sparse as possible, we only want to insert shortcuts which are necessary for exact query answering. So the only way to omit the insertion of a shortcut (u, w) which spans the edges (u, v) and (v, w) with costs $c(u, v), c(v, w)$ is to certify that there exists no α such that $\alpha^T c(u, v) + \alpha^T c(v, w)$ is the minimum cost among all paths p from u to w.

[1] `https://www.dropbox.com/s/tclrjdkfhabu27h/ger10.zip?dl=0`

Convex Hull Oracle. In [11], it was shown that the shortcut insertion problem is solvable in polynomial time for fixed dimension d by reduction to the following geometric problem: A path p with d-dimensional cost vector c can be interpreted as point in $\mathbb{N}^d$. It was shown that this path is optimal for some choice of α if and only if the respective point is part of the lower convex hull of all points that correspond to paths between the same source and target. Then an algorithm was described which decides if a point is part of the lower convex hull or not. As the convex hull might have exponential complexity, its explicit construction and inspection is impractical. The described algorithm avoids the explicit construction by iteratively computing optimal paths for α vectors chosen from a certain multi-dimensional interval. If the optimal path for some α equals p, the respective shortcut is necessary for sure and the search can be aborted. Otherwise, the optimal path p' allows to decrease the volume of the multi-dimensional interval of α vectors for which p could still be optimal by at least a constant fraction. Therefore, after at most polynomially (in the input size) many steps, the algorithm terminates and then certifies that no α exists for which p is optimal. The main problem with this approach is that calculating the new multi-dimensional interval for α and choosing a new reasonable α within this interval requires the computation of d-dimensional hyperplane cuts, which is expensive and numerically unstable. The total runtime of this approach was shown to be in $\mathcal{O}(d^2 \log(nMd)(n \log n + dm + d^{4+d} \log^{d-1}(nMd)))$ with $M = \max_{e \in E} |c(e)|_\infty$ being the maximum cost of any vector in any dimension.

We will now introduce an oracle based on linear programming which is simpler and leads to an improved theoretical runtime of $\mathcal{O}(d^2 \log(nMd)(n \log n + dm))$.

Naive LP Oracle. Our goal is still to either find an α for which our reference path p from u to w with costs $c(p) \in \mathbb{N}^d$ is optimal or to certify that no such α exists. We will achieve this by setting up an LP which has a feasible solution if and only if p is optimal for some α.

Let P be the set of all possible paths from u to w in G. We want to find an α such that $\alpha^T c(p)$ is not larger than $\alpha^T c(p')$ for all $p' \in P$, that is, p is optimal for this choice of α. We can express this as a simple system of linear (in)equalities or constraints:

$$\begin{aligned} \sum_{i=1}^{d} \alpha_i &= 1 & & \\ \alpha_i &\geq 0 & \qquad & i = 1, \dots, d \\ \alpha^T c(p) - \alpha^T c(p') &\leq 0 & & \forall p' \in P \end{aligned}$$

Obviously, any α which is a solution for this linear program certifies that p is optimal for some queries. If there is no feasible solution, then for all choices of α there exists some path p' in P with lower costs and hence the shortcut encoding p can be omitted.

The problem with this LP is that – like for the convex hull oracle – the set P of alternative paths from u to w might be exponentially large. Hence setting up the LP is already too expensive to be practical.

Improved LP Oracle. The basic question is whether we really have to consider all alternative paths in P to decide whether the shortcut is necessary or not.

The ellipsoid method [16] for LP solving does not require all constraints to be explicitly available, but instead demands the existence of an efficient separation oracle. A separation oracle is fed with a possible solution and then either has to verify that this is indeed a solution for the complete LP (with all constraints) or otherwise has to return a (so far not explicit) constraint that is violated by the current solution.

In our setting, Dijkstra's algorithm can serve as separation oracle: For some choice of α, we can efficiently check whether the path p is optimal for this choice or not by running Dijkstra with edge costs $\alpha^T c(e)$. If p is not optimal, Dijkstra returns an alternative path p' which provides us with the new constraint $\alpha^T(c(p) - c(p')) < 0$. This constraint excludes all choices of α for which p' is a better route than p.

