Route Planning with Flexible Objective Functions*

Robert Geisberger, Moritz Kobitzsch and Peter Sanders
Karlsruhe Institute of Technology, 76128 Karlsruhe, Germany
{geisberger,kobitzsch,sanders}@kit.edu

Abstract

We present the first fast route planning algorithm that answers shortest paths queries for a customizable *linear combination* of two different metrics, e. g. travel time and energy cost, on large scale road networks. The precomputation receives as input a directed graph, two edge weight functions $t(e)$ and $c(e)$, and a discrete interval $[L, U]$. The resulting *flexible* query algorithm finds for a parameter $p \in [L, U]$ an exact shortest path for the edge weight $t(e)+p\cdot c(e)$. This allows for different tradeoffs between the two edge weight functions at query time. We apply precomputation based on *node contraction*, which adds all necessary *shortcuts* for any parameter choice efficiently. To improve the *node ordering*, we developed the new concept of *gradual parameter interval splitting*. Additionally, we improve performance by combining node contraction and a goal-directed technique in our flexible scenario.

1 Introduction

In our world, shaped by globalization, transportation becomes more important every day. People and goods travel from one location to another every hour in quantities beyond imagination. This implies a huge demand for computing shortest paths, for example in road networks. This requirement has triggered a significant amount of research in algorithm engineering. Most of this effort was concentrated on the classical shortest path problem with a fixed objective function. However, today's transportation requirements must be flexible and so must be our algorithms. We contribute the first fast algorithm for flexible queries in the following sense:

Let $G = (V, E)$ be a directed graph with *two* nonnegative edge weight functions $t(e)$ and $c(e)$, e. g. travel time and energy cost. Our algorithm augments G in such a way, as to provide fast queries for a shortest path between a source and target node $s, t \in V$ and parameter $p \in [L, U] := \{x \in \mathbb{N} \mid L \leq x \leq U\}$ for an objective edge function $w_p(e) := t(e) + p \cdot c(e)$. We require $0 \leq L \leq U$ to ensure w_p is nonnegative. Note that we can choose a different parameter $p \in [L, U]$ for every query.

To allow fast queries, we apply precomputation based on *node contraction*; a node v is contracted by removing it from the network in such a way that shortest paths in the remaining graph are preserved. This property is achieved by replacing paths of the form $\langle u, v, w\rangle$ by a *shortcut* edge (u, w). Note that the shortcut (u, w) is only required if there is a $p \in [L, U]$ for which $\langle u, v, w\rangle$ is the only shortest path from u to w. The order in which the nodes are contracted is crucial; 'important' nodes should be contracted later. However, the 'importance' of a node depends on the parameter p, e. g. highways are important for travel time optimization but less important for distance optimization. So we developed *gradual parameter interval splitting* (shortened *parameter splitting*): we start to contract nodes while regarding the whole interval $[L, U]$, selecting the next node to contract only after the contraction of the previous one. When there are too many shortcuts that are not necessary for the whole interval, we split this interval into two intervals $[L, M]$ and $[M + 1, U]$. The remaining graph is contracted separately for every interval, so we can compute different node orders for the remaining nodes. So we can contract nodes multiple times, but at most once for each value of p. Of course, we can repeat the split several times, depending on the size of the interval. The shortcuts are marked with the parameter interval for which they are necessary, and the query algorithm only uses shortcuts necessary for the current value of p. Furthermore, we combined our algorithm with a goal-directed technique. This technique can either be applied to an uncontracted core, thus allowing to reduce the time needed for precomputation, or to a contracted core, thus yielding even faster query times.

Related Work There has been extensive work on single-criteria speed-up techniques. We refer to [22, 23, 3] for an overview. These techniques fix a single

*Partially supported by DFG grant SA 933/5-1, BMWi project 'MEREGIOmobil' and the 'Concept for the Future' of Karlsruhe Institute of Technology within the framework of the German Excellence Initiative.

objective function before the precomputation. The objective function can be arbitrary but the empirically best results are achieved with the travel time metric. We could obtain the same functionality as our algorithm by performing the precomputation for every value of p separately. However, this would require too much precomputation time and space.

The classic algorithm for the shortest path problem is *Dijkstra's algorithm* [6]. Starting with a source node s as root, it grows a tree that contains shortest paths from s to all other nodes. A node that already belongs to the tree is *settled*. If a node u is settled, a shortest path P^* from s to u has been found. A node that is adjacent to a settled node is *reached*. Note that a settled node is also reached. Nodes that are not reached are *unreached*.

We can classify current algorithms into three categories: *hierarchical* algorithms, *goal-directed* approaches and combinations of both. The most closely related hierarchical technique is contraction hierarchies (CH) [8]. A CH uses a single node order and contracts the nodes in this order. A slightly modified bidirectional Dijkstra shortest path search then answers a query request, touching only a few hundred nodes. For our algorithm, we adopt the basic concept of CH but add significant contributions to adapt it to our flexible objective function. Namely we had to change crucial parts of the contraction procedure and also further engineered the query algorithm.

The most successful goal-directed approaches are ALT and arc flags. The ALT algorithm [9, 11] is based on A* search [13], landmarks, and the triangle inequality. After selecting a small number of landmarks, for all nodes, the distances to and from each landmark are precomputed. For a target node, the triangle inequality yields for each landmark two lower bounds, which are used to add a sense of direction to the search process. The approach of arc flags [17, 19, 14] is, as the name suggests, to precompute for each edge 'signposts' that support the decision whether the target can possibly be reached on a shortest path via this edge.

The combination of hierarchy and goal-direction is currently the most successful approach [2]. We will show how to adapt the CALT algorithm to our flexible scenario. CALT is essentially the ALT algorithm applied to a small *core* of a hierarchical algorithm. However, ALT needs lower bounds to the source and target, even when they are not in the core. So CALT uses *proxy nodes* [10]: the proxy node of a node is the closest core node. We adapt CALT to our flexible scenario and exploit the properties of our objective function to compute lower bounds for whole intervals.

Recently, speed-up techniques for *time-dependent* road networks have been developed ([5] presents an overview). The cost of an edge depends on the arrival time at this edge, but current algorithms can only optimize the travel time. In our opinion, flexible query algorithms are at least as important as time-dependent query algorithms and should get more attention.

There is some work on using several optimization criteria simultaneously without fixing their relative importance before a query. In the classic approach all *Pareto-optimal* paths are computed, paths being better than any other path for respectively at least one criteria. Such a path P is said to dominate another path P' if P' is not better in any criterion. While this is even more flexible than our approach, it makes the shortest path problem NP-hard and also in practice generates a plethora of potential paths that then have to be pruned using additional techniques. Even then, the computational overhead is considerably larger than with our approach of fixing the objective function *after* preprocessing, but *before* each query. The most common algorithm to compute all Pareto-optimal paths is a generalization of Dijkstra's algorithm (*Pareto-Dijkstra*) [12, 18]. It does no longer have the node settling property, as already settled nodes may have to be updated again, and multiple Pareto-optimal paths to a node are represented by multi-dimensional *labels*. To the best of our knowledge, the most successful result on speed-up techniques for multi-criteria is an adaptation [4] of the SHARC algorithm [1], a combination of shortcuts and arc flags, to compute Pareto-optimal paths. However, Pareto-SHARC only works when the number of optimal paths between a pair of nodes (*target labels*) is small. Pareto-SHARC achieves this either by very similar edge weight functions or by tightening the dominance relation that causes it to omit some Pareto-optimal paths (*label reduction*). Yet, the label reduction entails serious problems since it may rule out possibly interesting paths to early, as not all subpaths of a path that fulfills those tightened domination criteria have to fulfill the criteria themselves. Therefore, the domination criteria *do not fulfill subpath optimality*. Thus, Pareto-SHARC with label reduction cannot guarantee to find all optimal paths w. r. t. to the tightened domination criteria and is therefore a heuristic. In a setup similar to ours, with two edge weight functions (travel time, cost), and without label reduction, they can only handle small networks of the size of a city. As simple label reduction, they propose to dominate a label if the travel time is more than ϵ times longer the fastest. This is reasonable but only works well for very small $\epsilon \leq 0.02$ with around 5 target labels. Even for slightly larger ϵ, the preprocessing and query times become unacceptable. Also, stronger label reduction is applied to drop the average number of target labels to 5 even for $\epsilon = 0.5$. It seems

like Pareto-optimality is not yet an efficient way to add flexibility to fast exact shortest path algorithms.

