Fast Exact Computation of Isochrones in Road Networks

Moritz Baum, Valentin Buchhold(✉), Julian Dibbelt, and Dorothea Wagner

Karlsruhe Institute of Technology, Karlsruhe, Germany
{moritz.baum,valentin.buchhold,julian.dibbelt,dorothea.wagner}@kit.edu

Abstract. We study the problem of computing isochrones in static and dynamic road networks, where the objective is to identify the boundary of the region in range from a given source within a certain amount of time. While there is a wide range of practical applications for this problem (e. g., urban planning, geomarketing, visualizing the cruising range of a vehicle), there has been little research on fast algorithms for large, realistic inputs, and existing approaches tend to compute more information than necessary. Our contribution is twofold: (1) We propose a more compact but sufficient definition of isochrones, based on which, (2) we provide several easy-to-parallelize, scalable algorithmic approaches for faster computation. By extensive experimental analysis, we demonstrate that our techniques enable fast isochrone computation within milliseconds even on continental networks, significantly faster than the state-of-the-art.

1 Introduction

Online map services, navigation systems, and other route planning and location-based applications have gained wide usage, driven by significant advances [2] in shortest path algorithms for, e. g., location-to-location, many-to-many, POI, or kNN queries. Less attention has been given to the fast computation of *isochrones*, despite its relevance in urban planning [3,23,24,33,35], geomarketing [17], range visualization for (electric) vehicles [4,28], and other applications [30].

Interestingly, there is no canonical definition of isochrones in the literature. A unifying property, however, is the consideration of a range limit (time or some other limited resource), given only a source location for the query and no specific target. As a basic approach, a pruned variant of Dijkstra's algorithm [16] can be used to compute shortest path distances to all vertices within range. Newer approaches [18,23,24] still subscribe to the same model (computing distances). However, for the applications mentioned above it suffices to identify only the set of vertices or edges within range (and no distances). Moreover, for visualization [4] it serves to find just the vertices and edges on the boundary of the range. Exploiting these observations, we derive new approaches for faster computation of isochrones.

Supported by the EU FP7 under grant agreement no. 609026 (project MOVE-SMART).

A.V. Goldberg and A.S. Kulikov (Eds.): SEA 2016, LNCS 9685, pp. 17–32, 2016.
DOI: 10.1007/978-3-319-38851-9_2

Related Work. Despite its low asymptotic complexity, Dijkstra's algorithm [16] is too slow in practice. *Speedup techniques* [2] accelerate online shortest-path queries with data *preprocessed* in an offline phase. Many employ *overlay edges (shortcuts)* that maintain shortest path distances, allowing queries to skip parts of the graph. Contraction Hierarchies (CH) [27] contracts vertices in increasing order of importance, creating shortcuts between yet uncontracted neighbors. Customizable Route Planning (CRP) [7] adds shortcuts between separators of a multilevel partition [10,29,31]. As separators are independent of routing costs, CRP offers fast, dynamic *customization* of preprocessed data to a new cost metric (e. g., user preferences, traffic updates). Customizable CH (CCH) was evaluated in [14,15].

While proposed for point-to-point queries, both CH and CRP can be extended to other scenarios. Scanning the hierarchy (induced by a vertex order or multi-level partition, respectively) in a final top-down sweep enables *one-to-all* queries: PHAST [5] applies this to CH, GRASP [18] to CRP. For *one-to-many* queries, RPHAST [5] and reGRASP [18] restrict the downward search by initial target selection. POI, kNN, and similar queries are possible [1,12,19,22,25,26,32].

Since the boundary of an isochrone is not known in advance but part of the query output, target selection (as in one-to-many queries) or backward searches (as in [19]) are not directly applicable in our scenario. To the best of our knowledge, the only speedup technique extended to isochrone queries is GRASP.[1] However, isoGRASP [18] computes distances to all vertices in range, which is more than we require. MINE [23] and MINEX [24] consider multimodal networks (including road and public transit), however, due to the lack of preprocessing, running times are prohibitively slow, even on instances much smaller than ours.

Our Contribution. We give a compact definition of isochrones that serves the applications mentioned above, but requires no output of distances (Sect. 2). We propose several techniques that enable fast computation of isochrones and are easy to parallelize. First, we describe a new algorithm based on CRP (Sect. 3). Moreover, we present a faster variant of isoGRASP [18], exploiting that distances are not required (Sect. 4). Then, we introduce novel approaches that combine graph partitions with variants of (R)PHAST (Sect. 5). Our experimental evaluation (Sect. 7) on large, realistic input reveals that our techniques compute isochrones in a few milliseconds, clearly outperforming the state-of-the-art.

2 Problem Statement and Basic Approach

Let $G = (V, E, \text{len})$ be a directed, weighted graph, representing the road network, with *length function* $\text{len}\colon E \to \mathbb{R}_{\geq 0}$, representing, e. g., travel time. Denote

[1] Extension of CRP to isochrones is outlined in a patent (US Patent App. 13/649,114; http://www.google.com/patents/US20140107921), however, in a simpler than our intended scenario. Furthermore, the approach was neither implemented nor evaluated.

by $d\colon V \times V \mapsto \mathbb{R}_{\geq 0}$ the associated shortest path distance. We assume that G is *strongly connected*. Our *isochrone problem* takes as input a source $s \in V$ and a limit $\tau \in \mathbb{R}_{\geq 0}$. We say that a vertex $v \in V$ is *in range* if $d(s, v) \leq \tau$, else it is *out of range*. We define the output of the isochrone problem as the set of all *isochrone edges* that separate vertices in range from those out of range. Observe that these are the edges $(u, v) \in E$ with exactly one endpoint in range. To distinguish, we call e *outward (isochrone)* if and only if $d(s, u) \leq \tau, d(s, v) > \tau$ and *inward (isochrone)* if and only if $d(s, u) > \tau, d(s, v) \leq \tau$. This set of edges compactly represents the area in range [4]. However, all approaches presented below can be modified to serve other output definitions (requiring, e. g., the set of vertices in range); see Sect. 6. In what follows, we first describe a basic approach for the isochrone problem as specified above.[2] Afterwards, we propose speedup techniques that employ offline preprocessing on the graph G to quickly answer online queries consisting of a source $s \in V$ and a limit $\tau \in \mathbb{R}_{\geq 0}$. We distinguish *metric-independent* preprocessing (must be run when the topology of the input graph changes) and *metric-dependent* customization (only the length function changes).