This gives rise to the following algorithm which returns true if p is optimal for some choice of α and false otherwise:

1. Initialize the LP with α with $\sum_{i=1}^{d} \alpha_i = 1$ and $\alpha_i \geq 0, i = 1, \dots, d$.
2. Solve the LP using the ellipsoid method. If there is no solution, return false. Otherwise let the solution be α^*.
3. Run the Dijkstra separation oracle with α^*. If the optimal path is p, return true. If the optimal path is $p' \neq p$, add the constraint $\alpha^T(c(p) - c(p')) < 0$ to the LP. Go to 2.

The separation oracle runs in $\mathcal{O}(n \log n + dm)$. The number of iterations when using the ellipsoid method is bounded by $\mathcal{O}(k^2 L)$ where k is the number of variables and L the number of bits needed to represent the constraints. We have $k = d$ and as the coefficients represent costs of paths in G, they are bounded by nMd, hence we get $L = \log(nMd)$. In total, this results in a runtime of $\mathcal{O}(d^2 \log(nMd)(n \log n + dm))$.

2.3 Compacting Edges for Query Answering

As the result of the CH preprocessing, we get an overlay graph with shortcut edges. Each shortcut edge (u, w) is augmented with a d-dimensional cost vector, which is the result of aggregating the cost vectors along an optimal path from u to w. Naturally, there might be many such shortcuts between u and w, representing different optimal paths for different choices of α. During query answering, the relaxation of each of those edges might lead to a new temporary distance label for w, hence inducing multiple decrease key operations in the Dijkstra priority queue. If we instead merge all edges between u and w into a single edge – now with an associated set of cost vectors S – the relaxation of this edge consists of first computing $\min_{s \in S} \alpha^T s$, followed by at most a single decrease key operation, which is much more efficient. Furthermore, the data structure we get when using customizable route planning with personalization (as exploited in [10], [8], [12]) is also an overlay graph with sets of cost vectors per edge (but based on a different construction process).

From now on, we assume that some overlay graph for exact query answering is available. We can modify and augment any such overlay graph to enable efficient answering of approximate queries, as described in the next section.

3 Adaptive Approximation Guarantees

Among the set of potentially optimal routes between A and B, there are often many similar ones from a user perspective. For example, one route might have a travel time of 34 minutes and a gas price of 0.96 Euro, while another one has a travel time of 32 minutes and a gas price of 0.97 Euro. Furthermore, it is not easy for a user to specify α precisely in a meaningful way. Should the travel time be twice as important as the gas price, i.e. $\alpha^T = (2/3, 1/3)$, or three times as important, i.e. $\alpha^T = (3/4, 1/4)$? Slight variations in the choice of α might also result in different optimal routes. Hence we argue that two routes p and p' with very similar costs $\alpha^T c(p) \approx \alpha^T c(p')$ are almost indistinguishable for a user.

From the overlay construction, we get shortcut edges (v, w) with sets of cost vectors, encoding paths from v to w that are optimal for some choice of α. In particular for shortcuts between nodes far away from each other, the set of cost vectors is typically quite large and hence relaxing the shortcut edge during query processing becomes very expensive (evaluating each vector with the given α). If one is willing to accept some small error, it might suffice just to inspect a few of these vectors. This gives rise to the following problem:

Consider a set of vectors $S = \{v \in \mathbb{R}^d\}$, $|S| = k$ and their objective function values $\alpha^T v$. We are interested in determining an ordering $v^{(1)}, v^{(2)}, \dots v^{(k)}$ of the vectors in S as well as a respective sequence of error bounds $err^{(1)} \geq err^{(2)} \geq \dots \geq err^{(k)} = 1$, such that for any optimization direction $\alpha = (\alpha_1, \dots, \alpha_d)$, $\alpha_i \in \mathbb{R}_0^+$:

$$\frac{\min\limits_{j \leq i} \alpha^T v^{(j)}}{\min\limits_{v \in V} \alpha^T v} \leq err^{(i)}$$

that is, when only considering the first i vectors, we will never experience an objective function value worse than a factor $err^{(i)}$ than the optimum, no matter how the optimization direction α is chosen.