Even though the Pareto-optimal shortest path problem has attracted more interest in the past, the parametric shortest path problem [16] has been studied. However it provides less flexibility as compared to our approach. Given a value p, we are just allowed to subtract p from the weight of a predefined subset of edges. All well-defined shortest path trees (when p is too large, we may get negative cycles) can be computed in $O(nm + n^2 \log n)$ [20].

2 Contraction

Node contraction is a central point in our algorithm. When we contract a node $v \in V$, we remove it and all its adjacent edges from the graph and add shortcuts to preserve shortest path distances in the remaining graph. We thus generate a graph $G' = (V', E')$ with edge set E' that preserves shortest paths distances w. r. t. the input graph. This is accomplished by inserting shortcuts that represent the paths between adjacent nodes of v over v. All original edges together with all shortcuts are the result of the preprocessing, a flexible contraction hierarchy. As we want to keep the graph G' as sparse as possible, we face the following many-to-many shortest path problem: For each source node $u \in V'$ with $(u, v) \in E'$, each target node $w \in V'$ with $(v, w) \in E'$ and each integer parameter $p \in [L, U]$, we want to compare the shortest u-w-path Q_p with minimal $w_p(Q_p)$ with the shortcut length $w_p(u, v) + w_p(v, w)$ in order to decide whether the shortcut is really needed.

Compared to single-criteria contraction, we cannot avoid parallel edges. However, the number of parallel edges is bound by the number of possible values of p. In this paper, we keep identifying edges by their two endpoints since the particular edge should always be clear from the context. Because we use a linear combination of the two edge weight functions, we do not need to keep all Pareto-optimal weight pairs.

Witness Search. A *witness* is a path that is not longer than the potential shortcut for a specific value of p and does not lead through the currently contracted node v. Hence a witness allows to omit this shortcut for this specific value of p. A simple implementation of the witness search could perform for each parameter p a forward shortest-path search in G' from each source, ignoring node v, until all targets have been settled. Since source and target nodes are usually not far apart, the search usually only needs to settle a small fraction of the nodes in the graph and is therefore called a *local search*. We can also limit the depth of the computed shortest paths tree (*hop limit*) to improve performance. This preserves correctness because we may only add superfluous shortcuts, but we never omit a necessary one. Still, if the cardinality of $[L, U]$ is large, this is still infeasible as we need to perform too many local searches. Another possible implementation could use a Pareto-Dijkstra. However, this requires the storage of multiple labels per node and also the results of [4] suggest that there can be too many labels. So instead, we do something tailored to our objective functions: Let Q_p be a u-w-path with $w_p(Q_p) \leq w_p(u, v) + w_p(v, w)$. We observe that if $c(Q_p) \leq c(u, v) + c(v, w)$, then $w_q(Q_p) \leq w_q(u, v) + w_q(v, w)$ for all $q \in [L, p]$. And if $c(Q_p) \geq c(u, v) + c(v, w)$, then $w_q(Q_p) \leq w_q(u, v) + w_q(v, w)$ for all $q \in [p, U]$. This observation implies Lemma 2.1.

Lemma 2.1 *Let $\langle u, v, w \rangle$ be a potential shortcut. Any parameter interval $[L, U]$ can be partitioned into three, possibly empty partitions $[L, L']$, $[L' + 1, U' - 1]$ and $[U', U]$ with integers L', U' and the following properties:*
(a) If $[L, L']$ is not empty, then there exists a single witness path for all $p \in [L, L']$.
(b) If $[U', U]$ is not empty, then there exists a single witness path for all $p \in [U', U]$.
(c) If $[L' + 1, U' - 1]$ is not empty, then for all values of p in it, there exists no witness path.

Algorithm 1: Parameter Incr. Witness Search

input : path $\langle u, v, w \rangle$, interval $[L, U]$
output : max. p with witness on $[L, p]$

$p := L$;
while *p smaller or equal to U* **do**
  `// Calculate the shortest path` $\langle u, \ldots, w \rangle_{\not\ni v}$ `for parameter` p
  $P :=$ PerformLocalDijkstra(u, w, v, p);
  if $w_p(P) > w_p(\langle u, v, w \rangle)$ **return** $p - 1$;
  $p := \max \{k \in \mathbb{N} \mid w_k(P) \leq w_k(\langle u, v, w \rangle)\} + 1$;
return U;

Algorithm 2: Parameter Decr. Witness Search

input : path $\langle u, v, w \rangle$, interval $[L, U]$
output : min. p with witness on $[p, U]$

$p := U$;
while *p greater or equal to L* **do**
  `// Calculate the shortest path` $\langle u, \ldots, w \rangle_{\not\ni v}$ `for parameter` p
  $P :=$ PerformLocalDijkstra(u, w, v, p);
  if $w_p(P) > w_p(\langle u, v, w \rangle)$ **return** $p + 1$;
  $p := \min \{k \in \mathbb{N} \mid w_k(P) \leq w_k(\langle u, v, w \rangle)\} - 1$;
return L;

Algorithm 3: Witness Search

input : path $\langle x, y, z \rangle$, interval $[L, U]$
output : min. interval $[L' + 1, U' - 1]$ for which a shortcut is necessary

L':= ParamIncrWitnessSearch($\langle x, y, z \rangle$, $[L, U]$);
// no witness necessary
if $L' \geq U$ **return** $\emptyset$;
// else $[L' + 1, U' - 1] \neq \emptyset$
U':= ParamDecrWitnessSearch($\langle x, y, z \rangle$, $[L, U]$);
return $[L' + 1, U' - 1]$;

Algorithm 4: Contraction

input : node v, interval $[L, U]$

foreach $(u, v) \in E'$, $(v, w) \in E'$ **do**
  // potential interval of shortcut
  I:= $NI(u, v) \cap NI(v, w) \cap [L, U]$;
  if $I = \emptyset$ **then continue**;
  $[L', U']$:= WitnessSearch($\langle u, v, w \rangle$, I);
  if $[L', U'] \neq \emptyset$ **then**
    // add shortcut (u, w)
    e:= (u, w);
    E':= $E' \cup \{e\}$;
    $t(e)$:= $t(u, v) + t(v, w)$;
    $c(e)$:= $c(u, v) + c(v, w)$;
    $NI(e)$:= $[L', U']$; // necessity interv.
remove v from $G' = (V', E')$;

Our algorithm performs multiple single source shortest path searches from each source but with different p (Algorithm 1). First, we start with $p := L$. When we find a witness path Q, we compute the largest $p' \geq p$ for that Q is still a witness path. This works in constant time since every path is basically a linear function over the parameter p, and p' is the possible intersection of the witness path and the possible shortcut. Then, we continue with $p := p' + 1$ and repeat the procedure increasing p until we either reach U or find no witness path. Note that because of this '+1', we only support discrete parameter values. If we reach U, we know that no shortcut is necessary and our witness search (Algorithm 3) is done. If no witness can be found, we perform the search decreasing p, starting with $p := U$ (Algorithm 2), to constrain the shortcut e during the contraction of node v to the smallest parameter interval $NI(e)$ necessary (Algorithm 4).

In practice, we observe an average of two performed shortest path queries. Since in some cases a large number of queries may be necessary to find the minimal necessity interval, we limit the number of single source shortest paths queries to 30. Also note that we can restrict the witness searches to the values of p in the intersection of the necessity intervals of the two edges constituting the potential shortcut.

Parameter Interval Reduction. Due to the locality of our witness searches that only use a constant hop limit, we may insert unnecessary shortcuts. Also edges of the original graph might not be necessary for all parameters. So whenever we perform a witness search from u, we compare the incident edges (u, x) with the computed path $\langle u, \ldots, x \rangle$. When there are values of p where $\langle u, \ldots, x \rangle$ is shorter than (u, x), we reduce its necessity interval and delete the edge when the interval is empty.

