Customizable Route Planning in Road Networks*

Daniel Delling[1], Andrew V. Goldberg[1],
Thomas Pajor[2], and Renato F. Werneck[1]

[1] Microsoft Research Silicon Valley
{dadellin,goldberg,renatow}@microsoft.com
[2] Karlsruhe Institute of Technology
pajor@kit.edu

July 24, 2013

Abstract. We propose the first routing engine for computing driving directions in large-scale road networks that satisfies all requirements of a real-world production system. It supports arbitrary metrics (cost functions) and turn costs, enables real-time queries, and can incorporate a new metric in less than a second, which is fast enough to support real-time traffic updates and personalized cost functions. The amount of metric-specific data is a small fraction of the graph itself, which allows us to maintain several metrics in memory simultaneously. The algorithm is the core of the routing engine currently in use by Bing Maps.

1 Introduction

A key ingredient of modern online map applications is a routing engine that can find best routes between two given location of a road network. This task can be translated into finding point-to-point shortest paths in a graph representing the road network. Although the classic algorithm by Dijkstra [25] runs in almost linear time with very little overhead, it still takes a few seconds on continental-sized graphs. This has motivated a large amount of research (see [19] or [62] for overviews) on speedup techniques that divide the work in two phases: *preprocessing* takes a few minutes (or even hours) and produces a (limited) amount of auxiliary data, which is then used to perform *queries* in a millisecond or less. Most previous research has been evaluated on benchmark data representing simplified models of the road networks of Western Europe and the United States, using the most natural metric, driving times, as the optimization function. The fastest technique, called HL [1, 2], can compute the driving time between any two points in a few hundred nanoseconds, or millions of times faster than Dijkstra's algorithm. Unfortunately, using such techniques in an actual production system is far more challenging than one might expect.

An efficient real-world routing engine must satisfy several requirements. First, it must incorporate all details of a road network—simplified models are not enough. In particular, previous work often neglected turn costs and restrictions, since it has been widely believed that any algorithm can be easily augmented to handle these efficiently. We show that, unfortunately, most methods actually have a significant performance penalty, especially if turns are represented space-efficiently. Moreover, a practical algorithm should be able to handle other natural metrics (cost functions) besides travel times, such as shortest distance, walking, biking, avoid U-turns, avoid/prefer freeways, avoid left turns, avoid ferries, and height and weight restrictions. In fact, the routing engine should provide reasonable performance guarantees for *any* metric, thus enabling personalized driving directions with no risk of timeouts. To support multiple cost functions efficiently, the algorithm should store as little data per metric as possible. In addition, new metrics should be incorporated quickly, thus enabling real-time traffic information to be part of the cost function. Moreover, updates to the traversal cost of a small number of road segments (due to road blocks, for example) should be handled even more efficiently. The engine should support not only the computation of point-to-point shortest paths,

* This is the full version of two conference papers [15, 22]. This work was done while the third author was at Microsoft Research Silicon Valley.

but also extended query scenarios, such as alternative routes. Finally, the routing engine should not be the bottleneck of a map application. All common types of queries should run in real time, i.e., fast enough for interactive applications.

In this paper, we report our experience in building a routing engine that meets *all* the above-mentioned requirements for a modern map server application. Surprisingly, the work goes far beyond simply implementing an existing technique in a new scenario. It turns out that no previous technique meets all requirements: Even the most promising candidates fail in one or more of the critical features mentioned above. We will argue that methods with a strong hierarchical component, the fastest in many situations, are too sensitive to metric changes. We choose to focus on separator-based methods instead, since they are much more robust. Interestingly, however, these algorithms have been neglected in recent research, since previously published results made them seem uncompetitive. The highest reported speedups [39] over Dijkstra's algorithm were lower than 60, compared to thousands or millions with other methods. By combining new concepts with careful engineering, we significantly improve the performance of this approach, easily enabling interactive applications.

One of our main contributions is the distinction between *topological* and *metric* properties of the network. The *topology* is the graph structure of the network together with a set of static properties of each road segment or turn, such as physical length, number of lanes, road category, speed limit, one- or two-way, and turn types. The *metric* encodes the actual cost of traversing a road segment or taking a turn. It can often be described compactly, as a function that maps (in constant time) the static properties of an arc/turn into a cost. For example, in the *travel time* metric (assuming free-flowing traffic), the cost of an arc may be its length divided by its speed limit. We assume the topology is shared by the metrics and rarely changes, while metrics may change quite often and can even be user-specific.

To exploit this separation, we consider algorithms for realistic route planning with *three stages.* The first, *metric-independent preprocessing*, may be relatively slow, since it is run infrequently. It takes only the graph topology as input, and may produce a fair amount of auxiliary data (comparable to the input size). The second stage, *metric customization*, is run once for each metric, and must be much quicker (a few seconds) and produce little data—a small fraction of the original graph. Finally, the *query stage* uses the outputs of the first two stages and must be fast enough for real-time applications. We call the resulting approach *Customizable Route Planning* (CRP).

We stress that CRP is not meant to compete with the fastest existing methods on individual metrics. For "well-behaved" metrics (such as travel times), our queries are somewhat slower than the best hierarchical methods. However, CRP queries are robust and suitable for real-time applications with arbitrary metrics, including those for which the hierarchical methods fail. CRP can process new metrics very quickly (orders of magnitude faster than any previous approach), and the metric-specific information is small enough to allow multiple metrics to be kept in memory at once. We achieve this by revisiting and thoroughly reengineering known acceleration techniques, and combining them with recent advances in graph partitioning.

This paper is organized as follows. In Section 2 we formally define the problem we solve, and explore the design space by analyzing the applicability of existing algorithms to our setting. Section 3 discusses overlay graphs, the foundation of our approach. We discuss basic data structures and modeling issues in Section 4. Our core routing engine is described in Section 5, with further optional optimizations discussed in Section 6. In Section 7 we present an extensive experimental evaluation of our method, including a comparison with alternative approaches. Section 8 concludes with final remarks.

2 Framing the Problem

This section provides a precise definition of the basic problem we address, and discusses potential approaches to solve it.

Formally, we consider a graph $G = (V, A)$, with a nonnegative *cost function* $\ell(v, w)$ associated with each arc $(v, w) \in A$. Our focus is on road networks, where vertices represent intersections, arcs represent road segments, and costs are computed from the properties of the road segments (e.g., travel time or length). A path $P = (v_0, \ldots, v_k)$ is a sequence of vertices with $(v_i, v_{i+1}) \in A$, and its *cost* is defined as $\ell(P) =$

$\sum_{i=0}^{k-1} \ell(v_i, v_{i+1})$. The basic problem we consider is computing point-to-point shortest paths. Given a source s and a target t, we must find the distance $dist(s,t)$ between them, defined as the length $\ell(Opt)$ of the shortest path Opt in G from s to t. We will ignore turn restrictions and turn costs for now, but will discuss them in detail in Section 4.1.

This problem has a well-known solution: Dijkstra's algorithm [25]. It processes vertices in increasing order of distance from s, and stops when t is reached. Its running time depends on the data structure used to keep track of the next vertex to scan. It takes $O(m + n \log n)$ time (with $n = |V|$ and $m = |A|$) with Fibonacci heaps [28], or $O(m+n \log C/\log\log C)$ time with multilevel buckets [24], which is applicable when arc lengths are integers bounded by C. In practice, the data structure overhead is quite small, and the algorithm is only two to three times slower than a simple breadth-first search [34]. One can save time by running a bidirectional version of the algorithm: in addition to running a standard *forward* search from s, it also runs a *reverse* (or *backward*) search from t. It stops when the searches meet.

On continental road networks, however, even carefully tuned versions of Dijkstra's algorithm would still take a few seconds on a modern server to answer a long-range query [56, 14]. This is unacceptable for interactive map services. Therefore, in practice one must rely on speedup techniques, which use a (relatively slow) preprocessing phase to compute additional information that helps accelerate queries. There is a remarkably wide selection of such techniques, with different tradeoffs between preprocessing time, space requirements, query time, and robustness to metric changes. The remainder of this section discusses how well these techniques fit the design goals set forth in Section 1. Recall that our main design goals are as follows: we must support interactive queries on *arbitrary* metrics, preprocessing must be quick (a matter of seconds), and the additional space overhead per cost function should be as small as possible.

Some of the most successful existing methods—such as reach-based routing [36], contraction hierarchies [32], SHARC [10], transit node routing [6, 5, 55, 4], and hub labels [1, 2, 17]—rely on the strong *hierarchy* of road networks with travel times. Intuitively, they use the fact that shortest paths between vertices in two faraway regions of the graph tend to use the same major roads.

The prototypical method among these is contraction hierarchies (CH). During preprocessing, CH heuristically sorts the vertices in increasing order of importance, and *shortcuts* them in this order. (To *shortcut* v, one temporarily removes it from the graph and adds as few arcs between its neighbors as necessary to preserve distances.) Queries run bidirectional Dijkstra, but only follow arcs or shortcuts to more important vertices. This rather simple approach works surprisingly well. On a continental-sized road network with tens of millions of vertices, using travel times as cost function (but ignoring turn costs), preprocessing is a matter of minutes, and queries visit only a few hundred vertices, resulting in query times well below 1 ms.

For metrics that exhibit strong hierarchies, such as travel times, CH has many of the features we want. Queries are sufficiently quick, and preprocessing is almost fast enough to enable real-time traffic. Moreover, Geisberger et al. [32] show that if a metric changes only slightly (as in most traffic scenarios), one can keep the order and recompute the shortcuts in about a minute on a standard server. Unfortunately, an order that works for one metric may not work for a substantially different metric (e.g., travel times and distances). Furthermore, queries are much slower on metrics with less-pronounced hierarchies [11]. For instance, minimizing distances instead of travel times leads to queries up to 10 times slower. This does not render algorithms such as CH impractical, but one should keep in mind that the distance metric is still far from the worst case. It still has a fairly strong hierarchy: since major freeways tend to be straighter than local roads, they are still more likely to be used by long shortest paths. Other metrics, including some natural ones, are less forgiving, in particular in the presence of turns. More crucially, the preprocessing stage can become impractical (in terms of space and time) for bad metrics, as Section 7 will show.

Other techniques, such as PCD [48], ALT [35], and arc flags [38, 45], are based on *goal direction*, i.e., they try to reduce the search space by guiding the search towards the target. Although they produce the same amount of auxiliary data for any metric, queries are not robust, and can be as slow as Dijkstra for bad metrics. Even for travel times, PCD and ALT are not competitive with other methods.

A third approach is based on *graph separators* [42, 60, 43, 61, 39]. During preprocessing, one computes a multilevel partition of the graph to create a series of interconnected *overlay graphs* (smaller graphs that preserve the distances between a subset of the vertices in the original graph). A query starts at the lowest

(local) level and moves to higher (global) levels as it progresses. These techniques, which predate hierarchy-based methods, have been widely studied, but recently dismissed as inadequate. Their query times are generally regarded as uncompetitive in practice, and they have not been tested on continental-sized road networks. The exceptions are recent extended variants [18, 52]; they achieve good query times, but only by adding many more arcs during preprocessing, which is costly in time and space. Despite these drawbacks, the fact that preprocessing and query times are essentially metric-independent makes separator-based methods the most natural fit for our problem.

There has also been previous work on variants of the route planning problem that deal with multiple metrics in a nontrivial way. The preprocessing of SHARC [10] can be modified to handle multiple (known) metrics at once. In the *flexible routing problem* [30, 29], one must answer queries on linear combinations of a small set of metrics (typically two or three) known in advance. Geisberger et at. [31] extend this idea to handle a predefined set of constraints on arcs, which can be combined at query time to handle scenarios like height and weight restrictions. Delling and Wagner [20] consider multicriteria optimization, where one must find Pareto-optimal paths among multiple metrics. ALT [35] and CH [32] can adapt to small changes in a benign base metric without rerunning preprocessing in full. All these approaches must know the base metrics in advance, and for good performance the metrics must be few, well-behaved, and similar to one another. In practice, even seemingly small changes to the metric (such as moderate U-turn costs) render some approaches impractical. In contrast, we must process metrics as they come (in the presence of traffic jams, for instance), and assume nothing about them.

3 Overlay Graphs

Having concluded that separator-based techniques are the best fit for our requirements, we now discuss this approach in more detail. In particular, we formally define partitions and revisit the existing technique of *partition-based overlay graphs* and its variants, with emphasis on how it fits our purposes.

3.1 Partitions

A *partition* of V is a family $\mathcal{C} = \{C_0, \dots, C_k\}$ of *cells* (sets) $C_i \subseteq V$ with each $v \in V$ contained in exactly one cell C_i. Let U be the size (number of vertices) of the biggest cell. A multilevel partition of V is a family of partitions $\{\mathcal{C}^0, \dots, \mathcal{C}^L\}$, where l denotes the *level* of a partition $\mathcal{C}^l$ and U^l represents the size of the biggest cell on level l. We set $U^0 = 1$, i.e., the level 0 contains only singletons. To simplify definitions, we also set $\mathcal{C}^{L+1} = V$. Throughout this paper, we only use *nested* multilevel partitions, i.e., for each $l \leq L$ and each cell $C_i^l \in \mathcal{C}^l$, there exists a cell $C_j^{l+1} \in \mathcal{C}^{l+1}$ (called the *supercell* of C_i^l) with $C_i^l \subseteq C_j^{l+1}$. Conversely, we call C_i^l a *subcell* of C_j^{l+1} if C_j^{l+1} is the supercell of C_i^l. Note that we denote by L the number of levels and that the supercell of a level-L cell is V. We denote by $\mathrm{c}_l(v)$ the cell that contains v on level l. To simplify notation, when $L = 1$ we may use $\mathrm{c}(v)$ instead of $\mathrm{c}_1(v)$. A *boundary* (or *cut*) arc on level l is an arc with endpoints in different level-l cells; a *boundary vertex* on level l is a vertex with at least one neighbor in another level-l cell. Note that, for nested multilevel partitions, boundary arcs are nested as well: a boundary arc at level l is also a boundary arc on all levels below.

3.2 Basic Algorithm

The preprocessing of the partition-based overlay graphs speedup technique [60] first finds a partition of the input graph and then builds a graph H containing all boundary vertices and boundary arcs of G. It then builds a *clique* for each cell C: for every pair (v, w) of boundary vertices in C, it creates an arc (v, w) whose cost is the same as the shortest path (restricted to C) between v and w (or infinity if w is not reachable from v using only arcs in C). See Fig. 1. One can determine the costs of these *shortcut* arcs by running Dijkstra from each boundary vertex.

Theorem 1. *H is an overlay of G.*

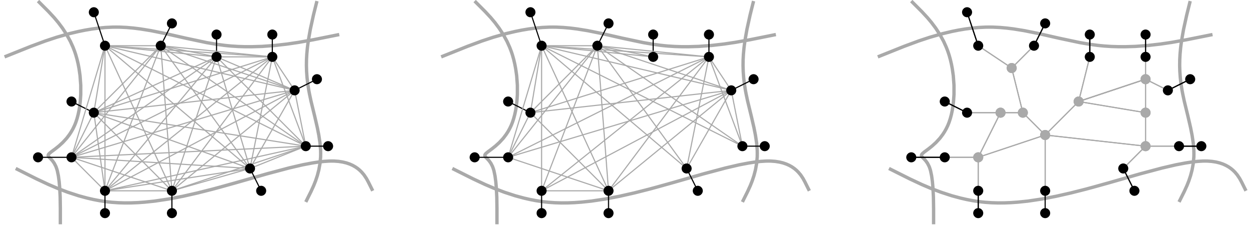

Fig. 1. Three possible ways of preserving distances within the overlay graph. Storing full cliques (left), performing arc reduction on the clique arcs (middle), and storing a skeleton (right).

