Customizable Contraction Hierarchies

JULIAN DIBBELT, BEN STRASSER, and DOROTHEA WAGNER,
Karlsruhe Institute of Technology

We consider the problem of quickly computing shortest paths in weighted graphs. Often, this is achieved in two phases: (1) derive auxiliary data in an expensive preprocessing phase, and (2) use this auxiliary data to speed up the query phase. By adding a fast weight-customization phase, we extend Contraction Hierarchies to support a three-phase workflow. The expensive preprocessing is split into a phase exploiting solely the unweighted topology of the graph and a lightweight phase that adapts the auxiliary data to a specific weight. We achieve this by basing our Customizable Contraction Hierarchies (CCHs) on nested dissection orders. We provide an in-depth experimental analysis on large road and game maps showing that CCHs are a very practicable solution in scenarios where edge weights often change.

CCS Concepts: • **Theory of computation** → **Shortest paths**; **Dynamic graph algorithms**; • **Mathematics of computing** → **Graph algorithms**; **Paths and connectivity problems**; • **Theory of computation** → *Shared memory algorithms*

Additional Key Words and Phrases: Route planning, speedup technique, distance oracle, contraction

ACM Reference Format:
Julian Dibbelt, Ben Strasser, and Dorothea Wagner. 2016. Customizable contraction hierarchies. ACM J. Exp. Algor. 21, 1, Article 1.5 (April 2016), 49 pages.
DOI: http://dx.doi.org/10.1145/2886843

1. INTRODUCTION

Computing optimal routes in road networks has many applications, such as navigation, logistics, traffic simulation, or Web-based route planning. Road networks are commonly formalized as weighted graphs, and the optimal route is formalized as the shortest path in this graph. Unfortunately, road graphs tend to be huge in practice, with vertex counts in the tens of millions, rendering Dijkstra's algorithm [Dijkstra 1959] impracticable for interactive use: it incurs running times in the order of seconds even for a single path query. For practical performance on large road networks, preprocessing techniques that augment the network with auxiliary data in a (possibly expensive) offline phase have proven useful. Bast et al. [2015] provides an overview. Many techniques work by adding extra edges called *shortcuts* to the graph that allow query algorithms to bypass large regions of the graph efficiently. Although variants of the optimal shortcut selection problem have been proven to be NP-hard [Bauer et al. 2012], determining good shortcuts is feasible in practice even on large road graphs. Among the most successful speedup techniques using this building block are Contraction Hierarchies (CHs) by Geisberger et al. [2008, 2012]. At its core, the technique consists of a systematic way of

Partial support was provided by DFG grant WA654/16-2 and EU grant 288094 (eCOMPASS) and the Google Focused Research Award.
Authors' addresses: J. Dibbelt, B. Strasser, and D. Wagner; emails: {Julian.Dibbelt, Ben.Strasser, Dorothea.Wagner}@kit.edu.

DOI: http://dx.doi.org/10.1145/2886843

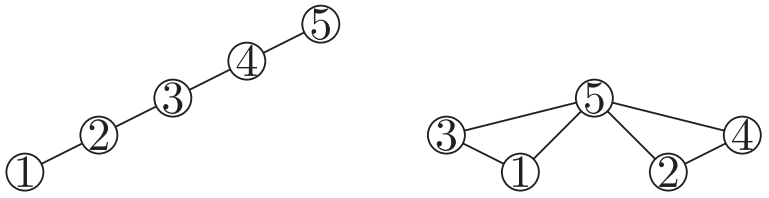
adding shortcuts by iteratively contracting vertices along a given order. Even though ordering heuristics exist that work well in practice [Geisberger et al. 2012], the problem of computing an optimal ordering is NP-hard in general [Bauer et al. 2010]. Worst-case bounds have been proven in Abraham et al. [2013a] in terms of a weight-dependent graph measure called *highway dimension*, and Milosavljević [2012] have shown that many of these bounds are tight on many graph classes.

A central restriction of Contraction Hierarchies as proposed by Geisberger et al. [2012] is that their preprocessing is *metric dependent*—that is, edge weights, also called *metric*, need to be known. Substantial changes to the metric, e.g., due to user preferences or traffic congestion, may require expensive recomputation. For this reason, a Customizable Route Planning (CRP) approach was proposed in Delling et al. [2011a], extending the multilevel-overlay MLD techniques of Schulz et al. [2000] and Holzer et al. [2008]. It works in three phases. In a first and expensive phase, auxiliary data is computed that solely exploits the topological structure of the network, disregarding its metric. In a second and much less expensive phase, this auxiliary data is *customized* to a specific metric, enabling fast queries in the third phase. In this work, we extend Contraction Hierarchy to support such a three-phase approach.

Game scenario. Most existing Contraction Hierarchy papers focus solely on road graphs, with Storandt [2013] being a notable exception, but there are many other applications with differently structured graphs in which fast shortest path computations are important. One of such applications is games. Think of a real-time strategy game where units quickly have to navigate across a large map with many choke points. The basic topology of the map is fixed; however, when buildings are constructed or destroyed, fields are rendered impassable or freed up. Furthermore, every player has his own knowledge of the map. Many games include a feature called *fog of war*. Initially, only the fields around the player's starting location are revealed. As his units explore the map, new fields are revealed. Since a unit must not route around a building that the player has not yet seen, every player needs his own metric. Furthermore, units such as hovercrafts may traverse water and land, whereas other units are bound to land. This results in vastly different, evolving metrics for different unit types per player, making metric-dependent preprocessing difficult to apply. Contrary to road graphs, one-way streets tend to be extremely rare; thus, being able to exploit the symmetry of the underlying graph is a useful feature.

Metric-independent orders for Contraction Hierarchies. One of the central building blocks of this article is the use of metric-independent *nested dissection orders* (ND-orders) [George 1973] for Contraction Hierarchy precomputation instead of the metric-dependent order of Geisberger et al. [2012]. This approach was proposed by Bauer et al. [2013], and a preliminary case study can be found in Zeitz [2013]. A similar idea was followed by Delling and Werneck [2013], where the authors employ partial Contraction Hierarchies to engineer subroutines of their customization phase. They also refer to preliminary experiments on full Contraction Hierarchy but did not publish results. Similar ideas have also appeared in Planken et al. [2012]. They consider graphs of low *treewidth* (see later discussion) and leverage this property to compute Contraction Hierarchy-like structures, without explicitly using the term *Contraction Hierarchy*. Related techniques by Wei [2010] and Chaudhuri and Zaroliagis [2000] work directly on the tree decomposition. Interestingly, our experiments show that even large road networks have relatively low treewidth: real-world road networks with vertex counts in the 10^7 have treewidth in the 10^2.

Tree decompositions, sparse matrices, and minimum fill-in. Customizable speedup techniques for shortest path queries are a very recent development, but the idea to order vertices along ND-orders is significantly older. To the best of our knowledge, the



(a) No shortcuts, maximum search space is four arcs

(b) Two shortcuts, maximum search space is two arcs

Fig. 1. Contraction Hierarchies for a path graph. Vertices are numbered by contraction order.

idea first appeared in 1973 in George [1973] and was refined in Lipton et al. [1979]. They use ND-orders to reorder the columns and rows of *sparse matrices* to ensure that Gaussian elimination preserves as many zeros as possible. From the matrix, they derive a graph and show that vertex contraction in this graph corresponds to Gaussian variable elimination. Inserting an extra edge in the graph destroys a zero in the matrix. The additional edges are called the *fill-in*. The *minimum fill-in problem* asks for a vertex order that results in a minimum number of extra arcs. In Contraction Hierarchy terminology, these extra edges are called *shortcuts*. The super graph constructed by adding the additional edges is a *chordal graph*. The *treewidth* of a graph G can be defined using chordal supergraphs. For every supergraph, consider the number of vertices in the maximum clique minus one. The treewidth of a graph G is the minimum of this number over all chordal supergraphs of G. This establishes a relation between sparse matrices and treewidth and in consequence with Contraction Hierarchies. We refer to Bodlaender [1993, 2007] for an introduction to the broad field of treewidth and tree decompositions.

