Route Planning for Bicycles — Exact Constrained Shortest Paths Made Practical Via Contraction Hierarchy

Sabine Storandt
Institut für formale Methoden der Informatik
Universität Stuttgart
70569 Stuttgart, Germany
sabine.storandt@fmi.uni-stuttgart.de

Abstract

We consider the problem of computing shortest paths subject to an additional resource constraint such as a hard limit on the (positive) height difference of the path. This is typically of interest in the context of bicycle route planning, or when energy consumption is to be limited. So far, the *exact* computation of such *constrained shortest paths* was not feasible on large networks; we show that state-of-the-art speed-up techniques for the shortest path problem, like *contraction hierarchies*, can be instrumented to solve this problem efficiently in practice despite the NP-hardness in general.

Introduction

Large detours are not desired when planning a trip from A to B by bicycle. On the other hand, one is certainly willing to ride a few extra kilometers if this saves a hard climb (see Figure 1 for a small example). Similarly, a vehicle/robot might save some energy if a path avoiding steep climbs is chosen instead of the shortest path. This gives rise to two natural optimization problems:

1. *Find the route from A to B with the least (positive) height difference (summed over all segments) which has length at most D.*
2. *Find the route from A to B which is shortest among all paths which have height difference of at most H.*

In practice one might choose the distance limit D for example as 1.2 times the shortest path distance, or the height difference limit H as 1.5 times the path of minimal height difference (both of which can be easily computed using standard shortest path algorithms). Unfortunately, these optimization problems are incarnations of the *constrained shortest path problem (CSP)* which is NP-hard in general. In this paper we will show that state-of-the-art speed-up techniques for the shortest-path problem like *contraction hierarchies (CH)* (Geisberger et al. 2008) can be instrumented to solve such problems very efficiently in practice.

In recent years several speed-up techniques, developed originally for the one-to-one shortest path problem, found their way into more sophisticated applications. For example, it was shown that SHARC (Bauer and Delling 2009) is also very useful to identify pareto paths (Delling and Wagner 2009), contraction hierarchy can be used for route planning scenarios with flexible objective functions (Geisberger, Kobitzsch, and Sanders 2010) and both can be applied to networks with time-dependent edge costs (Delling 2008), (Batz et al. 2009). All these papers, except (Delling and Wagner 2009), are concerned with routing problems, which can be solved efficiently in *polynomial* time, and query answering is always based on Dijkstra's algorithm (or a modification thereof).

Copyright © 2012, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

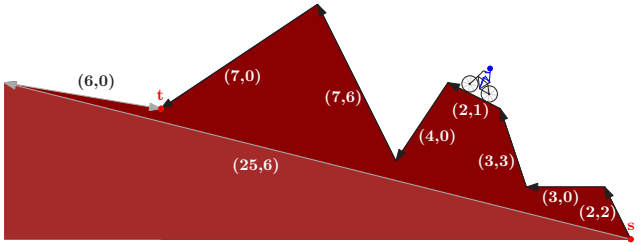

Figure 1: Example of a CSP-instance. Tuples contain the length and the positive height difference of an edge. While the upper path (black) from s to t is shorter, the lower path (gray) bears less climbs. Hence it depends on the choice of the resource bound, which of them is declared optimal.

One method to solve CSP exactly – the *label setting algorithm* – is based on the same concept as Dijkstra's algorithm. Like in Dijkstra's algorithm, labels get stored in a priority queue and every time we extract the label with minimal key value, we go through the list of outgoing edges of the respective node and update their target node labels if an improvement is possible. In addition, both algorithms allow for storing predecessors along with the labels and hence the optimal path can be found efficiently by backtracking. Moreover, like Dijkstra's algorithm, the label setting can be performed in a bidirectional manner. Due to these similarities, Dijkstra-based speed-up techniques like CH and arc-flags seem to be applicable to the label setting algorithm.

As these preprocessing methods extract a sparse subgraph for query answering without compromising optimality, they promise significantly decreased run time as well as space consumption during a label setting computation.

Related Work

Due to its relevance for real-world applications, various approaches have been developed to solve the CSP problem exactly or with an approximation guarantee. Common methods are label setting and label correcting algorithms, as well as dynamic programming. CSP can also be formulated as an ILP, giving rise to relaxations combined with gap closing algorithms. Moreover, path ranking and enumeration algorithms have been applied. We refer to (Muhandiramge and Boland 2009) and (Mehlhorn and Ziegelmann 2000) for a more detailed overview.

In (Muhandiramge and Boland 2009) the authors introduce the Aggressive Edge Elimination(AEE) procedure, which is based on Lagrangian relaxation techniques. With the help of AAE they could solve instances previously known as intractable, and tackle rather large networks (up to about 2.2 million nodes and 6.7 million edges). In contrast to our scenario, they do not focus on street networks, but on grid-like graphs in their experimental evaluation. Here simple pruning strategies were proven to be very effective, allowing to reduce the number of edges to only a few thousand. We will show, that simple pruning has a significantly lower impact on our inputs. Moreover contraction hierarchy takes advantage of the (hierarchical) structure of street networks, therefore a comparison of the two approaches on the same set of inputs seems difficult. The paper of (Köhler, Möhring, and Schilling 2005) also explores the idea of adapting speed-up techniques for conventional shortest path computations to the CSP scenario. The authors show, that goal-directed search leads to a significant speed-up in street networks. Unfortunately they restrict their evaluations to a comparatively small graph (Berlin, about 12000 nodes). It is not clear whether the speed-up translates to large networks. As our approach extracts a sparse subgraph for query answering, it can be seen as a basis for applying several other speed-up techniques – including goal-directed search – on top.