Clearly, any ordering of S will yield a monotonously decreasing sequence of error bounds, yet it is desirable to find an ordering where the errors drop as quickly as possible.

3.1 Two Greedy Strategies

In the following we present two greedy strategies to compute such an ordering. They both make use of a subroutine which for a given set of vectors $S' \subset S$ and an additional vector $w \notin S'$ computes the maximum relative approximation error of S' with respect to w, that is, how much worse the objective function value can get if S' is used to represent w. We will elaborate on this subroutine in the next subsection but treat it as a black box for now and denote by $T_{err}(k)$ the running time of this subroutine for a set of size k.

Iterative Error Minimization (bestNext). Assume we have already determined the first $i-1$ vectors $S^{(i-1)} := v^{(1)}, \dots, v^{(i-1)}$ in this ordering. As next vector $v^{(i)}$ we want to choose the one in $S - S^{(i-1)}$ which minimizes the maximum relative error with respect to the remaining set. We can do so by iterating through all vectors $v \in S - S^{(i-1)}$ and computing the maximum relative approximation error of the set $S^{(i-1)} \cup \{v\}$ with respect to each $w \in S - (S^{(i-1)} \cup \{v\})$. As next $v^{(i)}$ we choose the candidate v minimizing this maximum relative approximation error. The error bound $err^{(i)}$ is set accordingly. For $|S| = k$, $O(k^2)$ calls to the black box subroutine are necessary to determine $v^{(i)}$. So in $O(k^3 T_{err}(k))$ time the ordering of S can be computed. Note that if for each $v \in S - S^{(i-1)}$ we remember its relative error wrt $S^{(i-1)}$, very often a call to the black box subroutine is not necessary if this 'old' relative error is below the worst relative error already found, effectively reducing the total running time to $O(k^2 T_{err}(k))$ in practice.

Worst-Error Next (worstNext). The number of calls to the black box subroutine might be prohibitive for very large sets S. The following alternative greedy strategy also produces an ordering using considerably less calls to the black box subroutine.

Again we assume that we have already determined the first $i-1$ vectors $S^{(i-1)} := v^{(1)}, v^{(2)}, \dots v^{(i-1)}$ in this ordering. As next vector $v^{(i)}$ we choose a $v' \in S - S^{(i-1)}$ for which the relative approximation error of $S^{(i-1)}$ with respect to v' is maximized; $err^{(i)}$ is set accordingly. This requires $O(k)$ calls to the black box subroutine and hence $O(k^2 T_{err}(k))$ time to compute the ordering of S. As in the previous approach, calls to the black box subroutine can often be saved if the 'old' relative errors are memorized, effectively reducing the running time in practice.

3.2 Bounding the Relative Approximation Error

Consider a set of vectors $S' = \{u^{(1)}, \dots u^{(k)}\}$ and an additional vector $w \notin S'$. We are interested in the maximum *relative* approximation error of U with respect to w or more formally:

$$err_{rel} = \max_{\alpha} \frac{\min\limits_{u \in S'} u^T \alpha}{w^T \alpha}.$$

We will compute err_{rel} indirectly by determining the minimal factor $\delta \geq 1$ such that the vector δw is superfluous in (can be pruned from) the set $U \cup \{\delta w\}$ without affecting the optimum objective function value for any α. Since δ contributes linearly to the objective function, the minimal δ is exactly the relative approximation error err_{rel}, since then

$$\max_{\alpha} \frac{\min\limits_{u \in S'} u^T \alpha}{\delta w^T \alpha} = 1.$$

Determining whether a vector can be pruned from a set has been characterized in [12] as follows:

▶ **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 other vectors from S dominates v.*

Here domination refers to component-wise $\leq$.