Basic Approach. Dijkstra's algorithm [16] computes distances $d(s, v)$ from a source s to all $v \in V$. It maintains *distance labels* $d(\cdot)$ for each vertex, initially set to ∞ (except $d(s) = 0$). In each iteration, it extracts a vertex u with minimum $d(u)$ from a priority queue (initialized with s) and *settles* it. At this point, $d(u)$ is *final*, i. e., $d(u) = d(s, u)$. It then *scans* all edges (u, v): If $d(u) + \text{len}(u, v) < d(v)$, it updates $d(v)$ accordingly and adds (or updates) v in the queue. For our problem setting, *isoDijkstra* can be stopped once the distance label of the minimum element in the queue exceeds the limit τ (*stopping criterion*). Then, outward isochrone edges are easily determined: We sweep over all vertices left in the queue, which must be out of range, outputting incident edges where the other endpoint is in range. Inward isochrone edges can be determined during the same sweep if we apply the following modification to the graph search. When settling a vertex u, we also scan incoming edges (v, u). If $d(v) = \infty$, we insert v into the queue with a key of infinity. Thereby, we guarantee that for both types of isochrone edges the unreachable endpoint is contained in the queue when the search terminates.

Partitions. Below, we propose speedup techniques based on graph partitions. Formally, a *(vertex) partition* is a family $\mathcal{V} = \{V_1, \ldots, V_k\}$ of *cells* $V_i \subseteq V$, such that $V_i \cap V_j = \emptyset$ for $i \neq j$ and $\bigcup_{i=1}^{k} V_i = V$. A *(nested) multilevel partition* with L levels is a family $\Pi = \{\mathcal{V}^1, \ldots, \mathcal{V}^L\}$ of partitions of nested cells, i. e., for each level $\ell \leq L$ and cell $V_i^\ell \in \mathcal{V}^\ell$, there is a cell $V_j^{\ell+1} \in \mathcal{V}^{\ell+1}$ at level $\ell + 1$ with $V_i^\ell \subseteq V_j^{\ell+1}$. For consistency, we define $\mathcal{V}^0 := \{\{v\} \mid v \in V\}$ (the trivial partition where each vertex has its own cell) and $\mathcal{V}^{L+1} := \{V\}$ (the trivial single-cell partition). An edge $(u, v) \in E$ is a *boundary edge* (u and v are *boundary*

[2] Strictly speaking, iso*chrone* implies time as a resource. While *isoline* or *isocontour* would be more precise, we have settled for the term most common in the literature.

vertices) on level ℓ, if u and v are in different cells of $\mathcal{V}^\ell$. Similar to vertex partitions, we define *edge partitions* $\mathcal{E} = \{E_1, \ldots, E_k\}$, with $E_i \cap E_j = \emptyset$ for $i \neq j$ and $\bigcup_{i=1}^{k} E_i = E$. A vertex $v \in V$ is *distinct* (wrt. $\mathcal{E}$) if all its incident edges belong to the same cell, else v is a *boundary vertex* or *ambiguous*.

3 IsoCRP

The three-phase workflow of CRP [7] distinguishes preprocessing and metric customization. Preprocessing finds a (multilevel) vertex partition of the road network, inducing for each level ℓ an *overlay graph* H^ℓ containing all boundary vertices and boundary edges wrt. $\mathcal{V}^\ell$, and *shortcut edges* between pairs of boundary vertices that belong to the same cell $V_i^\ell \in \mathcal{V}^\ell$. Metric customization computes the lengths of all shortcuts. The basic idea of *isoCRP* is to run isoDijkstra on the overlay graphs. Thus, we use shortcuts to skip cells that are entirely in range, but *descend* into lower levels in cells that intersect the isochrone frontier, to determine isochrone edges. There are two major challenges. First, descending into cells where shortcuts exceed the limit τ is not sufficient (we may miss isochrone edges that are part of no shortcut, but belong to shortest paths leading into the cell), so we have to precompute additional information. Second, descents into cells must be consistent for all boundary vertices (i. e., we have to descend at all vertices), motivating two-phase queries.

Customization. Along the lines of plain CRP, we obtain shortcut lengths by running Dijkstra's algorithm restricted to the respective cell. Additionally, we make use of the same searches to compute *eccentricities* for all boundary vertices. Given a boundary vertex u in a cell V_i^ℓ, its (level-ℓ) eccentricity, denoted $\mathrm{ecc}_\ell(u)$, is the maximum finite distance to some $v \in V_i^\ell$ in the subgraph induced by V_i^ℓ. This subgraph is not strongly connected in general (i. e., some vertices may be unreachable), but restricting eccentricities to cells allows fast customization.

At the lowest level, the eccentricity of a boundary vertex u is the distance label of the last vertex settled in the search from u. To accelerate customization, previously computed overlays are used to obtain shortcuts on higher levels. We compute *upper bounds* on eccentricities for those levels. When settling a vertex v, we check if the sum of the label $d(v)$ and $\mathrm{ecc}_{\ell-1}(v)$ exceeds the current bound on $\mathrm{ecc}_\ell(u)$ and update it if needed. Shortcuts of a cell are represented as a square matrix for efficiency, and storing eccentricities adds a single column to them.

To improve data locality and simplify index mapping, vertices are reordered in descending order of level during preprocessing, breaking ties by cell [7].

Queries. We say that a cell is *active* if its induced subgraph contains at least one isochrone edge. Given a source $s \in V$ and a limit τ, queries work in two phases. The first phase determines active cells, while the second phase descends into active cells to determine isochrone edges. The *upward phase* runs isoDijkstra on the search graph consisting of the union of the top-level overlay and all subgraphs induced by cells containing s. To determine active cells, we maintain two

flags $\texttt{i}(\cdot)$ (initially false) and $\texttt{o}(\cdot)$ (initially true) per cell and level, to indicate whether the cell contains at least one vertex that is *in* or *out* of range, respectively. When settling a vertex $u \in V_i^\ell$, we set $\texttt{i}(V_i^\ell)$ to true if $d(u) \leq \tau$. Next, we check whether $d(u) + \text{ecc}_\ell(u) \leq \tau$. Observe that this condition is not sufficient to unset $\texttt{o}(V_i^\ell)$, because $\text{ecc}_\ell(u)$ was computed in the subgraph of V_i^ℓ. If this subgraph is not strongly connected, $d(u) + \text{ecc}_\ell(u)$ is not an upper bound on the distance to any vertex in V_i^ℓ in general. Therefore, when scanning an outgoing shortcut (u, v) with length ∞ (such shortcuts exist due to the matrix representation), we also check whether $d(v) + \text{ecc}_\ell(v) \leq \tau$. If the condition holds for u and all boundary vertices v unreachable from u (wrt. V_i^ℓ), we can safely unset $\texttt{o}(V_i^\ell)$. Toggled flags are *final*, so we no longer need to perform any checks for them. After the upward phase finished, cells V_i^ℓ that have both $\texttt{i}(V_i^\ell)$ and $\texttt{o}(V_i^\ell)$ set are active (isochrone edges are only contained in cells with vertices both in and out of range).