Parameter Splitting. Preliminary experiments revealed that a single node order for a large parameter interval will result in too many shortcuts. That is because a single node order is no longer sufficient when the shortest paths significantly change over the whole parameter interval. One simple solution would be to split the intervals into small adequate pieces beforehand, and contract the graph for each interval separately. But we can do better: we observed that a lot of shortcuts are required for the whole parameter interval. Such a shortcut for a path $\langle u, v, w \rangle$ would be present in each of the constructed hierarchies that contracts v before u and w. Therefore, we use a classical divide and conquer approach as already outlined in the introduction. We repeatedly split the parameter intervals during the contraction and compute separated node orders for the remaining nodes. For that, we need to decide on when to split and how we split the interval.

A split should happen when there are too many 'differences' in the classification of importance between different values of p in the remaining graph. An indicator for these 'differences' are the parameter intervals of the added shortcuts. When a lot of *partial shortcuts*, i. e. shortcuts not necessary for the whole parameter interval, are added, a split seems advisable. So we trigger the split when the number of partial shortcuts exceeds a certain limit. However, in our experience, this heuristic needs to be adjusted depending on the metrics used. One reason for this imperfection is the difficult prediction of the future contraction behaviour. Sometimes it is good to split earlier although the contraction currently works well and not too many partial shortcuts are created. But due to the shared node order, the contraction becomes more difficult later, even when we split then.

A very simple method to split a parameter interval is to cut into halves (*half split*). However, this may not be the best method in any case. The 'different' parameters can be unequally distributed among the parameter interval and a half split would return two

unequally difficult intervals. So we may also try a *variable split* where we look again on the shortcuts and their necessity intervals to improve the parameter interval splitting. One possibility is to split at the smallest (largest) parameter which lets more than half of the edges become necessary. If no such parameter exists, we cut into halves.

To reduce main memory consumption, we also use hard disk space during contraction. Before we split, we swap the shortcuts introduced since the last split to disk. When we have finished contraction of one half of a interval, and we need to contract the other half, we load the state of the graph before the split from disk. This saves us a significant amount of main memory and allows the processing of large graphs.

Node Ordering. The node order is selected using a heuristic that keeps the nodes in a priority queue, sorted by some estimate of how attractive it is to contract a node. We measure the attractiveness with a linear combination of several priority terms. After some initial tests with the available priority terms from [8], we decided to use the following terms. The first term is a slightly modified version of the *edge difference*; instead of the difference, we count the number of *added shortcuts* (factor 150) and the number of *deleted edges* (factor -40). The second term is for uniformity, namely *deleted neighbors* (factor 150). The third term favors the contraction in more sparse regions of the graph. We count the number of relaxed edges during a witness search as the *search space* (factor 80). Our last terms focus on the shortcuts we create. For each shortcut we store the number of original edges it represents. Furthermore, for every new shortcut (u, w) the contraction of a node v would yield, we calculate the difference between the number of original edges represented by (u, v) and (u, w) (both factor 10). Also, we use the same heuristics for priority updates, i. e. updating only neighbors of a contracted node and using *lazy updates*.

3 Goal-Directed Technique in the Core Graph

For a hierarchy given by some node order, we call the K most important nodes the *core* of the hierarchy. We observed that preprocessing the core takes long because the remaining graph is dense and we have to do it several times due to parameter splitting. To speed-up the preprocessing, we may omit contracting the core and utilize some additional speed-up techniques on the remaining core. In contrast, we can also utilize further speed-up techniques on a contracted core to improve query times at the cost of preprocessing. For both variants, we extended CALT [2] to our bi-criteria scenario. We need to compute appropriate landmarks and distances for that.

Again, the straightforward approach, to compute landmarks and distances for every value of p separately, is too time-consuming. Also, space consumption becomes an issue if we compute a landmark set for every value of p. So given a core for a parameter interval $[L, U]$, we compute two sets of landmarks and distances: one for $w_L(\cdot)$ and one for $w_U(\cdot)$. A lower bound d_L for parameter L and a lower bound d_U for parameter U can be combined to a lower bound d_p for any parameter $p \in [L, U]$:

$$d_p := (1-\alpha) \cdot d_L + \alpha \cdot d_U \text{ with } \alpha := (p-L)/(U-L) \tag{3.1}$$

The correctness of the lower bound (3.1) is a direct consequence of Lemma 3.1.

Lemma 3.1 *Let the source and target nodes of a query be fixed. Let $d(p)$ be the shortest path distance for a real-valued parameter p in $[L, U]$. Then, $d(\cdot)$ is* concave, *i. e. for $p, q \in [L, U], \alpha \in [0, 1] : d((1-\alpha)p + \alpha q) \geq (1-\alpha)d(p) + \alpha d(q)$.*

Proof. Assume p, q, α as defined above exist with $d((1-\alpha)p + \alpha q) < (1-\alpha)d(p) + \alpha d(q)$. Let P be a shortest path for parameter $(1-\alpha)p + \alpha q$. In the case $w_p(P) \geq d(p)$, we deduce directly $w_q(P) < d(q)$, and analogously for $w_q(P) \geq d(p)$, we deduce $w_p(P) < d(p)$. This contradicts the definition of $d(\cdot)$. □

The lower bound (3.1) is lower in quality as an individual bound for any value of p, since shortest paths are not all the same for $p \in [L, U]$, but is better than the lower bound obtained from landmarks for the two input edge weight functions $t(\cdot)$ and $c(\cdot)$.

Due to the nature of a CH query, which we will describe in full detail in the next section, we have to distinguish between different possibilities with respect to the used landmark approach. There are two possible methods for landmark-based potential function, as described by Pohl et al. [21] and Ikeda et al. [15]. These approaches are the *symmetric* approach and the *consistent* approach. In the consistent approach, the potential functions are usually worse but it may still be faster since we can stop a bidirectional search as soon as a node is settled in both search directions. The symmetric approach can only stop when the minimal distance of all reached but unsettled nodes is not smaller than the tentative distance between source and target. However, a CH query, which is bidirectional, also needs the same stop criterion as the symmetric approach. So when we use landmarks on a contracted core, it makes no sense

to use the worse potential functions of the consistent approach since we are not allowed stop earlier. On an uncontracted core, we could make use of the better stopping criterion of the consistent approach. However, this did not pay off in our experiments.

4 Query

Our query algorithm is an enhanced CH query algorithm, optionally augmented with CALT. We will first describe the original algorithm, and then our enhancements.

Original CH Query. After the contraction, we split the resulting graph with shortcuts into an *upward graph* $G_\uparrow := (V, E_\uparrow)$ with $E_\uparrow := \{(u,v) \in E \mid u \text{ contracted before } v\}$ and a *downward graph* $G_\downarrow := (V, E_\downarrow)$ with $E_\downarrow := \{(u,v) \in E \mid u \text{ contracted after } v\}$. Our *search graph* is $G^* = (V, E^*)$ where $\overline{E_\downarrow} := \{(v,u) \mid (u,v) \in E_\downarrow\}$ and $E^* := E_\uparrow \cup \overline{E_\downarrow}$. And we store a forward and a backward flag such that for any edge $e \in E^*$, $\uparrow(e) = \text{true}$ iff $e \in E_\uparrow$ and $\downarrow(e) = \text{true}$ iff $e \in \overline{E_\downarrow}$. For a shortest path query from s to t, we perform a modified bidirectional Dijkstra shortest path search, consisting of a forward search in $G_\uparrow$ and a backward search in $G_\downarrow$. Iff there exists an s-t-path in the original graph, then both search scopes eventually meet at a node u with the highest order of all nodes in a shortest s-t-path. The query alternates between forward and backward search. Whenever we settle a node in one direction which is already settled in the other direction, we get a new candidate for a shortest path. Search is aborted in one direction if the smallest weight in the queue is at least as large as the best candidate path found so far. We also prune the search space using the *stall-on-demand* technique [24]: Before a node u is settled in the forward search, we check the incident downward edges available in $G_\downarrow$. If there is a shorter path to u via such an edge, we stop (*stall*) the search at u. Moreover, stalling can propagate to additional nodes w in the neighborhood of u, stalling even more nodes.