Based on this Lemma and similarly to [12] we search for a convex combination u' of the k vectors in S' dominating the vector δw, minimizing δ.

$$\min \delta \tag{1}$$

$$\gamma_1 u^{(1)} + \gamma_2 u^{(2)} + \dots \gamma_k u^{(k)} \leq \delta w \tag{2}$$

$$\sum \gamma_i = 1 \tag{3}$$

$$\gamma_i \geq 0 \tag{4}$$

Note that the $\leq$ in inequality (2) is to be understood component-wise in each of the d dimensions. Clearly, this LP has a feasible solution (just choose δ large enough). Since its dual is a linear program with $O(k)$ constraints in $O(d+1)$ variables, theoretically it can be solved in $O(k)$ time for fixed d using approaches like see [17]. In practice, state-of-the-art linear programming solvers suffice to quickly compute an optimum solution.

3.3 Query Answering with an Approximation Guarantee

We will use the introduced greedy strategies to sort the cost vectors for each shortcut and remember the approximation factor for each prefix. Then, in a query with desired approximation factor δ, we only consider the cost vectors on each shortcut until the respective approximation factor is at least as good as δ. In that way, it is guaranteed that the outputted path in the end has costs at most δc^* with c^* being the optimal costs. Note, that it does not make sense to apply the sorting schemes to shortcuts with only a few cost vectors, as the potential for saving is small but we introduce some space overhead by storing the approximation factors.

4 Experimental Results

We implemented the new CH preprocessing technique as well as the algorithms to obtain adaptive approximation guarantees in C++ (g++ 6.2.0). The edge costs as well as the weights c, α are represented as *doubles*. Performance was measured on a single core of an Intel Xeon CPU E3-1225v3 with 3.20GHz and 32GB RAM running Ubuntu Linux 16.10. For solving LPs we called the GLPK library in version 4.60.

4.1 Benchmark

We use the street network of Germany with 22,046,972 and 44,702,123 edges extracted from OSM[2] as a benchmark graph. As in [12], we constructed the following ten meaningful metrics: *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[3] (precision of one meter, 0 for downhill edges), *distance on large/medium/small roads* using OSM road categories as basis (allowing to penalize e.g. large streets which tend to be more crowded, or small village streets as they might be too narrow), *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. We make the graph with all metrics available at `https://www.dropbox.com/s/tclrjdkfhabu27h/ger10.zip?dl=0`.

4.2 LP-based Contraction Hierarchy

We first investigate the performance of our new CH construction variant which uses linear programming with a Dijkstra-based oracle for efficiently deciding which shortcuts are necessary. We compare it to the previous version of CH which used a convex hull approach to decide the importance of a shortcut [11]; and to the CH variant based on customizable route planning where shortcuts are inserted independently of the metric and coated with cost vectors in a second preprocessing phase followed by some basic pruning, see [12]. We refer to these three variants as LP-CH (linear programming CH), H-CH (hull CH), and PC-CH (personalized customizable CH). For $d = 1$, we just use conventional CH.

4.2.1 Preprocessing

In Table 1, we provide the preprocessing time as well as the number of edges in the CH-graph and the number of cost vectors for all variants. We make the following observations: While H-CH is only applicable up to $d = 3$, LP-CH can also be computed for $d = 10$. The resulting graphs for H-CH and LP-CH are unsurprisingly similar. LP-CH is more efficiently computable for $d = 2$ as we use a simple oracle before actually starting the LP solver: We just compute the optimal path for all d possible α with a single 1 entry first. If the shortcut represents one of those paths, we add it to the graph. If one of those paths dominates the shortcut, we are sure the shortcut is unnecessary. Only if both cases do not apply, we start the LP-based

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

Table 1 Preprocessing results for real metrics. For $d = 2$, the process was stopped after 99.95% nodes were contracted, for $d = 3$ after 99.75% and for $d = 5, d = 10$ after 99% (to make PC-CH applicable). Timings are given in minutes, for # edges/vectors the 'm' denots millions. For comaprison, a CH for $d = 1$ can be constructed (with full contraction) in less than 10 minutes resulting a graph with 84.1 million edges.