The *downward phase* has L subphases. In descending order of level, and for every active cell at the current level ℓ, each subphase runs isoDijkstra restricted to the respective cell in $H_{\ell-1}$. Initially, all boundary vertices are inserted into the queue with their distance labels according to the previous phase as key. As before, we check eccentricities on-the-fly to mark active cells for the next subphase. Isochrone edges are determined at the end of each isoDijkstra search (see Sect. 2). On overlays, only boundary edges are reported.

Parallelization. For faster customization, cells of each level are processed in parallel [7]. During queries, the (much more expensive) downward phase is parallelized in a natural way, as cells at a certain level can be handled independently. We assign cells to threads and synchronize results between subphases. To reduce the risk of false sharing, we assign blocks of consecutive cells (wrt. vertex ordering) to the same thread. Moreover, to reduce synchronization overhead, we process cells on lower levels in a top-down fashion within the same thread.

4 Faster IsoGRASP

GRASP [18] extends CRP to batched query scenarios by storing for each level-ℓ boundary vertex, $0 \leq \ell < L$, (incoming) *downward shortcuts* from boundary vertices of its supercell at level $\ell + 1$. Customization follows CRP, collecting downward shortcuts in a separate *downward graph* $H^\downarrow$. Original *isoGRASP* [18] runs Dijkstra's algorithm on the overlays (as in CRP), marks all in-range top-level cells, and propagates distances in marked cells from boundary vertices to those at the levels below in a sweep over the corresponding downward shortcuts. We accelerate isoGRASP significantly by making use of eccentricities.

Customization. Metric customization of our variant of isoGRASP is similar to isoCRP, computing shortcuts and eccentricities with Dijkstra's algorithm as in Sect. 3. We obtain downward shortcuts in the same Dijkstra searches. We apply

edge reduction (removing shortcuts via other boundary vertices) [18] to downward shortcuts, but use the matrix representation for overlay shortcuts.

Queries. As in isoCRP, queries run two phases, with the *upward phase* being identical to the one described in Sect. 3. Then, the *scanning phase* handles levels from top to bottom in L subphases to process active cells. For an active level-ℓ cell V_i^ℓ, we sweep over its *internal* vertices (i. e., all vertices of the overlay $H_{\ell-1}$ that lie in V_i^ℓ and are no level-ℓ boundary vertices). For each internal vertex v, its incoming downward shortcuts are scanned, obtaining the distance to v. To determine active cells for the next subphase, we maintain flags $\texttt{i}(\cdot)$ and $\texttt{o}(\cdot)$ as in isoCRP. This requires checks at all boundary vertices that are unreachable from v within $V_i^{\ell-1}$. We achieve some speedup by precomputing these vertices, storing them in a separate adjacency array.

Similar to isoCRP, the upward phase reports all (original) isochrone edges contained in its search graph. For the remaining isochrone edges, we sweep over internal vertices and their incident edges a second time after processing a cell in the scanning phase. To avoid duplicates and to ensure that endpoints of examined edges have correct distances, we skip edges leading to vertices with higher indices. Both queries and customization are parallelized in the same fashion as isoCRP.

5 IsoPHAST

Preprocessing of PHAST [5] contracts vertices in increasing order of (heuristic) importance, as in the point-to-point technique CH [27]. To contract a vertex, shortcut edges are added between yet uncontracted neighbors to preserve distances, if necessary. Vertices are assigned *levels* $\ell(\cdot)$, initially set to zero. When contracting u, we set $\ell(v) = \max\{\ell(v), \ell(u) + 1\}$ for each uncontracted neighbor v. Given the set E^+ of all shortcuts added during preprocessing, PHAST handles one-to-all queries from some given source s as follows. During the *forward CH search*, it runs Dijkstra's algorithm on $G^\uparrow = (V, E^\uparrow)$, $E^\uparrow = \{(u, v) \in E \cup E^+ \colon \ell(u) < \ell(v)\}$. The subsequent *downward phase* is a linear sweep over all vertices in descending order of level, reordered accordingly during preprocessing. For each vertex, it scans its incoming edges in $E^\downarrow = \{(u, v) \in E \cup E^+ \colon \ell(u) > \ell(v)\}$ to update distances. Afterwards, distances from s to all $v \in V$ are known. RPHAST [9] is tailored to one-to-many queries with given target sets T. It first extracts the relevant subgraph $G_T^\downarrow$ that is reachable from vertices in T by a backward search in $G^\downarrow = (V, E^\downarrow)$. Then, it runs the linear sweep for $G_T^\downarrow$.

Our *isoPHAST* algorithm builds on (R)PHAST to compute isochrones. Since the targets are not part of the input, we use graph partitions to restrict the subgraph that is examined for isochrone edges. Queries work in three phases, in which we (1) run a forward CH search, (2) determine active cells, and (3) perform linear sweeps over all active cells as in PHAST. Below, we describe preprocessing of isoPHAST, before proposing different strategies to determine active cells.

First, we find a (single-level) partition $\mathcal{V} = \{V_1, \ldots, V_k\}$ of the road network and reorder vertices such that boundary vertices (or *core* vertices) are pushed to the front, breaking ties by cell (providing the same benefits as in CRP). Afterwards, we use CH to contract all cell-induced subgraphs, but leave core vertices *uncontracted*. Non-core vertices inside cells are reordered according to their CH levels to enable linear downward sweeps. The output of preprocessing consists of an *upward graph* $G^{\uparrow}$, containing for each cell all edges leading to vertices of higher level, added shortcuts between core vertices, and all boundary edges. We also obtain a *downward graph* $G^{\downarrow}$ that stores for each non-core vertex its incoming edges from vertices of higher level. Further steps of preprocessing depend on the query strategy and are described below.