Proof. For any two vertices u, v in H, we must show that the distance between them is the same in G and H (i.e., that $dist_G(u, v) = dist_H(u, v)$). By construction, every arc we add to H corresponds to a path of equal cost in G, so $d_H(u, v) \geq d_G(u, v)$ for all $u, v \in H$ (distances cannot decrease). Consider the shortest path P_{uv} between u and v in G. It can be seen a sequence of subpaths between consecutive boundary vertices. Each of these subpaths is either a boundary arc (which belongs to H by construction) or a (shortest) path within a cell between two boundary vertices (which corresponds to a shortcut arc in H, also by construction). This means there is a u–v path in H with the same cost as P_{uv}, ensuring that $d_H(u, v) \leq d_G(u, v)$ and concluding our proof.

Note that the theorem holds even though some shortcuts added to H are not necessarily shortest paths in either G or H. Such shortcuts are redundant, but do not affect correctness.

To perform a query between s and t, one runs a bidirectional version of Dijkstra's algorithm on the graph consisting of the union of H, $\text{C}(s)$, and $\text{C}(t)$, called the *search graph*. The fact that H is an overlay of G ensures queries are correct, as shown by Holzer et al. [39]. Intuitively, consider the shortest path in G from s to t. It consists of three parts: a maximal prefix entirely in $\text{C}(s)$, a maximal suffix entirely in $\text{C}(t)$, and the remaining "middle" part. By construction, the segments in $\text{C}(s)$ and $\text{C}(t)$ are part of the search space, and the overlay H contains a path of equal cost as the path representing the middle part, with the same start and end vertex. Since the overlay does not decrease the distances between any two vertices, we find a path of equal cost to the shortest in G.

To accelerate queries, partition-based approaches often use multiple levels of overlay graphs. For each level i of the partition, one creates a graph H_i as before: it includes all boundary arcs, plus an overlay linking the boundary vertices within a cell. If one builds the overlays in a bottom-up fashion, one can use H_{i-1} (instead of G) when running the Dijkstra searches from the boundary vertices of H_i. This accelerates the computation of the high-level overlay graphs significantly. During queries, we can skip cells that contain neither s nor t. More precisely, an s–t query runs bidirectional Dijkstra on a restricted search graph G_{st}. An arc (v, w) from H_i will be in G_{st} if both v and w are in the same cell as s or t on level $i + 1$, but not on level i.

3.3 Pruning the Overlay Graph

The basic overlay approach stores a full clique per cell for each metric, which seems wasteful. Many shortcuts are not necessary to preserve the distances within the overlay graph because the shortest path between their endpoints actually uses some arcs outside the cell. (These shortcuts are introduced because construction considers only paths within a cell.) This happens particularly often for well-behaved metrics. A first approach to identify and remove such arcs is *arc reduction* [60]. After computing all cliques, Dijkstra's algorithm is run from each vertex u in H, stopping as soon as all neighbors of v (in H) are scanned. Then, one can remove all arcs (u, v) from H with $dist(u, v) < \ell(u, v)$. These searches are usually quick (they only visit the overlay), and we can avoid pathological cases by having a hard bound on the size of the search space. Although this can miss some opportunities for pruning, it preserves correctness.

A more aggressive technique to further reduce the size of the overlay graph is to preserve some internal cell vertices [61, 39, 18]. If $B = \{v_1, v_2, \ldots, v_k\}$ is the set of boundary vertices of a cell, let T_i be the shortest path tree (restricted to the cell) rooted at v_i, and let T'_i be the subtree of T_i consisting of the vertices with descendants in B. One can take the union $S = \cup_{i=1}^{k} T'_i$ of these subtrees, and shortcut all internal vertices with two neighbors or fewer. We call the resulting object a *skeleton graph*; it is technically not an overlay, since the distances between its internal vertices may not be preserved. But it does preserve the distances between all *boundary* vertices, which is enough to ensure correctness [39]. Both arc reduction and skeletons can be naturally extended to work with multiple levels of overlay graphs. In Section 5.4, we will discuss which of these optimizations carry over to a realistic routing environment.

4 Modeling

The first step for building a fully-fledged routing engine is to incorporate all modeling constraints, such as turn restrictions and turn costs. Moreover, we want to be able to handle multiple metrics without explicitly storing a traversal cost for each arc and metric. In this section, we explain the data structures we use to support these two requirements and show how Dijkstra's algorithm can be adapted. These are both building blocks to the fast routing engine we will present in Section 5.

4.1 Turns

Most previous work on route planning algorithms has considered a simplified representation of road networks, with each intersection corresponding to a single vertex (see Fig. 2). This is not enough for a real-world production system, since it does not account for turn costs (or restrictions, a special case). In principle, any algorithm can handle turns simply by working on an expanded graph. The most straightforward approach is to introduce two vertices, a *head vertex* and a *tail vertex* per directed arc. Tail and head vertices are connected as before by a *road arc*. In addition, *turn arcs* model allowed turns at an intersection by linking the head vertex of an arc to the tail vertex of another. This *fully-blown* approach is wasteful, however; in particular, tail vertices now always have out-degree 1 and head vertices have in-degree 1. A slightly more compact representation is *arc-based*: one only keeps the tail vertices of each intersection, and each arc is a road segment followed by a turn.

To save even more space, we propose a *compact representation* in which each intersection becomes a single vertex with some associated information. If a vertex u has p incoming and q outgoing arcs, we associate a $p \times q$ *turn table* T_u to it, where $T_u[i, j]$ represents the cost of turning from the i-th incoming arc into the j-th outgoing arc at u. In addition, we store with each arc (v, w) its *tail order* (its position among v's outgoing arcs) and its *head order* (its position among w's incoming arcs). These orders may be arbitrary. Since degrees are small in practice, 8 bits for each suffice. For each arc (u, v), we say that its head corresponds to an *entry point* at v, and its tail corresponds to an *exit point* at u. Note that the entry/exit points translate directly to the head/tail vertices in the fully blown model.

In practice, many vertices tend to share the same turn table. The total number of such *intersection types* is modest—in the thousands rather than millions. For example, many degree-four vertices in the United

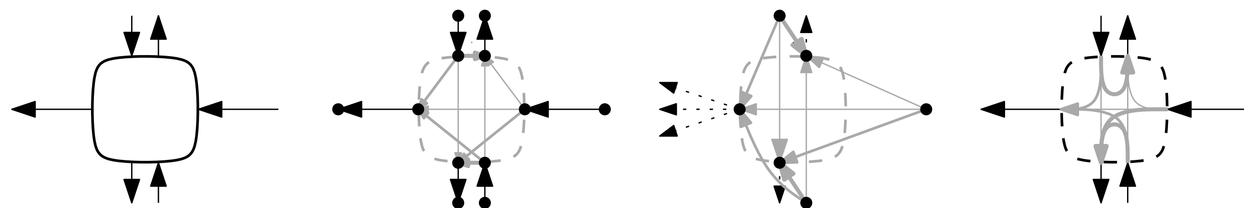

Fig. 2. Turn representations (from left): none, fully expanded, arc-based, and compact.

States have four-way stop signs with exactly the same turn costs. Each distinct turn table is thus stored only once, and each vertex keeps a pointer to the appropriate type, with little overhead.

Finally, we note that data representing real-world road networks often comes with so-called *polyvalent turn restrictions*, as observed before [58, 59]. Depending on which turn a driver takes at a particular intersection, certain turns may be forbidden at the next. For example, if one turns right onto a multilane street, one is often forbidden to take a left exit that is just a few meters ahead. Drivers who are already in the multilane street have no such constraint. Such (rare) scenarios can be handled by locally "blowing up" the graph with additional arcs and/or vertices for each affected intersection. In this example, representing this segment of the multilane street by two (parallel) arcs is enough to accurately represent all valid paths. On our proprietary data, the number of such polyvalent turns is very small, and thus the size of the routing graph increases only slightly.

4.2 Queries

The arc-based representation allows us to use Dijkstra's algorithm (unmodified) to answer point-to-point queries. Since the graph is bigger, the algorithm becomes about three times slower than on a graph with no turns at all, as Section 7 will show. Similarly, most speedup techniques can be used without further modifications, although the effects on preprocessing and query times are not as predictable, since adding turn costs may change the shortest path structure of a road network significantly.

Dijkstra's Algorithm on the Compact Graph. Recall that on a standard graph (without turn information), Dijkstra's algorithm computes the distance from a vertex s to any other vertex by scanning all vertices in non-increasing distance from s. This ensures that each vertex is scanned exactly once. Dijkstra's algorithm becomes more complicated on the compact representation. For correctness, the algorithm can no longer maintain one distance label per vertex (intersection); it must operate on entry points instead. As a result, it may now visit each vertex multiple times, once for each entry point. We implement this by maintaining triples (v, i, d) in the heap, where v is a vertex, i the order of an entry point at v, and d a distance label. The algorithm is initialized by $(s, i, 0)$ indicating that we start the query from entry point i at vertex s. (Note that one can generalize this to allow queries starting anywhere along an arc by inserting s, i with an offset into the queue.) The distance value d of a label (v, i, d) then indicates the cost of the best path seen so far from the source to the entry point i at vertex v. Intuitively, this approach essentially simulates the execution of the arc-based representation; accordingly, it is roughly three times slower than the non-turn version.

We propose a *stalling* technique that can reduce this slowdown to a factor of about 2 in practice. The general idea is as follows. During the search, scanning an entry point of an intersection immediately gives us upper bounds on the (implicit) distance labels of its exit points. This allows us to only scan another entry point if its own distance label is small enough to potentially improve at least one exit point.

To determine quickly whether a distance label at an entry point can improve the implicit distance labels of the exit points, we keep an array of size p (the number of entry points) for each vertex v, with each entry denoted by $b_v[i]$. We initialize all values in the array with ∞. Whenever we scan an entry point i at a vertex v with a distance d, we set each $b_v[k]$ to $\min\{b_v[k], d + \max_j\{T_v[i, j] - T_v[k, j]\}\}$, with j denoting the exit points of v. However, to properly deal with turn restrictions, we must not update $b_v[k]$ if there exists an exit point that can be reached from k, but not from i. We then prune the search as follows: when processing an element (u, i, d), we only insert it into the heap if $d \leq b_u[i]$.

We use two optimizations for stalling. First, we do not maintain $b_v[i]$ explicitly, but instead store it as the tentative distance label for the i-th entry point of v, which has the same effect. Second, we precompute (during customization) the $\max_k\{T_v[i, k] - T_v[j, k]\}$ entries for all pairs of entry points of each vertex. Note that these *stalling tables* are unique per intersection type, so we again store each only once, reducing the overhead. Since the number of unique intersection types is small, the additional precomputation time is negligible.

Bidirectional Search. To implement bidirectional search on the compact model, we maintain a tentative shortest path distance μ, initialized by ∞. Then, we perform a forward search from an entry point at s, operating as described above, and a backward search from an *exit point* of t that operates on the exit points of the vertices. Both searches use stalling as described above, but the backward search uses stalling arrays (and a precomputed stalling table) for the exit points. Whenever we scan a vertex that has been seen from the other side, we evaluate all possible turns between all entry and exit points of the intersection and check whether we can improve μ. We can stop the search as soon as the sum of the minimum keys in both priority queues exceeds μ. Note that this algorithm basically performs a search from the head vertex (s) of an arc to the tail (t) of another.

Arc-based Queries. In real-world applications, we often do not want to compute routes between intersections, but between points (addresses) along road segments. Hence, we define the input to our routing engine to be two arcs a_s and a_t with real-valued offsets $o_s, o_t \in [0, 1]$ representing the exact start/end point along the arc.

For unidirectional Dijkstra, we can handle this as follows. We maintain a tentative shortest path cost μ, initialized by ∞. We then initialize the search with the triple $(h(a_s), i, (1 - o_s) \cdot \ell(a_s))$, where $h(a_s)$ is the head vertex of a_s and i is the head order of a_s. The algorithm then processes vertices as described above, but whenever we scan the head vertex $h(a_t)$ of a_t, we update μ taking o_t into account. Handling arc-to-arc queries with bidirectional Dijkstra is even easier, since the backward search starts from a tail vertex anyway. We only need to use the correct offset o_t to initialize the backward search with $(t(a_t), j, o_t \cdot \ell(a_t))$, where $t(a_t)$ is the tail vertex of a_t and j the tail order of a_t. The special case where source and target are on the same arc can be handled explicitly. For simplicity, whenever we talk about point-to-point queries in the rest of this paper, we actually mean arc-to-arc queries.

4.3 Graph Data Structure

Our implementation represents the topology of the original graph using standard adjacency arrays, in which each vertex has a reference to an array representing its incident arcs; see [49] for further details on this data structure. In addition, each arc stores several attributes (if available), such as length, speed limit, road category, slope, and a bitmask encoding further properties (height restricted, open only for emergency vehicles, one-way street, and so on). The actual traversal cost (such as travel times or travel distances) is then computed as a function of these attributes. This allows us to define a metric using only a few bytes of memory.

Similarly, we do not store the turn tables (described in Section 4.1) explicitly. Instead, the turn table only stores identifiers of *turn types* (such as "right turn" or "left turn against incoming traffic"). The number of different turn types is quite small in practice (no more than a few hundred), which allows us to define specific metric-dependent turn costs and restrictions with a few bytes per metric. It also allows us to further decrease the space overhead of the compact turn representation, since the entries of the turn table can be packed into a few bits each.

Note that not all metrics can be encoded as described above. Although many traffic jams can be modeled by temporally changing the category of a road segment, personal preferences (like avoiding a particular intersection) cannot. We use hashing for these special cases, eliminating the need to store specific costs for all arcs for each user.

5 Realistic Routing with Overlays

In this section we describe our routing algorithm in full. The foundation of our approach (CRP) is the basic partition-based overlay approach from Section 3. As already mentioned, however, this approach has been previously tested in practice and deemed too slow. This section proposes several modifications and enhancements that make it practical.

One key element of CRP is the separation of the standard preprocessing algorithm in two parts. The first, the *metric-independent* preprocessing, only considers the topology (and *no* arc costs) of the road network and generates some auxiliary data. The second phase, *metric customization*, takes the metric information, the graph, and the auxiliary data to compute some more data, which is metric-specific. The query algorithm can then use the graph and the data generated by both preprocessing phases.

Our motivation for dividing the preprocessing algorithm in two phases is that they have very different properties. The *metric-independent* data changes very infrequently and is shared among all metrics; in contrast, the data produced by the second stage is specific to a single metric, and can change quite frequently. This distinction is crucial to guide our design decisions: our goal is to optimize the time and space requirements of the second stage (customization) by shifting as much effort as possible to the first stage (metric-independent preprocessing). Therefore, the metric-independent phase can be relatively slow (several minutes) and produce a relatively large amount of data (but still linear in the size of the input). In contrast, the customization phase should run much faster (ideally within a few seconds) and produce as little data as possible. Together, these properties enable features such as real-time traffic updates and support for multiple (even user-specific) metrics simultaneously. Finally, we cannot lose sight of queries, which must fast enough for interactive applications.