Minimizing the number of extra edges, i.e., minimizing the fill-in, is NP-hard [Yannakakis 1981] but fixed parameter tractable in the number of extra edges [Kaplan et al. 1999]. Note, however, that from the Contraction Hierarchy point of view, optimizing the number of extra edges, i.e., the number of shortcuts, is not the sole optimization criterion. For example, consider a path graph as depicted in Figure 1: optimizing the Contraction Hierarchy search space and the number of shortcuts are competing criteria. A tree relevant in the theory of treewidth is the *elimination tree*. Bauer et al. [2013] have shown that the maximum search space size in terms of vertices corresponds to the height of this elimination tree. Unfortunately, minimizing the elimination tree height is also NP-hard [Pothen 1988]. For planar graphs, it has been shown that the number of additional edges is in $O(n \log n)$ [Gilbert and Tarjan 1986]. However, this does not imply a $O(\log n)$ search space bound in terms of vertices, as search spaces can share vertices.

Directed and undirected graphs. Real-world road networks contain one-way streets and highways. Such networks are thus usually modeled as directed graphs. Our algorithms fully support direction of traffic—however, we introduce it at a different stage of the toolchain than most related techniques, which should not be confused with only supporting undirected networks. Our first preprocessing phase works exclusively on the underlying undirected and unweighted graph, obtained by dropping all edge directions and edge weights. Direction of traffic as well as traversal weights are only introduced in the second (customization) phase, where every edge can have two weights: an upward and a downward weight. If an edge corresponds to a one-way street, then one of these weights is set to ∞. Note that this setup is a strength of our algorithm: throughout large parts of the toolchain, we are not confronted with additional algorithmic complexity induced by directed edges.

Our contribution. The main contribution of our work is to show that Customizable Contraction Hierarchies (CCHs) solely based on the nested dissection principle are

feasible and practical. Compared to CRP [Delling et al. 2011a], we achieve a similar preprocessing–query trade-off, albeit with slightly better query performance at slightly slower customization speed and we need somewhat more space. Interestingly, for less well behaved metrics such as travel distance, we achieve query times below the original metric-dependent Contraction Hierarchy of Geisberger et al. [2012]. Besides this main result, we show that given a fixed contraction order, a metric-independent Contraction Hierarchy can be constructed in time essentially linear in the size of the Contraction Hierarchy with working memory consumption linear in the input graph. Our specialized algorithm has better theoretic worst-case running time and performs significantly better empirically than the dynamic adjacency arrays used in Geisberger et al. [2012]. Another contribution of our work is perfect witness searches. We show that for a fixed metric-independent vertex order, it is possible to construct Contraction Hierarchies with a provably minimum number of arcs in a few seconds on continental road graphs. Our construction algorithm has a running time performance completely independent of the weights used. We further show that an order based on nested dissection gives a constant factor approximation of the maximum and average search space sizes in terms of the number of arcs and vertices for metric-independent Contraction Hierarchies on a class of graphs with very regular recursive vertex separators. Experimentally, we show that road graphs have such a recursive separator structure.

Outline. Section 2 presents the necessary notation. Section 3 discusses metric-dependent orders as used by Geisberger et al. [2012], highlighting specifics of our implementation. Next, we discuss metric-independent orders in Section 4. In Section 5, we describe how to efficiently construct the arcs of the Contraction Hierarchy. Section 6 discusses how to efficiently enumerate triangles in the Contraction Hierarchy—an operation needed throughout the customization process detailed in Section 7. In Section 7, we further describe the details of the perfect witness search. Finally, Section 8 concludes the algorithm description by introducing the algorithms used in the query phase to compute shortest path distances and corresponding paths in the input graph. We then present an extensive experimental study of the proposed approach: Section 9 thoroughly evaluates ND-order computation, Contraction Hierarchy construction and resulting search space size, triangle enumeration, customization, query performance, and comparison with related work. Section 10 explores alternative metric-independent ordering strategies, whereas Section 11 presents summary experiments on further benchmark instances. In Section 12, we conclude and discuss directions for future work.

2. BASICS

We denote by $G = (V, E)$ an *undirected* n-vertex *graph*, where V is the set of *vertices* and E the set of *edges*. Furthermore, $G = (V, A)$ denotes a *directed graph*, where A is the set of *arcs*. A graph is *simple* if it has no loops or multiedges. Graphs in this article are simple unless noted otherwise, e.g., in parts of Section 5. Furthermore, we assume that input graphs are strongly connected. We denote by $N(v)$ the neighborhood of vertex $v \in G$, i.e., the set of vertices adjacent to v; for directed graphs, the neighborhood ignores arc direction. A *vertex separator* is a vertex subset $S \subseteq V$ whose removal separates G into two disconnected subgraphs induced by the vertex sets A and B. The sets S, A, and B are disjoint, and their union forms V. Note that the subgraphs induced by A and B are not necessarily connected and may be empty. A separator S is *balanced* if $\max\{|A|, |B|\} \leq 2n/3$.

A *vertex order* $\pi : \{1 \dots n\} \to V$ is a bijection. Its inverse π^{-1} assigns each vertex a *rank*. Every undirected graph can be transformed into a *upward directed graph* with respect to a vertex order π, i.e., every edge $\{\pi(i), \pi(j)\}$ with $i < j$ is replaced by an

arc $(\pi(i), \pi(j))$. Note that all upward directed graphs are acyclic. We denote by $N_u(v)$ the *upward neighborhood* of v, i.e., the neighbors of v with a higher rank than v, and by $N_d(v)$ the *downward neighborhood* of v, i.e., the vertices with a lower rank than v. We denote by $d_u(v) = |N_u(v)|$ the *upward degree* and by $d_d(v) = |N_d(v)|$ the *downward degree* of a vertex.

Undirected edge weights are denoted using $w : E \to \mathbb{R}_{>0}$. With respect to a vertex order π, we define an *upward weight* $w_u : E \to \mathbb{R}_{>0}$ and a *downward weight* $w_d : E \to \mathbb{R}_{>0}$. For directed graphs, one-way streets are modeled by setting w_u or w_d to ∞.

A path P is a sequence of adjacent vertices and incident edges. Its *hop length* is the number of edges in P. Its *weight length* with respect to w is the sum over all edges' weights. Unless noted otherwise, length always refers to weight length in this article. A shortest st-path is a path of minimum length between vertices s and t. The minimum length in G between two vertices is denoted by $\text{dist}_G(s, t)$. We set $\text{dist}_G(s, t) = \infty$ if no path exists. An *up-down path* P with respect to π is a path that can be split into an upward path P_u and a downward path P_d. The vertices in the upward path P_u must occur by increasing rank π^{-1}, and the vertices in the downward path P_d must occur by decreasing rank π^{-1}. The upward and downward paths meet at the vertex with the maximum rank on the path. We call this vertex the *meeting vertex*.

The vertices of every directed acyclic graph (DAG) can be partitioned into *levels* $\ell : V \to \mathbb{N}$ such that for every arc (x, y), it holds that $\ell(x) < \ell(y)$. We only consider levels such that each vertex has the lowest possible level. Note that such levels can be computed in linear time given a DAG.