IsoPHAST-CD. Our first strategy (Core-Dijkstra) performs isoDijkstra on the core graph to determine active cells. This requires eccentricities for core vertices, which are obtained during preprocessing as follows. To compute $\text{ecc}(u)$ for some vertex u, we run (as last step of preprocessing) Dijkstra's algorithm on the subgraph induced by all core vertices of $G^{\uparrow}$ in the cell V_i of u, followed by a linear sweep over the internal vertices of V_i. When processing a vertex v during this sweep, we update the eccentricity of u to $\text{ecc}(u) = \max\{\text{ecc}(u), d(v)\}$.

Queries start by running isoDijkstra from the source in $G^{\uparrow}$. Within the source cell, this corresponds to a forward CH search. At core vertices, we maintain flags $\texttt{i}(\cdot)$ and $\texttt{o}(\cdot)$ to determine active cells (as described in Sect. 3, using an adjacency array to store unreachable core neighbors as in Sect. 4). If the core is not reached, only the source cell is set active. Next, we perform for each active cell a linear sweep over its internal vertices, obtaining distances to all vertices that are both in range and contained in an active cell.

Isochrone edges crossing cell boundaries are added to the output during the isoDijkstra search, whereas isochrone edges connecting non-core vertices are obtained in the linear sweeps as follows. When scanning incident edges of a vertex v, neighbors at higher levels have final distance labels. Moreover, the label of v is final after scanning incoming edges $(u, v) \in G^{\downarrow}$. Thus, looping through incoming original edges a second time suffices to find the remaining isochrone edges. Since original edges $(v, u) \in E$ to vertices u at higher levels are not contained in $G^{\downarrow}$ in general, we add dummy edges of length ∞ to $G^{\downarrow}$ to ensure that neighbors in G are also adjacent in $G^{\downarrow}$.

isoPHAST-CP. Instead of isoDijktra, our second strategy (Core-PHAST) performs a linear sweep over the core. Eccentricities are precomputed after generic preprocessing as described above. Next, we use CH preprocessing to contract vertices in the core, and reorder core vertices according to their levels. Finally, we update $G^{\uparrow}$ and $G^{\downarrow}$ by adding core shortcuts.

Queries strictly follow the three-phase pattern discussed above. We first run a forward CH search in $G^{\uparrow}$. Then, we determine active cells and compute distances for all core vertices in a linear sweep over the core. Again, we maintain flags $\texttt{i}(\cdot)$ and $\texttt{o}(\cdot)$ for core vertices (cf. Section 3) and use an adjacency array storing unreachable core neighbors (cf. Section 4). To find isochrone edges between core

vertices, we insert dummy edges into the core to preserve adjacency. The third phase (linear sweeps over active cells) is identical to isoPHAST-CD.

isoPHAST-DT. Our third strategy (Distance Table) uses a *distance (bounds) table* to accelerate the second phase, determining active cells. Working with such tables instead of a dedicated core search benefits from *edge partitions*, since the unique assignment of edges to cells simplifies isochrone edge retrieval. Given a partition $\mathcal{E} = \{E_1, \ldots, E_k\}$ of the edges, the table stores for each pair E_i, E_j of cells a lower bound $\underline{d}(E_i, E_j)$ and an upper bound $\overline{d}(E_i, E_j)$ on the distance from E_i to E_j, i. e., $\underline{d}(E_i, E_j) \leq d(u, v) \leq \overline{d}(E_i, E_j)$ for all $u \in E_i$, $v \in E_j$ (we abuse notation, saying $u \in E_i$ if u is an endpoint of at least one edge $e \in E_i$). Given a source $s \in E_i$ (if s is ambiguous, pick any cell containing s) and a limit τ, cells E_j with $\underline{d}(E_i, E_j) \leq \tau < \overline{d}(E_i, E_j)$ are set active.

Preprocessing first follows isoPHAST-CP, with three differences: (1) We use an edge partition instead of a vertex partition; (2) Eccentricities are computed on the *reverse* graph, with Dijkstra searches that are not restricted to cells but stop when all boundary vertices of the current cell are reached; (3) After computing eccentricities, we recontract the whole graph using a CH order (i. e., contraction of core vertices is not delayed), leading to sparser graphs $G^{\uparrow}$ and $G^{\downarrow}$. Afterwards, to quickly compute (not necessarily tight) distance bounds, we run for each cell E_i a (multi-source) forward CH search in $G^{\uparrow}$ from all boundary vertices of E_i. Then, we perform a linear sweep over $G^{\downarrow}$, keeping track of the minimum and maximum distance label per target cell. This yields, for all cells, lower bounds $\underline{d}(E_i, \cdot)$, and upper bounds on the distance from *boundary* vertices of E_i to each cell. To obtain the desired bounds $\overline{d}(E_i, \cdot)$, we increase the latter values by the *(backward) boundary diameter* of E_i, i. e., the maximum distance from any vertex in E_i to a boundary vertex of E_i. This diameter equals the maximum eccentricity of the boundary vertices of E_i on the reverse graph (which we computed before). As last step of preprocessing, we extract and store the relevant search graph $G_i^{\downarrow}$ for each $E_i \in \mathcal{E}$. This requires a target selection phase as in RPHAST for each cell, using all (i. e., distinct and ambiguous) vertices of a cell as input.

Queries start with a forward CH search in $G^{\uparrow}$. Then, active cells are determined in a sweep over one row of the distance table. The third phase performs a linear sweep over $G_i^{\downarrow}$ for each active cell E_i, obtaining distances to all its vertices. Although vertices can be contained in multiple search graphs, distance labels do not need to be reinitialized between sweeps, since the source remains unchanged. To output isochrone edges, we proceed as before, looping through incoming downward edges twice (again, we add dummy edges to $G_i^{\downarrow}$ for correctness). To avoid duplicates (due to vertices contained in multiple search graphs), edges in $G_i^{\downarrow}$ have an additional flag to indicate whether the edge belongs to E_i.

Search graphs may share vertices, which increases memory consumption and slows down queries (e. g., the vertex with maximum level is contained in every search graph). We use *search graph compression*, i. e., we store the topmost vertices of the hierarchy (and their incoming edges) in a separate graph $G_{\mathsf{c}}^{\downarrow}$ and

remove them from all $G_i^{\downarrow}$. During queries, we first perform a linear sweep over $G_{\mathtt{c}}^{\downarrow}$ (obtaining distances for all $v \in G_{\mathtt{c}}^{\downarrow}$), before processing search graphs of active cells. The size of $G_{\mathtt{c}}^{\downarrow}$ (more precisely, its number of vertices) is a tuning parameter.