To achieve all these goals simultaneously, we must engineer all aspects of the partition-based overlay approach. For example, we must make careful choices of data structures to enable the separation between metric-dependent and metric-independent information. Moreover, we introduce new concepts that significantly improve on previous implementations. In this section, we will discuss each phase in turn, and also explain why we choose *not* to use some of the optimizations described in Section 3.3 for the partition-based overlay approach.

5.1 Metric-Independent Preprocessing

During the metric-independent part of the preprocessing, we perform a multilevel partitioning of the network, build the topology of the overlay graphs, and set up additional data structures that will be used by the customization phase. As already mentioned, we try to perform as much work as possible in this phase, since it only runs when the topology of the network changes. This is quite rare, especially considering that temporary road blocks can be handled as cost increases (to infinity) rather than changes in topology.

Partitioning. The performance of the overlay approach depends heavily on the number of boundary arcs of the underlying partition. Still, previous implementations of the overlay approach used out-of-box graph partitioning algorithms like METIS [44], SCOTCH [53], planar separators [46, 40], or grid partitions [43] for this step. Since these algorithms are not tailored to road networks, they output partitions of rather poor quality. More recently, Delling et al. [16] developed PUNCH, a graph partitioning algorithm that routinely finds solutions with half as many boundary arcs (or fewer) as the general-purpose petitioners do. The main reason for this superior quality is that PUNCH exploits the natural cuts that exist in road networks, such as rivers, mountains, and highways. It identifies these cuts by running various local maximum flow computations on the full graph, then contracts all arcs of the graph that do not participate in natural cuts. The final partition is obtained by running heavy heuristics on this much smaller *fragment graph*. PUNCH outputs a high-quality partition with at most U vertices per cell, where U is an input parameter. Users can also specify additional parameters to determine how much effort (time) is spent in finding natural cuts and the heavy heuristics. In general, more time leads to better partitions, with fewer cut arcs. For further details, we refer the interested reader to the original work [16]. Moreover, note that, since the initial publication of PUNCH, great advances in general graph partitioning have been achieved. State-of-the-art partitioners like KaHiP [57], which use natural cut heuristics from PUNCH, generate partitions of quality similar to PUNCH.

We run PUNCH on the input graph with the following arc costs. Road segments open in both directions (most roads) have cost 2, while one-way road segments (such as freeways) get cost 1. We then use PUNCH to generate an L-level partition (with maximum cell sizes $U_1, \ldots, U_L$) in top-down fashion. We first run PUNCH with parameter U_L to obtain the top-level cells. Cells in lower levels are then obtained by running

PUNCH on individual cells of the level immediately above. To ensure similar running times for all levels, we set the PUNCH parameters so that it spends more effort on the topmost partition, and less on lower ones. The exact parameters can be found in Section 7.

As our experiments will show, PUNCH is slower than popular general partitioners (such as METIS). Since the metric-independent preprocessing is run very infrequently, however, its running time is not a major priority in our setting. Customization and query times, in contrast, should be as fast as possible, and (because we create cliques) their running times roughly depend on the square of the cut size. Still, PUNCH runs in tens of minutes on a standard server, which is fast enough to repartition the network frequently. Moreover, one can handle a moderate number of new road segments by simply patching the original partition. If both endpoints of a new arc are in the same bottom-level cell, no change is needed; if they are in different cells, we simply add a new cut arc. A full reoptimization is only needed when the quality of the partition decreases significantly.

To represent a multilevel partition within our algorithm, we use an approach similar to the one introduced for SHARC [10]. Each cell (at level 1 or higher) is assigned a unique sequential identifier within its supercell. Since the number of levels and the number of cells per supercell are small, all identifiers for a vertex v can be packed in a single 64-bit integer $\textsc{pv}(v)$, with the lower bits representing lower levels. For additional space savings, we actually store $\textsc{pv}$ only once for each cell on level 1, since it can be shared by all vertices in the cell. To determine $\textsc{pv}(v)$, we first look up the cell v is assigned to on level 1, then access its $\textsc{pv}$ value. This data structure requires $4 \cdot n + 8 \cdot |\mathcal{C}^1|$ bytes, including the 32-bit integer we store with each vertex to represent its level-1 cell.

Overlay Topology. After partitioning the graph, we set up the metric-independent data structures of the overlay graph. The goal is to store the overlay topology only once, and allow it to be shared among all metrics. Regardless of the cost function used during customization and queries, we know the number of vertices and arcs of each overlay graph. For each boundary arc (u, v), we store two overlay vertices: u'_H, the appropriate exit point of intersection u, and v''_H, the appropriate entry point of intersection v. We call u'_H an *exit vertex* of cell $\textsc{c}_1(u)$ and v''_H an *entry vertex* of cell $\textsc{c}_1(v)$. We also add an arc (u'_H, v''_H) to the overlay, and store a pointer from this arc to the original arc in G to determine its cost when needed. Note that if the reverse arc (v, u) exists, we will add two additional vertices as well. See Figure 3 for an example. Also note that by introducing entry and exit vertices, we actually have a complete bipartite graph per cell (and not a clique).

Recall that we use nested partitions. Hence, a vertex (or cut arc) in any of the overlay graphs must be in H_1 as well. Therefore, we can store each overlay vertex only once with markers indicating whether it is a boundary vertex on each level. To improve locality, we assign IDs to overlay vertices such that the boundary vertices of the highest level have the lowest IDs, followed by the boundary vertices of the second highest level (which are not on the highest), and so on. Within a level, we keep the same relative ordering as in the original graph.

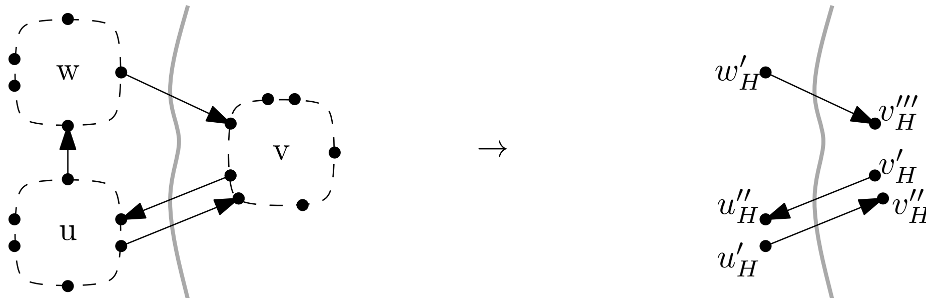

Fig. 3. Building the overlay graph from the cut arcs. Each cut arc (u, v) yields two vertices u'_H,v''_H in the overlay graph.

During queries, we must be able to switch efficiently (in both directions) between the original graph and the overlay. As Section 5.3 will explain, the switch can only happen at cut arcs. Therefore, for each vertex of the overlay graph, we explicitly store the corresponding vertex in the original graph, as well as which entry/exit point it refers to. For the other direction, we use a hash table to map triples containing the vertex (intersection), turn order, and point type (exit or entry) to vertices in the overlay. Recall that only boundary vertices must be in the hash table.

Moreover, to speed up the customization phase, for each vertex in G we also store a *local identifier*. Within each level-1 cell, vertices have local identifiers that are unique and sequential (between 0 and $U_1 - 1$). For the same reason, each level-1 cell keeps the list of vertices it contains.

Finally, for each cell we must represent an overlay graph connecting its boundary vertices. Since our overlays are collections of complete bipartite graphs, we can represent them efficiently as matrices. A cell with p entry points and q exit points corresponds to a $p \times q$ matrix in which position (i, j) contains the cost of the shortest path (within the cell) from the cell's i-th entry vertex to its j-th exit vertex. We need one matrix for each cell in the overlay (on all levels). They can be compactly represented as a single (one-dimensional) array W: it suffices to interpret the matrix associated to each cell C as an array of length $p_C q_C$ (in row-major order), then concatenate all arrays.

Note that the actual contents of W cannot be computed during the metric-independent preprocessing, since they depend on the cost function. In fact, each metric will have its own version of W. But we still need auxiliary data structures to access and evaluate the matrices efficiently, however. Since they are shared by all metrics, they are set up during the metric-independent preprocessing stage.

First, for each cell C in the overlay graph, we keep three integers: p_C (the number of entry points), q_C (the number of exit points), and f_C (the position in W where the first entry of C's matrix is represented). During customization and queries, the cost of the shortcut between the i-th entry point and the j-th exit point of C will be stored in $W[f_C + i q_C + j]$.

In addition, we maintain maps to translate (both ways) between a vertex identifier in the overlay graph and its position among the entry or exit vertices in the corresponding cells. More precisely, if an overlay vertex v is the i-th entry (or exit) point of its cell C at level l, we must be able to translate (v, l) into i, as well as (C, i) into v. Since identifiers are sequential and the number of levels is a small constant, simple arrays suffice for that.

Finally, to allow access to the cell number of an overlay vertex on each level, we also store the encoded level information PV for the overlay graph. This increases the memory overhead slightly (by 8 bytes per overlay vertex), but by avoiding expensive indirections it accelerates customization and queries.

Note that the vertices of the overlay graph *do not* have turn tables; the actual turn costs are encoded into the cost of the shortcut arcs, which we determine during the next phase.

5.2 Customization

The customization phase has access to the actual cost function that must be optimized during queries. Because we have the metric-independent data structures in place, all we need to do is compute the entries of the above-mentioned array W, which represents the costs of all shortcuts between entry and exit vertices within cells.

We compute these distances in a bottom-up fashion, one cell at a time. Consider a cell C in H_1 (the first overlay level). For each entry (overlay) vertex v in C, we run Dijkstra's algorithm in G (restricted to C) until the priority queue is empty. This computes the distances to all reachable exit vertices of C. Since we work on the underlying graph, we must use the turn-aware implementation of Dijkstra, as explained in Section 4.1.

A cell C at a higher level H_i (for $i > 1$) can be processed similarly, with one major difference. Instead of working on the original graph, we can work on the subgraph of H_{i-1} (the overlay level immediately below) corresponding to subcells of C. This subgraph is much smaller than the corresponding subgraph of G. In addition, since overlay graphs have no (explicit) turns, we can just apply the standard version of Dijkstra's algorithm, which tends to be faster.

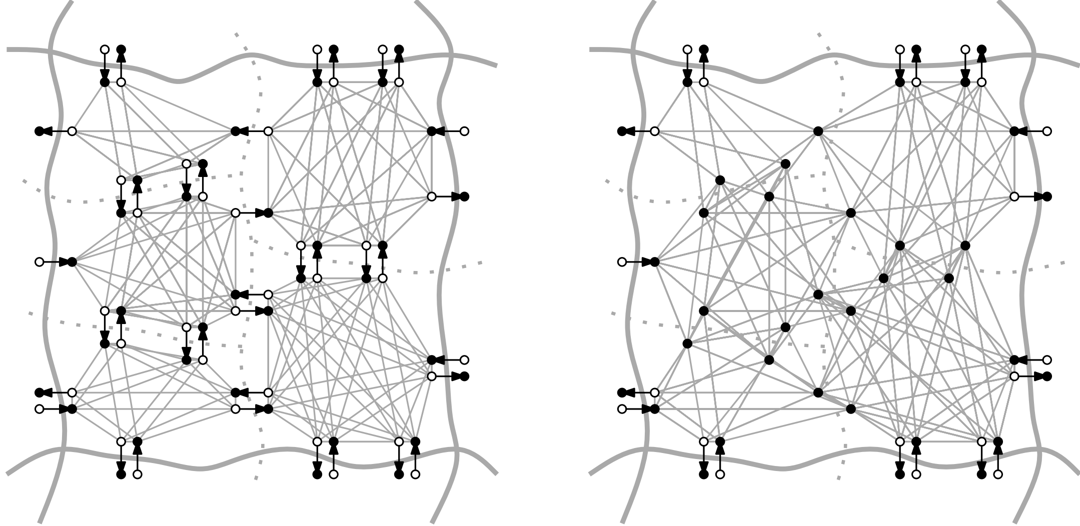

Fig. 4. The overlay graph before (left) and after (right) pruning.

Acceleration Techniques. Since customization is executed whenever a new metric must be optimized, we want this phase to be as fast as possible. We propose several optimizations that can speed up the basic algorithm describe above. We explain each optimization in turn, but they can be combined in the final algorithm.

Improving Locality. Conceptually, to process a cell C on level i we could operate on the full overlay graph H_{i-1}, but restricting the searches to vertices inside C. For efficiency, we actually create a temporary copy of the relevant subgraph G_C of H_{i-1} in a separate memory location, run our searches on it, then copy the results to the appropriate locations in W. This simplifies the searches, allows us to use sequential local IDs, and improves locality. For the lowest level, instead of operating on the turn-aware graph, we extract an arc-based (see Section 4.1) subgraph. This allows us to use standard (non-turn-aware) graph search algorithms and other optimizations, which we discuss next.

Pruning the Search Graph. To process a cell C of H_i, we must compute the distances between its entry and exit points. For a level $i > 1$, the graph G_C on which we operate is the union of subcell overlays (complete bipartite graphs) with some boundary arcs between them (see Fig. 4). Instead of searching G_C directly, we first contract its internal exit points. Since each such vertex has out-degree one (its outgoing arc is a boundary arc within C), this reduces the number of vertices and arcs in the search graph. Although C's own exit points must be preserved (they are the targets of our searches), they do not need to be scanned (they have no outgoing arcs). We do not perform this optimization on the lowest level (H_1), since the number of degree-one vertices is very small.

Alternative Algorithms. We can further accelerate customization by replacing Dijkstra's algorithm with the well-known Bellman-Ford algorithm [12, 27]. It starts by setting the distance label of the source vertex to 0, and all others to ∞. Each round then scans each vertex once, updating the distance label of its neighbors appropriately. For better performance, we only scan *active* vertices (i.e., those whose distance improved since the previous round) and stop when there is no active vertex left. While Bellman-Ford cannot scan fewer vertices than Dijkstra, its simplicity and better locality make it competitive for small graphs (such as the ones we operate on during customization). The number of rounds is bounded by the maximum number of arcs on any shortest path, which is small for reasonable metrics but linear in the worst case. One could therefore switch to Dijkstra's algorithm whenever the number of Bellman-Ford rounds reaches a given (constant) threshold.

For completeness, we also tested the Floyd-Warshall algorithm [26]. It computes shortest paths among *all* vertices in the graph, and we just extract the relevant distances. Its running time is cubic, but with its tight inner loop and good locality, it could be competitive with Bellman-Ford on denser graphs.

Multiple-source executions. Multiple runs of Dijkstra's algorithm (from different sources) can be accelerated if combined into a single execution [38, 64]. We apply this idea to the Bellman-Ford executions we perform within each cell. Let k be the number of simultaneous executions, from sources $s_1, \ldots, s_k$. For each vertex v, we keep k distance labels: $d_1(v), \ldots, d_k(v)$. All $d_i(s_i)$ values are initialized to zero (each s_i is the source of its own search), and all remaining $d_i(\cdot)$ values to ∞. All k sources s_i are initially marked as active. When Bellman-Ford scans an arc (v, w), we try to update all k distance labels of w at once: for each i, we set $d_i(w) \leftarrow \min\{d_i(w), d_i(v) + \ell(v, w)\}$. If any such distance label actually improves, we mark w as active. This simultaneous execution needs as many rounds as the worst of the k sources, but, by storing the k distances associated with a vertex contiguously in memory, locality is much better. In addition, it enables instruction-level parallelism [64], as discussed next.