The unweighted *vertex contraction* of v in G consists of removing v and all incident edges and inserting edges between all neighbors $N(v)$ if not already present. The inserted edges are called *shortcuts*, and the other edges are *original edges*. Given an order π, the *core graph* $G_{\pi,i}$ is obtained by contracting all vertices $\pi(1) \dots \pi(i-1)$ in order of their rank. We call the original graph G augmented by the set of shortcuts a *contraction hierarchy* $G^*_\pi = \bigcup_i G_{\pi,i}$. Furthermore, we denote by $G^\wedge_\pi$ the corresponding upward directed graph.

Given a fixed weight w, one can exploit that in many applications, it is sufficient to only preserve all shortest path distances [Geisberger et al. 2012]. *Weighted vertex contraction* of a vertex v in the graph G is defined as the operation of removing v and inserting (a minimum number) of shortcuts among the neighbors of v to obtain a graph G' such that $\text{dist}_G(x, y) = \text{dist}_{G'}(x, y)$ for all vertices $x \neq v$ and $y \neq v$. To compute G', one iterates over all pairs of neighbors x, y of v increasing by $\text{dist}_G(x, y)$. For each pair, one checks whether an xy-path of length $\text{dist}_G(x, y)$ exists in $G \backslash \{v\}$, i.e., one checks whether removing v destroys the xy-shortest path. This check is called *witness search* [Geisberger et al. 2012], and the xy-path is called *witness*, if it exists. If a witness is found, the considered vertex pair is skipped and no shortcut added. Otherwise, if an edge $\{x, y\}$ already exists, its weight is decreased to $\text{dist}_G(x, y)$, or a new shortcut edge with that weight is added to G. This new shortcut edge is considered in witness searches for subsequent neighbor pairs as part of G. If shortest paths are not unique, it is important to iterate over the pairs increasing by $\text{dist}_G(x, y)$, because otherwise more edges than strictly necessary can be inserted: shorter shortcuts can make longer shortcuts superfluous. However, if we insert the shorter shortcut after the longer ones, the witness search will not consider them. Figure 2 presents an example. (This effect was independently observed by Rice and Tsotras [2010] in a different setting.) Note that the witness searches are expensive, and therefore the witness search is usually aborted after a certain number of steps [Geisberger et al. 2012]. If no witness was found, we assume that none exists and add a shortcut. This does not affect the correctness of the technique but might result in slightly more shortcuts than necessary. To distinguish, *perfect witness search* is without such a one-sided error.

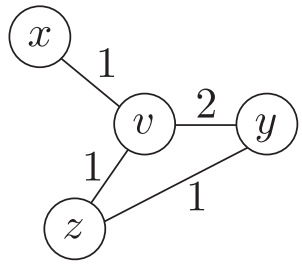


Fig. 2. Contraction of v. If the pair x, y is considered first, a shortcut $\{x, y\}$ with weight 3 is inserted. If the pair x, z is considered first, an edge $\{x, z\}$ with weight 2 is inserted. This shortcut is part of a witness $x \to z \to y$ for the pair x, y. The shortcut $\{x, y\}$ is thus *not* added if the pair x, z is considered first.

For an order π and a weight w, the *weighted core graph* $G_{w,\pi,i}$ is obtained by contracting all vertices $\pi(1) \dots \pi(i-1)$. The original graph G augmented by the set of weighted shortcuts is called a *weighted contraction hierarchy* $G^*_{w,\pi}$. The corresponding upward directed graph is denoted by $G^{\wedge}_{w,\pi}$.

The search space $\mathrm{SS}(v)$ of a vertex v is the subgraph of $G^{\wedge}_{\pi}$ (respectively, $G^{\wedge}_{w,\pi}$) reachable from v. For every vertex pair s and t, it has been shown that a shortest up-down path must exist. This up-down path can be found by running a bidirectional search from s restricted to $\mathrm{SS}(s)$ and from t restricted to $\mathrm{SS}(t)$ [Geisberger et al. 2012]. A graph is *chordal* if for every cycle of at least four vertices there exists a pair of vertices that are nonadjacent in the cycle but are connected by an edge. An alternative characterization is that a vertex order π exists such that for every i the neighbors of $\pi(i)$ in $G_{\pi,i}$, i.e., the core graph before the contraction of $\pi(i)$, form a clique [Fulkerson and Gross 1965]. Such an order is called a *perfect elimination order*. Another way to formulate this characterization in Contraction Hierarchy terminology is as follows. A graph is chordal if and only if a contraction order exists such that the Contraction Hierarchy construction without witness search does not insert any shortcuts. A chordal supergraph can be obtained by adding the Contraction Hierarchy shortcuts.

The elimination tree $T_{G,\pi}$ is a tree directed toward its root $\pi(n)$. The parent of vertex $\pi(i)$ is its upward neighbor $v \in N_u(\pi(i))$ of minimal rank $\pi^{-1}(v)$. Note that this definition already yields a straightforward algorithm for constructing the elimination tree. As shown in Bauer et al. [2013], the set of vertices on the path from v to $\pi(n)$ is the set of vertices in $\mathrm{SS}(v)$. Computing a Contraction Hierarchy without witness search of graph G consists of computing a chordal supergraph G^*_{π} with perfect elimination order π. The height of the elimination tree corresponds to the maximum number of vertices in the search space. Note that the elimination tree is only defined for undirected unweighted graphs.

3. METRIC-DEPENDENT ORDERS

Most publications on applications and extensions of Contraction Hierarchy use greedy orders in the spirit of Geisberger et al. [2012], but details of vertex order computation and witness search vary. For reproducibility, we describe our precise approach in this section, extending on the general description of metric-dependent Contraction Hierarchy preprocessing given in Section 2. Our witness search aborts once it finds some path shorter than the shortcut—or when both forward and backward search each have settled at most p vertices. For most experiments, we choose $p = 50$. The only exception is the distance metric on road graphs, where we set $p = 1500$. We found that a higher value of p increases the time per witness search but leads to sparser cores. For the distance metric, we needed a high value because otherwise our cores become too dense.

This effect did not occur for the other weights considered in the experiments. Our weighting heuristic is similar to the one of Abraham et al. [2012b]. We denote by $L(x)$ a value that approximates the level of vertex x. Initially, all $L(x)$ are 0. If x is contracted, then for every incident edge $\{x, y\}$, we perform $\ell(y) \leftarrow \max\{\ell(y), \ell(x) + 1\}$. We further store for every arc a a hop length $h(a)$. This is the number of arcs that the shortcut represents if fully unpacked. Denote by $D(x)$ the set of arcs removed if x is contracted and by $A(x)$ the set of arcs that are inserted. Note that $A(x)$ is not necessarily a full clique because of the witness search and because some edges may already exist. We greedily contract a vertex x that minimizes its *importance* $I(x)$, defined by

$$I(x) = L(x) + \frac{|A(x)|}{|D(x)|} + \frac{\sum_{a \in A(x)} h(a)}{\sum_{a \in D(x)} h(a)}.$$

We maintain a priority queue that contains all vertices weighted by I. Initially, all vertices are inserted with their exact importance. As long as the queue is not empty, we remove a vertex x with minimum importance $I(x)$ and contract it. This modifies the importance of other vertices. However, our weighting function is chosen such that only the importance of adjacent vertices is influenced, if the witness search was perfect. We therefore only update the importance values of all vertices y in the queue that are adjacent to x. In practice, with a limited witness search, we sometimes choose a vertex x with a sightly suboptimal $I(x)$. However, preliminary experiments have shown that this effect can be safely ignored. Hence, for the experiments presented in Section 9, we do not use lazy updates or periodic queue rebuilding as proposed in Geisberger et al. [2012].