Parameter CH Query. Given a parameter p, we relax only those edges which are necessary for this parameter. The parameter splitting during the contraction results in multiple node orders. Therefore we have to augment the definition of the upward and downward graph. Our upward/downward graph contains all edges that are directed upwards/downwards in the hierarchy for their parameter interval. Then $E_\uparrow := \{(u,v) \in E \mid \exists p : p \in NI(u,v), u \text{ contracted before } v \text{ for } p\}$ and $E_\downarrow := \{(u,v) \in E \mid \exists p : p \in NI(u,v), u \text{ contracted after } v \text{ for } p\}$. If there are edges that are directed in both directions, depending on a parameter in their interval, we split them into several parallel edges and adjust their interval. Otherwise the search graph definitions remain. Then, our query Algorithm 5[1] can recognize an edge not being directed upwards/downwards for the given value of p by looking at its interval.

Algorithm 5: FlexibleCHQuery(s, t, p)

```
//tentative distances
d_↑:= ⟨∞,...,∞⟩; d_↓:= ⟨∞,...,∞⟩;
d_↑[s]:= 0; d_↓[t]:= 0; d:= ∞;
//priority queues
q_↑ = {(0,s)}; q_↓ = {(0,t)}; r:= ↑;
while (q_↑ ≠ ∅) or (q_↓ ≠ ∅) do
    if d > min{q_↑.min(), q_↓.min()} then break;
    //interleave direction
    //¬↑=↓ and ¬↓=↑
    if q_¬r ≠ ∅ then r:= ¬r;
    //u is settled and new candidate
    (·,u):= q_r.deleteMin();
    d:= min{d, d_↑[u] + d_↓[u]};
    foreach e = (u,v) ∈ E* do
        //have we found a shorter path?
        if r(e) and p ∈ NI(e) and
        (d_r[u] + w_p(e) < d_r[v]) then
            //update tentative distance;
            d_r[v]:= d_r[u] + w_p(e);
            //update priority queue;
            q_r.update(d_r[v], v);
return d;
```

During experiments, we observed that a query scans over a lot of edges that are not necessary for the respective parameter value. We essentially scan much more edges than we relax. We alleviate this problem with buckets for smaller parameter intervals. As data structure, we use an adjacency array, with a node array and an edge array. As an additional level of indirection, we add a bucket array, storing the edge indices necessary for each bucket. These indices are ordered like the edge array, so a single offset into this additional array is enough per node. Each node stores one offset per bucket array. By this method we essentially trade fast edge access for space consumption. A lot of edges are necessary for the whole parameter interval, almost all edges of the original graph and additionally many shortcuts. We store them separately, since otherwise, we would have an entry in each bucket array for each of these edges. This single action makes the buckets twice more space-efficient. Note that we can access the

[1] The pseudocode does not include stalling of nodes.

edges resident in all buckets without the further level of indirection, we just use another offset per node that points into a separate array.

Parameter CALT Query. We perform the core query, as described in [2], in two phases. In the first phase, we perform the parameter CH query described before, but stop at core nodes. In the second phase, we perform a core-based bidirectional ALT algorithm. We split the second phase into two parts itself. The first part is the calculation of the proxy nodes [10]. We perform a backward search from the source node and a forward search from the target node until we settle the first core node. Note that it is not guaranteed to find a path to the core in the respective searches. This can only happen if our input graph does not consist of strongly connected components. If this should be the case we disable the goal-direction for the respective direction by setting the potential function to zero. The second part is a bidirectional ALT algorithm using the symmetric approach [21]. Alternatively to performing a proxy search we could calculate the respective proxy nodes and their distances in advance for every node and every core. This would result in an extra number of $3 \cdot |\#cores|$ integers we would have to store for every node. Still, compared to the computation time needed for the search in the core, the proxy search is very fast. We observed a number of 7% to 13% of the settled nodes to be a result of the proxy searches. For the relaxed edges, the part of the proxy search varied between 3% and 5%.

Profile Query. Our algorithm can also be utilized to find all possible shortest paths between a given source and target node for arbitrary values of p in our interval. This can be done with a number of searches in the number of existing paths. We give an algorithm to find all existing $k > 1$ paths with $3 \cdot k - 2$ queries. Due to case differentiations for distinct parameter values, we will only prove a simpler bound of $2 \cdot k - 1$ queries for a continuous parameter interval.

Lemma 4.1 *For two s-t-paths P_1 and P_2 with $(t(P_1), c(P_1)) \neq (t(P_2), c(P_2))$ the following holds: Let p_1 be a parameter value for which P_1 is a shortest path and p_2 be a parameter value for which P_2 is a shortest path. If a shortest path P exists for a parameter value $p_1 < p < p_2$ with $(t(P_1), c(P_1)) \neq (t(P), c(P)) \neq (t(P_2), c(P_2))$, we can find such a path P at p chosen so that $w_p(P_1) = w_p(P_2)$ as in Figure 1.*

Note that we rely on a continuous parameter interval to choose p. If we use distinct parameter values as in our algorithm and the 'cutting point' p lies between two integer values, we would have to choose $p' = \lfloor p \rfloor$ or $p' = \lceil p \rceil$. Which one of these values, if any, yields another path cannot be decided in advance.

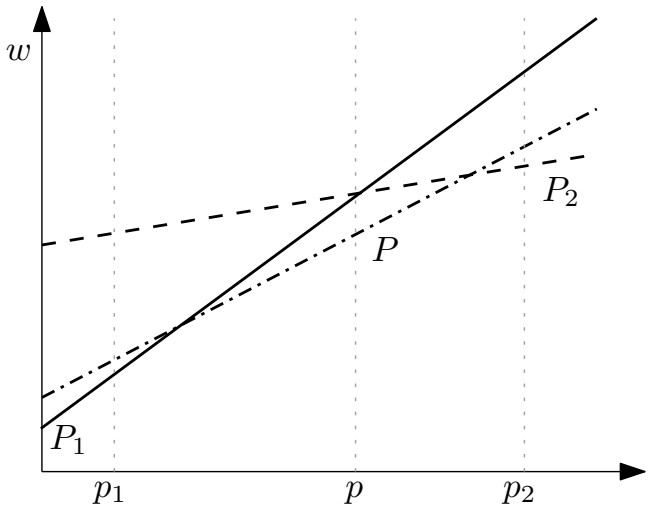

Figure 1: The profile query can locate an additional shortest path P in the interval (p_1, p_2) by looking at the 'cutting point' p of the shortest paths P_1 and P_2.

Proof. Let us assume P and p exist with $p_1 < p < p_2$ and $w_p(P) < w_p(P_1)$ and $w_p(P) < w_p(P_2)$. Furthermore, we assume that at $\widetilde{p}$ with $w_{\widetilde{p}}(P_1) = w_{\widetilde{p}}(P_2)$ no path $\widetilde{P}$ exists that fulfills $w_{\widetilde{p}}(\widetilde{P}) < w_{\widetilde{p}}(P_1) = w_{\widetilde{p}}(P_2)$. Also let w.l.o.g. $p_1 < p < \widetilde{p}$. Since $w_{p_1}(P_1)$ is minimal over all existing s-t-paths at p_1, $w_{p_1}(P) \geq w_{p_1}(P_1)$. With the assumption also $w_{\widetilde{p}}(P) \geq w_{\widetilde{p}}(P_1)$ holds. This cannot be true for linear functions of degree one, as $w_p(P) < w_p(P_1)$. □

We can now use Lemma 4.1 to find all existing paths in a continuous parameter interval. We start our profile query with two point to point shortest path queries for the parameter values $p_{\min}$ and $p_{\max}$. Due to Lemma 4.1, we can recursively calculate the cutting point of the respective weight functions and perform a new query for the cutting point p_c of the weight functions and continue the search between the respective paths for $p_{\min}$ and p_c as well as p_c and $p_{\max}$. Thus, we perform one query for each existing path. Furthermore, we perform one query without obtaining a new path between each adjacent pair of paths. This yields a number of $k - 1$ further queries. Thus, we get a number of $2 \cdot k - 1$ queries in total. For the case $k = 1$ we obviously need a number of two queries.