Parallelism. Modern CPUs have extended instruction sets with SIMD (single instruction, multiple data) operations, which work on several pieces of data at once. In particular, the SSE instructions available in x86 CPUs can manipulate special 128-bit registers, allowing basic operations (such as additions and comparisons) on four 32-bit words in parallel.

Consider the simultaneous execution of Bellman-Ford from $k = 4$ sources, as above. When scanning v, we first store v's four distance labels in one SSE register. To process an arc (v, w), we store four copies of $\ell(v, w)$ into another register and use a single SSE instruction to add both registers. With an SSE comparison, we check if these tentative distances are smaller than the current distance labels for w (themselves loaded into an SSE register). If so, we take the minimum of both registers (in a single instruction) and mark w as active.

Besides using SIMD instructions, we can use core-level parallelism by assigning cells to distinct cores. In addition, we parallelize the highest overlay levels (where there are few cells per core) by further splitting the sources in each cell into sets of similar size, and allocating them to separate cores (each accessing the entire cell).

Phantom Levels. As our experiments will show, using more levels (up to a point) tends to lead to faster customization, since each level can operate on smaller graphs. Unfortunately, adding more levels also increases the metric-dependent space consumption. To avoid this downside, it often pays to introduce *phantom levels*, which are additional partition levels that are used only to accelerate customization, but are not kept for queries. This keeps the metric-dependent space unaffected. Note, however, that we still need to build the metric-independent data structures for all levels, increasing the metric-independent space consumption during customization.

Incremental Updates. Today's online map services receive a continuous stream of traffic information. If we are interested in the best route according to the current traffic situation, we need to update the overlay graphs. The obvious approach is to rerun the entire customization procedure as if we get a new metric. If only a few arcs $(u_0, v_0), \ldots, (u_k, v_k)$ change their costs, however, we can do better. We first identify all cells $c_l(u_i)$ and $c_l(v_i)$ for all updated arcs i and all levels l. Only these cells are *affected* by the update. Since we restrict searches during customization to the cells, we know that it is sufficient to rerun the customization only for these affected cells.

5.3 Query

Our query algorithm takes as input a source arc a_s, a target arc a_t, the original graph G, the overlay graph $H = \cup_i H_i$, and computes the shortest path between the head vertex s of a_s and the tail vertex t of a_t. We first explain a unidirectional algorithm that computes a path with shortcuts, then consider a bidirectional approach and explain how to translate shortcuts into the corresponding underlying arcs.

Basic Algorithm. Given any vertex v, define its *query level* $l_{st}(v)$ as the highest level such that v is not at the same cell as s or t. Equivalently, $l_{st}(v)$ is the maximum i such that $\textsc{c}_i(v) \cap \{s,t\} = \emptyset$. As discussed by Holzer et al. [39], this is the level at which v must be scanned if we find it during the search. The main idea is that one can skip cells that do not contain s or t, and use shortcuts instead.

To compute $l_{st}(v)$, we first determine the most significant differing bit of $\textsc{pv}(s)$ and $\textsc{pv}(v)$. (Recall that $\textsc{pv}(v)$ encodes the cell number of v on each level.) This bit indicates the topmost level $l_s(v)$ in which they differ. We do the same for $\textsc{pv}(t)$ and $\textsc{pv}(v)$ to determine $l_t(v)$. The minimum of $l_s(v)$ and $l_t(v)$ is $l_{st}(v)$.

The query algorithm maintains a distance label $d(u)$ for each entry u which can either be a vertex on the overlay or a pair (v,i) corresponding to the i-th entry point of v in the original graph. It also maintains the cost μ of the shortest path seen so far; all variables are initialized to ∞. Since we always start a query on the original graph, we set $d(s,i) = 0$ and add the corresponding entry to the priority queue.

Each iteration of the algorithm takes the minimum-distance entry from the queue, representing either an overlay vertex u or a pair (u,i) from the original graph. If the entry is a pair, we scan it using the turn-aware version of Dijkstra's algorithm (and look at its neighbors in G). Otherwise, we use the overlay graph at level $l_{st}(u)$, which does not have turns. In either case, the neighbors v of u are added to the priority queue with the appropriate distance labels. Note that a level transition occurs when u and v have different query levels; in particular, for transitions from or to level 0, we must translate between the two vertex identifiers for v (in G and in H), which can be done in constant time using the metric-independent data structures described in Section 5.1.

As described in arc-to-arc queries with plain unidirectional Dijkstra (Section 4.2), we update μ whenever we scan t, and the search terminates when it is about to scan a vertex whose distance label is greater than μ. We can retrieve the path with shortcuts by keeping a parent pointer for each vertex.

We apply several optimizations. First, by construction, each exit vertex u in the overlay has a single outgoing arc (u,v). Therefore, during the search we do not add u to the priority queue; instead, we traverse the arc (u,v) immediately and process v. This reduces the number of heap operations. We still store a distance value at u, though. A second optimization uses the fact that the maximum heap size can be computed in advance. Any search will scan at most all overlay vertices and the vertices in the cells of s and t on the lowest level. This allows us to preallocate all necessary data structures. To index the heap, we use local identifiers (for vertices in the source and target cells, with different offsets), and overlay identifiers otherwise. Finally, we keep track of all vertices touched by one quick, allowing for a quick initialization of the next one.

Bidirectional Search. We can accelerate queries even further by running bidirectional search. The forward search works as before, starting from the head vertex of a_s and operating on entry points of vertices. The backward search is symmetric: it starts from the tail vertex of a_t and works on the reverse graph, which means it maintains distance labels for *exit points* instead. Whenever we scan (during the forward or backward search) a vertex v that has been seen by the opposite direction, we update μ. Note that if $v \in G$, we have to evaluate all possible turns at v because the forward search operates on the entry points of v, and the backward search on the exit points. The search terminates as soon as the sum of the minimum keys of both priority queues (forward and backward) exceeds μ.

Note that, when processing the overlay graph, the backward search may access the matrices within each cell in a cache-inefficient way, since they are represented in row-major order. Keeping a second copy of each matrix (transposed) would only improve overall running times by about 15%, which is not enough to justify doubling the amount of metric-dependent data. We therefore opt to store each matrix only once.

Path Unpacking. Up to now, we have only discussed how to compute the distance between two arcs. Following the parent pointers of the meeting vertex (the vertex that was responsible for the last update of μ) of forward and backward searches, we obtain a path that potentially containing shortcuts. To obtain the complete path description as a sequence of arcs (or vertices) in the original graph, each shortcut in the result must be translated (*unpacked*) into the corresponding subpath. An obvious approach is to store this information for each shortcut explicitly, but this is wasteful in terms of space. Instead, we recursively unpack a level-i shortcut (v,w) by running bidirectional Dijkstra between v and w on level $i-1$, restricted

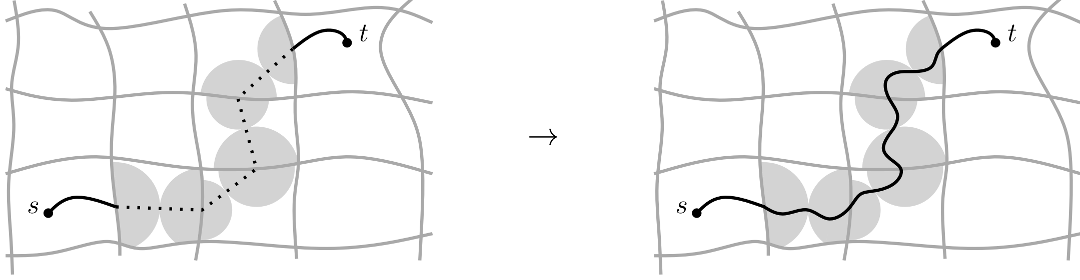

Fig. 5. Shortcut unpacking. A shortcut on the shortest path is processed by running a local version of bidirectional Dijkstra on the level below (left). This results in a path consisting of arcs from the level below (right).

to subcells of the level-i cell containing the shortcut. See Fig. 5 for an illustration. This does not increase the metric-dependent space consumption, and query times are still small enough. Note that disjoint cells can be handled in parallel.

If even faster unpacking times are needed, the customization phase could store a bit with each arc at level $i-1$ indicating whether it appears in a shortcut at level i. Only arcs with this bit set need to be visited during the unpacking process. This approach increases the metric-dependent amount of data stored only slightly, and does accelerate queries. Unfortunately, it complicates the customization step.

In practice, we use a simpler approach: we maintain a cache of frequently-used shortcuts. Each entry in the cache represents a level-i shortcut together with the corresponding sequence of level-$(i-1)$ shortcuts. Note that we have one cache with all shortcuts instead of having one cache for each level. As the experiments will show, with a standard least-recently used (LRU) update policy, even a small cache can accelerate path unpacking significantly.

Alternatives. In road networks, there often exist several routes from s to t with similar overall costs. That is why a common feature for map service applications is to report *alternative* routes besides the best route. While the best route corresponds to the shortest path in the underlying graph, characterizing a "good" alternative is less obvious. We follow the approach based on the concept of *admissible paths* [3]. Intuitively, an admissible path is significantly different from the shortest path, but it still "feels" reasonable and is not much longer. More precisely, given a shortest path Opt, they define a path P between s and t to be an (α, β, γ)-*admissible alternative* if it has the following properties:

1. *Limited sharing.* The sum of the costs of the arcs that appear in both Opt and P is at most $\gamma \cdot dist(s,t)$.
2. *Local optimality.* Every subpath of P shorter than $\alpha \cdot dist(s,t)$ is a shortest path.
3. *Bounded stretch.* For each pair of vertices $u, v \in P$, it holds that $\ell(P_{uv}) \leq (1+\epsilon) dist(u,v)$.

Finding such paths in general is hard, so Abraham et al. [3] restrict themselves to *single via paths*, defined as the concatenation of the shortest paths Opt_{sv} and Opt_{vt} for some vertex v (the *via vertex*). They show that one can find such alternatives by running bidirectional query algorithms, such as bidirectional Dijkstra, contraction hierarchies (CH), or reach (RE). The main idea is to relax the stopping criterion so that many vertices are scanned by both searches (forward and backward). Each doubly-scanned vertex v is a candidate via vertex, and is evaluated according to the cost $\ell(v)$ of the path s–v–t, how much that path shares with the shortest (indicated by $\sigma(v)$), and the so-called *plateau cost* $pl(v)$, which gives a bound on the local optimality of the path [13]. The plateau cost of a vertex v is defined by the cost of the subpath of P_v containing all vertices u with $dist(s,u) + dist(u,t) = \ell(v)$. To ensure no bad routes are reported to the user, they discard any vertex v that violates any of the admissibility constraints: $\ell(v) < (1+\epsilon) \cdot \ell(Opt)$, $\sigma(v) < \alpha \cdot \ell(Opt)$, or $pl(v) > \gamma \cdot \ell(Opt)$. They then return as alternative the path P_v through the vertex v that minimizes the function $f(v) = 2\ell(v) + \sigma(v) - pl(v)$. As suggested by Abraham et al. [3], typical parameter values are $\alpha = 0.8$, $\gamma = 0.25$, and $\epsilon = 0.3$. Note that they return no alternative if all double-scanned vertices violate the admissibility constraints.

They also show that one can compute multiple alternatives by redefining *sharing* to be the sum of costs of the arcs a path has in common with *Opt* and any previously selected alternative.

Adapting CRP to find single via paths is straightforward. We run the normal point-to-point query, but stop only when each search scans a vertex with distance label greater than $(1 + \epsilon) \cdot \mu$. The candidate via vertices (those visited by both searches) are then be evaluated using the method described above. One advantage of using this approach with CRP instead of CH or RE is that we do not need to run additional point-to-point queries to reconstruct P_v. CH and RE need such queries because a vertex that is not part of the shortest path may have the wrong distance label, which is not the case for CRP.

Handling Traffic. One of the most important features of CRP is its fast customization, which enables (among other features) real-time traffic updates. As explained in Section 5.2, we can quickly update the overlay graphs whenever new traffic information is available. A standard query in the updated graph then will find the shortest path in traffic. When computing a very long journey, however, it may make sense to take the current traffic into account only when evaluating arcs that are sufficiently close to the source. After all, by the time the driver actually gets to faraway arcs, the traffic situation will most likely have changed. CRP can easily handle this scenario by keeping two cost functions, with and without the current traffic situation. A modified (unidirectional) query algorithm then starts from s incorporating traffic and then switches to a non-traffic scenario after a certain time.

In practice, many traffic jams can be predicted from historic traffic information. Accordingly, the *time-dependent route planning problem* associates a travel time *function* to each arc in the graph, representing the time to traverse the road segment *at a certain time of the day* [21]. Dijkstra's algorithm can still find the optimal solution in this scenario, as long as the functions are FIFO (i.e., no overtaking is allowed within an arc, which is a natural assumption in practice). CRP can therefore still be used, as long as we store time-dependent shortcuts. These are more costly to compute and take more space, but compression methods developed for time-dependent CH [7] should work here as well. Since historic traffic information is itself only an approximation (for future traffic), in practice one can allow small errors to improve efficiency and space consumption in the time-dependent scenario.

One can get the most accurate results by incorporating the current traffic situation at the beginning of the journey, then switching to a time-dependent scenario when far away from s.

5.4 Discussion

As already mentioned, CRP follows the basic strategy of separator-based methods. In particular, HiTi [43] uses arc-based separators and cliques to represent each cell, just as we do. Unfortunately, HiTi has not been tested on large road networks; experiments were limited to small grids, and the original proof of concept does not appear to have been optimized using modern algorithm engineering techniques. The same holds for other implementations [40].

CRP improves on various aspects of previous implementations. First, by separating metric-independent from metric-dependent data, we can easily handle multiple metrics by changing a single pointer (to a different array W, representing the costs of the shortcuts). Second, we have carefully engineered every aspect of the algorithm to fit our target application. Finally, we have developed a partitioning algorithm that, by exploiting the natural cuts of road networks, finds significantly better partitions than previous approaches.

An important aspect of our work is to explore the design space comprehensively. For example, by forgoing known acceleration techniques such as arc reduction or sparsification (see Section 3.3), we were able to keep the topology of the overlay metric-independent, allowing it to be shared among all cost functions. Although using such techniques would make queries slightly faster (by less than a factor of 2), it would significantly slow down (and complicate) customization. Even worse, it would not allow the metric-independent data to be shared among cost functions.

Similarly, we decided not to use goal-directed speedup techniques such as ALT [35, 37], PCD [48], or Arc-Flags [45, 38] on the topmost overlay graph, despite the apparent advantages reported in previous work [41, 11]. We tested some of these techniques, and the impact on query performance was again limited (less

than a factor of 2), but its implementation became much more complicated. For example, queries now need to be executed in 2 phases [11], which requires a more complicated algorithm and makes local queries slower. Moreover, these techniques require metric-dependent preprocessed data, which significantly increases customization times. On balance, using goal-direction on the topmost level is not worth the effort.

We stress that CRP could be further optimized to handle a large number of (perhaps personalized) cost functions. Most likely, a particular cell will have the exact same cost matrix (overlay) for multiple cost functions. (For example, a metric such as “avoid tolls” could only differ from the standard metric in cells containing toll booths.) We could avoid duplication (and save space) by maintaining a pool of unique matrices, and making each cost function keep pointers to the relevant cells in the pool.