4. METRIC-INDEPENDENT ORDER

The metric-dependent orders presented in the previous section lead to very good results on road graphs with a travel time metric. However, the results for the distance metric are not as good and the orders are completely impracticable to compute Contraction Hierarchy without witness search, as our experiments in Section 9 show. To support metric independence, we therefore use ND-orders as suggested in Bauer et al. [2013]. An order π for G is computed recursively by determining a balanced separator S of minimum cardinality that splits G into two parts induced by the vertex sets A and B. The vertices of S are assigned to $\pi(n - |S| + 1) \dots \pi(n)$ in an arbitrary order. Orders π_A and π_B are computed recursively and assigned to $\pi(1) \dots \pi(|A|)$ and $\pi(|A|+1) \dots \pi(|A|+|B|)$, respectively. The base case of the recursion is reached when the subgraphs are empty. Computing ND-orders requires good graph bisectors, which in theory is *NP*-hard. However, recent years have seen heuristics that solve the problem very well even for continental road graphs [Sanders and Schulz 2013; Delling et al. 2011b, 2012]. This justifies assuming in our particular context that an efficient bisection oracle exists. We experimentally examine the performance of ND-orders computed by NDMetis [Karypis and Kumar 1999] and KaHIP [Sanders and Schulz 2013] in Section 9. After having obtained the ND-order, we reorder the in-memory vertex IDs of the input graph accordingly, i.e., the contraction order of the reordered graph is the identity function. This improves cache locality, and we have seen a resulting acceleration of a factor 2 to 3 in query times. In the remainder of this section, we prepare and provide a theoretical approximation result.

For $\alpha \in (0, 1)$, let K_α be a class of graphs that is closed under subgraph construction and admits balanced separators S of cardinality $O(n^\alpha)$.

LEMMA 4.1. *For every $G \in K_\alpha$ an ND-order results in $O(n^\alpha)$ vertices in the maximum search space.*

The proof of this lemma is a straightforward argument using a geometric series as described in Bauer et al. [2013]. As a direct consequence, the average number of vertices is also in $O(n^\alpha)$ and the number of arcs in $O(n^{2\alpha})$.

LEMMA 4.2. *For every connected graph G with minimum balanced separator S and every order π, the chordal supergraph G^*_π contains a clique of $|S|$ vertices. Furthermore, there are at least $n/3$ vertices such that this clique is a subgraph of their search space in $G^\wedge_\pi$.*

This lemma is a minor adaptation and extension of Lipton et al. [1979], who only prove that such a clique exists but not that it lies within enough search spaces. We provide the full proof for self-containedness.

PROOF. Consider the subgraph G_i of G^*_π induced by the vertices $\pi(1)\dots\pi(i)$. Do not confuse with the core graph $G_{\pi,i}$. Choose the smallest i, such that a connected component A exists in G_i such that $|A| \geq n/3$. As G is connected, such an A must exist. We distinguish two cases:

(1) $|A| \leq 2n/3$: Consider the set of vertices S' adjacent to A in G^*_π but not in A. Let B be the set of all remaining vertices. S' is by definition a separator. It is balanced because $|A| \leq 2n/3$ and $|B| = n - \underbrace{|A|}_{\geq n/3} - \underbrace{|S'|}_{\geq 0} \leq 2n/3$. As S is minimum, we have that $|S'| \geq |S|$. For every pair of vertices $u \in S'$ and $v \in S'$, there exists a path through A as A is connected. The vertices u and v are not in G_i, as otherwise they could be added to A. The ranks of u and v are thus strictly larger than i. On the other hand, the ranks of the vertices in A are at most i, as they are part of G_i. The vertices u and v thus have the highest ranks on the path. They are therefore contracted last, and therefore an edge $\{u, v\}$ in G^* must exist. S' is therefore a clique. Furthermore, from every $u \in A$ to every $v \in S'$, there exists a path such that v has the highest rank. Hence, v is in the search space of u, i.e., there are at least $|A| \geq n/3$ vertices whose search space contains the full S'-clique.

(2) $|A| > 2n/3$: As i is minimum, we know that $\pi(i) \in A$ and that removing it disconnects A into connected subgraphs $C_1 \dots C_k$. We know that $|C_j| < n/3$ for all j because i is minimum. We further know that $|A| = 1 + \sum |C_j| > 2n/3$. We can therefore select a subset of components C_k such that the number of their vertices is at most $2n/3$ but at least $n/3$. Denote by A' their union. Note that A' does not contain $\pi(i)$. Consider the vertices S' adjacent to A' in G^*_π. The set S' contains $\pi(i)$. Using an argument similar to Case 1, one can show that $|S'| \geq |S|$. But since A' is not connected, we cannot directly use the same argument to show that S' forms a clique in G^*. Observe that $A' \cup \{\pi(i)\}$ is connected, and thus the argument can be applied to $S' \backslash \{\pi(i)\}$, showing that it forms a clique. This clique can be enlarged by adding $\pi(i)$, as for every $v \in S' \backslash \{\pi(i)\}$, a path through one of the components C_k exists where v and $\pi(i)$ have the highest ranks, and thus an edge $\{v, \pi(i)\}$ must exist. The vertex set S' therefore forms a clique of at least the required size. It remains to show that enough vertices exist whose search space contains the S' clique. As $\pi(i)$ has the lowest rank in the S' clique, the whole clique is contained within the search space of $\pi(i)$. It is thus sufficient to show that $\pi(i)$ is contained in enough search spaces. As $\pi(i)$ is adjacent to each component C_k, a path from each vertex $v \in A'$ to $\pi(i)$ exists such that $\pi(i)$ has maximum rank showing that S' is contained in the search space of v. This completes the proof as $|A'| \geq n/3$. □

THEOREM 4.3. *Let G be a graph from K_α with a minimum balanced separator with $\Theta(n^\alpha)$ vertices. Then an ND-order gives an $O(1)$-approximation of the average and*

maximum search spaces of an optimal metric-independent Contraction Hierarchy in terms of vertices and arcs.

PROOF. The key observation of this proof is that the top-level separator solely dominates the performance. Denote by π_{nd} the ND-order and by π_{opt} an optimal order. First, we show a lower bound on the performance of π_{opt}. We then demonstrate that π_{nd} achieves this lower bound showing that π_{nd} is an $O(1)$-approximation.

As the minimum balanced separator has cardinality $\Theta(n^\alpha)$, we know by Lemma 4.2 that a clique with $\Theta(n^\alpha)$ vertices exists in $G^*_{\pi_{opt}}$. As this clique is in the search space of at least one vertex with respect to π_{opt}, we know that the maximum number of vertices in the search space is at least $\Omega(n^\alpha)$. Further, as this clique contains $\Theta(n^{2\alpha})$ arcs, we also have a lower bound of $\Omega(n^{2\alpha})$ on the maximum number of arcs in a search space. From these bounds for the worst-case search space, we cannot directly derive bounds for the average search space. Fortunately, Lemma 4.2 does not only tell us that this clique exists but that it must also be inside the search space of at least $n/3$ vertices. For the remaining $2n/3$ vertices, we use a very pessimistic lower bound: we assume that their search space is empty. The resulting lower bound for the average number of vertices is $2/3 \cdot \Omega(0) + 1/3 \cdot \Omega(n^\alpha) = \Omega(n^\alpha)$, and the lower bound for the average number of arcs is $2/3 \cdot \Omega(0) + 1/3 \cdot \Omega(n^{2\alpha}) = \Omega(n^{2\alpha})$.

We required that $G \in K_\alpha$, i.e., that recursive $O(n^\alpha)$ balanced separators exist. This allows us to apply Lemma 4.1. We therefore know that the number of vertices in the maximum search space of $G^\wedge_{\pi_{nd}}$ is in $O(n^\alpha)$. In the worst case, this search space contains $O(n^{2\alpha})$ arcs. As the average case can never be better than the worst case, these upper bounds directly translate to upper bounds for the average search space.