Parallelization. The first preprocessing steps are executed in parallel, namely, building cell graphs, contracting non-core vertices, inserting dummy edges, and reordering non-core vertices by level. Afterwards, threads are synchronized, and $G^{\uparrow}$ and $G^{\downarrow}$ are built sequentially. Eccentricities are again computed in parallel. Since our CH preprocessing is sequential, the core graph is contracted in a single thread (if needed). Computation of distance bounds is parallelized (if needed).

Considering queries, the first two phases are run sequentially. Both isoDijkstra and the forward CH search are difficult to parallelize. Executing PHAST (on the core) in parallel does not pay off (the core is rather dense, resulting in many levels). Distance table operations, on the other hand, are very fast, and parallelization is not necessary. In the third phase, however, active cells can be assigned to different threads. We store a copy of the graph $G^{\downarrow}$ once per NUMA node for faster access during queries. Running the third phase in parallel can make the second phase of isoPHAST-CP a bottleneck. Therefore, we alter the way of computing flags $\mathtt{i}(\cdot)$ and $\mathtt{o}(\cdot)$. When settling a vertex $v \in V_i$, we set $\mathtt{i}(V_i)$ if $d(v) \leq \tau$, and $\mathtt{o}(V_i)$ if $d(v) + \mathrm{ecc}(v) > \tau$. Note that these checks are less accurate (more flags are toggled), but we no longer have to check unreachable boundary vertices. Correctness of isoPHAST-CP is maintained, as no stopping criterion is applied and $\max_{v \in V_i}(d(v) + \mathrm{ecc}(v))$ is a valid upper bound on the distance to each vertex in V_i. Hence, no active cells are missed.

6 Alternative Outputs

Driven by our primary application, visualizing the cruising range of a vehicle, we introduced a compact, yet sufficient representation of isochrones. However, all approaches can be adapted to produce a variety of other outputs, without increasing their running times significantly. As an example, we modify our algorithms to output a list of all vertices in range (instead of isochrone edges).

Even without further modifications, we can check in constant time if a vertex is in range after running the query. Consider the (top-level) cell V_i of a vertex v. If $\mathtt{i}(V_i)$ is not set, the cell contains no in-range vertices and v must be out of range. Similarly, if $\mathtt{o}(V_i)$ is not set, v is in range. If both flags are set, we run the same check for the cell containing v on the level below. If both flags are set for the cell on level 1 containing v, we check if the distance label of v exceeds the time limit (since all cells considered are active, the distance label of v is correct).

A simple approach to output the vertices in range performs a sweep over all vertices and finds those in range as described above. We can do better by collecting vertices in range on-the-fly. During isoDijkstra searches and when scanning active cells, we output each scanned vertex that is in range. In the scanning phase we also add all internal vertices for cells V_i where $\mathtt{o}(V_i)$ is not set.

7 Experiments

Our code is written in C++ (using OpenMP) and compiled with g++ 4.8.3 -O3. Experiments were conducted on two 8-core Intel Xeon E5-2670 clocked at 2.6 Ghz, with 64 GiB of DDR3-1600 RAM, 20 MiB of L3 and 256 KiB of L2 cache. Results are checked against a reference implementation (isoDijkstra) for correctness.

Input and Methodology. Experiments were done on the European road network (with 18 million vertices and 42 million edges) made available for the 9th DIMACS Implementation Challenge [13], using travel times in seconds as edge lengths.

We implemented CRP following [7], with a matrix-based clique representation. Our GRASP implementation applies implicit initialization [18] and (downward) shortcut reduction [20]. The CH preprocessing routine follows [27], but takes priority terms and hop limits from [5]. We use PUNCH [8] to generate multilevel partitions for isoCRP/isoGRASP, and Buffoon [37] to find single-level partitions for isoPHAST. Edge partitions are computed following the approach in [36,38].

We report parallel customization times, and both sequential and parallel query times. Parallel execution uses all available cores. Customization times for isoPHAST exclude partitioning, since it is metric-independent. For queries, reported figures are averages of 1 000 random queries (per individual time limit τ).

Tuning Parameters. We conducted preliminary studies to obtain reasonable parameters for partitions and search graph compression. For isoCRP/isoGRASP, we use the 4-level partition from [6], with maximum cell sizes of 2^8, 2^{12}, 2^{16}, 2^{20}, respectively. Although [18] uses 16 levels, resorting to a 4-level partition had only minor effects in preliminary experiments (similar observations are made in [19]).

For sequential isoPHAST-CD (CP) queries, a partition with $k = 2^{12}$ (2^{11}) cells yields best query times. For fewer cells (i. e., coarser partitions), the third query phase scans a large portion of the graph and becomes the bottleneck. Using more fine-grained partitions yields a larger core graph, slowing down the second query phase. Consequently, fewer cells ($k = 256$) become favorable when queries are executed in parallel (as the third phase becomes faster due to parallelization).

For isoPHAST-DT, similar effects occur for different values of k. Moreover, search graph compression has a major effect on query times (and space consumption). If there are few vertices in $G_c^{\downarrow}$, then vertices at high levels occur in search graphs of multiple cells, but large $G_c^{\downarrow}$ cause unnecessary vertex scans. Choosing $k = 2^{14}$ (2^{12}) and $|G_c^{\downarrow}| = 2^{16}$ (2^{13}) yields fastest sequential (parallel) queries.

Evaluation. Table 1 summarizes the performance of all algorithms discussed in this paper, showing figures on customization and queries. We report query times for medium-range ($\tau = 100$) and long-range time limits ($\tau = 500$, this is the hardest limit for most approaches, since it maximizes the number of isochrone edges).

Table 1. Performance of our algorithms. We report parallel customization time and space consumption (space per additional metric is given in brackets, if it differs). The table shows the average number of settled vertices (Nmb. settled, in thousands) and running times of sequential and parallel queries, using time limits $\tau = 100$ and $\tau = 500$. Best values (except Dijkstra wrt. space) are highlighted in bold.