6 Accelerating Customization by Contraction

As our experiments will show, separating preprocessing from metric customization allows CRP to incorporate a new cost function on a continental road network in about 10 seconds (sequentially) on a modern server. This is enough for real-time traffic, but still too slow to enable on-line personalized cost functions. Accelerating customization even further requires speeding up its basic operation: computing the costs of the shortcuts within each cell.

In the Section 5.2, we already discussed several optimization techniques for computing shortcut costs. In this section, we discuss how we can use contraction, the basic building block of the contraction hierarchies (CH) algorithm [32], to accelerate the customization phase even further. Instead of computing shortest paths explicitly, we eliminate internal vertices from a cell one by one, adding new arcs as needed to preserve distances; the arcs that eventually remain are the desired shortcuts. For efficiency, not only do we precompute the order in which vertices are contracted, but also abstract away the graph itself. During customization, we simply simulate the actual contraction by following a (precomputed) series of *instructions* describing the basic operations (memory reads and writes) the contraction routine would perform.

The *contraction* approach is based on the *shortcut* operation [56, 32]. To shortcut a vertex v, one removes it from the graph and adds new arcs as needed to preserve shortest paths. For each incoming arc (u, v) and outgoing arc (v, w), one creates a *shortcut* arc (u, w) with $\ell(u, w) = \ell(u, v) + \ell(v, w)$. A shortcut is only added if it represents the only shortest path between its endpoints in the remaining graph (without v), which can be tested by running a *witness search* (local Dijkstra) between its endpoints. CH uses contraction as follows. During preprocessing, it heuristically sorts all vertices in increasing order of importance and shortcuts them in this order; the order and the shortcuts are then used to speed up queries.

We propose using contraction during customization. To process a cell, we can simply contract its internal vertices while preserving its boundary vertices. The arcs (shortcuts) in the final graph are exactly the ones we want. If we want to compute multiple overlay levels with contraction, we also need to remember which intermediate shortcuts contribute to W. To deal with turn costs appropriately, we run contraction on the arc-based graph.

The performance of contraction strongly depends on the cost function. With travel times in free-flow traffic (the most common case), it works very well. Even for continental instances, sparsity is preserved during the contraction process [32], and the number of arcs less than doubles. Unfortunately, other metrics often need more shortcuts, which leads to denser graphs and makes finding the contraction order much more expensive. Even if a good order is given, simply performing the contraction can still be quite costly [32].

Within the CRP framework, we can deal with these issues by exploiting the separation between metric-independent preprocessing and customization. During preprocessing, we compute a unique contraction order to be used by all metrics. Unlike previous approaches [32], to ensure this order works well even in the worst case, we simply assume that every potential shortcut will be added. Accordingly, we do not perform witness searches during customization. Moreover, for maximum efficiency, we precompute a sequence of *microinstructions* to describe the entire contraction process in terms of basic operations. We detail each of these improvements next.

6.1 Contraction Order

Computing a contraction order that minimizes the number of shortcuts added is NP-hard [8]. In practice, one uses on-line heuristics that pick the next vertex to contract based on a priority function that depends on local properties of the graph [32]. A typical criterion is the difference between the number of arcs added and removed if a vertex v were contracted. We tested similar greedy priority functions to evaluate each vertex v, taking into account parameters such as the number $ia(v)$ of incoming arcs, the number $oa(v)$ of outgoing arcs, and the number $sc(v)$ of shortcuts created or updated if v is contracted (this may be less than $ia(v) \cdot oa(v)$, since self-loops are never needed). We found that picking vertices v that minimize $h(v) = 100sc(v) - ia(v) - oa(v)$ works well. This essentially minimizes the number of shortcuts added, using the current degree as a tiebreaker. The algorithm is not very sensitive to the values of the coefficients.

This approach gives reasonable orders, but one can do even better by taking the graph topology into account. There exist natural orders that lead to a provably small number of shortcuts for graphs with small separators [9, 51] or treewidth [9]. It suffices to find a small separator for the entire graph, recursively contract the two resulting components, then contract the separating vertices themselves. For graphs with $O(\sqrt{n})$-separators (such as planar graphs), such nested dissection leads to $O(n \log n)$ shortcuts. Although real-world road networks are far from planar, they have even smaller separators [16].

This suggests using partitions to guide the contraction order. We create additional *guidance levels* during the preprocessing step, extending our standard CRP multilevel partition downward, to even smaller cells. We subdivide each level-1 cell (of maximum size U) into nested subcells of maximum size U/σ^i, for $i = 1, 2, \ldots$ (until cells become too small). Here $\sigma > 1$ is the *guidance step*. For each internal vertex v in a level-1 cell, let $g(v)$ be the smallest i such that v is a boundary vertex on the guidance level with cell size U/σ^i (intuitively, more important nodes have smaller $g(v)$ values). We use the same contraction order as before, but delay vertices according to $g(\cdot)$. If $g(v) > g(w)$, v is contracted before w; within each guidance level, we use $h(v)$. When computing multiple overlay levels with contraction, we also use guidance levels between them.

6.2 Microinstructions

While the contraction order is determined during the metric-independent phase of CRP, we can only *execute* the contraction (follow the order) during customization, when the arc costs are known. Even with the order given, this execution is still expensive [32]. Conceptually, to contract v one must first retrieve the costs (and endpoints) of its incident arcs, then process each potential shortcut (u, w) by either inserting it or updating its current value. This requires data structures supporting arc insertions and deletions, and even checking if a shortcut already exists gets costlier as degrees increase. Each fundamental operation, however, is rather simple: we read the costs of two arcs, add them up, compare the result with the cost of a third arc, and update it if needed. The entire contraction routine can therefore be fully specified by a sequence of triples (a, b, c). Each element in the triple is a memory position holding an arc (or shortcut) cost. We must read the values in a and b and write the sum to c if there is an improvement.

Since the sequence of operations is the same for any cost function, we use the metric-independent preprocessing stage to set up, for each cell, an *instruction array* describing the contraction as a list of triples. Each element of a triple represents an offset in a separate *memory array*, which stores the costs of all arcs (temporary or otherwise) touched during the contraction. The preprocessing stage outputs the entire instruction array as well as the size of the memory array.

During customization, entries in the memory array representing input arcs (or shortcuts) are initialized with their costs; the remaining entries (new shortcuts) are set to ∞. We then execute the instructions one by one, and eventually copy the output values (costs of shortcuts from entry to exit points in the cell) to the overlay graph. With this approach, the graph itself is abstracted away during customization. We do not keep track of arc endpoints, and there is no notion of vertices at all. The code just manipulates numbers (which happen to represent arc costs). This is cheaper (and simpler) than operating on an actual graph.

Although the space required by the instruction array is metric-independent (shared by all cost functions), it can be quite large. We can keep it manageable by representing each triple with as few bytes as necessary to address the memory array associated with the cell. In addition, we use a single *macroinstruction* to represent

the contraction of a vertex v whenever the resulting number of writes exceeds an *unrolling threshold* τ. This instruction explicitly lists the addresses of v's c_{in} incoming and c_{out} outgoing arcs, followed by the corresponding $c_{in} \cdot c_{out}$ write positions (self-loops are written to a dummy position). The customization phase must explicitly loop over all incoming and outgoing positions, which is slightly slower than reading tuples but saves space. By default, we set $\tau = 3 + c_{in} \cdot c_{out}$, minimizing the space usage as much as possible.

7 Experiments

In this section, we present an extensive experimental evaluation of CRP. Our code is written in C++ (with OpenMP for parallelization) and compiled with Microsoft Visual C++ 2010. We use 4-heaps as priority queues. Our test machine runs Windows Server 2008 R2 and has 96 GiB of DDR3-1333 RAM and two 6-core Intel Xeon X5680 3.33 GHz CPUs, each with 6×64 KB of L1, 6×256 KB of L2, and 12 MB of shared L3 cache.

Inputs. Our main benchmark instance is the European road network, with 18 million vertices and 42 million arcs, made available by PTV AG for the 9th DIMACS Implementation Challenge [23]. This instance builds on Navteq data from 2003. We also tested other inputs, including the (proprietary) data Bing Maps uses for computing driving directions, in Section 7.4. To ensure our results are reproducible, however, we perform most of our experiments on publicly available (rather than proprietary) data.

The PTV instance has been published in two versions, with arc costs representing either travel times or travel distances. From these two graphs, we determined the average traversal speed on each arc. It turns out that each arc belongs to one of 15 categories, from which 13 model roads with different speed limits (10 to 130 km/h in steps of 10), while the remaining two refer to ferries. For each arc, we store its length in meters (derived from the distances graph) and the deducted speed category packed in one 32-bit integer. This allows us to change metrics by assigned different traversal speeds to each category. Also, we allow offsets for each category to model roll-on penalties for ferries. Unfortunately, we are not aware of any publicly-available realistic turn data, so we augment our standard benchmark instance instead. For every vertex v, we add a turn between each incoming and each outgoing arc. A turn from (u, v) to (v, w) is either a *U-turn* (if $u = w$) or a *standard turn* (if $u \neq w$), and each of these two types has a cost. We have not tried to further distinguish between turn types, since any automated method would not reflect real-life turns. However, just adding U-turn costs is enough to reproduce the issues we found on the real-life Bing Maps data. Vertex IDs are 32-bit integers.

As already mentioned, our data structures allow fast changes to the cost functions by assigning speed limits to road categories as well as setting the costs for turns (U-turns and non U-turns). In most cases, we consider two standard (but quite different) metrics: travel times and travel distances. For travel times, we use speed limits from 10 to 130 km/h in steps of 10 for the different road categories. For ferries, we used 5 km/h. This matches the travel time version of the PTV graph rather accurately. U-turns are set to 100 seconds, while other turns are free. For travel distances, we set the costs of turns to 0, and each road segment (including ferries) can be traversed in constant speed.

Methodology. We want to minimize *metric customization time*, *metric-dependent space* (excluding the original graph), and *query times*. We also report the time and space requirements of the metric-independent stage, which are less relevant as long as they remain reasonable (as they do). Unless otherwise mentioned, we run queries on a single core and report *sequential* running times, while preprocessing and customization uses all cores available. We run 100 000 point-to-point queries with the source arc s and the target arc t picked uniformly at random. We also use randomly chosen offset on both the source and target arcs. Although our focus in on finding the *cost* of the shortest path, in some (clearly marked) experiments we also consider the time to retrieve the actual path description.

Default Settings. First we discuss PUNCH settings. CRP uses PUNCH as part of the metric-dependent preprocessing step. As discussed in Section 5.1, PUNCH has several input parameters that yield different

tradeoffs between running time and solution quality. Delling et al. [16] define four main parameters: coverage (which defines how carefully natural cuts are sought), number of reoptimization attempts during local search (φ), number of restarts in the assembly phase (M), and size of the pool of elite solutions (to allow recombination). By varying these parameters, we considered five different variants of PUNCH. The *standard* version sets the parameters above to $(2, 16, 9, 3)$. The parameters for *fast*, *fast+*, *heavy*, and *heavy+* are $(1, 16, 4, 0)$, $(1, 4, 1, 0)$, $(2, 32, 16, 4)$, and $(2, 64, 100, 10)$, respectively. For all parameters, higher values generally lead to better solutions, but higher running times. By default, we run PUNCH multi-threaded, using all available cores.

We implemented contraction hierarchies following [32], with some optimizations that lead improved performance in our setting. In particular, we use a different priority term during contraction. The priority of a vertex u is given by $EQ(u) + HQ(u) + L(u)$, where $EQ(u)$ is the ratio between the number of arcs added and removed (if u were contracted), $HQ(u)$ is the ratio between the total number of arcs represented by all shortcuts added and the total number of arcs represented by all arcs removed, and $L(u)$ is the level u would be assigned to. We use the parallelization techniques introduced by Vetter [63] to accelerate the preprocessing, using a neighborhood of 2 to determine independent sets. Note that, as previous implementations [32], our CH implementation only works on standard graphs (without explicit turn information), but it can be used on the fully-blown or expanded graph. For the compact graph representation, Christian Vetter provided us with the source code of his turn-aware implementation [33], which uses a very similar way of representing turns. Due to some portability issues (our code runs on Windows, his on Linux), we conducted experiments with Vetter's code on a machine comparable to ours, a 2.6 GHz dual 8-core Intel Xeon E5-2670 with 64 GiB of DDR3-1600 RAM.

7.1 Main Results

Table 1 summarizes the main results of our work. It compares the performance of Dijkstra, Contraction Hierarchies, and CRP for Europe using different cost functions and graph representations (see Section 4.1). For CRP, we use a six-level setup ($U_1 = 2^4, U_2 = 2^8, U_2 = 2^{11}, U_3 = 2^{14}, U_4 = 2^{17}, U_5 = 2^{20}$) using PUNCH heavy+, heavy, standard, fast, fast, and fast+ from the highest to the lowest level; the lowest level is used as phantom level, i.e., it is used for customization but discarded during queries. This choice of parameters makes PUNCH spend more time on higher levels, whose cells are touched by more queries. Customization uses Bellman-Ford with SSE optimization run from 16 simultaneous sources ($k = 16$), but does *not* use contraction. (Subsequent experiments will justify this choice of parameters.) For CRP, we report the (parallel) time and space for the metric-independent preprocessing (partitioning and setting up the overlay topology) and for the customization phase (computing W); for CH, we report the (parallel) time and space usage of the preprocessing phase (contraction). For queries, we report the number of vertices scanned (delete-min operations on the priority queues) and the resulting sequential query times (to determine the cost of the shortest path) for random queries. For each graph representation, we also report the amount of space required to represent the graph itself in memory. Note that we use two tailored implementations for Dijkstra's algorithm and CH, depending whether the graph uses the compact turn representation or not. For CRP, though, we use the same (turn-aware) implementation and simply disregard turn costs (and restrictions) for the "no turns" scenario.

We observe that the impact of incorporating turns is huge for both Dijkstra and CH, but limited for CRP. For Dijkstra's algorithm, the slowdown is between 2.2 (with the compact turn representation) and 5.2 (fully blown). The memory consumption of the graph itself also increases significantly, unless the compact turn representation is used. Since memory is limited, especially on mobile devices, we consider this representation superior. CH preprocessing takes over an order of magnitude longer on turn-aware graphs. Even with 12 cores, preprocessing is not fast enough with real-time traffic updates, and the preprocessed metric-dependent data increases significantly. Similarly, the performance of CH is severely affected when optimizing travel distances instead of travel times, even without turns.

CRP is hardly affected by turns or metric changes. Although queries may not be as fast as in CH on good metrics (by up to an order of magnitude), they are fast enough for interactive applications. Another clear advantage is that *by design*, CRP will perform similarly for *any* metric. In contrast, the performance of

Table 1. Performance of Dijkstra, Contraction Hierarchies, and CRP on our benchmark instance, using different graph representations. Preprocessing and customization times are given for multi-threaded execution, while queries are run single-threaded.