The bound of $3 \cdot k - 2$ follows from the fact that we can only perform queries for integers p and therefore have to perform two queries between every adjacent pair of found paths if the cutting point is not an integer.

Profile Query with Sampling. As an alternative to a complete profile query, our algorithm can also be used to perform queries for a sample subset $S = \{p_1, p_2, \ldots, p_k\} \subseteq [L, U]$. To do so, we adapt our algorithm from the profile query. We start with a query for the minimal parameter value p_1 and the maximal parameter value p_k in S. For two paths P_i at parameter p_i and P_j at parameter p_j with $p_i < p_j$ we calculate the next query parameter as p_ℓ, $\ell = \lfloor i + (j - i)/2 \rfloor$. By this method we recursively continue. If we find the already known path P_i (P_j) again at p_ℓ, we need not continue the recursion between p_ℓ and p_i (p_j).

Approximated Profile Query. Since the large number of queries result in a relatively long computation time for a profile query, we also offer the possibility of an approximated profile query with ε-guarantee:

Lemma 4.2 *For two shortest s-t-paths P_1 at parameter value p_1 and P_2 at parameter value p_2, $p_1 < p_2$ and $t(P_2) \leq (1+\varepsilon) \cdot t(P_1)$ or $c(P_1) \leq (1+\varepsilon) \cdot c(P_2)$ holds: if a shortest s-t-path P at parameter value p exists with $p_1 < p < p_2$ then either $t(P) \leq (1+\varepsilon) \cdot t(P_1) \wedge c(P) \leq (1+\varepsilon) \cdot c(P_1)$ or $t(P) \leq (1+\varepsilon) \cdot t(P_2) \wedge c(P) \leq (1+\varepsilon) \cdot c(P_2)$ holds.*

Proof. First let $t(P_2) \leq (1+\varepsilon) \cdot t(P_1)$. In this case we can use P_2 to approximate P. From $p < p_2$ follows $c(P_2) < c(P)$. From $p_1 < p$ follows $t(P) > t(P_1)$ and with $t(P_2) \leq (1+\varepsilon) \cdot t(P_1)$ follows $t(P_2) \leq (1+\varepsilon) \cdot t(P)$. In the same way we can use P_1 in case that $c(P_1) \leq (1+\varepsilon) \cdot c(P_2)$ holds. □

Therefore, we can guarantee that for every omitted path P a path $P' = (t', c')$ will be found with $t(P') \leq (1+\varepsilon) \cdot t(P)$ and $c(P') \leq (1+\varepsilon) \cdot c(P)$. Since we essentially need two queries to terminate our search for further paths between two already found paths, the approximation might help to reduce the number of 'unnecessary' searches without omitting too many paths.

Note that the approximation method can of course also be applied to our sampling method.

Correctness. The correctness of our query algorithm is based on the correctness of the CH query algorithm and the CALT query algorithm. We already described that we add all necessary shortcut for any integer parameter p in a given interval $[L, U]$. Thus the parameter CH query is correct for any parameter inside the interval. The correctness of the CALT query depends on the correct lower bounds. Therefore, our query algorithm is correct.

5 Experiments

Test Instances. We present experiments performed on road networks from the year 2006, provided by PTV AG. The German road network consists of 4 692 751 nodes and 10 806 191 directed edges. The European road network consists of 29 764 455 nodes and 67 657 778 directed edges. For comparison with Pareto-SHARC, we also performed experiments with their older network of Western Europe having 18 017 748 nodes and 42 189 056 directed edges.

Environment. We did experiments on one core of a single AMD Opteron Processor 270 clocked at 2.0 GHz, with 8 GB main memory and 2×1 MB L2 cache. Only the preprocessing of the European road network has been done on one core of a Intel Xeon 5345 processor clocked at 2.33 GHz with 16 GB main memory and 2×4 MB L2 cache. We run SuSE Linux 11.1 (kernel 2.6.27) and use the GNU C++ compiler 4.3.2 with optimization level 3.

Annotation. In our tables we denote with *param* the number of different parameters in the interval $[0, x-1]$. For core approaches we may give the kind of core used by a string of two signifiers *core size/#landmarks*. The core size indicates how many nodes the core contains. The second number gives the number of landmarks used in the core. A core is usually uncontracted, only the contracted cores are additionally marked with *C*. The *preprocessing* time is given in the format hh:mm and may split it into the *compute* time and the *I/O* time needed for reading/writing to disk. *Query* performance is given in milliseconds. For the *speed-up* we compare our algorithm to the timings of a plain unidirectional Dijkstra algorithm.

Weight Functions. For our experiments, we combined the travel time and the approximate monetary cost for traversing an edge. The travel time was computed from the length of an edge and the provided average speed. To approximate the cost, we chose to calculate the needed mechanical work for a standard car. We use the standard formulas for rolling and air resistance to compute the force $F(v) = F_N \cdot c_r + A \cdot c_w \cdot \frac{\rho}{2} \cdot v^2$ with F_N the normal force, c_r the rolling resistance coefficient, A representing the reference area in square meters, c_w being the drag coefficient, ρ the air density and v the average speed on the desired road. We estimated the constants as $F_N = 15\,000 \mathrm{kg\,m/s^2}$, $c_r = 0.015$, $A = 2.67 \mathrm{m^2}$, $c_w = 0.3$ and $\rho = 1.2 \mathrm{kg/m^3}$. The resulting cost function is defined as $\widetilde{C}(v, \ell) = (\ell \cdot c \cdot F(v))/\eta$ with ℓ denoting the length of the road, c denoting the cost of energy and η denoting the efficiency of the en-

gine. We estimated $c = 0.041€/MJ$, this corresponds to a fuel price of about $1.42€/\ell$, and $\eta = 0.25$. For inner city roads, we multiply the cost by a factor of 1.5 to compensate for traffic lights and right of way situations. Note that this method is a good approximation for higher speeds but disregards the bad efficiency of the engine and transmission on low speeds. To get a more realistic model, we favor an average speed of 50 km/h and define our final weight function as $C(v,\ell) = \left\{ \begin{array}{ll} \widetilde{C}(50+\sqrt{50-v},\ell) & \text{if } v < 50 \\ \widetilde{C}(v,\ell) & \text{otherwise} \end{array} \right\}$. Figure 2 plots $C(v,1)$ against the travel speed.

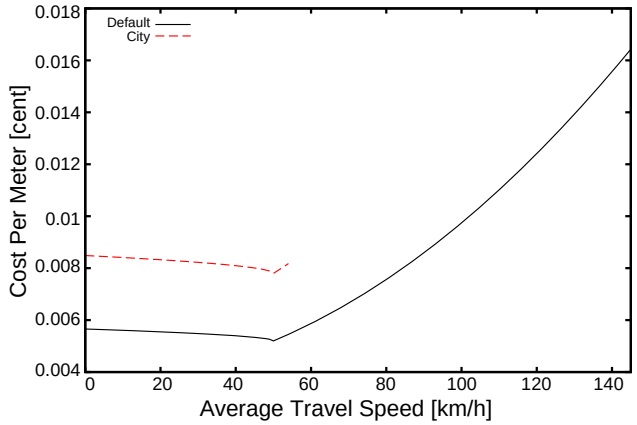

Figure 2: Energy costs per meter against the travel speed. The upper curve is for inner city roads.

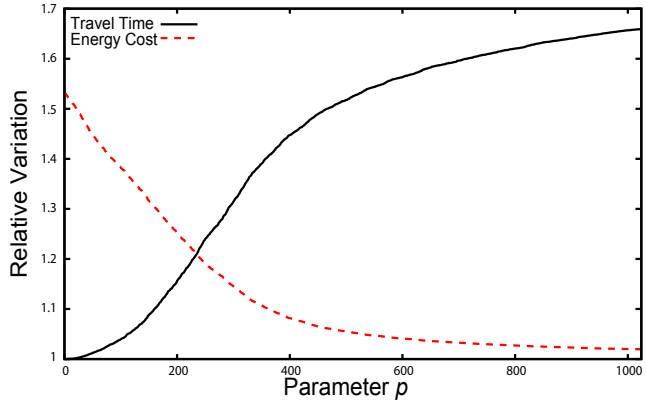

Figure 3: Average increase of the travel time and decrease of the energy cost as ratio to the best possible routes.