In (Geisberger, Kobitzsch, and Sanders 2010) the authors also consider two edge metrics. Here the goal is to find the optimal path wrt to a linear combination of the two edge metrics (with the linear combination being revealed at query time only). The authors show how CH can be adapted for this scenario to achieve very fast query answering. Note that this is a subproblem of CSP, which can be solved in polynomial time. On the other hand, finding all pareto-paths in a network, as described in (Delling and Wagner 2009), is also NP-hard. In this context, several metrics are considered, and a path is declared pareto-optimal if it is superior to all other possible paths with respect to at least one of the metrics. The authors use a modified version of SHARC to speed-up such queries, but have to restrict themselves to a subset of all solutions in order to cope with space consumption.

Contribution

In this paper we show that Dijkstra-based speed-up techniques can also be used to reduce the query time and space consumption for instances of CSP. We describe in detail how CH can be modified to maintain all pareto-optimal solutions in a given street network. We propose several approaches to decide whether a shortcut is necessary, allowing to trade reduced preprocessing time for graph sparseness. Apart from answering queries in the CH-graph with the label setting algorithm, we also outline how the dynamic programming approach can take advantage of CH. Additionally, we introduce a modified version of arc-flags that allows to speeding up CSP queries on its own or in combination with CH. Experimental evaluations show that our methods can solve bicycle route planning problems exactly in networks with millions of nodes and edges.

Preliminaries

We are given a digraph $G(V, E)$, a cost function $c : E \to \mathbb{R}_0^+$ and a resource consumption $r : E \to \mathbb{R}_0^+$ on the edges. In a query we are given the source and destination node, s and t, as well as the budget or resource bound R. We want to determine the minimal cost path from s to t, whose resource consumption does not exceed R. With $c(p) = \sum_{e \in p} c(e)$ and $r(p) = \sum_{e \in p} r(e)$ we refer to the cost and resource consumption of a path p. We say, that a v-w-path p dominates another v-w-path p' if $c(p) \leq c(p')$ and $r(p) \leq r(p')$. We call a v-w-path p pareto-optimal, if there exists no dominating path for p. Moreover, we can represent each tuple $(c(p), r(p))$ as line segment $\lambda c(p) + (1-\lambda) r(p),\ \lambda \in [0, 1]$. A v-w-path p lies on the lower convex hull (LCH) of (v, w), if there exists a $\lambda \in [0, 1]$ for which $\lambda c(p) + (1 - \lambda) r(p)$ is minimal among all v-w-paths.

In this paper we will focus on the problem of finding a shortest or most energy-efficient path, while restricting the total sum of climbs. To that end we are also given an elevation function $h : V \to \mathbb{Z}$. Based on that the resource consumption of an edge $e = (v, w)$ can be defined as follows: $r(e) = \max(h(w) - h(v), 0)$.

Dynamic Programming

As proposed in (Joksch 1966), we can solve the CSP problem using a dynamic programming (DP) approach: Let $c_{i,w}$ denote the minimal costs for an $s - w$-path with a resource consumption smaller or equal to i. It can be computed recursivly using

$$c_{i,w} = \min\{c_{i-1,w}, \min_{e=(v,w) \in E} \{c_{i-r(e),v} + c(e)\}\}$$

and initial values $c_{0,s} = 0, c_{0,w} = \infty$ if $w \neq s$ and $c_{i,w} = \infty$ if $i < 0$. As we are interested in $c_{R,t}$, we have to store all $c_{i,w}, i = 0, \cdots, R, \quad w = 1, \cdots, n$ in the dynamic programming table, leading to a space consumption of $\mathcal{O}(nR)$ and a runtime of $\mathcal{O}(mR)$. Observe that this approach can only be applied if the resource consumption for all edges is really greater than zero. As this is not the case in our application, we have to use an alternative dynamic programming formulation: Let $r_{i,w}$ be the minimal resource consumption of an $s - w$-path with costs equal to i. We can compute $r_{i,w}$ as follows:

$$r_{i,w} = \min_{e=(v,w) \in E} \{r_{i-c(e),v} + r(e)\}$$

The optimal solution then equals $r_{i,t} \leq R$ with i being minimal. Therefore the computation stops at $i = L_{OPT}$, leading to a to pseudopolynomial runtime of $\mathcal{O}(mL_{OPT})$. This

formulation can be extended to a PTAS using scaling and rounding of the edge costs, see (Hassin 1992).

Labeling

The label setting (LS) method, introduced by (Aggarwal, Aneja, and Nair 1982) in the context of minimum spanning trees with constraints, can be viewed as a variant of the DP approach, but has the advantage of not expanding dominated paths. LS assigns to each node v the list of all pareto-optimal tuples $(c(p), r(p))$ for an s-v-path p. This can be achieved by using an approach that adopts the idea of Dijkstra's algorithm for computing shortest paths. Here, we store labels in a priority queue (PQ). A label can be seen as triple consisting of a node ID, cost and resource consumption. The PQ sorts the labels in the increasing order of costs. We start with the PQ containing only the label $(s, 0, 0)$. In every round we extract the label with minimal cost and check for the respective node v if any of its outgoing edges $e = (v, w)$ leads to a new pareto-optimal solution (c, r) for w. If this is the case, we push (w, c, r) into the PQ. If (c, r) dominates any solution that was already assigned to w the dominated solution gets pruned.

Knowing all pareto-optimal solutions assigned to t after termination, we can easily extract the cheapest one that does not exceed the maximal allowed resource consumption R. Of course if we do not push any labels into the PQ whose resource value exceed R, we can stop when t is popped out of the PQ for the first time.