As the given upper and lower bounds match, we can conclude that π_{nd} is a $O(1)$-approximation in terms of average and maximum search space in terms of vertices and arcs. □

5. CONSTRUCTING THE CONTRACTION HIERARCHY

In this section, we describe how to efficiently compute the hierarchy $G^\wedge_\pi$ for a given graph G and order π. Weighted Contraction Hierarchies are commonly constructed using a dynamic adjacency array representation of the core graph. Our experiments show that this approach also works for the unweighted case, however, requiring more computational and memory resources because of the higher growth in shortcuts. It has been proposed by Zeitz [2013] to use hash tables on top of the dynamic graph structure to improve speed but at the cost of significantly increased memory consumption. In this section, we show that the Contraction Hierarchy construction can be done significantly faster on unweighted and undirected graphs. Note that in our toolchain, graph weights and arc directions are accounted for during the customization phase.

Denote by n the number of vertices in G (and $G^\wedge_\pi$), by m the number of edges in G, by $\hat{m}$ the number of arcs in $G^\wedge_\pi$, and by $\alpha(n)$ the inverse $A(n, n)$ Ackermann function. For simplicity, we assume that G is connected. Our approach enumerates all arcs of $G^\wedge_\pi$ in $O(\hat{m}\alpha(n))$ running time and has a memory consumption in $O(m)$. To store the arcs of $G^\wedge_\pi$, additional space in $O(\hat{m})$ is needed. The approach is heavily based upon the method of the quotient graph [George and Liu 1978]. To the best of our knowledge, it has not yet been applied in the context of route planning, and there exists no complexity analysis for the specific variant employed by us. Therefore, we discuss both the approach and present a running time analysis in the remainder of the section.

Recall that to compute the Contraction Hierarchy $G^\wedge_\pi$ from a given input graph G and order π, one iteratively contracts each vertex, adding shortcuts between its neighbors. Let $G' = G_{\pi,i}$ be the core graph in iteration i. We do not store G' explicitly but employ a

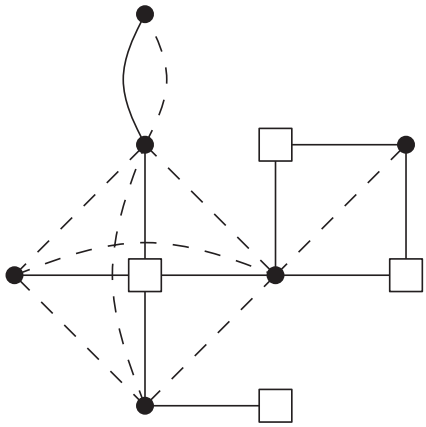


Fig. 3. Dots represent vertices in G' and H. Squares are additional supervertices in H. Solid edges are in H and dashed ones in G'. Notice how the neighbors of each supervertex in H forms a clique in G'. Furthermore, there are no two adjacent supervertices in H, i.e., they form an independent set.

special data structure called the *contraction graph* for efficient contraction and neighborhood enumeration. The contraction graph H contains both yet uncontracted core vertices and an independent set of virtually contracted *supervertices* (Figure 3 provides an illustration). These supervertices enable us to avoid the overhead of dynamically adding shortcuts to G'. For each vertex in H, we store a marker bit indicating whether it is a supervertex. Note that G' can be obtained by contracting all supervertices in H.

5.1. Contracting Vertices

A vertex x in G' is contracted by turning it into a supervertex. However, creating new supervertices can violate the independent set property. We restore it by merging neighboring supervertices: denote by y a supervertex that is a neighbor of x. We rewire all edges incident to y to be incident to x and remove y from H. To support efficiently merging vertices in H, we store a linked list of neighbors for each vertex. When merging two vertices, we link these lists together. Unfortunately, combining these lists is not enough, as the former neighbors z of y still have y in their list of neighbors. We therefore further maintain a union-find data structure: initially all vertices are within their own set. When merging x and y, the sets of x and y are united. We chose x as representative, as y was deleted.[1] When z enumerates its neighbors, it finds a reference to y. It can then use the union-find data structure to determine that the representative of y's set is x. The reference in z's list is thus interpreted as pointing to x.

It is possible that merging vertices can create multiedges and loops. For example, consider that the neighborhood list of y contains x. After merging, the united list of x will therefore contain a reference to x. Similarly, it will contain a reference to y, which after looking up the representative is actually x. Two loops are thus created at x per merge. Furthermore, consider a vertex z that is a neighbor of both y and x. In this case, the neighborhood list of x will contain two references to z. These multiedges and loops need to be removed. We do this lazily and remove them in the neighborhood enumeration instead of removing them in the merge operation.

[1]Or alternatively, we can let the union-find data structure choose the new representative. We then denote by x the new representative and by y the other vertex. In this variant, it is possible that the new x is the old y, which can be confusing. For this reason, we describe the simpler variant, where x is always chosen as representative and thus x always refers to the same vertex.

5.2. Enumerating Neighbors

Suppose that we want to enumerate the neighbors of a vertex x in H. Note that x's neighborhood in H differs from its neighborhood in G'. The neighborhood of x in H can contain supervertices, as supervertices are only contracted in G'. We maintain a Boolean marker that indicates which neighbors have already been enumerated. Initially, no marker is set. We iterate over x's neighborhood list. For each reference, we lookup the representative v. If v was already marked or is x, we remove the reference from the list. If v was not marked and is not x, we mark it and report it as a neighbor. After the enumeration, we reset all markers by enumerating the neighbors again.

However, during the execution of our algorithm, we are not interested in the neighborhood of x in H, but we want the neighborhood of x in G', i.e., the algorithm should not list supervertices. Our algorithm conceptually first enumerates the neighborhood of x and then contracts x. We actually do this in reversed order. We first contract x. After the contraction x is a supervertex. Because of the independent set property, we know that x has no supervertex neighbors in H. We can thus enumerate x's neighbors in H and exploit that in this particular situation the neighborhoods of x in G' and H coincide.

5.3. Performance Analysis

As there are no memory allocations, it is clear that the working space memory consumption is in $O(m)$. Proving a running time in $O(\hat{m}\alpha(n))$ is less clear. Denote by $d(x)$ the degree of x just before x is contracted. $d(x)$ coincides with the upward degree of x in $G^{\wedge}_{\pi}$, and thus $\sum d(x) = \hat{m}$. We first prove that we can account for the neighborhood cleanup operations outside of the actual algorithm. This allows us to assume that they are free within the main analysis. We then show that contracting a vertex x and enumerating its neighbors is in $O(d(x)\alpha(n))$. Processing all vertices thus has a running time in $O(\hat{m}\alpha(n))$.

The neighborhood list of x can contain duplicated references, and thus its length can be larger than the number of neighbors of x. Further, for each entry in the list, we need to perform a union find lookup. The costs of a neighborhood enumeration can therefore be larger than $O(d(x)\alpha(n))$. Fortunately, the first neighborhood enumeration compactifies the neighborhood list, and thus every subsequent enumeration runs in $O(d(x)\alpha(n))$. Removing a reference has a cost in $O(\alpha(n))$. Our algorithm never adds references. Initially, there are $\Theta(m)$ references. The total costs for removing references over the whole algorithm are therefore in $O(m\alpha(n))$. As our graph is assumed to be connected, we have that $m \in O(m')$ and thus $O(m\alpha(n)) \subseteq O(\hat{m}\alpha(n))$. We can therefore assume that removing references is free within the algorithm. As removing a reference is free, we can assume that even the first enumeration of the neighbors of x is within $O(d(x)\alpha(n))$. Merging two vertices consists of redirecting a constant number of references within a linked list. The merge operation is thus in $O(1)$.