	GRAPH		DIJKSTRA QUERIES		CH PREPRO		CH QUERIES		CRP PREPRO		CRP CUSTOM		CRP QUERIES	
metric	data structure	space [MiB]	scans [$\times 10^6$]	time [ms]	time [s]	space [MiB]	nmb. scans	time [ms]	time [s]	space [MiB]	time [s]	space [MiB]	nmb. scans	time [ms]
dist	no turns	408	9.3	1779	726	270	858	0.87	654	411.8	1.04	71.0	2942	1.91
time	no turns	408	9.4	2546	109	196	280	0.11	654	411.8	1.05	71.0	2766	1.65
	fully-blown	2739	42.7	13306	1967	2682	409	0.20	–	–	–	–	–	–
	arc-based	1620	21.6	7888	1392	1520	404	0.19	–	–	–	–	–	–
	compact	445	15.1	5582	1753	642	1998	2.27	654	411.8	1.10	71.0	3049	1.67

CH depends on how well the contraction process works, which is a function of the metric. We note that CH preprocessing can be faster if the order in which vertices are processed is fixed in advance. Unfortunately, the order computed for one metric to rebuild another is only efficient if the metrics are very similar [30]. In addition, Delling et al. [19] show that the space consumption can be reduced by storing only the upper part of the hierarchy, at the expense of query times. Still, none of these optimizations brings neither customization times nor metric-dependent space consumption close to those achieved by CRP.

To summarize, CRP queries are fast enough for interactive applications and, with customization times of around one second, it can handle very frequent traffic updates. Moreover, the space per metric is small enough to enable personalization, with each user in the system having their own cost function. (After all, 71 MiB is a small fraction of a standard mailbox.) Also, CRP works with *any* metric, which is a big advantage in production systems where the function to be optimized is not always known in advance. Together, these factors make CRP a superior approach for building a real-world route planning engine for road networks.

7.2 Parameters

Next, we consider the impact of parameters and design choices on the performance of CRP.

Impact of the Partition. We start by comparing PUNCH, which has been developed for road networks, to a popular general-purpose partitioner. For each method tested, Table 2 reports the total partitioning time, the resulting number of cells ($|\mathcal{C}|$), and the size of the corresponding overlay graph (given by the number of overlay vertices and the space required to represent the corresponding cost matrices). We consider three different maximum cell sizes U, and compare our five versions of PUNCH with METIS [44]. Instead of taking U as input, METIS takes the number k of cells and a maximum imbalance ϵ, so we set $\epsilon = 0.03$ (the default for METIS) and $k = \lceil 1.03 \cdot n/U \rceil$. METIS has two parameters (`ncuts` and `niter`) which influence the quality of the obtained partitions. We test two settings: the *default* version uses the preset values (`ncuts=1` and `niter=10`), while the *heavy* version sets `ncuts=100` and `niter=10000`. Moreover, since METIS may produce disconnected cells (which are not useful for CRP), we also consider *METIS merge*, in which we greedily combine adjacent disconnected cells until no merge is possible without violating our upper bound U. In each step, we merge the pair of cells that reduces the overall arc cut the most. Note that METIS can also be made to produce connected cells, but we observed that the obtained cut size is up to 20% higher. For a fair comparison, we run both METIS and PUNCH single-threaded in this experiment.

We observe that default METIS is much faster than PUNCH, but produces significantly worse partitions. METIS yields twice as much metric-independent data as PUNCH with $U = 2^{10}$, and three times as much with $U = 2^{20}$. Surprisingly, letting METIS run (much) longer has hardly any impact on solution quality. As we argued before, the topology of a network does not change too often, thus justifying spending more time and using PUNCH for partitioning. For PUNCH, we see that different parameters have limited impact on the partition quality: the heavy+ variant is much slower than fast+, but reduces the amount of metric-dependent data by only about 10%. Since the performance gap is smaller for bigger cells, for the remainder

Table 2. Impact of the partition on preprocessing time and size of the overlay graph. $|\mathcal{C}|$ denotes the number of cells in the partition, $|V_H|$ the number of nodes of the overlay graph, and the last column denotes the size of the metric-dependent data in MiB. Execution times are sequential.

		TIME	OVERLAY		
U	algorithm	[s]	$\lvert\mathcal{C}\rvert$	$\lvert V_H\rvert$	[MiB]
1024	METIS	22.7	21413	825180	39.97
	METIS merge	22.7	20453	818642	40.02
	METIS heavy	3091.2	21306	819932	39.51
	METIS heavy merge	3091.2	20395	813680	39.56
	PUNCH fast+	115.5	20389	634564	23.66
	PUNCH fast	659.0	20117	626488	23.20
	PUNCH standard	4358.6	20402	615584	22.06
	PUNCH heavy	14972.9	20298	613052	21.93
	PUNCH heavy+	195112.1	20207	610758	21.83
32768	METIS	7.9	662	96512	18.93
	METIS merge	7.9	611	95948	18.91
	METIS heavy	788.9	648	95544	18.90
	METIS heavy merge	788.9	618	95340	18.87
	PUNCH fast+	131.4	631	67716	9.31
	PUNCH fast	210.3	625	65976	8.83
	PUNCH standard	907.8	628	64986	8.46
	PUNCH heavy	2531.1	627	64888	8.44
	PUNCH heavy+	29468.8	625	64330	8.33
1048576	METIS	7.6	27	9922	6.54
	METIS merge	7.6	22	9894	6.52
	METIS heavy	761.8	29	9424	6.33
	METIS heavy merge	761.8	22	9318	6.33
	PUNCH fast+	368.9	21	5444	2.16
	PUNCH fast	369.8	22	5240	2.00
	PUNCH standard	756.2	20	5254	2.17
	PUNCH heavy	852.8	20	5112	1.91
	PUNCH heavy+	2483.4	21	5038	1.95

of our experiments (when we compute multilevel partitions) we use heavier variants for large values of U_i, the maximum allowed cell size at level i. More precisely, we use fast+ for $U_i < 256$, fast for $256 \leq U_i < 4096$, standard for $4096 \leq U_i < 32768$, heavy for $32768 \leq U_i < 524288$, and heavy+ for $U_i \geq 524288$. To accelerate the remaining experiments, we run a multi-threaded version of PUNCH (using all 12 cores) by default.

To evaluate the impact of the multilevel partition on CRP, we consider varying the number of levels and of cell sizes on each level. Table 3 reports the time and space for the metric-independent preprocessing (partitioning and overlay topology) and the customization phase (the matrices). For queries, we report the number of vertices scanned (delete-min operations on the priority queues) and the resulting query times (to determine the cost of the shortest path) for random queries. Finally, we report the time it takes (on average and in the worst observed case) to update the matrices after a single arc change. For all sets of parameters, we always use a phantom level with $U = 16$. All runs in the experiment are sequential (except for PUNCH) and use the travel time metric.

We observe that customization times are around 12 seconds for $L \geq 3$ and between 11 and 35 seconds for $L < 3$. During this time all Bellman-Ford searches scan about 140 million vertices in total. Recall, however, that each scan updates up to 16 distance labels; if we processed each source independently, the total number of scans would be about 330 million. Processing the phantom level (with $U = 16$) actually takes up a significant fraction of the total time (around 7.3 seconds). This explains why the overall times are relatively stable: the remaining levels are rather cheap, unless the gap between the levels is large.

Table 3. Impact of the multilevel partition on the performance of CRP for varying number of levels (L) and different cell sizes. Our default configuration is highlighted in bold.

		PREPRO		CUSTOM		QUERIES		UPDATES	
		time	space	time	space	nmb.	time	time [ms]	
L	parameter	[s]	[MiB]	[s]	[MiB]	scans	[ms]	avg	max
1	$[2^{10}]$	160.53	401.21	11.95	23.05	87771	38.64	0.23	1.40
	$[2^{12}]$	428.79	399.31	18.31	14.91	46315	19.09	2.16	12.41
	$[2^{14}]$	262.30	398.62	34.52	10.31	61500	21.69	21.31	119.55
2	$[2^{10}:2^{16}]$	371.03	402.51	13.73	29.61	10584	4.75	6.31	41.12
	$[2^{10}:2^{18}]$	263.76	402.46	16.34	26.86	11411	4.55	59.64	347.83
	$[2^{12}:2^{18}]$	387.09	399.80	19.85	18.69	16656	5.65	23.01	138.98
	$[2^{12}:2^{20}]$	621.91	399.79	22.11	16.83	20709	7.04	215.09	675.19
3	$[2^{7}:2^{14}:2^{21}]$	530.89	417.17	14.85	60.33	8066	3.51	204.92	565.29
	$[2^{8}:2^{14}:2^{20}]$	607.81	409.91	13.13	48.23	5927	2.76	77.01	247.08
	$[2^{9}:2^{14}:2^{19}]$	780.02	405.47	12.55	40.95	5304	2.42	23.61	112.30
	$[2^{10}:2^{15}:2^{20}]$	713.81	402.67	13.86	32.80	6662	2.75	46.02	169.03
4	$[2^{7}:2^{11}:2^{15}:2^{19}]$	839.45	418.85	11.26	77.97	3566	1.94	14.73	61.53
	$[2^{8}:2^{12}:2^{16}:2^{20}]$	767.60	410.69	11.43	58.87	3603	1.87	27.05	107.11
	$[2^{9}:2^{13}:2^{17}:2^{21}]$	650.70	405.79	12.11	46.36	4271	1.95	47.36	158.36
	$[2^{10}:2^{14}:2^{18}:2^{22}]$	535.44	402.91	13.47	36.84	5607	2.21	69.31	196.20
5	$[2^{7}:2^{10}:2^{13}:2^{16}:2^{19}]$	1048.43	419.92	10.79	89.63	2995	1.76	9.63	41.86
	$[2^{8}:2^{11}:2^{14}:2^{17}:2^{20}]$	**653.50**	**411.81**	**11.09**	**71.01**	**3049**	**1.67**	**17.94**	**73.77**
	$[2^{9}:2^{12}:2^{15}:2^{18}:2^{21}]$	716.35	406.44	11.76	55.57	3570	1.65	28.84	99.86
	$[2^{10}:2^{13}:2^{16}:2^{19}:2^{22}]$	604.54	403.26	13.17	44.43	4911	1.93	46.30	132.11
6	$[2^{8}:2^{10}:2^{12}:2^{14}:2^{16}:2^{18}]$	1234.26	414.33	10.80	94.22	3582	2.03	3.61	29.80
	$[2^{9}:2^{11}:2^{13}:2^{15}:2^{17}:2^{19}]$	871.69	408.05	11.61	75.34	3632	1.81	6.26	34.02
	$[2^{10}:2^{12}:2^{14}:2^{16}:2^{18}:2^{20}]$	677.37	404.21	12.92	60.80	4701	2.02	12.80	64.68
7	$[2^{8}:2^{10}:2^{12}:2^{14}:2^{16}:2^{18}:2^{20}]$	712.61	414.39	10.97	96.71	2583	1.50	12.06	73.49
	$[2^{9}:2^{11}:2^{13}:2^{15}:2^{17}:2^{19}:2^{21}]$	1007.51	408.10	11.66	76.69	3101	1.47	19.63	74.16
	$[2^{10}:2^{12}:2^{14}:2^{16}:2^{18}:2^{20}:2^{22}]$	770.44	404.29	12.99	60.71	4392	1.71	29.94	105.96

Increasing the number of levels tends to improve query performance, but increases the amount of metric-dependent data used. Two levels are enough to reduce query times below 10 ms, which is fast enough for interactive applications; even faster queries are possible with more levels. Updates are also quite fast (usually under 100 ms) for most parameters tested.

For the remainder of our experiments, we use a phantom level of size 16, and five additional levels with maximum cell sizes $[2^8, 2^{11}, 2^{14}, 2^{17}, 2^{20}]$. This setup gives a reasonable tradeoff between query performance, space requirements, and update times.

Parallelization. Table 4 reports how customization scales as the number of CPU cores increases. Besides our two default metrics, we evaluate four additional cost functions. All use travel times as the baseline, with a few modifications: *free turns* sets U-turn costs to 0; *avoid ferries* reduces ferry speeds to 1 km/h; *avoid freeways* reduces the traversal speed of the top four road categories to 30 km/h; and *prefer freeways* halves the traversal speed of all other road categories. Besides customization times, we also report the sequential query performance. Recall that the amount of the metric-dependent data is the same in all cases: 71 MiB.

We observe that the sequential customization time is around 11 s, independent of the metric. Moreover, customization scales very well with the cores used. With 12 cores, we see a speedup of around 10, resulting in customization times of about 1 second, which is fast enough for real-time metric updates. Moreover, query times are below 2 ms for all metrics considered.

Table 4. Impact of parallelization and of the metric on the performance of CRP. Customization times are given for $c = 1, 3, 6, 12$ cores in seconds, queries are executed sequentially and given in milliseconds.

	CUSTOM. TIME [S]				QUERIES	
	number of cores				nmb.	time
metric	$c = 1$	$c = 3$	$c = 6$	$c = 12$	scans	[ms]
distances	10.37	3.67	1.88	1.04	2942	1.91
times	11.12	3.92	2.00	1.10	3049	1.68
avoid ferries	11.14	3.92	2.00	1.10	3052	1.68
avoid freeways	11.11	3.91	2.00	1.09	3210	1.87
free turns	10.55	3.72	1.91	1.05	2766	1.65
prefer freeways	11.00	3.87	1.98	1.09	2743	1.52

Path Retrieval. So far, we have only reported the time to compute the *cost* of the shortest path. Next, we evaluate how long is takes to actually retrieve the full path, which is essential for applications that compute actual driving directions. Table 5 reports the overall execution time (including determining the cost of the path) when we use different degrees of parallelization for unpacking shortcuts and different sizes of the LRU cache. We again use travel times and distances as cost functions. In this experiment, we warm up the cache by running 1 000 000 random s–t queries before measuring the execution times (over 1 000 000 queries).

Without a cache, the full path can be retrieved (sequentally) in about 8 ms for travel distances and 4 ms for travel times. The difference reflects the fact that bidirectional Dijkstra searches (which we use during unpacking) meet earlier when optimizing travel times instead of travel distances. Unpacking times can be reduced by using either more cores or an LRU cache. In particular, caching up to 262 144 shortcuts provides a good tradeoff between unpacking time (0.2 ms for travel times and about 0.5 ms for travel distances) and additional metric-dependent memory consumption (less than 15 MiB). The fast unpacking can be explained by the high cache hit ratio (well above 90%). Interestingly, with large enough caches, using more cores for unpacking does not help much; it even hurts when switching from 6 to 12 cores. This happens due to memory contention: the shortcut cache resides on the memory bank of the first CPU, thus making accesses by the second CPU more expensive.

In general, the table shows that caching is very useful for common cost functions (such as travel times), but unpacking is fast enough even with no cache. Note, however, that performance would be worse with some adversarial (and unnatural) cost functions. Consider, for example, a metric in which the path between two points contains a large fraction of the arcs of the graph. Unpacking would take a long time because it would run bidirectional on a large number of cells (often more than once). Of course, other speed-up techniques would not perform much better than plain Dijkstra's algorithm for such an adversarial cost function. We stress, however, that customization times and cost-only CRP queries would remain essentially unaffected even by such a cost function.

Table 5. Performance of unpacking shortcuts for travel times and distances; c is the number of CPU cores used.