In the bidirectional version of the label setting computation (LSC) we also run a backwards search from t simultaneously. Whenever the two search networks meet or a new label is assigned to a meeting node we select the best possible combination of an upward and a downward label, i.e. the one with minimal summed costs, that is not dominated by any other combination and the summed resource consumption does not exceed R. If the cost of the selected combination is lower than the previous cost bound C, we found a new upper bound on cost of our optimal path, which can be used to prune the remaining search space even more. As soon as both PQs become empty, the optimal path has been found, namely the one that is responsible for the minimal C. While LS often outperforms DP in practice, the theoretical runtime and space consumption are the same.

Simple Pruning

The first attempt to reduce the graph size without compromising correctness was based on resource labels (Aneja, Aggarwal, and Nair 1983). Here the resource label r_{min} of a node v is the minimal resource consumption of an s-t-path, that visits v. Obviously, all nodes with $r_{min}(v) > R$ can never be on a feasible path, and hence these nodes as well as their adjacent edges can be excluded a priori. Checking this condition for all nodes in G can be done very efficiently by running two conventional Dijkstra computations on the resource consumption starting in s, and t (on the reversed graph). Aftwerwards we sum the two resulting labels $r_s(v)$ and $r_t(v)$ for each node to receive $r_{min}(v)$. Nodes with a single label already exceeding R do not need to be pushed into the respective priority queue.

If we keep the labels r_s and r_t for all feasible nodes, we can extend this pruning method to minimize the number of polls (extract min operations of the priority queue) during the label setting. Namely, we only push a label (w, c, r) into the PQ during the forward phase if $r + r_t(w) \leq R$ (analogously $r + r_s(w) \leq R$ in the backward run).

Note that if the allowed resource bound R is rather large or – as in our case – there exist many edges with a resource consumption of zero, simple pruning might not eliminate enough nodes and edges to obtain a subgraph on which queries can be answered efficiently in practice. Hence in the following we will introduce further pruning techniques based on the idea of contraction hierarchy and arc-flags.

Advanced Pruning with Contraction Hierarchy

To reduce the runtime, we want to exclude as many nodes and edges as possible a priori and hence thin out the search space for the labeling algorithm. Conventional shortest path computations methods like contraction hierarchy or arc-flags achieve exactly this. Therefore, we will show in the following how these techniques can be adapted to our scenario with resource constraints.

Conventional CH

The contraction hierarchy (CH) technique was introduced in (Geisberger et al. 2008) and leads to a remarkable speed-up for Dijkstra computations.

In a preprocessing phase an importance value is assigned to each node, and then the nodes are sorted in the increasing order of importance. Afterwards the nodes get removed/contracted one by one in that order, while preserving all shortest path distances among the remaining nodes by inserting additional edges (so called shortcuts). More precisely, an edge $e = (u, w)$ is added, when contracting a node v, if u and w are adjacent to v, and the shortest path from u to w is uvw. The costs of e result from the chained costs of the edges (u, v) and (v, w). If the shortest path does not equal uvw, then we found a witness path testifying that the shortcut can be omitted. After all nodes have been removed, we build a new graph G' consisting of all nodes and edges of the original graph and all shortcuts. We call an edge $e = (v, w)$ (original or shortcut) upwards if the importance of v is smaller than that of w and downwards otherwise. In G' s-t-queries can be answered by a bidirectional Dijkstra computation, with the forward run (starting at s) considering only upward edges and the backward run (starting at t) considering exclusively downward edges. We call the respective sets of edges *the upward/downward graph induced by s / t*. This strategy prunes the respective search spaces dramatically and leads to a speed-up of more than two orders of magnitude for query answering.

Node Ordering

To decide in which order the nodes get contracted several heuristics have been developed, evaluating the importance of a node. As one goal is to keep the resulting graph as sparse as possible, the *edge-difference* (Geisberger et al. 2008)

seems promising. The edge-difference of a node v is the actual number of shortcuts we have to add minus the number of edges that can be removed when contracting v. Also, weighted versions of edge-difference have been evaluated, penalizing addition of shortcuts even more, see (Geisberger, Kobitzsch, and Sanders 2010).
In our case nodes seem more important if they belong to cost-optimal paths with low total resource consumption. Therefore we also use a weighted edge-difference, but break ties by first contracting nodes whose incoming edges have a high resource consumption.

Witness Search for Constrained Shortest Paths

Like every subpath of a shortest path has to be a shortest path itself, every subpath of a pareto-optimal path has to be pareto-optimal. Hence we have to maintain all pareto-optimal paths on a local level to guarentee correct query answering in the CH-graph. Therefore, while contracting a node v, we can avoid adding a shortcut between two of its neighbors u and w only if we find an alternative path from u to w which dominates the reference path $p = uvw$.

A straightforward way to find such a dominating witness is starting a LSC in u with a resource bound $R = r(p)$. As soon as w pops out of the PQ the first time, we stop the LSC and check if the label's cost exceeds $c(p)$. If this is the case, p is pareto-optimal and hence the shortcut $sc = (u, w)$ with $c(sc) = c(u, v) + c(v, w)$ and $r(sc) = r(u, v) + r(v, w)$ is needed for sure. Otherwise we found a dominating path and therefore the shortcut can be omitted. Of course, we can already abort the LSC if w is assigned a label that dominates the label of the reference path. Note that for conventional shortest paths there can be at most one shortcut (v, w) for any pair of nodes $v, w \in V$ in G'. On the other hand, in our scenario there might be as many shortcuts as there are pareto-optimal paths in G between v and w.
Unfortunately LSC might be too time-consuming to apply to every pair of neighboring nodes in every contraction, even if we apply simple pruning first. Because of that, we now propose a procedure which can help to avoid some of these computations:

The basic idea is to first restrict ourselves to paths on the lower convex hull (LCH) of all paths. If p is a part of the LCH, there can exist no dominating path and so we have to insert $sc = (u, w)$. On the other hand if there is a dominating path p', it is likely (not required!), that $p' \in LCH$. To get the candidate paths on the LCH we introduce the parameter $\lambda \in [0, 1]$. With G^λ we refer to the graph, that consists of the same vertices and edges as the original one, but has only one weight assigned to each edge, namely $\lambda c(e) + (1 - \lambda)r(e)$. In G^λ we can identify an optimal path between two nodes using plain Dijkstras algorithm. If we extract this path and evaluate it in G we can easily check if this path dominates p or is equal to p. Otherwise we can retry using another value of λ.
In practice we want to restrict ourselves to a small set of support points $\lambda_1, \cdots, \lambda_t$. Of course, we could choose these values randomly or sample the interval $[0, 1]$ uniformly, but in both cases many λ_i might lead to the same

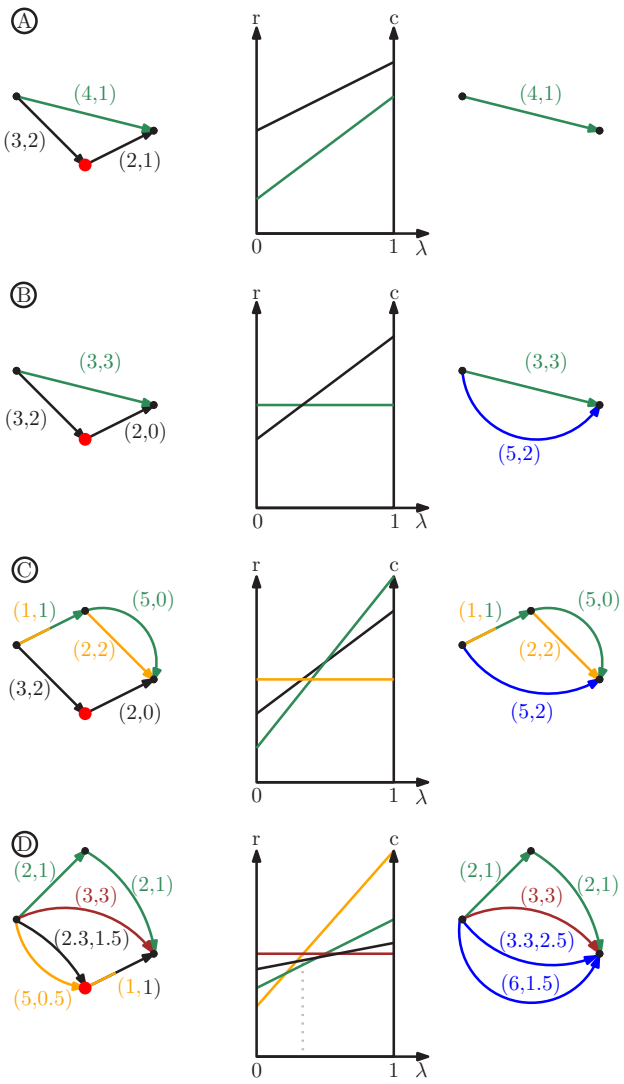

Figure 2: Examples for contracting a node (large red mark). The resulting graph is given on the right. In (A) the reference path (black) is dominated by the green path. The latter describes the complete LCH, therefore a single Dijkstra run in G^λ with $\lambda = 0$ or $\lambda = 1$ is sufficient to make sure that the respective shortcut for the black path can be omitted. In (B) the green path does not dominate the black one. Instead, the reference path is a part of the LCH, as a Dijkstra run with $\lambda = 0$ will reveal. Hence the shortcut must be inserted (indicated by the blue edge on the right side). In (C) the shortcut is needed as well, but the exploration of the LCH will be inconclusive, as the reference path is neither a part of the LCH nor dominated by the ones that are. In (D) the black path is again a part of the LCH. Using $\lambda_1 = 0$ and $\lambda_2 = 1$ we will discover the orange and the brown path respectively. The λ-value of their intersection point will lead us then to the green path as indicated by the dotted gray line. Hence our checker needs more than three support points for a conclusive result.

resulting path. We can do better if we search for the support points systematically. To that end we start with $\lambda_1 = 0$ and $\lambda_2 = 1$, i.e. we compute the resource-minimal path p_1 and the cost-minimal path p_2. If we are not done yet, we set λ_3 to the λ-value of the intersection point of $g(p_1)$ and $g(p_2)$ with $g(p_i) = (c(p_i) - r(p_i))\lambda + r(p_i)$, namely $\lambda_3 = \frac{r(p_2)-r(p_1)}{c(p_1)-c(p_2)+r(p_2)-r(p_1)}$. The new path p_3 – which is optimal for λ_3 – will either be the same as p_1 or p_2 or will have an intersection point with both $g(p_1)$ and $g(p_2)$. If the former is the case, we have explored the complete LCH. Otherwise the two intersection points are candidates for further support points, therefore we push them onto a stack Q. For every further λ_i extracted from Q we proceed similarly. As soon as Q becomes empty, we have explored the complete LCH and are done. Moreover, we can fix a maximal value for t, restricting the number of possible Dijkstra computations a priori. Figure 2 shows some examples that illustrate the connection between the witness search and the check procedure.
Of course, the Dijkstra computations in G^λ can also be sped up by simple pruning. To that end we just have to store and update the cost and resource consumption for a node simultaneously with the transformed cost label. Observe, that the simple pruning is independent of the choice of λ and hence has only to be done once for every pair of nodes.