Parameter Interval. A good parameter interval for the travel time and the energy cost function is between 0 and 0.1. For $p = 0$, we find the fastest path and for

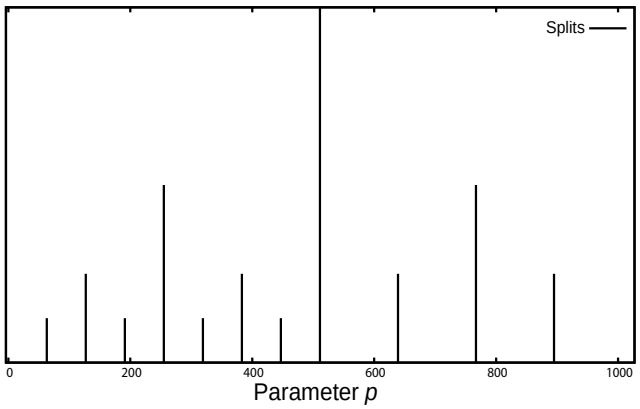

Figure 4: The vertical lines present the split positions. The higher a line, the lower is the recursion depth of the split.

$p = 0.1$, we observe an increase of the average travel time by 65%. This increase is reasonable, and also Pareto-SHARC considers at most 50% increase.

Since we only support integer parameters, we scale all weights by a factor of 10 000 to be able to adjust the parameter with four digits after the decimal point. This results in a parameter interval of $[0, 1\,000]$, but actually we rounded to $[0, 1\,023]$. To prove the robustness of our approach, we additionally perform some experiments for the larger parameter interval $[0, 2\,047]$ using the same scale factor of 10 000.

Parameter Splitting. During the contraction, we maintain a threshold T; when there are more than T partial shortcuts since the last split, we trigger a split. After a split, we increase T since there are more shortcuts necessary in the upper hierarchy. We start with $T = 1.3\%$ of the number of edges in the input graph. After a split, we multiply T with 1.2. We do not split parameter intervals of size 16 or below.

We use only half split, preliminary experiments with variable split did not yield significant improvements. In Figure 4, the splits of the parameter interval are visualised. The longer a line, the earlier the split happened. We see that in the interval $[0, 511]$ 7 splits happened whereas in $[512, 1023]$ only 3 happen. This correlates quite well with the observations from Figure 3, where great changes are especially visible for the first half of the interval.

Performance. We summarize the performance of different variants of our algorithm in Table 1. When we only use contraction, preprocessing on Germany takes 2.4 hours, resulting in a average query time of 2.9 ms. Note that the average is over 100 000 queries, where

Table 1: Preprocessing and query performance for different *graph*s. *param* specifies the number of different parameters x in the interval $[0, x-1]$. The *core* size and number of *landmark*s are given. The *preproc*essing time in hh:mm is split into the *compute* time and the *I/O* time needed for reading/writing to disk. Also the number of *split*s and the *space* consumption in Byte/node are given. The query performance is given as *query* time in milliseconds, *speed-up* compared to plain Dijkstra, number of *settled nodes* and number of *relaxed edges*.

		core/	preproc [hh:mm]		#	space	query	speed	settled	relaxed
graph	param	landmark	contr	IO	splits	[B/node]	[ms]	-up	nodes	edges
Ger	1 024	-/-	-	-	-	60	2 037.52	1	2 359 840	5 437 900
Ger	1 024	0/0	1 : 54	29	11	159	2.90	698	579	4 819
Ger	1 024	10k,C/64	2 : 06	29	11	183	0.63	3 234	170	2 059
Ger	1 024	3k/64	1 : 13	29	11	164	2.33	874	620	9 039
Ger	1 024	5k/32	1 : 05	30	11	161	2.76	738	796	11 137
Ger	1 024	5k/64	1 : 06	30	11	167	2.58	789	735	10 191
Ger	2 048	5k/64	1 : 30	37	14	191	2.64	771	734	9 835
Eur	1 024	-/-	-	-	-	59	15 142.31	1	6 076 540	13 937 000
Eur	1 024	5k/64	12 : 55	2 : 32	11	142	6.80	2 226	1 578	32 573
Eur	1 024	10k/64	11 : 58	2 : 31	11	144	8.48	1 784	2 151	39 030
Eur	1 024	10k,C/64	18 : 37	2 : 35	11	145	1.87	8 097	455	7 638
WestEur	16	10k,C/64	1 : 00	0 : 10	0	60	0.42	14 427	270	2 103
WestEur	1 024	10k,C/64	5 : 12	1 : 12	7	151	0.98	6 183	364	3 360

source, target and the value of p are selected uniformly at random. We split the precomputation time into the compute part and the I/O part. The I/O part is the time to write the intermediate results to disk when a split happens. You see that it can take a significant percentage of the overall precomputation time, up to one third, but easily be avoided by using more main memory.

We usually select 64 *avoid* landmarks [11] per core. Compared to full contraction, a 5k uncontracted core has 12% better query time and significantly decreases the precomputation by one third. As expected, a 3k core results in even better query times, at the cost of precomputation time. However, switching to 32 landmarks is not significantly better for precomputation time and space, but increases the query time by 7%. Our best query times are with a 10k contracted core yielding speed-up of more than 3 000.

You cannot directly compare our performance to previously published results of single-criteria CH, since the performance heavily depends on the edge weight function. We computed single-criteria CH for Germany with $p = 0$, $p = 1000$ and for $p = 300$, one of the most difficult parameters from Figure 5. The preprocessing time varied by about 100% between these parameters and the query time even by 270%. Only space consumption[2] is quite constant, it changed by less than 3% and is around 22 B/node. We compare our 5k/64 core to economical CH [7] as both have the best preprocessing times. Our space consumption is a factor 7.6 larger, however we could greatly reduce space in relation to our 1 024 different parameters, even below the number of 12 different cores that exist due to 11 splits. Also, we could compute 18.9 different hierarchies within the time needed by our new algorithm. For this comparison, we ignored the preprocessing I/O time since the single-criteria CH also needs to be written to disk. The reduction in preprocessing time is therefore not as large as for space, but still good. However, our efficient preprocessing comes at the cost of higher query times. Single-criteria CH has 0.82 ms query time on average but our query time is about three times larger. One reason for our higher query time is the shared node order within a parameter interval that is not split, but we also have a larger constant factor because of the data structure: there is an additional level of indirection due to the buckets, and we store two weights and a parameter interval per edge, resulting in more cache faults. The frequently occurring weight comparisons are also more expensive and noticeable, since multiplications and additions are necessary. But e. g. for web services, the query time is still much lower than other delays, e. g., for communication latency. Using landmarks on a contracted core would yield even faster query times than single-criteria CH, but this would not be a fair comparison as we should use landmarks there as well.

Our algorithm scales well with the size of the interval. Increasing it to twice the size only increases the preprocessing time by 32% and the space by 14%

[2] do not mistake it for space overhead

without affecting query time much. We also did some experiments on Europe with parameters that worked well for Germany, but we could not perform a thorough investigation, since the precomputation took very long. The query times are very good, yielding speed-ups of more than 8 000. Better speed-ups are expected for larger graphs, as for single-criteria CH, too. The space consumption is even better, the dense road network of Germany is more difficult than the rest of Europe. Preprocessing time is however super-linear, but we expect that tuning the node ordering parameters and the split heuristic will alleviate the problem.