Algorithm	Thr.	Custom			$\tau = 100$ min		$\tau = 500$ min	
		Time [s]	Space	[MiB]	Nmb. settled	Time [ms]	Nmb. settled	Time [ms]
isoDijkstra	1	–	646		460 k	68.32	7041 k	1184.06
isoCRP	1	**1.70**	900	(138)	101 k	15.44	354 k	60.67
isoGRASP	1	2.50	1856	(1094)	120 k	10.06	387 k	37.77
isoPHAST-CD	1	26.11	785		440 k	**6.09**	1501 k	31.63
isoPHAST-CP	1	1221.84	**781**		626 k	15.02	2029 k	31.00
isoPHAST-DT	1	1079.11	2935		597 k	9.96	1793 k	**24.80**
isoCRP	16	**1.70**	900	(138)	100 k	2.73	354 k	7.86
isoGRASP	16	2.50	1856	(1094)	120 k	2.35	387 k	5.93
isoPHAST-CD	16	38.07	769		918 k	**1.61**	4578 k	8.22
isoPHAST-CP	16	1432.39	**766**		944 k	4.47	5460 k	7.86
isoPHAST-DT	16	865.50	1066		914 k	1.74	2979 k	**3.80**

As expected, techniques based on multilevel overlays provide better customization times, while isoPHAST achieves the lowest query times (CD for medium-range and DT for long-range queries, respectively). Customization of isoCRP and isoGRASP is very practical (below three seconds). The lightweight preprocessing of isoPHAST-CD pays off as well, allowing customization in less than 30 s. The comparatively high preprocessing times of isoPHAST-CP and DT are mainly due to expensive core contraction. Still, metric-dependent preprocessing is far below half an hour, which is suitable for applications that do not require real-time metric updates. Compared to isoCRP, isoGRASP requires almost an order of magnitude of additional space for the downward graph (having about 110 million edges).

Executed sequentially, all approaches take well below 100 ms, which is significantly faster than isoDijkstra. The number of settled vertices is considerably larger for isoPHAST, however, data access is more cache-efficient. IsoPHAST provides faster queries than the multilevel overlay techniques for both limits, with the execption of isoPHAST-CP for small ranges (since the whole core graph is scanned). Again, the performance of isoPHAST-CD is quite notable, providing the fastest queries for (reasonable) medium-range limits and decent query times for the long-range limit. Finally, query times of isoPHAST-DT show best scaling behavior, with lowest running times for hardest queries.

The lower half of Table 1 reports parallel times for the same set of queries. Note that preprocessing times of isoPHAST change due to different parameter choices. Most approaches scale very well with the number of threads, providing a speedup of (roughly) 8 using 16 threads. Note that factors (according to the table) are much lower for isoPHAST, since we use tailored partitions for sequential queries. In fact, isoPHAST-DT scales best when run on the same preprocessed data (speedup of 11), since its sequential workflow (forward CH search,

table scan) is very fast. Considering multilevel overlay techniques, isoGRASP scales worse than isoCRP (speedup of 6.5 compared to 7.7), probably because it is memory bandwidth bounded (while isoCRP comes with more computational overhead). Consequently, isoGRASP benefits greatly from storing a copy of the downward graph on each NUMA node. As one may expect, speedups are slightly lower for medium-range queries. The isoPHAST approaches yield best query times, below 2 ms for medium-range queries, and below 4 ms for the long-range limit. To summarize, all algorithms enable queries fast enough for practical applications, with speedups of more than two orders of magnitude compared to isoDijkstra.

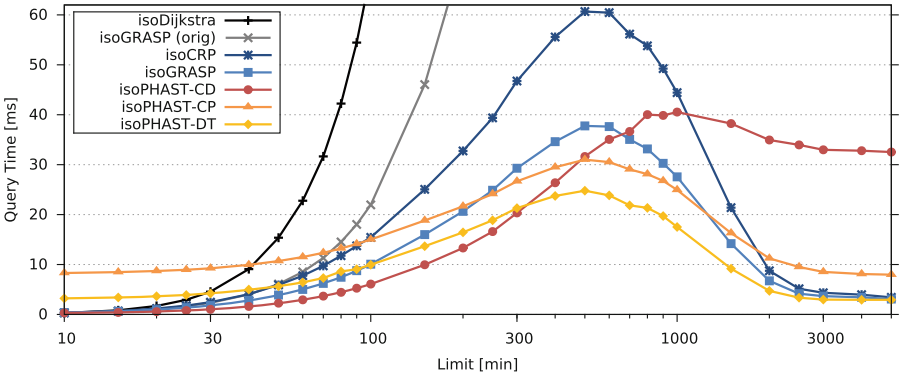

Fig. 1. Sequential query times for various time limits, ranging from 10 to (roughly) 4700 min (the diameter of our input graph).

Figure 1 shows how (sequential) query times scale with the time limit. For comparability, we also show (sequential) query times of original isoGRASP as described in [18] (computing distances to all in-range vertices, but no isochrone edges). Running times of all proposed algorithms (except isoDijkstra and original isoGRASP) follow a characteristic curve. Times first increase with the limit τ (the isochrone frontier is extended, intersecting more active cells), before dropping again once τ exceeds 500 min (the isochrone reaches the boundary of the network, decreasing the number of active cells). For $\tau > 4\,710$ min, all vertices are in range, making queries very fast, as there are no active cells. For small τ, the multilevel overlay techniques and isoPHAST-CD are fastest. IsoPHAST-CP is slowed down by the linear sweep over the core graph (taking about 6 ms, independent of τ), while isoPHAST-DT suffers from distance bounds not being tight. However, since Dijkstra's algorithm does not scale well, isoPHAST-CD becomes the slowest approach for large τ (while the other isoPHAST techniques benefit from good scaling behavior). Considering multilevel overlays, our isoGRASP is up to almost twice as fast as isoCRP, providing a decent trade-off between customization effort and query times. Note that while isoDijkstra is fast enough for some realistic time limits, it is not robust to user inputs. When executed in

Table 2. Impact of different outputs on the performance of isoCRP, isoGRASP, and a variant of isoPHAST (CP). We report sequential (seq.) and parallel (par.) query times as well as output size (# out., in thousands for vertices in range) when computing isochrone edges and vertices in range.