If the checker does not provide a conclusive result, we can either start the LSC on top, or add the shortcut $sc(u, w)$ without further investigations in order to reduce the preprocessing time. Of course, without the LSC we might add some superfluous shortcuts. Note that this does not compromise correctness, but can lead to worse query times and increased space consumption.

Observe that in the resulting CH-graph G' – consisting of all original nodes and edges as well as all shortcuts – we can now answer *both* kinds of queries mentioned in the introduction more efficiently, i.e., we can either restrict the distance and ask for the path with the minimal positive height difference or we could also restrict the latter and compute the shortest path fulfilling the height constraint. Note, that this is a direct consequence of omitting shortcuts only if a dominating path can be found.

Answering queries

To answer an s-t-query in the constructed CH-graph, we only have to consider edges that lie in the upward graph $G^\uparrow$ induced by s or in the downward graph $G^\downarrow$ induced by t *and* are adjacent to a feasible node according to the simple pruning with resource labels. Therefore we can also speed up the required Dijkstra computations for the simple pruning by considering only edges in $G^\updownarrow = G^\uparrow \cup G^\downarrow$. If t does not receive a feasible resource label in this step, there exists no path from s to t fulfilling the resource constraint and we are done. Note that, as a nice side effect of simple pruning in $G^\updownarrow$, all nodes that do not lie on any path from s to t receive a resource label of ∞ automatically. Hence they will not be considered anymore, pruning the search space additionally. Also we run a (bidirectional) Dijkstra computation in $G^\updownarrow$ from s to t, now considering the edge costs. If the resulting path is feasible wrt to its resource consumption, we found the optimal solution straight away. Otherwise futher computations are required:
The *bidirectional* version of the *LS* can be easily modified to take advantage of CH. The only difference is that the forward run from s considers only upward edges, while the backward run from t considers only downward edges. Of course we do not push any nodes into the PQ, that were declared infeasible in the simple pruning step.

For combination with other speed-up techniques it might be desirable to answer queries in a *unidirectional* manner. For that purpose we start a breadth-first search (BFS) at t to mark all downward inedges recursively, omitting edges that are adjacent to infeasible nodes. Afterwards we can start a conventional LSC at s that runs on upward and marked edges. The first time t is popped out of the PQ, the optimal label is found.

Up to now we only outlined how CH can be useful to answer queries with LS more efficiently. But we can also use this preprocessing to reduce space consumption and runtime for the *dynamic programming* approach. To that end we perform a BFS not only from t on downward edges, but also from s to mark all possible upward edges. Afterwards we extract the subgraph by keeping only marked edges and their adjacent nodes (if they are feasible).

Extension: Pruning wth Arc-Flags

Arc-flags (Lauther 1997) are based on the observation that the set of all shortest paths from one region to another (far away) region contains only a very few edges. This is derived from the fact that long drives in a certain direction require almost always the usage of a specific highway or autobahn. Therefore the idea behind arc-flags is to identify this set of useful edges a priori and explore only these during query answering.
To assign arc-flags, the nodes are first partitioned $V = P_1 \uplus P_2 \uplus \cdots \uplus P_l$. Then a list of boolean flags is assigned to each edge, one flag for every partition P_i. This flag will be set true if the edge lies on a shortest path $\pi(v, t)$ with $v \in V$ and $t \in P_i$. While answering a query, we only have to consider edges marked true for the partition in which the target lies. Naively, we would have to compute the shortest path between any pair of nodes to determine all arc-flags. But even for small graphs this is too time-consuming to be pratical. Using the observation that any path between two nodes v, w with $v \in P_i, w \in P_j, P_i \neq P_j$, has to go through at least one node on the boundary of P_i and P_j respectively, we can restrict ourselves to shortest paths between boundary nodes. Of course, with this approach the flag for edges with their source and target in the same partition P_i has to be set true for P_i, in order to still allow queries to any node.

For CSP the number of edges on pareto-optimal paths between two regions is suprisingly small (see Figure 3), but of course larger than the number of edges on shortest paths only. In order to further reduce the number of important

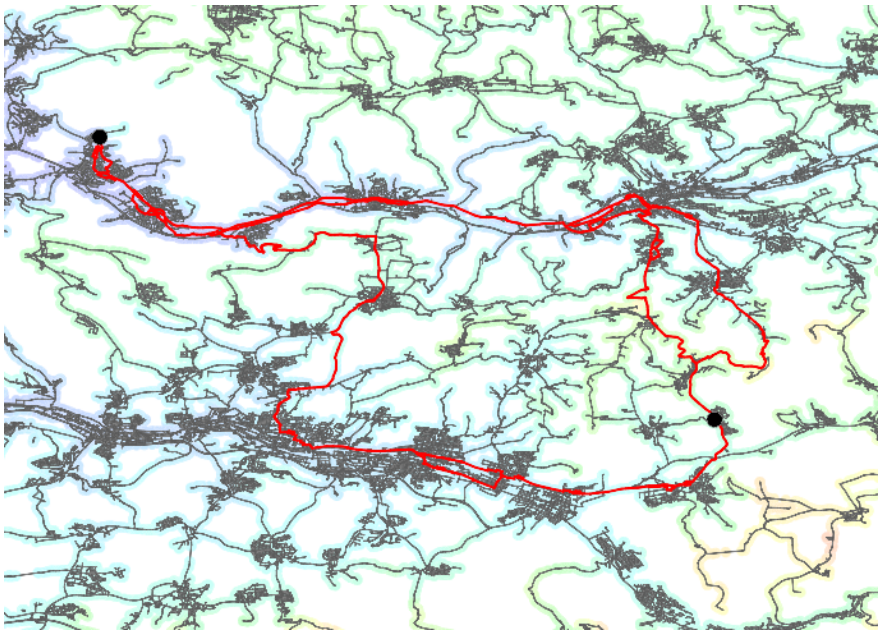

Figure 3: Pareto-optimal paths (red) between source (top left) and destination node (bottom right). Node colours indicate elevation (from blue-243 m up to red-786m). While there are 87 different pareto-optimal solutions, the total number of used edges is rather small.