Edge Buckets. As already explained earlier, the number of scanned edges has a large impact on the quality of the query. When the number of memory accesses for non-necessary edges is large enough, it pays of to omit the respective edges, even if an additional level of indirection has to be used for some of the edges. By default, for each parameter interval that was not further split, we have one bucket, i. e. with 11 splits we have 12 buckets that split the parameter interval as in Figure 4. To investigate the effect further, we take two computed hierarchies from Table 1, remove the original buckets and use different numbers buckets that split the parameter interval in equally spaced pieces. The results are in Table 2 and show the different tradeoffs between query time and space consumption. If we compare the data to Table 1, we can see that even with buckets, the number of scanned edges is more than two times larger than the number of relaxed edges. Comparing the different numbers of buckets, we notice that around a quarter of all edges would be stored in all buckets. Therefore, storing them separately helps to control the space, since buckets already consume a major part. About 40% of our space is due to buckets but we get almost the same percentage as improvement of the query time. Using just two buckets even increases the query time as another level of indirection is necessary to address the edges in the buckets even though we only halve the number of scanned edges. Our choice of 12 buckets is in favor of fast query times, more buckets bring only marginal improvements. Figure 5 visualises the effects of buckets for different values of p. We see that ≥ 4 buckets improve the query time for all parameters but in the interval $[0, 511]$ more buckets are necessary than in $[512, 1023]$ as 12 buckets show the most uniform performance over the whole interval. When we ignore the different bucket sizes, we also note that our algorithm achieves the best query times for $p = 0$, when we optimize solely for travel time. Therefore, our performance depends on the chosen value of p and furthermore on the chosen edge weights functions as they have a strong impact on the hierarchical structure of the network.

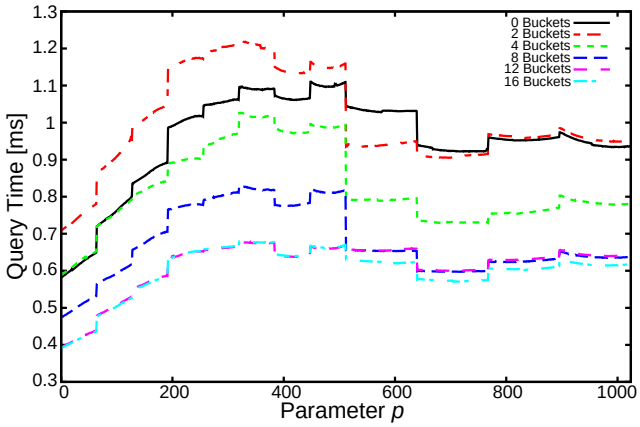

Figure 5:

Profile Query. Utilizing our flexible fast query algorithm, we can also compute the shortest paths for *all* values of p. We get about $k = 31$ different paths on average and need 88 queries for different values of p to compute them (Table 3). This is close to the bound of $3k - 2 = 91$ queries, which are theoretically necessary. So applications that need all shortest paths between two locations should use our fastest version.

Table 3: Performance of the profile search in different versions.

graph	core/ landmark	# paths	performed queries	time [ms]
Ger	10k,C/64	30.7	88.1	59.4
Ger	3k/64	30.7	88.1	307.0
Ger	5k/64	30.7	88.1	349.6

When a reasonable subset of all such paths is sufficient, our sampling approach provides better performance. A possible application could be to present a number of samples for the user to choose from and maybe calculate more detailed information on demand for certain parameter intervals of the presented choices. In Table 4, we give an overview over the performance of our sampling version of the profile query, with the main focus on the number of found routes and running time for different numbers of equidistant sample values of p. As expected, we see a linear increase of the query time compared to the number of performed queries. Up to 9 samples, we get a new path with almost every query. However, the more samples we choose, the more queries are done in vain. Still, we can chose at query time very fine-grained how many different paths we want. This is a clear advantage over Pareto-SHARC, were the av-

Table 2: Impact of edge buckets onto the query times. Experiments on the German road network with complete contraction for different numbers of buckets.

graph	core/ landmark	# buckets	% edges all buckets	query [ms]	scanned edges	memory overhead [B/node]
Ger	10k,C/64	1	100.0	0.96	41 613	0
Ger	10k,C/64	2	26.9	1.02	23 486	19
Ger	10k,C/64	4	26.5	0.81	13 388	28
Ger	10k,C/64	8	26.4	0.69	7 757	49
Ger	10k,C/64	12	26.4	0.63	5 769	71
Ger	10k,C/64	16	26.4	0.62	5 270	96
Ger	3k/64	1	100.0	3.75	170 483	0
Ger	3k/64	2	27.5	3.90	94 876	18
Ger	3k/64	4	27.0	3.18	53 082	28
Ger	3k/64	8	26.9	2.60	29 866	48
Ger	3k/64	12	26, 9	2.33	21 694	70
Ger	3k/64	16	26.9	2.32	20 754	95

erage number of target labels (= #paths) is limited to 5–6 since otherwise the precomputation and query time gets out of hand.

Table 4: Performance of sampled profile search on the German road network with a contracted core of 10 000 nodes using 64 landmarks.

# samples	# queries	# paths	time [ms]
2	1.8	1.8	1.0
3	2.7	2.7	1.7
5	4.4	4.3	2.9
9	8.0	7.2	5.5
17	15.8	11.4	10.2
33	31.4	17.3	18.9
65	58.3	22.8	34.3
129	95.9	27.1	57.1

Approximated Profile Query. Another possibility to reduce the profile query time is using ε-approximation. We prune our profile query when all paths that we may miss are within an ε factor of the currently found paths. By that, we balance the query time without missing significant differently valued paths as it may happen with sampling. In Table 5 we see the same ratio between found paths and query time as for the sampling approach. Many of the 31 different paths are very close as for a small $\varepsilon = 1\%$ the number of paths is cut in halves. Still, there are significant differences in the paths as even $\varepsilon = 16\%$ still has more than 4 different paths.

Table 5: Performance of the approximated profile query on the German road network with a contracted core of 10 000 nodes an 64 avoid landmarks.

ε	# paths	# queries	time [ms]
0.00	30.7	88.3	59.4
0.00125	26.5	55.0	36.4
0.0025	23.7	46.0	30.0
0.005	20.4	36.9	24.4
0.01	17.0	28.3	18.9
0.02	13.3	20.1	13.5
0.04	9.7	12.9	8.7
0.08	6.5	7.6	5.3
0.16	4.2	4.4	3.0
0.32	2.7	2.7	1.7

Comparison with Previous Work. Even though Pareto-SHARC can handle continent-sized networks only heuristically and we have exact shortest paths, we perform very well in comparison since we do not rely on Pareto-optimality. We used the same network of Western Europe (*WestEur*) and costs as Pareto-SHARC. With 16 values of p, the average travel time increases by 4.3%. This case allows in our opinion the closest comparison to Pareto-SHARC with simple label reduction with $\epsilon = 0.02$ (4:10 hours preprocessing and 48.1 ms query). Our precomputation is 3.6 times faster (see Table 1) and our profile query (2 ms) is more than 24 times faster. Computing a single path is even faster, taking just 0.4 ms. For 1024 different values of p, the average travel time increases by 43.4%, that might be close to heuristic Pareto-SHARC with strong label reduction ($\epsilon = 0.5$ and $\gamma = 1.0$ in [4], 7:12 hours preprocessing and 35.4 ms query). We could not reach an average in-

crease of 50%, even doubling the values of p yields less than 44% travel time increase. We need 12% less preprocessing time, and an approximate profile query with $\varepsilon = 0.01$ returning 5.7 different paths (6.2 ms) is 5.7 times faster. A query for a single value of p is even below 1 ms. Furthermore, we provide more flexibility with 12.5 different paths available on average over the whole parameter interval. But compared to our German road network, we have less different paths. This is due to the different 'costs': Pareto-SHARC uses a simple model to calculate fuel- and toll-costs whereas our model of the energy cost is based upon laws of physics. The only downside of our algorithm is that we need more space than Pareto-SHARC (22.5 B/node preprocessing + 23.7 B/node input graph for Pareto-SHARC). However, we can also provide more different routes. In conclusion, our linear combination of two edge weight functions is superior to Pareto-optimality. We scale better since we can split parameter intervals, whereas Pareto-optimal weights naturally cannot be split and distributed to buckets. Adapting the SHARC algorithm to our flexible scenario is possible. However, such a Flexible-SHARC algorithm also needs, for efficiency, necessity intervals, edge buckets, or maybe multiple arc flags for different parts of the parameter interval. Therefore it is not clear, whether it will turn out to be more space-efficient than our CH-based algorithm.