	LP-CH			H-CH			PC-CH		
d	#edges	#vectors	time	#edges	#vectors	time	#edges	#vectors	time
2	85.6m	86.3m	25	85.6m	86.2m	46	94.5m	94.8m	24
3	84.1m	86.1m	32	84.2m	86.1m	23	89.3m	90.3m	28
5	75.2m	79.7m	28	-	-	-	84.4m	90.7m	47
10	76.4m	82.4m	56	-	-	-	84.4m	91.0m	152

algorithm described in Section 2.2. For $d = 2$, this reduces the number of LPs to solve significantly. Hence LP-CH is faster here as H-CH. For $d = 3$, more LPs have to be solved and the convex hull approach used for H-CH is slightly more efficient. In comparison to PC-CH, LP-CH leads to significantly smaller graphs. More advanced vector and edge pruning techniques could reduce the number of edges and vectors in the PC-CH graph further, but we observe that the preprocessing times are already worse for PC-CH for larger d. Also, further contraction of nodes for high values of d was completely impossible for PC-CH as the number of vectors grows too quickly. For LP-CH on the other hand, we could continue the contraction process up to 99.5% of the nodes. For $d = 5$ this took 32 minutes and resulted in a graph with 86.0 million vectors, for $d = 10$ it took about 2 hours and the final graph contained 92.6 million vectors. Note, that there might be some superfluous vectors contained in our LP-CH graphs as we allowed at most 100 LPs to be solved per shortcut in question, and just inserted the shortcut if we had no conclusive answer so far. Also, to accommodate for not having exact arithmetic on real numbers, we not only inserted a shortcut if it has exactly the optimal cost c^* for some α but also if the costs were less than $c^* + \epsilon$ with ϵ being an upper bound on the error that might occur.

4.2.2 Query Answering

Next, we compare LP-CH to PC-CH in terms of query processing efficiency. We measure speed-up towards the bi-directional Dijkstra in the original graph. Our results for varying d are summarized in Table 2. We observe that LP-CH outperforms PC-CH for all choices of d but the advantage decreases with higher dimension. While the PC-CH graph contains many more edges and vectors than the LP-CH graph, the query times for larger d are dominated by the relaxation of few shortcuts with large cost vector sets for both approaches. For $d = 5$, the LP-CH graph contains shortcuts with 455 vectors, for PC-CH up to 986 vectors. Figure 1 shows the distribution of the vector set sizes. Indeed, most of the shortcuts have very small vector sets. But the few with large sets are typically shortcuts between important nodes (contracted late in the preprocessing) and hence are contained in the search space of many long-distance queries. Therefore, if the the set sizes of such shortcuts can be reduced – as envisioned with our approximation framework – the query times can be further improved.

4.3 Adaptive Approximation via Vector Ordering

In Section 3 we have devised two strategies for ordering the vector set associated with a shortcut edge which are evaluated in the following.

As initial test data we used randomly generated (non-dominated) vector sets in 5 and 10 dimensions, as well as actual vector sets as constructed by our LP-CH; the results can be

Table 2 Query results (averaged over 100 source-target pairs) for randomly chosen α vectors. A poll refers to an extraction operation from the priority queue. For LP-CH and PC-Ch we list the improvements towards the Dijkstra baseline.

	Dijkstra		LP-CH		PC-CH	
d	time (ms)	polls	speed-up	poll ratio	speed-up	poll ratio
2	5,192	$1.4 \cdot 10^7$	425	616	112	286
3	5,617	$1.4 \cdot 10^7$	311	422	98	224
5	6,357	$1.5 \cdot 10^7$	118	259	87	185
10	7,126	$1.6 \cdot 10^7$	71	192	54	174

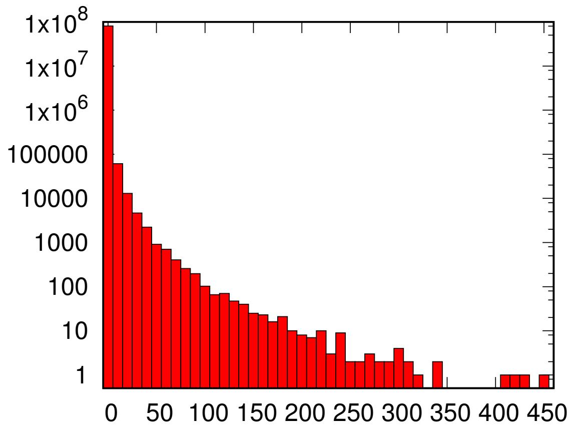

Figure 1 Number of shortcuts in dependency of the vector set size. The y-axis is in logscale.