edges for a query, we introduce the concept of resource-dependent arc-flags. To that end we divide the range of possible bounds R into intervals $[0, R_1], (R_1, R_2], \cdots (R_k, \infty]$. Now we split the edge flag for a certain partition P_i in k flags. Such an interval flag for $(R_{i-1}, R_i]$ is true iff a pareto-optimal path to a node $t \in P_i$ starts starting with this edge has a resource consumption of at most R_i. Using again only boundary nodes to determine arc-flags, we set all interval flags for the partition that contain the edge to true.
Answering a query changes slightly: now we consider only edges $e = (v, w)$ if the resource consumption of the actual label assigned to v lies in a true flagged interval for the target's partition.

Combination

Combining arc-flags with contraction hierarchy promises an even smaller search space and therefore better query times. Computing the CH first, and assigning arc-flags to all edges including shortcuts afterwards is very time-consuming, because adding shortcuts also increases the number of nodes that are located on partition boundaries. Moreover, the edge flags of shortcuts can be easily derived if the edge flags for the original graph are known previously. Conventionally, a shortcut that skips two edges e_1, e_2 can only have a true edge flag for partition P_i if both respective edge flags of e_1 and e_2 are true. Therefore, conventional arc-flags for shortcuts can be derived by applying the bitwise and-operator to the vectors of edge flags for e_1, e_2. Using resource-dependent arc-flags we set the shortcut's interval flag for $(R_{i-1}, R_i]$ to true if the according edge flag of e_1 is true and the flag of e_2 for the interval containing $R_i - r(e_1)$ is true as well.

Experimental Results

In this section we evaluate the impact of the introduced speed-up techniques on real-world instances. We used five test graphs (named according to the number of their nodes 10k, 100k, 500k, 1m, 5.5m), which are all cutouts of the street network of Germany, based on OpenStreetMap[1] data and augmented with SRTM (Shuttle Radar Topography Mission)[2] height information. Distances and elevations were used with a precision of 1m. The average path lengths and the average positive height differences for all test graphs can be found in Table 1. Our implementations are written in C++. Preprocessing times were taken on a 2-core, Intel Core i3-2310M CPU with 2.10GHz and 8 GB RAM. Query times were taken on a single core.

	10k	100k	500k	1m	5.5m
shortest path					
avg length	5618	28066	50102	77599	197334
avg height diff	189	715	1164	1677	4038
minimal height difference path					
avg length	6925	30938	70657	98333	287461
avg height diff	116	553	766	1179	2116

Table 1: Characteristics of the test graphs: Average path length as well as height difference for shortest paths and for paths with minimal height difference. All values are given in meters and are averaged over 1000 random queries.

We preprocessed all the test graphs using contraction hierarchy – once in the conventional way, considering only costs and aiming to maintain all shortest paths (SP), and once using the new variant for CSP. For the latter we used the described check procedure (considering first paths on

[1] www.openstreetmap.org/
[2] www2.jpl.nasa.gov/srtm/

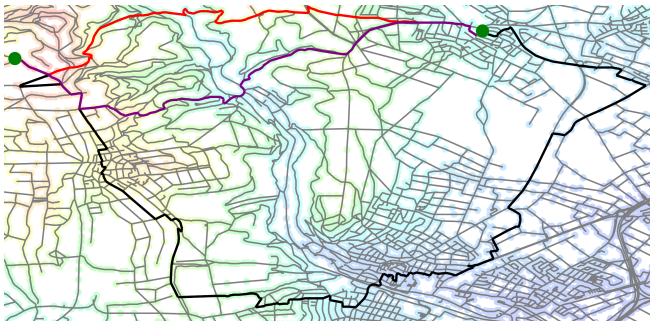

Figure 4: Example of a shortest path (top, red, distance 7.5km, height diff. 517m) and the respective path with minimal height difference (bottom, black, distance 19.1 km, height diff. 324m), which makes a very large detour. The shortest path under the constraint of having a height difference smaller than $1.5 \cdot 324\text{m} = 486\text{m}$ is a fair compromise (middle, purple, distance 7.7km, height diff. 410m).

the lower convex hull) with three support points. Based on that we could avoid about 62% of the local label setting computations. We restricted ourselves to contracting 99.5% of the nodes in each graph, in order to achieve reasonable preprocessing times for the CSP-CH (4 seconds for the 10k graph, about 2h for the 5.5m graph). Edges between uncontracted nodes were declared downward edges. The total number of edges in the final CH-graphs can be found in Table 2 for all five graphs. Surprisingly, the total number of edges in the CSP-CH graph is only about $3 - 10\%$ larger than the number in the SP-CH graph. Hence we also have only about twice the number of original edges in the CH-graph. This might partly result from the fact that the positive height difference is bounded in the path length, as one can not achieve to gain 10 meters in height without riding for at least 10 meters at the same time. Moreover, the large number of edges with zero resource consumption is beneficial here, as finding a witness for a path without any resource consumption simply reduces to finding a path with smaller costs, which is equivalent to the conventional scenario.

		edges		
graph	nodes	original	SP-CH	CSP-CH
10k	11220	24119	50641	54383
100k	100242	213096	407034	419210
500k	500011	1074458	2090628	2160438
1m	999591	2131490	4150204	4276478
5.5m	5588146	11711088	23970043	26586530

Table 2: Number of nodes and edges for the used test graphs. 'Original' describes the number of edges in G before applying CH. The last two columns show how conventional CH for shortest paths (SP-CH) and CH for CSP augments the set of edges.