	TRAVEL TIMES						TRAVEL DISTANCES					
cached	size	hits	query time [ms]				size	hits	query time [ms]			
shortcuts	[MiB]	[%]	$c = 1$	$c = 3$	$c = 6$	$c = 12$	[MiB]	[%]	$c = 1$	$c = 3$	$c = 6$	$c = 12$
0	0.00	0.00	4.20	2.91	2.56	2.50	0.00	0.00	8.01	4.55	3.58	3.25
1024	0.03	8.58	4.15	2.92	2.60	2.60	0.06	2.56	7.99	4.57	3.63	3.37
16384	0.54	79.31	2.53	2.24	2.14	2.22	0.80	33.67	6.31	3.99	3.30	3.17
65536	2.65	95.50	1.98	1.96	1.93	2.02	3.26	70.18	4.11	3.08	2.78	2.79
262144	12.02	98.94	1.85	1.87	1.87	1.97	13.32	94.57	2.49	2.34	2.29	2.39
1048576	50.89	99.94	1.80	1.84	1.84	1.94	54.50	99.59	2.12	2.14	2.13	2.24

Table 6. Time (in seconds) spent on each overlay level for different algorithms. The best value in each column is highlighted.

method	2^4	2^8	2^{11}	2^{14}	2^{17}	2^{20}	total
Dijkstra	10.39	3.87	1.09	0.73	0.58	0.42	17.08
4-Dijkstra	12.42	4.35	1.25	0.87	0.71	0.53	20.13
4-Dijkstra (SSE)	11.43	4.73	1.13	0.68	0.47	0.33	18.79
16-Dijkstra	11.90	4.10	1.19	0.83	0.66	0.48	19.16
16-Dijkstra (SSE)	10.15	4.48	1.13	0.70	0.47	0.34	17.27
Bellman-Ford	9.92	4.19	1.15	0.83	0.71	0.57	17.37
4-Bellman-Ford	9.29	3.30	0.94	0.74	0.69	0.60	15.57
4-Bellman-Ford (SSE)	7.67	2.76	0.71	0.49	0.39	0.29	12.31
16-Bellman-Ford	9.41	3.34	1.00	0.81	0.76	0.64	15.95
16-Bellman-Ford (SSE)	7.35	2.18	0.56	**0.39**	**0.34**	**0.28**	**11.10**
Floyd-Warshall	52.34	35.14	7.52	7.19	7.19	5.97	115.36
Contraction	**1.92**	**0.46**	**0.38**	1.67	2.01	5.41	11.84

Other Customization Algorithms. Up to now, we used Bellman-Ford (with SSE optimization) and $k = 16$ during customization. Table 6 reports the impact on the customization time if we use different algorithms, including the contraction approach discussed in Section 6 (with guidance step size 2). We report the total time (on a single core) to compute the shortcut costs on each of the 6 levels, as well as the total customization time. It shows that individual executions of Dijkstra's algorithm are slightly faster than Bellman-Ford; Floyd-Warshall is not competitive. Computing distances from four boundary vertices at once (prefix "4-" in the table) helps Bellman-Ford, but hurts Dijkstra (which needs more scans). SSE instructions help both algorithms. Due to better locality, the fastest approach is Bellman-Ford with 16 simultaneous searches.

We note, however, that the contraction approach outperforms Bellman-Ford on the lowest three levels. By processing the lowest three levels with contraction, and the top three with Bellman-Ford, we can reduce the total (sequential) customization time to a mere 3.77 seconds.

The main disadvantage of using contraction is the increase in metric-independent space consumption: the instruction array can become quite big. In Table 7 we report the overall performance of variants of CRP that differ on whether phantom levels are used and the level up to which contraction is used. While queries are not affected by these parameters, we observe a tradeoff between metric-independent space consumption and customization times. Using a phantom level with $U = 16$ increases the metric-independent space consumption by a factor of about 3, but decreases customization times by a factor of 2.5. Although heavy use of contraction accelerates customization by another factor of 3 (to 360 ms), this requires significantly more metric-independent space (about 3 GiB). While this is tolerable for a server, it may be too costly in other scenarios. Moreover, processing a new metric may be fast enough even without contraction: it takes

Table 7. Performance of different CRP variants. We use our default partition, but vary whether we use a phantom level (p) and up to which point we customize with contraction (c) instead of Bellman-Ford. Customization times are given for single- (seq.) and multi-threaded (par.) execution.

		PREPRO		CUSTOM			QUERIES		
		time	space	seq.	par.	space	nmb.	dist	path
p	c	[s]	[MiB]	[s]	[s]	[MiB]	scans	[ms]	[ms]
—	—	620	143.1	25.95	2.32	71.0	3049	1.67	1.85
2^4	—	654	411.8	11.10	1.09	71.0	3049	1.67	1.85
2^4	2^4	688	2148.9	5.67	0.53	71.0	3049	1.67	1.85
—	2^8	706	3119.3	3.95	0.37	71.0	3049	1.67	1.85
—	2^{11}	713	3626.9	3.77	0.36	71.0	3049	1.67	1.85

only about a second on a 12-core server, which is orders of magnitude faster than observed for any previous method.

We stress that both the guidance levels and the microcode optimization (see Section 6) are critical to the effectiveness of contraction. Without guidance levels, the amount of metric-independent data increases by another 30%. Without microcode, customization times are slower than Bellman-Ford on all levels.

7.3 Extended Query Scenarios

Next, we evaluate the performance of the default (six levels with one phantom level, no contraction) version of CRP for other query scenarios. This includes queries for varying ranges and computing alternatives.

Local Queries. In real-world applications, most queries tend to be local. To evaluate CRP on queries of various ranges, Figure 6 plots query times against the Dijkstra rank [54]. When Dijkstra's algorithm is run from s, the Dijkstra rank of a vertex v is r if v is the r-th vertex scanned by the search. We run 1000 queries for each rank tested (all powers of two), with s chosen uniformly at random. Since we run arc-to-arc queries by default, we determine the Dijkstra ranks on the arc-based graph. We report the running times for computing the cost of the path excluding and including the time to unpack the path (with and without a cache of 262 144 shortcuts). We use one core for unpacking and, when using a cache, we perform 100 000 (random) warmup queries.

The plot shows that local queries are much faster than global ones. Up to ranks of 2^{15}, all queries can be answered in less than 1 ms, even if the full path description is required and no cache is used. Caching is particularly helpful for long-range queries, which are sped up by a factor of three. Even with no cache, however, all types of queries can be answered in 10 ms or less, which is still fast enough for interactive queries.

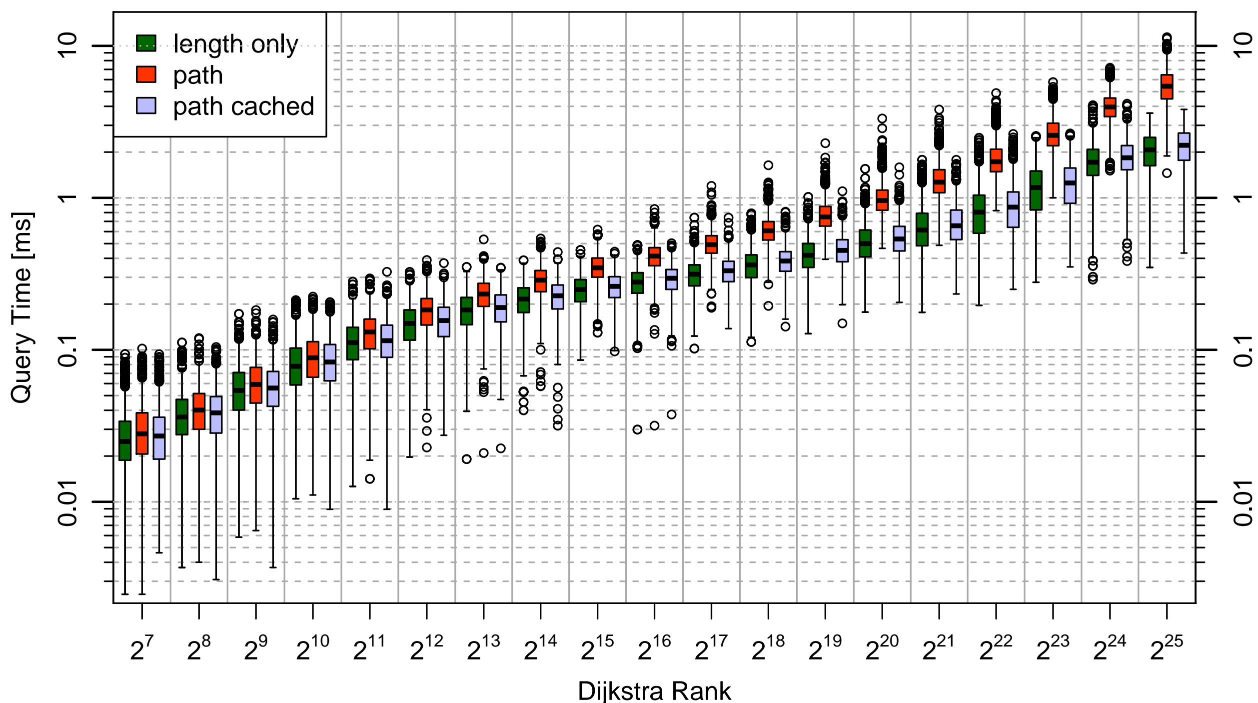

Fig. 6. Performance of CRP for various query ranges. The input is Europe with travel times and U-turn costs. Path unpacking is done on one core, and the cache cpaacity is set to 262 144 shortcuts.

Table 8. Performance of CRP when computing alternatives.

	TRAVEL TIMES				TRAVEL DISTANCES			
# alt.	success rate [%]	# scans	time [ms]	slow-down	success rate [%]	# scans	time [ms]	slow-down
0	–	3044	1.85	1.00	–	2936	2.49	1.00
1	90.92	8720	5.84	3.15	95.31	8047	6.86	2.75
2	65.40	8720	6.07	3.28	82.14	8047	7.37	2.96
3	39.23	8720	6.23	3.37	63.55	8047	7.76	3.12
4	19.03	8720	6.32	3.42	44.52	8047	8.04	3.23

Alternatives. We next evaluate the performance penalty when we also want to report alternatives, as discussed in Section 5.3. Table 8 shows the performance for finding up to 4 alternative routes, using both travel times and travel distances. We also report the *success rate* which is defined by how often the algorithm actually finds the desired number of alternatives. We use a cache of size 262 144, and warm it by 100 000 random queries. The running times include finding the shortest path, unpacking it, evaluating all via nodes, and retrieving the alternatives. We also report the *slowdown* compared to pure CRP point-to-point queries (including path retrieval).

We observe that we often succeed in finding one or two alternatives. These success rates are in line with existing algorithms based on CH or bidirectional Dijkstra [3, 47]. (A direct comparison is impossible since their code does not work with turns.) Interestingly, the success rates are higher for travel distances than for travel times, since low-category roads (such as rural roads or minor urban streets) become attractive candidates for alternatives. For both metrics, finding alternatives is roughly three times as expensive as shortest-path-only queries. The main reason is the increased search space, since we now cannot stop the search as soon as forward and backward search meet. Evaluating the search spaces to find the next best via node, however, is rather cheap and has limited effect on the running time. In any case, running times are still well below 10 ms, fast enough for interactive applications.

7.4 Other Inputs

Our last experiment considers other benchmark instances. From the 9th DIMACS Challenge [23], we take PTV Europe and TIGER USA, each with two cost functions: driving times (with 100 s U-turn costs) and distances. We also consider OpenStreetMap (OSM) data (v. 121812) representing major landmasses and with realistic turn restrictions. Finally, we test the instances used by Bing Maps, which build on Navteq data and include actual turn costs and restrictions; the proprietary "default" metric correlates well with driving times. The key figures of these networks, together with the average number of scans and running time (over 1000 random queries) of turn-aware Dijkstra, are shown in Table 9.

Table 9. Key figures and performance of Dijkstra's algorithm on various benchmark instances.

					Dijkstra	
source	input	$\lvert V\rvert$ [$\times 10^6$]	deg	cost func	scans [$\times 10^6$]	time [ms]
PTV	Europe	18.0	2.36	distance	9.1	3069
	Europe	18.0	2.36	time	15.2	6093
TIGER	US	23.9	2.41	distance	12.1	4790
	US	23.9	2.41	time	13.2	6124
OSM	Australia	4.9	1.97	time	3.4	919
	S. America	11.4	2.18	time	9.2	2549
	N. America	162.7	2.04	time	115.8	70542
	Old World	189.4	2.02	time	127.0	77121
Bing	N. America	30.3	2.41	default	28.3	11684
	Europe	47.9	2.23	default	37.0	17750

Table 10. Performance of CRP without and with contraction on various benchmark instances.

			DEFAULT				WITH CONTRACTION				QUERIES		
			PREPRO		CUSTOM		PREPRO		CUSTOM				
source	input	cost func	time [s]	space [MiB]	time [s]	space [MiB]	time [s]	space [MiB]	time [s]	space [MiB]	nmb. scans	dist [ms]	path [ms]
PTV	Europe	dist	654	412	1.04	71.0	706	3119	0.37	71.0	2930	1.91	2.49
	Europe	time	654	412	1.10	71.0	706	3119	0.37	71.0	3049	1.67	1.85
TIGER	US	dist	519	569	1.60	111.2	619	5356	0.71	111.2	3088	1.86	2.81
	US	time	519	569	1.62	111.2	619	5356	0.71	111.2	2961	1.61	1.90
OSM	Australia	time	63	88	0.18	4.7	80	435	0.04	4.7	1116	0.29	0.42
	S. America	time	175	245	0.61	20.5	224	2106	0.27	20.5	1241	0.34	0.66
	N. America	time	2173	2968	6.32	199.2	2766	15564	1.54	199.2	2996	1.63	3.68
	Old World	time	2843	3431	7.54	195.8	3659	17471	1.47	195.8	2582	1.49	4.25
Bing	N. America	dflt	769	726	2.45	136.5	941	6547	0.77	136.5	3382	1.61	1.91
	Europe	dflt	1246	1004	2.29	120.6	1447	6006	0.61	120.6	3667	1.98	2.33

Table 10 shows the performance of CRP on these instances. We evaluate our two variants of CRP: our default version (without contraction), and a version where we customize by contraction up to cells of size 256. For both versions, we report the time for preprocessing and customization, as well as how much data those two phases generate generates, followed by average statistics about queries (over 100 000 runs): number of scans, time to get the cost of the shortest path, and time to get a full description of the path (cost and underlying arcs). Queries are sequential and use a (prewarmed) LRU cache for 2^{18} shortcuts; preprocessing and customization run on 12 cores. Note that we use a seventh overlay level (cell size 2^{23}) for the two largest OSM instances, representing North America and Old World (Africa, Asia, and Europe).

The table shows that our default version of CRP is indeed robust, enabling consistently fast customization and queries. It is slowest for OSM instances, which are very large because (unlike other inputs) they use vertices to represent both intersections and geometry. Even so, customization takes less than 8 seconds, and queries take under 2 milliseconds. Preprocessing time is dominated by partitioning. The amount of metric-dependent data is relatively small, while metric-independent space usage is similar to the amount needed to store the original graph. While finding the cost of a path takes similar time on most instances, describing the path takes longer on OSM data. For every instance, customization is faster than a single Dijkstra search. The main reason is that Dijkstra's algorithm has poor locality, since its working set is spread throughout the graph. Moreover, customization is trivial to parallelize with almost perfect speedups, while known parallelization techniques for Dijkstra's algorithm [50] do not scale well on sparse graphs.

We also observe that using contraction to compute the bipartite graphs on the lowest level reduces customization times by a factor between 2 and 6, but increases the metric-independent space consumption by up to a factor of 10. As discussed before, this high price in memory consumption is only worth the effort if customization times need to be as fast as possible.