Our algorithm starts by enumerating all neighbors of x to determine all neighboring supervertices in $O(d(x)\alpha(n))$ time. There are at most $d(x)$ neighboring supervertices, and therefore the costs of merging all supervertices into x is in $O(d(x))$. We subsequently enumerate all neighbors a second time to output the arcs of $G^{\wedge}_{\pi}$. The costs of this second enumeration is also within $O(d(x)\alpha(n))$. The whole algorithm thus runs in $O(\hat{m}\alpha(n))$ time as $\sum d(x) = \hat{m}$, which completes the proof.

5.4. Adjacency Array

Although the described algorithm is efficient in theory, linked lists cause too many cache misses in practice. We therefore implemented a hybrid of a linked list and an adjacency array, which has the same worst-case performance but is more cache-friendly

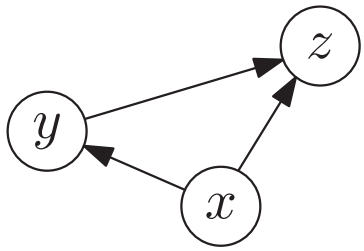


Fig. 4. A triangle in $G^{\wedge}_{\pi}$. The triple $\{x, y, z\}$ is a lower triangle of the arc (y, z), an intermediate triangle of the arc (x, z), and an upper triangle of the arc (x, y).

in practice. An element in the linked list not only holds a single reference but also holds a small set of references organized as small arrays called *blocks*. The neighbors of every original vertex form a single block. The initial linked neighborhood list are therefore composed of a single block. We merge two vertices by linking their blocks together. If all references are deleted from a block, we remove it from the list.

6. ENUMERATING TRIANGLES

A triangle $\{x, y, z\}$ is a set of three adjacent vertices. A triangle can be an upper, intermediate, or lower triangle with respect to an arc (x, y), as illustrated in Figure 4. A triangle $\{x, y, z\}$ is a *lower triangle* of (y, z) if x has the lowest rank among the three vertices. Similarly, $\{x, y, z\}$ is a *upper triangle* of (x, y) if z has the highest rank and $\{x, y, z\}$ is a *intermediate triangle* of (x, z) if y's rank is between the ranks of x and z. The triangles of an edge (a, b) can be characterized using the upward N_u and downward N_d neighborhoods of a and b. There is a lower triangle $\{a, b, c\}$ of an arc (a, b) if and only if $c \in N_d(a) \cap N_d(b)$. Similarly, there is an intermediate triangle $\{a, b, c\}$ of an arc (a, b) with $\pi^{-1}(a) < \pi^{-1}(b)$ if and only if $c \in N_u(a) \cap N_d(b)$ and an upper triangle $\{a, b, c\}$ of an arc (a, b) if and only if $c \in N_u(a) \cap N_u(b)$. The triangles of an arc can thus be enumerated by intersecting the neighborhoods of the arc's endpoints.

Efficiently enumerating all lower triangles of an arc is an important base operation of the customization (Section 7) and path unpacking algorithms (Section 8). It can be implemented using adjacency arrays or accelerated using extra preprocessing. Note that in addition to the vertices of a triangle, we are interested in the IDs of the participating arcs, as we need these to access the metric of an arc.

Basic triangle enumeration. Triangles can be efficiently enumerated by exploiting their characterization using neighborhood intersections. We construct an upward and a downward adjacency array for $G^{\wedge}_{\pi}$, where incident arcs are ordered by their head and respectively tail vertex ID. The lower triangles of an arc (x, y) can be enumerated by simultaneously scanning the downward neighborhoods of x and y to determine their intersection. Intermediate and upper triangles are enumerated analogously using the upward adjacency arrays. For later access to the metric of an arc, we also store each arc's ID in the adjacency arrays. This approach requires space proportional to the number of arcs in $G^{\wedge}_{\pi}$.

Triangle preprocessing. Instead of merging the neighborhoods on demand to find all lower triangles, we propose to create a *triangle adjacency array* structure that maps the arc ID of (x, y) to the set of pairs of arc IDs of (z, x) and (z, y) for every lower triangle $\{x, y, z\}$ of (x, y). This requires space proportional to the number of triangles t in $G^{\wedge}_{\pi}$ but allows for a very fast access. Analogous structures allow us to efficiently enumerate all upper triangles and all intermediate triangles.

Hybrid approach. For less well behaved graphs, the number of triangles t can significantly outgrow the number of arcs in $G^{\wedge}_{\pi}$. In the worst case, G is the complete graph

and the number of triangles t is in $\Theta(n^3)$, whereas the number of arcs is only in $\Theta(n^2)$. It can thus be prohibitive to store a list of all triangles. We therefore propose a hybrid approach, where only some triangles are precomputed.

The basic triangle enumeration algorithm computes the intersection of two lower neighborhoods and thus encounters two cases: either a neighbor is common (yielding a triangle that has to be processed) or it is not. With precomputed lower triangles, this second case can be eliminated, resulting in faster enumeration times.

Now, for arcs where both endpoints have a high level, many (of their numerous lower triangles) are contained in the top-level cliques of the Contraction Hierarchy. As a consequence, for them the ratio of common neighbors to noncommon neighbors is very high. For lower-level arcs, on the other hand, this ratio is often lower. This gives precomputed triangles for these lower levels the greater benefit over basic triangle enumeration. Hence, we propose to only precompute triangles for those arcs (u, v), where the level of u is below a certain threshold. The threshold is a tuning parameter that trades space for time.

Comparison with CRP. Triangle preprocessing has similarities with micro- and macrocode in CRP [Delling and Werneck 2013]. In the following, we compare the space consumption of these two approaches against our lower triangles' preprocessing scheme. However, note that at this stage, we do not yet consider travel direction on arcs. Hence, let t be the number of undirected triangles and m be the number of arcs in $G_\pi^\wedge$; further, let t' be the number of directed triangles and m' be the number of arcs used in Delling and Werneck [2013]. If every street is a one-way street, then $m' = m$ and $t' = t$; otherwise, without one-way streets, $m' = 2m$ and $t' = 2t$.

Micro code stores an array of triples of pointers to the arc weights of the three arcs in a directed triangle, i.e., it stores the equivalent of $3t'$ arc IDs. Computing the exact space consumption of macrocode is more difficult. However, it is easy to obtain a lower bound: macrocode must store for every triangle at least the pointer to the arc weight of the upper arc. This yields a space consumption equivalent to *at least* t' arc IDs. In comparison, our approach stores for each triangle the arc IDs of the two lower arcs. Additionally, the index array of the triangle adjacency array, which maps each arc to the set of its lower triangles, maintains $m+1$ entries. Each entry has a size equivalent to an arc ID. Our total memory consumption is thus $2t + m + 1$ arc IDs.

Hence, our approach always requires less space than microcode. It has similar space consumption as macrocode if one-way streets are rare; otherwise, it needs at most twice as much data. However, the main advantage of our approach over macrocode is that it allows for random access, which is crucial in the algorithms presented in the following sections.

7. CUSTOMIZATION

Up to now, we only considered the metric-independent first preprocessing phase. In this section, we describe the second, metric-dependent preprocessing phase, known as customization. In other words, we show how to efficiently extend the weights of the input graph to a corresponding metric with weights for all arcs in $G_\pi^\wedge$. We consider three different distances between the vertices: we refer to $\text{dist}_\text{I}(s, t)$ as the shortest st-path distance in the input graph G. With $\text{dist}_\text{UD}(s, t)$, we denote the shortest st-path distance in $G_\pi^\wedge$ when only considering up-down paths. Finally, let $\text{dist}_\text{A}(s, t)$ be the shortest st-path distance in G_π^*, i.e., when allowing arbitrary not-necessarily up-down paths in $G_\pi^\wedge$.