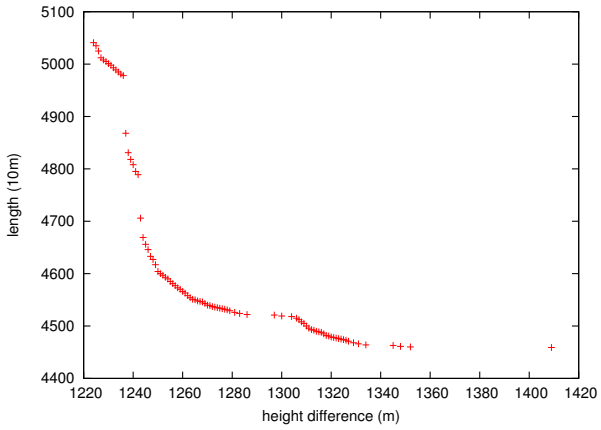

Figure 5: All 91 pareto-optimal labels (consisting of path length and corresponding height difference) assigned to a single node during the label setting algorithm in the 100k graph.

We measured query times and polls for the unidirectional label setting algorithm. At first we picked a source and a target vertex s, t randomly, and computed the path with the minimal positive height difference H. Then we started a LSC with a resource bound of $1.5H$ and aimed for the shortest path fulfilling this constraint (see Figure 4 for a an example in the 10k graph).
The results can be found in Table 3. In all test graphs the number of edges in the subgraph of the CH-graph induced by s and t is at least one order of magnitude smaller than in the original graph. With the only exception of the 5.5m graph, applying the pure CH reduced the number of edges far more than the simple pruning approach. Neverthless, in all cases the queries in the CH-graph could be answered with siginficantly less poll operations than in the feasible subgraph based on simple pruning. This is also reflected in the runtime. While pruning with resource labels only halves the runtime that was needed to answer a query in a completely unpreprocessed graph, the respective CH-query can be answered two oders of magnitude faster. The combination of both techniques leads to query times below 1 second in graphs with a size up to $5 \cdot 10^5$ nodes. The reason why for the 5.5m graph we achieve only a speed-up of 20 using CH (50 for the combination of CH and simple pruning) might be the incomplete contraction of the nodes during the preprocessing, because all of the edges between uncontracted nodes have to remain in the induced subgraph for query answering. Moreover the runtime of a LSC depends not only on the number of polls, but also on the time required to check the pareto-optimality of a label and prune dominated ones previously assigned to the respective node. The costs of these operations increase, of course, with the number of labels that are already assigned to a node. As one can see in Figure 5 exemplarily for the 100k graph this number might be very large in our scenario.
Nevertheless, reduction in the graph size was significant and it allowed us to answer all queries using only 8GB of RAM.

graph	subgraph edges naive	naive+p	CH	CH+p
10k	$2.4 \cdot 10^4$	$7.5 \cdot 10^3$	$2.8 \cdot 10^3$	$1.2 \cdot 10^3$
100k	$2.1 \cdot 10^5$	$1.0 \cdot 10^5$	$5.2 \cdot 10^3$	$2.9 \cdot 10^3$
500k	$1.1 \cdot 10^6$	$4.6 \cdot 10^5$	$3.6 \cdot 10^4$	$1.2 \cdot 10^4$
1m	$2.1 \cdot 10^6$	$5.3 \cdot 10^5$	$8.1 \cdot 10^4$	$3.4 \cdot 10^4$
5.5m	$1.2 \cdot 10^7$	$7.0 \cdot 10^5$	$1.8 \cdot 10^6$	$1.6 \cdot 10^5$
graph	number of polls naive	naive+p	CH	CH+p
10k	$3.6 \cdot 10^4$	$1.2 \cdot 10^4$	$4.9 \cdot 10^2$	$2.3 \cdot 10^2$
100k	$1.4 \cdot 10^6$	$6.5 \cdot 10^5$	$8.1 \cdot 10^3$	$4.9 \cdot 10^3$
500k	$8.9 \cdot 10^6$	$6.7 \cdot 10^6$	$8.9 \cdot 10^4$	$7.4 \cdot 10^4$
1m	$2.9 \cdot 10^7$	$6.8 \cdot 10^6$	$3.4 \cdot 10^5$	$9.1 \cdot 10^4$
5.5m	$9.2 \cdot 10^7$	$3.7 \cdot 10^7$	$1.7 \cdot 10^6$	$8.7 \cdot 10^5$
graph	query time (s) naive	naive+p	CH	CH+p
10k	0.0233	0.0117	0.0015	0.0002
100k	7.5942	2.7301	0.1294	0.0168
500k	123.0791	105.7814	1.0438	0.9895
1m	265.6990	117.6859	5.6712	2.5879
5.5m	2369.3361	1131.1603	124.2088	64.2364

Table 3: Experimental results for answering queries with the label setting algorithm in the original graph (naive) and the CH-graph, in combination with simple pruning ('+p') and without. All values are means based on 1000 random queries. For all measurements standard deviations range between 1.2 and 1.6 times the average.

In contrast, using the naive approach, part of the queries in 1m and 5.5m failed (these were excluded from timings). Even after switching to a server with 96GB RAM we ran out of memory for some queries in the 5.5m graph. But with our new techniques, the search space reduces remarkably, and therefore enabling us to handle such queries on desktop computers or even laptops.