Algorithm	Limit [min]	Isochrone edges			Vertices in range		
		# out.	seq. [ms]	par. [ms]	# out.	seq. [ms]	par. [ms]
isoCRP	100	5937	15.44	2.73	460 k	15.83	2.77
	500	14718	60.67	7.86	7041 k	76.35	9.26
	5000	0	3.42	3.17	18010 k	46.64	6.64
isoGRASP	100	5937	10.06	2.35	460 k	11.07	2.50
	500	14718	37.77	5.93	7041 k	56.83	7.57
	5000	0	3.08	3.10	18010 k	46.09	6.44
isoPHAST	100	5937	15.02	4.47	460 k	16.40	4.70
	500	14718	31.00	7.86	7041 k	49.57	9.67
	5000	0	7.96	3.61	18010 k	50.86	7.03

parallel, query times follow the same characteristic curves (not reported in the figure). The linear sweep in the second phase of isoPHAST-CP becomes slightly faster, since the core is smaller (due to a different partition).

Alternative Outputs. Table 2 compares query times when computing different outputs (isochrone edges and vertices in range, respectively). For medium-range time limits ($\tau = 100$ min), both sequential and parallel query times increase by less than 10 %. When using long-range limits, where roughly half of the vertices are in range, sequential and parallel queries are slower by a factor of about 1.5, but still significantly faster than the original isoGRASP algorithm. Only when considering the graph diameter as time limit, sequential queries are significantly slower when computing all vertices in range, since variants reporting only isochrone edges can already terminate after the (very fast) upward phase.

Comparison with Related Work. Since we are not aware of any work solving our compact problem formulation, we cannot compare our algorithms directly to competitors. Hence, to validate the efficiency of our code, we compare our implementations of some basic building blocks to original publications. Table 3 reports running times for our implementations of Dijkstra's algorithm, GRASP, PHAST and RPHAST on one core of a 4-core Intel Xeon E5-1630v3 clocked at 3.7 GHz, with 128 GiB of DDR4-2133 RAM, 10 MiB of L3 and 256 KiB of L2 cache (chosen as it most closely resembles the machines used in [9,19]). For comparison, we report running times (as-is)

Table 3. Running times [ms] of basic one-to-all and one-to-many building blocks.

Algorithm	[our]	[9,19]
Dij (1-to-all)	2653.18	–
PHAST	144.16	136.92
GRASP	171.11	169.00
Dij (1-to-many)	7.34	7.43
RPHAST (select)	1.29	1.80
RPHAST (query)	0.16	0.17

from [9,19]. For the one-to-many scenario, we adopt the methodology from [9], using a target and ball size of 2^{14}. Even when accounting for hardware differences, running times of our implementations are similar to the original publications.

8 Final Remarks

We proposed a compact definition of isochrones, and introduced a portfolio of speedup techniques for the resulting isochrone problem. While no single approach is best in all criteria (preprocessing effort, space consumption, query time, simplicity), the right choice depends on the application. If user-dependent metrics are needed, the fast and lightweight customization of isoCRP is favorable. Fast queries subject to frequent metric updates (e. g., due to real-time traffic) are enabled by our isoGRASP variant. If customization time below a minute is acceptable and time limits are low, isoPHAST-CD provides even faster query times. The other isoPHAST variants show best scaling behavior, making them suitable for long-range isochrones, or if customizability is not required.

Regarding future work, we are interested in integrating the computation of eccentricities into microcode [11], an optimization technique to accelerate customization of CRP. For isoPHAST, we want to separate metric-independent preprocessing and metric customization (exploiting, e. g., CCH [14]). We also explore approaches that do not (explicitly) require a partition of the road network. Another direction of research is the speedup of network Voronoi diagram computation [21,34], where multiple isochrones are grown simultaneously from a set of Voronoi generators. We are also interested in extending our speedup techniques to more involved scenarios, such as multimodal networks.

References

1. Abraham, I., Delling, D., Fiat, A., Goldberg, A.V., Werneck, R.F.: HLDB: location-based services in databases. In: Proceedings of the 20th ACM SIGSPATIAL International Symposium on Advances in Geographic Information Systems (GIS 2012), pp. 339–348. ACM Press, New York (2012)
2. Bast, H., Delling, D., Goldberg, A.V., Müller-Hannemann, M., Pajor, T., Sanders, P., Wagner, D., Werneck, R.F.: Route Planning in Transportation Networks. Technical report abs/1504.05140, ArXiv e-prints (2015)
3. Bauer, V., Gamper, J., Loperfido, R., Profanter, S., Putzer, S., Timko, I.: Computing isochrones in multi-modal, schedule-based transport networks. In: Proceedings of the 16th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (GIS 2008), pp. 78:1–78:2. ACM Press, New York (2008)
4. Baum, M., Bläsius, T., Gemsa, A., Rutter, I., Wegner, F.: Scalable Isocontour Visualization in Road Networks via Minimum-Link Paths. Technical report abs/1602.01777, ArXiv e-prints (2016)
5. Delling, D., Goldberg, A.V., Nowatzyk, A., Werneck, R.F.: PHAST: hardware-accelerated shortest path trees. J. Parallel Distrib. Comput. **73**(7), 940–952 (2013)