For correctness of the Contraction Hierarchy query algorithms (compare to Section 8), it is necessary that between any pair of vertices s and t, a shortest up-down st-path in $G_\pi^\wedge$ exists with the same distance as the shortest st-path in the input graph G. In other words, $\text{dist}_\text{I}(s, t) = \text{dist}_\text{A}(s, t) = \text{dist}_\text{UD}(s, t)$ must hold for all vertices s and t. We say that a metric that fulfills $\text{dist}_\text{I}(s, t) = \text{dist}_\text{A}(s, t)$ *respects* the input weights. If

additionally $\text{dist}_A(s,t) = \text{dist}_{UD}(s,t)$ holds, we call the metric *customized*. Note that customized metrics are not necessarily unique. However, there is a special customized metric, called *perfect* metric m_P, where for every arc (x,y) in $G_\pi^\wedge$, the weight of this arc $m_P(x,y)$ is equal to the shortest path distance $\text{dist}_I(x,y)$. We optionally use the perfect metric to perform perfect witness search.

Constructing a respecting metric is trivial: assign to all arcs of $G_\pi^\wedge$ that already exist in G their input weight and to all other arcs $+\infty$. Computing a customized metric is less trivial. We therefore describe in Section 7.1 the basic customization algorithm that computes a customized metric m_C given a respecting one. Afterward, we describe the perfect customization algorithm that computes the perfect metric m_P given a customized one (i.e., m_C). Finally, we show how to employ the perfect metric to perform a perfect witness search.

7.1. Basic Customization

A central notion of the basic customization algorithm is the *lower triangle inequality*, which is defined as follows. A metric m_C fulfills it if for all lower triangles $\{x,y,z\}$ of each arc (x,y) of $G_\pi^\wedge$, it holds that $m_C(x,y) \leq m_C(x,z) + m_C(z,y)$. We show that every respecting metric that also fulfills this inequality is customized. Our algorithm exploits this by transforming the given respecting metric in a coordinated way that maintains the respecting property and ensures that the lower triangle inequality holds. The resulting metric is thus customized. We first describe the algorithm and prove that the resulting metric is respecting and fulfills the inequality. We then prove that this is sufficient for the resulting metric to be customized.

Our algorithm iterates over all arcs $(x,y) \in G_\pi^\wedge$ ordered *increasingly* by the rank of x in a bottom-up fashion. For each arc (x,y), it enumerates all lower triangles $\{x,y,z\}$ and checks whether the path $x \to z \to y$ is shorter than the path $x \to y$. If this is the case, then it decreases $m_C(x,y)$ so that both paths are equally long. Formally, it performs for every arc (x,y) the operation $m_C(x,y) \leftarrow \min\{m_C(x,y), m_C(x,z) + m_C(z,y)\}$. Note that this operation never assigns values that do not correspond to a path length, and therefore m_C remains respecting. By induction over the vertex levels, we can show that after the algorithm is finished, the lower triangle inequality holds for every arc, i.e., for every arc (x,y) and lower triangle $\{x,y,z\}$, the inequality $m_C(x,y) \leq m_C(x,z) + m_C(z,y)$ holds. The key observation is that by construction, the rank of z must be strictly smaller than the ranks of x and y. The final weights of $m_C(x,z)$ and $m_C(z,y)$ have therefore already been computed when considering (x,y). In other words, when the algorithm considers the arc (x,y), the weights $m_C(x,z)$ and $m_C(z,y)$ are guaranteed to remain unchanged until termination.

THEOREM 7.1. *Every respecting metric that additionally fulfills the lower triangle inequality is customized.*

PROOF. We need to show that between any pair of vertices s and t, a shortest up-down st-path exists. As we assumed for simplicity that G is connected, there always exists a shortest not-necessarily up-down path from s to t. Either this is an up-down path or a subpath $x \to z \to y$ with $\pi^{-1}(x) > \pi^{-1}(z)$ and $\pi^{-1}(y) > \pi^{-1}(z)$ must exist. As z is contracted before x and y, an edge $\{x,y\}$ must exist. Because of the lower triangle inequality, we further know that $m(x,y) \leq m(x,z) + m(z,y)$. and thus replacing $x \to z \to y$ by $x \to y$ does not make the path longer. Either the path is now an up-down path or we can apply the argument iteratively. As the path has only a finite number of vertices, this is guaranteed to eventually yield the up-down path required by the theorem, and thus this completes the proof. □

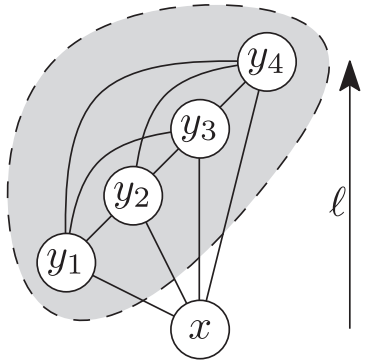


Fig. 5. The vertices $y_1 \dots y_4$ denote the upper neighborhood $N_u(x)$ of x. They form a clique (grey area) because x was contracted first. As $\ell(x) < \ell(y_j)$ for every j, we know by the induction hypothesis that the arcs in this clique are weighted by shortest path distances. We therefore have an all-pair shortest path distance table among all y_j. We have to show that using this information, we can compute shortest path distances for all arcs outgoing of x.

7.2. Perfect Customization

Given a customized metric m_C, we want to compute the perfect metric m_P. We first copy all values of m_C into m_P. Our algorithm then iterates over all arcs (x, y) *decreasing* by the rank of x in a top-down fashion. For every arc, it enumerates all intermediate and upper triangles $\{x, y, z\}$ and checks whether the path over z is shorter and adjusts the value of $m_P(x, y)$ accordingly, i.e., it performs $m_P(x, y) \leftarrow \min\{m_P(x, y), m_P(x, z) + m_P(z, y)\}$. After all arcs have been processed m_P is the perfect metric, as is shown in the following theorem.

THEOREM 7.2. *After the perfect customization, $m_P(x, y)$ corresponds to the shortest xy-path distance for every arc (x, y), i.e., m_P is the perfect metric.*

PROOF. We have to show that after the algorithm has finished processing a vertex x, all of its outgoing arcs in $G^{\wedge}_{\pi}$ are weighted by the shortest path distance. We prove this by induction over the level of the processed vertices. The top-most vertex is the only vertex in the top level. It does not have any upward arcs, and thus the algorithm does not have anything to do. This forms the base case of the induction. In the inductive step, we assume that all vertices with a strictly higher level have already been processed. As a consequence, we know that the upward neighbors of x form a clique weighted by shortest path distances. Denote these neighbors by y_i. The situation is depicted in Figure 5. The weights of the y_i encode a complete shortest path distance table between the upward neighbors of x.

Pick some arbitrary arc (x, y_j). We show the correctness of our algorithm by proving that either $m_C(x, y_j)$ is already the shortest path distance or a neighbor $y_k \in N_u(x)$ must exist such that $x \to y_k \to y_j$ is a shortest up-down path. For the rest of this paragraph, assume the existence of y_k; we prove its existence in the next paragraph. If $m_C(x, y_j)$ is already the shortest xy_j-path distance, then enumerating triangles will not change $m_C(x, y_j)$ and is thus correct. If $m_C(x, y_j)$ is not the shortest xy_j-path distance, then enumerating all intermediate and upper triangles of (x, y_j) is guaranteed to find the $x \to y_k \to y_j$ path, and thus the algorithm is correct. The upper triangles correspond to paths with $\ell(y_k) > \ell(y_j)$, whereas the intermediate triangles correspond to paths with $\ell(y_k) < \ell(y_j)$.