As outlined in the previous section, we can also take advantage of the CH-graph when answering queries using dynamic programming. Here we first have to extract the feasible subgraph to build the dynamic table upon. The numbers of nodes that remain in this subgraph (and therefore determine one of the table's dimensions) are given in Table 4. For all considered graphs the percentage of these nodes lies below 1% of the original nodes, leading to an overall reduction in table size of at least two orders of magnitude. Therefore, the average CH-table size for the 5.5m graph is now comparable to the original one for the 100k graph. Accordingly, we could again answer all queries using only 8GB og RAM, while no queries in the 1m and 5.5m graph could be answered at all in the original setting. The query times, given also in Table 4, show a speed-up by a factor of $10-30$, though the runtime for larger graphs is prohibitive for practical use, taking about one hour for a query in the 5.5m graph. But note that path lengths were computed with a very high precision of one meter, and as the value of the optimal solution determines the second dimension of the dynamic table, this leads to a large number of necessary table entries.

graph	nodes CH	table size naive	CH	query time (s) naive	CH
10k	100	$6.5 \cdot 10^7$	$5.6 \cdot 10^5$	13.7	1.9
100k	412	$2.9 \cdot 10^9$	$2.9 \cdot 10^6$	106.2	11.2
500k	2508	$3.1 \cdot 10^{10}$	$1.5 \cdot 10^8$	3729.1	104.4
1m	4724	$(8.5 \cdot 10^{10})$	$4.2 \cdot 10^8$	-	498.9
5.5m	21486	$(1.3 \cdot 10^{12})$	$4.6 \cdot 10^9$	-	3573.5

Table 4: Experimental results for the dynamic programming approach. The second column gives the number of nodes in $G^{\updownarrow}$, and hence equals the first dimension of the dynamic table. Query times for the naive approach could not be measured for the graphs 1m and 5.5m, because we ran out of memory. Values in brackets are estimates based on the optimal solutions returned by the CH-variant and the number of nodes in the respective graphs. All other values are averaged over 100 random queries.

Finally we evaluated (resource-dependent) arc-flags in the CSP setting. Due to very long preprocessing time and high space consumption, we restricted ourselves to measurements on the 10k and 100k graphs.
First we divided the 10k graph into 64 partitions (using a uniform 8x8 grid). We achieved a speed-up similar to CH (without simple pruning), but used about ten times the preprocessing time. Secondly, we applied resource-dependent arc-flags to the 100k graph, using 12 partitions and three intervals with $R_1 = 400$ and $R_2 = 800$. The preprocessing took about 4 hours and considered 2576 boundary nodes. For high resource consumption bounds the number of polls was reduced by an order of magnitude in comparison to the naive approach. The combination of CH and arc-flags led to an equal number of polls as CH plus simple pruning. In addition with the help of all three of these techniques, the number of polls and the subgraph size could be halved once more.

Conclusions and Future Work

In this paper we described in detail how the speed-up technique of contraction hierarchy can be adapted to solve instances of the NP-hard CSP problem exactly and efficiently even in large street networks. This allows for faster query answering in a wide range of applications, where resource constraints play a role. On the example of the bicycle route planning problem we showed that not only the runtime but also the space consumption of a query can be reduced remarkably with the help of CH.
Future work will include further investigations to reduce the preprocessing time for CH and arc-flags. Also the combination with other heuristics, like A^*-search, might lead to improved query times and less space consumption.

Acknowledgement

This work was partially supported by the Google Focused Grant Program on Mathematical Optimization and Combinatorial Optimization in Europe.

References

Aggarwal, V.; Aneja, Y.; and Nair, K. 1982. Minimal spanning tree subject to a side constraint. In *32nd ACM Symposium on Theory of Computing (STOC)*, 286–295.

Aneja, Y. P.; Aggarwal, V.; and Nair, K. P. K. 1983. Shortest chain subject to side constraints. *Networks* 13(2):295–302.

Batz, G. V.; Delling, D.; Sanders, P.; and Vetter, C. 2009. Time-dependent contraction hierarchies. In *ALENEX*, 97–105.

Bauer, R., and Delling, D. 2009. Sharc: Fast and robust unidirectional routing. *ACM Journal of Experimental Algorithmics* 14.

Delling, D., and Wagner, D. 2009. Pareto paths with sharc. In *SEA*, 125–136.

Delling, D. 2008. Time-dependent sharc-routing. In *ESA*, 332–343.

Geisberger, R.; Sanders, P.; Schultes, D.; and Delling, D. 2008. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *WEA*, 319–333.

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

Hassin, R. 1992. Approximation schemes for the restricted shortest path problem. *Mathematical Operational Research* 17(1):36–42.

Joksch, H. 1966. The shortest route problem with constraints. *Journal of Mathematical Analysis and Application* 14:191–197.

Köhler, E.; Möhring, R.; and Schilling, H. 2005. Acceleration of shortest path and constrained shortest path computation. In *Experimental and Efficient Algorithms*, volume 3503 of *Lecture Notes in Computer Science*. Springer. 126–138.

Lauther, U. 1997. Slow preprocessing of graphs for extremely fast shortest path calculations. *Lecture at the Workshop on Computational Integer Programming at ZIB*.

Mehlhorn, K., and Ziegelmann, M. 2000. Resource constrained shortest paths. In Paterson, M., ed., *Algorithms - ESA 2000*, volume 1879 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg. 326–337.

Muhandiramge, R., and Boland, N. 2009. Simultaneous solution of lagrangean dual problems interleaved with preprocessing for the weight constrained shortest path problem. *Netw.* 53:358–381.