seen in Table 3 (averages taken over 10 runs each). For example, inspecting only the first 32 vectors of a 10 dimensional vector set of size 130 guarantees on average an approximation ratio of 2.75 for the worstNext strategy and 2.90 for the bestNext strategy. Computing the former ordering took 23ms, the latter 873ms on average. As to be expected, the bestNext strategy takes considerably longer time, but sometimes yields better approximation guarantees when only considering very few vectors. We also see that random instances hardly mimic the instances actually appearing as vector sets of shortcuts due to the LP-CH construction. This is also comes as no surprise given the rather strong correlation between certain metrics in the real-world data. For the real-world instances, looking at the first 32 (out of 130) vectors yields almost perfect results (with an error below 1.0001). Just for comparison, using a *random order* leads to considerably worse approximation factors (e.g., 65 vs. 2.75 and 2.90 for the example mentioned above).

For our LP-CH we used the worstNext approach due to its quicker construction times compared to bestNext. We only considered shortcuts with 10 or more vectors for the ordering approach. There were less than $200,000$ such edges for all d. The results are given in Table 4. The additional preprocessing time is less than the time to compute the LP-CH in the first place. We see that already a very small set of vectors represent all vectors well, illustrated by the number of vectors necessary to guarantee an approximation factor of less than 1.001. We observe that in practice, the approximation factor is often even better, as parts of the path over shortcuts with small vector sets are exact and also the approximation factor computed by our approach assumes the worst possible α, which of course might not be the actual choice of the user.

We also observe that we indeed achieve larger speed-ups even for the very tight approximation factor of $\delta = 1.001$. This is more pronounced for larger dimensions d, where we have

Table 3 Performance of ordering for random and real-world vector sets of size $k = 130$ (averaged over 10 runs each).

	$d = 5$, random		$d = 10$, random		$d = 10$, real-world	
	worstNext	bestNext	worstNext	bestNext	worstNext	bestNext
err_2	4894.6100	918.6600	319.2000	159.8840	4.0977	1.8349
err_4	739.4000	141.3070	86.1920	54.0640	1.2210	1.1787
err_8	37.0480	28.3280	22.4890	21.1100	1.0338	1.0227
err_{16}	6.4786	7.5151	6.2003	7.0924	1.0042	1.0037
err_{32}	2.5864	2.9134	2.7582	2.9043	1.0000	1.0000
err_{64}	1.4722	1.4895	1.6170	1.6364	1.0000	1.0000
err_{128}	1.0094	1.0093	1.0184	1.0184	1.0000	1.0000
time	18ms	916ms	23ms	873ms	11ms	990ms

Table 4 Effect of vector reordering and approximate query answering. The ordering time (in minutes) is the total time taken by worstNext to reorder vectors and compute approximation guarantees for all shortcuts with at least 10 vectors. The avg. and max until good count the number of vectors after reordering until an approximation factor of 1.001 was achieved. The speed-up is then computed as average over 100 queries with $\delta = 1.001$.

d	ordering time	avg. until good	max until good	speed-up
2	< 1	4.1	15	478
3	5	6.5	20	357
5	14	8.4	22	176
10	23	12.1	47	131

larger cost sets per vector and hence save more by restricting us to looking at only a few of the vectors. So accepting only slightly suboptimal result, we achieve a speed-up of over 100 for $d = 10$ which was not possible before. We want to emphasize, that the approximation factor δ can be chosen *per query*, hence even faster query times can be achieved by relaxing δ, or more precise results at the cost of an increased running time.

5 Conclusions and Future Work

We introduced a new CH variant for the personalized route planning problem which allows to deal with more metrics than previous methods while exhibiting manageable preprocessing times and better speed-ups than approaches based on customization. But even with our new approach, the speed-up decreases considerably for a growing number of metrics, as large vector sets have to be inspected during query answering. As a remedy, we introduced a new scheme which allows to sort vectors in a way that the maximum relative error is guaranteed to be small when only few vectors are investigated. This scheme might be of independent interest and turn out to be useful in other scenarios with complicated edge costs or edge cost functions, as e.g. in time-dependent route planning. We experimentally proved that small approximation factors already allow to decrease the query times significantly.