8 Conclusion

We have proposed the first routing engine that satisfies all requirements of a real-world production system. To our own surprise, adapting the recent research on route planning algorithms to such a system went far beyond a simple engineering effort. In particular, we realized that the most promising approach, contraction hierarchies, suffers from some critical drawbacks, such as the high sensitivity to small metric changes. It turns out that the well-known separator-based multilevel approach is the most adequate, even though previous studies have (prematurely) considered it to be inferior. We have shown that careful engineering, together with recent advances in graph partitioning, can improve query speedups relative to Dijkstra quite drastically, from less than 60 (as reported by Holzer et al. [39]) to more than 1500. The speedup is even higher (3000) once turn costs are taken into account. This makes real-time queries possible with this approach. Another

major contribution of our work is the explicit separation of metric customization from metric-independent preprocessing. It allows us to process a completely new metric in a second or less. Not only is this orders of magnitude faster than any previous approach, but it also enables highly desirable features such as incorporating real-time traffic information or even personalized driving directions. The resulting routing engine is a flexible and practical solution to many real-life variants of the problem, making it a perfect fit for Bing Maps.

Acknowledgments We thank Ittai Abraham and Ilya Razenshteyn for interesting discussions and their valuable input. We are also grateful to Christian Vetter for making his code [33] available, and to Dennis Luxen for providing routable OSM instances (`http://project-osrm.org/`). Finally, we thank the Bing Maps team for illuminating discussions on the requirements of real-world production systems.

References

1. I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck. A Hub-Based Labeling Algorithm for Shortest Paths on Road Networks. In P. M. Pardalos and S. Rebennack, editors, *Proceedings of the 10th International Symposium on Experimental Algorithms (SEA'11)*, volume 6630 of *Lecture Notes in Computer Science*, pages 230–241. Springer, 2011.
2. I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck. Hierarchical Hub Labelings for Shortest Paths. In L. Epstein and P. Ferragina, editors, *Proceedings of the 20th Annual European Symposium on Algorithms (ESA'12)*, volume 7501 of *Lecture Notes in Computer Science*, pages 24–35. Springer, 2012.
3. I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck. Alternative Routes in Road Networks. *ACM Journal of Experimental Algorithmics*, 18(1):1–17, 2013.
4. J. Arz, D. Luxen, and P. Sanders. Transit Node Routing Reconsidered. In *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13)*, volume 7933 of *Lecture Notes in Computer Science*, pages 55–66. Springer, 2013.
5. H. Bast, S. Funke, and D. Matijevic. Ultrafast Shortest-Path Queries via Transit Nodes. In C. Demetrescu, A. V. Goldberg, and D. S. Johnson, editors, *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, volume 74 of *DIMACS Book*, pages 175–192. American Mathematical Society, 2009.
6. H. Bast, S. Funke, P. Sanders, and D. Schultes. Fast Routing in Road Networks with Transit Nodes. *Science*, 316(5824):566, 2007.
7. G. V. Batz, R. Geisberger, P. Sanders, and C. Vetter. Minimum Time-Dependent Travel Times with Contraction Hierarchies. *ACM Journal of Experimental Algorithmics*, 18(1.4):1–43, April 2013.
8. R. Bauer, T. Columbus, B. Katz, M. Krug, and D. Wagner. Preprocessing Speed-Up Techniques is Hard. In *Proceedings of the 7th Conference on Algorithms and Complexity (CIAC'10)*, volume 6078 of *Lecture Notes in Computer Science*, pages 359–370. Springer, 2010.
9. R. Bauer, T. Columbus, I. Rutter, and D. Wagner. Search-Space Size in Contraction Hierarchies. In *Proceedings of the 40th International Colloquium on Automata, Languages, and Programming (ICALP'13)*, volume 7965 of *Lecture Notes in Computer Science*, pages 93–104. Springer, 2013.
10. R. Bauer and D. Delling. SHARC: Fast and Robust Unidirectional Routing. *ACM Journal of Experimental Algorithmics*, 14(2.4):1–29, August 2009. Special Section on Selected Papers from ALENEX 2008.
11. R. Bauer, D. Delling, P. Sanders, D. Schieferdecker, D. Schultes, and D. Wagner. Combining Hierarchical and Goal-Directed Speed-Up Techniques for Dijkstra's Algorithm. *ACM Journal of Experimental Algorithmics*, 15(2.3):1–31, January 2010. Special Section devoted to WEA'08.
12. R. Bellman. On a Routing Problem. *Quarterly of Applied Mathematics*, 16:87–90, 1958.
13. Cambridge Vehicle Information Technology Ltd. Choice Routing, 2005. Available at `http://www.camvit.com`.
14. D. Delling, A. V. Goldberg, A. Nowatzyk, and R. F. Werneck. PHAST: Hardware-accelerated shortest path trees. *Journal of Parallel and Distributed Computing*, 73(7):940–952, 2013.
15. D. Delling, A. V. Goldberg, T. Pajor, and R. F. Werneck. Customizable Route Planning. In P. M. Pardalos and S. Rebennack, editors, *Proceedings of the 10th International Symposium on Experimental Algorithms (SEA'11)*, volume 6630 of *Lecture Notes in Computer Science*, pages 376–387. Springer, 2011.
16. D. Delling, A. V. Goldberg, I. Razenshteyn, and R. F. Werneck. Graph Partitioning with Natural Cuts. In *25th International Parallel and Distributed Processing Symposium (IPDPS'11)*, pages 1135–1146. IEEE Computer Society, 2011.

17. D. Delling, A. V. Goldberg, and R. F. Werneck. Hub Label Compression. In *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13)*, volume 7933 of *Lecture Notes in Computer Science*, pages 18–29. Springer, 2013.
18. D. Delling, M. Holzer, K. Müller, F. Schulz, and D. Wagner. High-Performance Multi-Level Routing. In C. Demetrescu, A. V. Goldberg, and D. S. Johnson, editors, *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, volume 74 of *DIMACS Book*, pages 73–92. American Mathematical Society, 2009.
19. D. Delling, P. Sanders, D. Schultes, and D. Wagner. Engineering Route Planning Algorithms. In J. Lerner, D. Wagner, and K. Zweig, editors, *Algorithmics of Large and Complex Networks*, volume 5515 of *Lecture Notes in Computer Science*, pages 117–139. Springer, 2009.
20. D. Delling and D. Wagner. Pareto Paths with SHARC. In J. 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.
21. D. Delling and D. Wagner. Time-Dependent Route Planning. In R. K. Ahuja, R. H. Möhring, and C. Zaroliagis, editors, *Robust and Online Large-Scale Optimization*, volume 5868 of *Lecture Notes in Computer Science*, pages 207–230. Springer, 2009.
22. D. Delling and R. F. Werneck. Faster Customization of Road Networks. In *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13)*, volume 7933 of *Lecture Notes in Computer Science*, pages 30–42. Springer, 2013.
23. C. Demetrescu, A. V. Goldberg, and D. S. Johnson, editors. *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, volume 74 of *DIMACS Book*. American Mathematical Society, 2009.
24. E. V. Denardo and B. L. Fox. Shortest-Route Methods: 1. Reaching, Pruning, and Buckets. *Operations Research*, 27(1):161–186, 1979.
25. E. W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1:269–271, 1959.
26. R. W. Floyd. Algorithm 97: Shortest path. *Communications of the ACM*, 5(6):345, 1962.
27. L. R. Ford Jr. and D. R. Fulkerson. *Flows in Networks*. Princeton U. Press, 1962.
28. M. L. Fredman and R. E. Tarjan. Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms. *Journal of the ACM*, 34(3):596–615, July 1987.
29. S. Funke and S. Storandt. Polynomial-time Construction of Contraction Hierarchies for Multi-criteria Objectives. In *Proceedings of the 15th Meeting on Algorithm Engineering and Experiments (ALENEX'13)*, pages 31–54. SIAM, 2013.
30. R. Geisberger, M. Kobitzsch, and P. Sanders. Route Planning with Flexible Objective Functions. In *Proceedings of the 12th Workshop on Algorithm Engineering and Experiments (ALENEX'10)*, pages 124–137. SIAM, 2010.
31. R. Geisberger, M. Rice, P. Sanders, and V. Tsotras. Route Planning with Flexible Edge Restrictions. *ACM Journal of Experimental Algorithmics*, 17(1):1–20, 2012.
32. R. Geisberger, P. Sanders, D. Schultes, and C. Vetter. Exact Routing in Large Road Networks Using Contraction Hierarchies. *Transportation Science*, 46(3):388–404, August 2012.
33. R. Geisberger and C. Vetter. Efficient Routing in Road Networks with Turn Costs. In P. M. Pardalos and S. Rebennack, editors, *Proceedings of the 10th International Symposium on Experimental Algorithms (SEA'11)*, volume 6630 of *Lecture Notes in Computer Science*, pages 100–111. Springer, 2011.
34. A. V. Goldberg. A Practical Shortest Path Algorithm with Linear Expected Time. *SIAM Journal on Computing*, 37:1637–1655, 2008.
35. A. V. Goldberg and C. 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. SIAM, 2005.
36. A. V. Goldberg, H. Kaplan, and R. F. Werneck. Reach for A*: Shortest Path Algorithms with Preprocessing. In C. Demetrescu, A. V. Goldberg, and D. S. Johnson, editors, *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, volume 74 of *DIMACS Book*, pages 93–139. American Mathematical Society, 2009.
37. A. V. Goldberg and R. 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.
38. M. Hilger, E. Köhler, R. H. Möhring, and H. Schilling. Fast Point-to-Point Shortest Path Computations with Arc-Flags. In C. Demetrescu, A. V. Goldberg, and D. S. Johnson, editors, *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, volume 74 of *DIMACS Book*, pages 41–72. American Mathematical Society, 2009.
39. M. Holzer, F. Schulz, and D. Wagner. Engineering Multi-Level Overlay Graphs for Shortest-Path Queries. *ACM Journal of Experimental Algorithmics*, 13(2.5):1–26, December 2008.
40. M. Holzer, F. Schulz, D. Wagner, G. Prasinos, and C. Zaroliagis. Engineering planar separator algorithms. *ACM Journal of Experimental Algorithmics*, 14(1):1–31, 2009.

41. M. Holzer, F. Schulz, D. Wagner, and T. Willhalm. Combining Speed-up Techniques for Shortest-Path Computations. *ACM Journal of Experimental Algorithmics*, 10(2.5):1–18, 2006.
42. Y.-W. Huang, N. Jing, and E. A. Rundensteiner. Effective Graph Clustering for Path Queries in Digital Maps. In *Proceedings of the 5th International Conference on Information and Knowledge Management*, pages 215–222. ACM Press, 1996.
43. S. Jung and S. Pramanik. An Efficient Path Computation Model for Hierarchically Structured Topographical Road Maps. *IEEE Transactions on Knowledge and Data Engineering*, 14(5):1029–1046, September 2002.
44. G. Karypis and G. Kumar. A Fast and Highly Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1999.
45. U. Lauther. An Experimental Evaluation of Point-To-Point Shortest Path Calculation on Roadnetworks with Precalculated Edge-Flags. In C. Demetrescu, A. V. Goldberg, and D. S. Johnson, editors, *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, volume 74 of *DIMACS Book*, pages 19–40. American Mathematical Society, 2009.
46. R. J. Lipton and R. E. Tarjan. A Separator Theorem for Planar Graphs. *SIAM Journal on Applied Mathematics*, 36(2):177–189, April 1979.
47. D. Luxen and D. Schieferdecker. Candidate Sets for Alternative Routes in Road Networks. In *Proceedings of the 11th International Symposium on Experimental Algorithms (SEA'12)*, volume 7276 of *Lecture Notes in Computer Science*, pages 260–270. Springer, 2012.
48. J. Maue, P. Sanders, and D. Matijevic. Goal-Directed Shortest-Path Queries Using Precomputed Cluster Distances. *ACM Journal of Experimental Algorithmics*, 14:3.2:1–3.2:27, 2009.
49. K. Mehlhorn and P. Sanders. *Algorithms and Data Structures: The Basic Toolbox*. Springer, 2008.
50. U. Meyer and P. Sanders. Δ-Stepping: A Parallelizable Shortest Path Algorithm. *Journal of Algorithms*, 49(1):114–152, 2003.
51. N. Milosavljevic. On optimal preprocessing for contraction hierarchies. In *Proceedings of the 5th ACM SIGSPATIAL International Workshop on Computational Transportation Science (IWCTS)*, page Paper 6, 2012.
52. L. F. Muller and M. Zachariasen. Fast and Compact Oracles for Approximate Distances in Planar Graphs. In *Proceedings of the 14th Annual European Symposium on Algorithms (ESA'07)*, volume 4698 of *Lecture Notes in Computer Science*, pages 657–668. Springer, 2007.
53. F. Pellegrini and J. Roman. SCOTCH: A Software Package for Static Mapping by Dual Recursive Bipartitioning of Process and Architecture Graphs. In *High-Performance Computing and Networking*, volume 1067 of *Lecture Notes in Computer Science*, pages 493–498. Springer, 1996.
54. P. Sanders and D. Schultes. Highway Hierarchies Hasten Exact Shortest Path Queries. In *Proceedings of the 13th Annual European Symposium on Algorithms (ESA'05)*, volume 3669 of *Lecture Notes in Computer Science*, pages 568–579. Springer, 2005.
55. P. Sanders and D. Schultes. Robust, Almost Constant Time Shortest-Path Queries in Road Networks. In C. Demetrescu, A. V. Goldberg, and D. S. Johnson, editors, *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, volume 74 of *DIMACS Book*, pages 193–218. American Mathematical Society, 2009.
56. P. Sanders and D. Schultes. Engineering Highway Hierarchies. *ACM Journal of Experimental Algorithmics*, 17(1):1–40, 2012.
57. P. Sanders and C. Schulz. Think Locally, Act Globally: Highly Balanced Graph Partitioning. In *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13)*, volume 7933 of *Lecture Notes in Computer Science*, pages 164–175. Springer, 2013.
58. J.-O. Sasse. Route Planning in Road Networks with Turn Costs and Multi Edge Restrictions. Diploma thesis, Karlsruhe Institute of Technology, November 2011.
59. H. Schilling. TomTom Navigation - How mathematics help getting through traffic faster, 2012. Talk given at ISMP.
60. F. Schulz, D. Wagner, and K. Weihe. Dijkstra's Algorithm On-Line: An Empirical Case Study from Public Railroad Transport. *ACM Journal of Experimental Algorithmics*, 5(12):1–23, 2000.
61. F. Schulz, D. Wagner, and C. Zaroliagis. Using Multi-Level Graphs for Timetable Information in Railway Systems. In *Proceedings of the 4th Workshop on Algorithm Engineering and Experiments (ALENEX'02)*, volume 2409 of *Lecture Notes in Computer Science*, pages 43–59. Springer, 2002.
62. C. Sommer. Shortest-Path Queries in Static Networks, 2012. submitted.
63. C. Vetter. Parallel Time-Dependent Contraction Hierarchies, 2009. Student Research Project. `http://algo2.iti.kit.edu/download/vetter_sa.pdf`.
64. H. Yanagisawa. A Multi-Source Label-Correcting Algorithm for the All-Pairs Shortest Paths Problem. In *24th International Parallel and Distributed Processing Symposium (IPDPS'10)*, pages 1–10. IEEE Computer Society, 2010.