6 Conclusion

Our algorithm is the first one for fast flexible queries, and still we achieve query times comparable with single-criteria algorithms. We were able to engineer the preprocessing to be very time- and space-efficient. This is perfectly suitable for a server scenario where a server answers a lot of subsequent queries. We successfully avoid the problems of Pareto-optimality and still provide very diverse routes.

There is some open work. The I/O-time is too high for larger cores, in-memory compression can maybe make the disk storage obsolete. To reduce the preprocessing time, we can parallelize it. We can process different parameter intervals after a split in parallel and additionally we can apply the parallelization techniques from [25]. Also, our parameter splitting is efficient but leaves room for further improvements. One could possibly merge intervals later when they seem to be very similar. This could save preprocessing time since a lot of time is spent contracting highest ordered nodes. Additionally, our heuristics for the split decision are very simple, and more sophisticated methods might achieve better and more robust results. To reduce the space overhead, we can reduce the number of splits by using approximate shortcuts, but this will also reduce the number of different routes.

It would be interesting to adapt our solution to a continuous range of parameter values. In principle, it should be possible to do this but we believe that the added level of complication required makes this more of an academic question than something worthwhile in practice.

Even more flexibility is possible with more than two edge weights functions or nonlinear combinations. However, it is a open problem how to efficiently compute shortcuts in such a scenario, since our current approach only works well for one linear parameter dimension. Time-dependent routing algorithms should also become flexible. This is a challenge, as there it is even difficult to optimize for a single criterion that is not travel time but also *depends on the time*. One such criterion might be a time-dependent cost, that is e. g. different in slow- and fast-moving traffic. The main problem is the time-dependency of the edge weights, and therefore most likely all Pareto-optimal pairs of travel time and the other weight need to be computed. But hopefully, road networks are good-natured enough that this causes no hard problems in practice.

Acknowledgements. We thank Veit Batz for support when starting this project and Daniel Delling for his ALT implementation.

References

[1] Reinhard Bauer and Daniel Delling. SHARC: Fast and Robust Unidirectional Routing. In Ian Munro and Dorothea Wagner, editors, *Proceedings of the 10th Workshop on Algorithm Engineering and Experiments (ALENEX'08)*, pages 13–26. SIAM, April 2008.

[2] Reinhard Bauer, Daniel Delling, Peter Sanders, Dennis Schieferdecker, Dominik Schultes, and Dorothea Wagner. Combining Hierarchical and Goal-Directed Speed-Up Techniques for Dijkstra's Algorithm. In Catherine C. McGeoch, editor, *Proceedings of the 7th Workshop on Experimental Algorithms (WEA'08)*, volume 5038 of *Lecture Notes in Computer Science*, pages 303–318. Springer, June 2008.

[3] Daniel Delling. *Engineering and Augmenting Route Planning Algorithms.* PhD thesis, Universität Karlsruhe (TH), Fakultät für Informatik, 2009. http://i11www.ira.uka.de/extra/publications/d-earpa-09.pdf.

[4] Daniel Delling and Dorothea Wagner. Pareto Paths with SHARC. In Jan Vahrenhold, editor, *Proceedings of the 8th International Symposium on Experimental Algorithms (SEA'09)*, volume 5526 of *Lecture Notes in Computer Science*, pages 125–136. Springer, June 2009.

[5] Daniel Delling and Dorothea Wagner. Time-Dependent Route Planning. In Ravindra K. Ahuja,

Rolf H. Möhring, and Christos Zaroliagis, editors, *Robust and Online Large-Scale Optimization*, volume 5868 of *Lecture Notes in Computer Science*, pages 207–230. Springer, 2009.

[6] Edsger W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1:269–271, 1959.

[7] Robert Geisberger. Contraction Hierarchies. Master's thesis, Universität Karlsruhe (TH), Fakultät für Informatik, 2008. http://algo2.iti.uni-karlsruhe.de/documents/routeplanning/geisberger_dipl.pdf.

[8] Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. In Catherine C. McGeoch, editor, *Proceedings of the 7th Workshop on Experimental Algorithms (WEA'08)*, volume 5038 of *Lecture Notes in Computer Science*, pages 319–333. Springer, June 2008.

[9] Andrew V. Goldberg and Chris Harrelson. Computing the Shortest Path: A* Search Meets Graph Theory. In *Proceedings of the 16th Annual ACM–SIAM Symposium on Discrete Algorithms (SODA'05)*, pages 156–165, 2005.

[10] Andrew V. Goldberg, Haim Kaplan, and Renato F. Werneck. Better Landmarks Within Reach. In Camil Demetrescu, editor, *Proceedings of the 6th Workshop on Experimental Algorithms (WEA'07)*, volume 4525 of *Lecture Notes in Computer Science*, pages 38–51. Springer, June 2007.

[11] Andrew V. Goldberg and Renato F. Werneck. Computing Point-to-Point Shortest Paths from External Memory. In *Proceedings of the 7th Workshop on Algorithm Engineering and Experiments (ALENEX'05)*, pages 26–40. SIAM, 2005.

[12] P. Hansen. Bricriteria Path Problems. In Günter Fandel and T. Gal, editors, *Multiple Criteria Decision Making – Theory and Application –*, pages 109–127. Springer, 1979.

[13] Peter E. Hart, Nils Nilsson, and Bertram Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4:100–107, 1968.

[14] Moritz Hilger. Accelerating Point-to-Point Shortest Path Computations in Large Scale Networks. Master's thesis, Technische Universität Berlin, 2007.

[15] T. Ikeda, Min-Yao Hsu, H. Imai, S. Nishimura, H. Shimoura, T. Hashimoto, K. Tenmoku, and K. Mitoh. A fast algorithm for finding better routes by AI search techniques. In *Proceedings of the Vehicle Navigation and Information Systems Conference (VNSI'94)*, pages 291–296. ACM Press, 1994.

[16] Richard M. Karp and James B. Orlin. Parameter Shortest Path Algorithms with an Application to Cyclic Staffing. *Discrete Applied Mathematics*, 2:37–45, 1980.

[17] Ulrich Lauther. An Extremely Fast, Exact Algorithm for Finding Shortest Paths in Static Networks with Geographical Background. In *Geoinformation und Mobilität - von der Forschung zur praktischen Anwendung*, volume 22, pages 219–230. IfGI prints, 2004.

[18] Ernesto Queiros Martins. On a Multicriteria Shortest Path Problem. *European Journal of Operational Research*, 26(3):236–245, 1984.

[19] Rolf H. Möhring, Heiko Schilling, Birk Schütz, Dorothea Wagner, and Thomas Willhalm. Partitioning Graphs to Speedup Dijkstra's Algorithm. *ACM Journal of Experimental Algorithmics*, 11:2.8, 2006.

[20] James B. Orlin, Neal E. Young, and Robert Tarjan. Faster Parametric Shortest Path and Minimum Balance Algorithms. *Networks*, 21(2):205–221, 1991.

[21] Ira Pohl. Bi-directional Search. In Bernard Meltzer and Donald Michie, editors, *Proceedings of the Sixth Annual Machine Intelligence Workshop*, volume 6, pages 124–140. Edinburgh University Press, 1971.

[22] Peter Sanders and Dominik Schultes. Engineering Fast Route Planning Algorithms. In Camil Demetrescu, editor, *Proceedings of the 6th Workshop on Experimental Algorithms (WEA'07)*, volume 4525 of *Lecture Notes in Computer Science*, pages 23–36. Springer, June 2007.

[23] Dominik Schultes. *Route Planning in Road Networks*. PhD thesis, Universität Karlsruhe (TH), Fakultät für Informatik, February 2008. http://algo2.iti.uka.de/schultes/hwy/schultes_diss.pdf.

[24] Dominik Schultes and Peter Sanders. Dynamic Highway-Node Routing. In Camil Demetrescu, editor, *Proceedings of the 6th Workshop on Experimental Algorithms (WEA'07)*, volume 4525 of *Lecture Notes in Computer Science*, pages 66–79. Springer, June 2007.

[25] Christian Vetter. Parallel Time-Dependent Contraction Hierarchies, 2009. Student Research Project. http://algo2.iti.kit.edu/documents/routeplanning/vetter_sa.pdf.