In future work, it should be investigated if the reordering could be applied already during the preprocessing to accelerate it and maybe also allow for a later stop in the contraction process, which in turn might lead to better query times.

References

1 Hannah Bast, Mirko Brodesser, and Sabine Storandt. Result diversity for multi-modal route planning. In *ATMOS-13th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems*, volume 33, pages 123–136, 2013.

2 Daniel Delling, Julian Dibbelt, Thomas Pajor, Dorothea Wagner, and Renato F. Werneck. Computing multimodal journeys in practice. In *Proc. 12th International Symposium on Experimental Algorithms (SEA)*, pages 260–271. Springer, 2013.

3 Daniel Delling, Andrew V. Goldberg, Moises Goldszmidt, John Krumm, Kunal Talwar, and Renato F. Werneck. Navigation made personal: Inferring driving preferences from GPS traces. In *Proc. 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems*, page 31. ACM, 2015.

4 Daniel Delling, Andrew V. Goldberg, Thomas Pajor, and Renato F. Werneck. Customizable route planning. In *Proc. 10th International Symposium on Experimental Algorithms (SEA)*, pages 376–387. Springer, 2011.

5 Daniel Delling, Moritz Kobitzsch, and Renato F. Werneck. Customizing driving directions with GPUs. In *Proc. 20th European Conference on Parallel Processing (EuroPar)*, pages 728–739. Springer, 2014.

6 Daniel Delling and Renato F. Werneck. Faster customization of road networks. In *Proc. 12th Int. Symposium on Experimental Algorithms (SEA)*, volume 13, pages 30–42. Springer, 2013.

7 Julian Dibbelt, Ben Strasser, and Dorothea Wagner. Customizable contraction hierarchies. In *Proc. 13th Int. Symposium on Experimental Algorithms (SEA)*, pages 271–282, 2014.

8 Julian Dibbelt, Ben Strasser, and Dorothea Wagner. Fast exact shortest path and distance queries on road networks with parametrized costs. In *Proc. 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems*, page 66. ACM, 2015.

9 Stefan Funke, Sören Laue, and Sabine Storandt. Deducing individual driving preferences for user-aware navigation. In *Proc. 24th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, page 14. ACM, 2016.

10 Stefan Funke, André Nusser, and Sabine Storandt. On k-path covers and their applications. *Proceedings of the VLDB Endowment*, 7(10), 2014.

11 Stefan Funke and Sabine Storandt. Polynomial-time construction of contraction hierarchies for multi-criteria objectives. In *Proc. 15th Meeting on Algorithm Engineering and Experiments, ALENEX 2013*, pages 41–54, 2013.

12 Stefan Funke and Sabine Storandt. Personalized route planning in road networks. In *Proc. 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems*, page 45. ACM, 2015.

13 Robert Geisberger, Moritz Kobitzsch, and Peter Sanders. Route planning with flexible objective functions. In *Proc. 12th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 124–137, 2010.

14 Robert Geisberger, Peter Sanders, Dominik Schultes, and Christian Vetter. Exact routing in large road networks using contraction hierarchies. *Transportation Science*, 46(3):388–404, 2012.

15 Andrew V. Goldberg and Chris Harrelson. Computing the shortest path: A search meets graph theory. In *Proc. 16th Annual ACM-SIAM Symposium on Discrete algorithms*, SODA'05, pages 156–165, Philadelphia, PA, USA, 2005. Society for Industrial and Applied Mathematics.

16 Martin Grötschel, László Lovász, and Alexander Schrijver. The ellipsoid method and its consequences in combinatorial optimization. *Combinatorica*, 1(2):169–197, 1981.

17 Raimund Seidel. Linear programming and convex hulls made easy. In *Proc. 6th Ann. Symp. on Computational Geometry (SCG)*, pages 211–215, New York, NY, USA, 1990. ACM.