6. Delling, D., Goldberg, A.V., Pajor, T., Werneck, R.F.: Customizable route planning. In: Pardalos, P.M., Rebennack, S. (eds.) SEA 2011. LNCS, vol. 6630, pp. 376–387. Springer, Heidelberg (2011)
7. Delling, D., Goldberg, A.V., Pajor, T., Werneck, R.F.: Customizable route planning in road networks. Transportation Science (2015)
8. Delling, D., Goldberg, A.V., Razenshteyn, I., Werneck, R.F.: Graph partitioning with natural cuts. In: Proceedings of the 25th International Parallel and Distributed Processing Symposium (IPDPS 2011), pp. 1135–1146. IEEE Computer Society (2011)
9. Delling, D., Goldberg, A.V., Werneck, R.F.: Faster batched shortest paths inroad networks. In: Proceedings of the 11th Workshop on Algorithmic Approachesfor Transportation Modeling, Optimization, and Systems (ATMOS 2011). OpenAccessSeries in Informatics, vol. 20, pp. 52–63. OASIcs (2011)
10. Delling, D., Holzer, M., Müller, K., Schulz, F., Wagner, D.: High-performance multi-level routing. In: The Shortest Path Problem: Ninth DIMACS Implementation Challenge, DIMACS Book, vol. 74, pp. 73–92. American Mathematical Society (2009)
11. Delling, D., Werneck, R.F.: Faster customization of road networks. In: Bonifaci, V., Demetrescu, C., Marchetti-Spaccamela, A. (eds.) SEA 2013. LNCS, vol. 7933, pp. 30–42. Springer, Heidelberg (2013)
12. Delling, D., Werneck, R.F.: Customizable point-of-interest queries in road networks. IEEE Trans. Knowl. Data Eng. **27**(3), 686–698 (2015)
13. Demetrescu, C., Goldberg, A.V., Johnson, D.S. (eds.): The Shortest Path Problem: Ninth DIMACS Implementation Challenge, DIMACS Book, vol. 74. American Mathematical Society (2009)
14. Dibbelt, J., Strasser, B., Wagner, D.: Customizable contraction hierarchies. In: Gudmundsson, J., Katajainen, J. (eds.) SEA 2014. LNCS, vol. 8504, pp. 271–282. Springer, Heidelberg (2014)
15. Dibbelt, J., Strasser, B., Wagner, D.: Customizable contraction hierarchies. ACM J. Exp. Algorithmics **21**(1), 108–122 (2016)
16. Dijkstra, E.W.: A note on two problems in connexion with graphs. Numer. Math. **1**(1), 269–271 (1959)
17. Efentakis, A., Grivas, N., Lamprianidis, G., Magenschab, G., Pfoser, D.: Isochrones, traffic and DEMOgraphics. In: Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (GIS 2013), pp. 548–551. ACM Press, New York (2013)
18. Efentakis, A., Pfoser, D.: GRASP. Extending graph separators for the single-source shortest-path problem. In: Schulz, A.S., Wagner, D. (eds.) ESA 2014. LNCS, vol. 8737, pp. 358–370. Springer, Heidelberg (2014)
19. Efentakis, A., Pfoser, D., Vassiliou, Y.: SALT. A unified framework for all shortest-path query variants on road networks. In: Bampis, E. (ed.) SEA 2015. LNCS, vol. 9125, pp. 298–311. Springer, Heidelberg (2015)
20. Efentakis, A., Theodorakis, D., Pfoser, D.: Crowdsourcing computing resources for shortest-path computation. In: Proceedings of the 20th ACM SIGSPATIAL International Symposium on Advances in Geographic Information Systems (GIS 2012), pp. 434–437. ACM Press, New York (2012)
21. Erwig, M.: The graph voronoi diagram with applications. Networks **36**(3), 156–163 (2000)
22. Foti, F., Waddell, P., Luxen, D.: A generalized computational framework for accessibility: from the pedestrian to the metropolitan scale. In: Proceedings of the

4th TRB Conference on Innovations in Travel Modeling. Transportation Research Board (2012)
23. Gamper, J., Böhlen, M., Cometti, W., Innerebner, M.: Defining isochrones in multimodal spatial networks. In: Proceedings of the 20th ACM International Conference on Information and Knowledge Management (CIKM 2011), pp. 2381–2384. ACM Press, New York (2011)
24. Gamper, J., Böhlen, M., Innerebner, M.: Scalable computation of isochrones with network expiration. In: Ailamaki, A., Bowers, S. (eds.) SSDBM 2012. LNCS, vol. 7338, pp. 526–543. Springer, Heidelberg (2012)
25. Geisberger, R.: Advanced Route Planning in Transportation Networks. Ph.D. thesis, Karlsruhe Institute of Technology (2011)
26. Geisberger, R., Luxen, D., Sanders, P., Neubauer, S., Volker, L.: Fast detour computation for ride sharing. In: Proceedings of the 10th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS 2010). OpenAccess Series in Informatics, vol. 14, pp. 88–99. OASIcs (2010)
27. Geisberger, R., Sanders, P., Schultes, D., Vetter, C.: Exact routing in large road networks using contraction hierarchies. Transp. Sci. **46**(3), 388–404 (2012)
28. Grubwinkler, S., Brunner, T., Lienkamp, M.: Range prediction for EVs via crowdsourcing. In: Proceedings of the 10th IEEE International Vehicle Power and Propulsion Conference (VPPC 2014), pp. 1–6. IEEE (2014)
29. Holzer, M., Schulz, F., Wagner, D.: Engineering multilevel overlay graphs for shortest-path queries. ACM J. Exp. Algorithmics **13**, 1–26 (2008)
30. Innerebner, M., Böhlen, M., Gamper, J.: ISOGA: a system for geographical reachability analysis. In: Liang, S.H.L., Wang, X., Claramunt, C. (eds.) W2GIS 2013. LNCS, vol. 7820, pp. 180–189. Springer, Heidelberg (2013)
31. Jung, S., Pramanik, S.: An efficient path computation model for hierarchically structured topographical road maps. IEEE Trans. Knowl. Data Eng. **14**(5), 1029–1046 (2002)
32. Knopp, S., Sanders, P., Schultes, D., Schulz, F., Wagner, D.: Computing many-to-many shortest paths using highway hierarchies. In: Proceedings of the 9th Workshop on Algorithm Engineering and Experiments (ALENEX 2007), pp. 36–45. SIAM (2007)
33. Marciuska, S., Gamper, J.: Determining objects within isochrones in spatial network databases. In: Catania, B., Ivanović, M., Thalheim, B. (eds.) ADBIS 2010. LNCS, vol. 6295, pp. 392–405. Springer, Heidelberg (2010)
34. Okabe, A., Satoh, T., Furuta, T., Suzuki, A., Okano, K.: Generalized network voronoi diagrams: concepts, computational methods, and applications. Int. J. Geogr. Inf. Sci. **22**(9), 965–994 (2008)
35. O'Sullivan, D., Morrison, A., Shearer, J.: Using desktop GIS for the investigation of accessibility by public transport: an isochrone approach. Int. J. Geogr. Inf. Sci. **14**(1), 85–104 (2000)
36. Pothen, A., Simon, H.D., Liou, K.P.: Partitioning sparse matrices with eigenvectors of graphs. SIAM J. Matrix Anal. Appl. **11**, 430–452 (1990)
37. Sanders, P., Schulz, C.: Distributed evolutionary graph partitioning. In: Proceedings of the 14th Meeting on Algorithm Engineering and Experiments (ALENEX 2012), pp. 16–29. SIAM (2012)
38. Schulz, C.: High Quality Graph Partitioning. Ph.D. thesis, Karlsruhe Institute of Technology (2013)

http://www.springer.com/978-3-319-38850-2

Experimental Algorithms
15th International Symposium, SEA 2016, St. Petersburg, Russia, June 5-8, 2016, Proceedings
Goldberg, A.V.; Kulikov, A.S. (Eds.)
2016, XVI, 386 p. 96 illus., Softcover
ISBN: 978-3-319-38850-2