It remains to show that the $x \to y_k \to y_j$ shortest up-down path actually exists. As the metric is customized at every moment during the perfect customization, we know

that a shortest up-down xy_j-path K exists. As K is an up-down path, we can conclude that the second vertex of K must be an upward neighbor of x. We denote this neighbor by y_k. K thus has the following structure: $x \to y_k \to \cdots \to y_j$. As y_k has a higher rank than x, $m_P(y_k, y_j)$ is guaranteed to be the shortest $y_k y_j$-path distance, and therefore we can replace the $y_k \to \cdots \to y_j$ subpath of K by $y_k \to y_j$ and we have proven that the required $x \to y_k \to y_j$ shortest up-down path exists, which completes the proof. □

7.3. Perfect Witness Search

Using the perfect customization algorithm, we can efficiently compute the weighted Contraction Hierarchy with a minimum number of arcs with respect to the same contraction order. We present two variants of our algorithm. The first variant consists of removing each arc (x, y) whose weight $m_C(x, y)$ after basic customization does not correspond to the shortest xy-path distance $m_P(x, y)$. Although simple and correct, this variant does not remove as many arcs as possible, if a pair of vertices a and b exists in the input graph such that there are multiple shortest ab-paths. The second variant[2] also removes these additional arcs. An arc (x, y) is removed if and only if an upper or intermediate triangle $\{x, y, z\}$ exists such that the shortest path from x over z to y is no longer than the shortest xy-path. However, before we can prove the correctness of the second variant, we need to introduce some technical machinery, which will also be needed in the correctness proof of the stalling query algorithm. We define the "height" of a not-necessarily up-down path in G^*_π. We show that with respect to every customized metric, for every path that is not up-down, an up-down path must exist that is strictly higher and is no longer.

7.3.1. Variant for Graphs with Unique Shortest Paths. The first algorithm variant consists of removing all arcs (x, y) from the Contraction Hierarchy for which $m_P(x, y) \neq m_C(x, y)$. It is optimal if shortest path are unique in the input graph, i.e., between every pair of vertices a and b there is only one shortest ab-path. This simple algorithm is correct, as the following theorem shows.

THEOREM 7.3. *If the input graph has unique shortest paths between all pairs of vertices, then we can remove an arc* (x, y) *from the Contraction Hierarchy if and only if* $m_P(x, y) \neq m_C(x, y)$.

PROOF. We need to show that after removing all arcs, there still exists a shortest up-down path between every pair of vertices s and t. We know that before removing any arc, a shortest up-down st-path K exists. We show that no arc of K is removed, and thus K also exists after removing all arcs. Every subpath of K must be a shortest path, as K is a shortest path. Every arc of K is a subpath. However, we only remove arcs such that $m_P(x, y) \neq m_C(x, y)$, i.e., which are not shortest paths.

To show that no further arcs can be removed, we need to show that if $m_P(x, y) = m_C(x, y)$, then the path $x \to y$ is the only shortest up-down path. Denote the $x \to y$ path by Q. Suppose that another shortest up-down path R existed. R must be different than Q, i.e., a vertex z must exist that lies on R but not on Q. As z must be reachable from x, we know that z is higher than x. Unpacking the path Q in the input graph yields a path where x and y are the highest-ranked vertices, and thus this unpacked path cannot contain z. Unpacking R yields a path that contains z and is therefore different. Both paths are shortest paths from x to y in the input graph. This contradicts the assumption

[2]Note that the second algorithm variant exploits that we defined weights as being nonzero. If zero weights are allowed, it may remove too many arcs. A work-around consists of replacing all zero weights with a very small but nonzero weight.

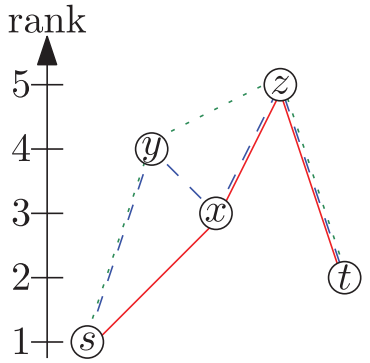


Fig. 6. The rank sequence of the solid red path is [3, 2, 1]. The 3 is the minimum of the ranks of the endpoints of the $\{x, z\}$ edge. Similarly, the 2 is induced by the $\{z, t\}$ edge and the 1 by the $\{s, x\}$ edge. The rank sequence of the blue dashed path is [3, 3, 2, 1], and the rank sequence of the green dotted path is [4, 2, 1]. The solid red path is the lowest followed by the blue dashed path, and the green dotted path is the highest.

that shortest paths are unique. We have thus proven that if the input graph has unique shortest paths, we can remove an arc (x, y) if and only if $m_P(x, y) \neq m_C(x, y)$. □

7.3.2. Variant for General Graphs. Using the first variant of our algorithm still yields correct results, even when shortest paths are not unique in the original graph. However, it is possible that some arcs are not removed that could be removed. Our second algorithm variant does not have this weakness. It removes all arcs (x, y) for which an intermediate or upper triangle $\{x, y, z\}$ exists such that $m_P(x, y) = m_P(x, z) + m_P(z, y)$. These arcs can efficiently be identified while running the perfect customization algorithm. An arc (x, y) is marked for removal if an upper or intermediate triangle $\{x, y, z\}$ with $m_C(x, y) \geq m_C(x, z) + m_C(z, y)$ is encountered. However, before we can prove the correctness of the second variant, we need to introduce some technical machinery.

We want to order paths by "height." To achieve this, we first define for each path K in G^*_π its *rank sequence*. We order paths by comparing the rank sequences lexicographically. Denote by v_i the vertices in K. For each edge $\{v_i, v_{i+1}\}$ in K, the rank sequence contains $\min\{\pi^{-1}(v_i), \pi^{-1}(v_{i+1})\}$. The numbers in the rank sequences are sorted in nonincreasing order. Two paths have the same height if one rank sequence is a prefix of the other. Otherwise, we compare the rank sequences lexicographically. This ordering is illustrated in Figure 6. We prove the following technical lemma.

LEMMA 7.4. *Let m_C be some customized metric. For every st-path K that is no up-down path, an up-down st-path Q exists such that Q is strictly higher than K and Q is no longer than K with respect to m_C.*

PROOF. Denote by v_i the vertices on the path K. As K is no up-down path, there must exist a vertex v_i on K that has lower ranks than its neighbors v_{i-1} and v_{i+1}. v_{i-1} and v_{i+1} are different vertices because they are part of a shortest path and zero weights are not allowed. Further, as v_i is contracted before its neighbors, there must be a edge between v_{i-1} and v_{i+1}. As the metric is customized, $m_C(v_{i-1}, v_{i+1}) \leq m_C(v_{i-1}, v_i) + m_C(v_i, v_{i+1})$ must hold. We can bypass v_i by replacing the subpath (v_{i-1}, v_i, v_{i+1}) with the single arc (v_{i-1}, v_{i+1}) without making the path longer. Denote this new path by R. R is higher than K, as we replaced $\pi^{-1}(v_i)$ in the rank sequence by $\min\{\pi^{-1}(v_{i-1}), \pi^{-1}(v_{i+1})\}$, which must be larger. Either R is an up-down path or we apply the argument iteratively. In each iteration, the path loses a vertex, and therefore we can guarantee that eventually we obtain an up-down path that is higher than K and no longer. This is the desired up-down path Q that is no longer than K and strictly higher. □

Note that this lemma does not exploit any property that is inherent to Contraction Hierarchies with a metric-independent contraction ordering and is thus applicable to every Contraction Hierarchy.

Given this technical lemma, we can prove the correctness of the second variant of our algorithm.

THEOREM 7.5. *We can remove an arc (x, y) if and only if an upper or intermediate triangle $\{x, y, z\}$ exists with $m_P(x, y) = m_P(x, z) + m_P(z, y)$.*

PROOF. We need to show that for every pair of vertices s and t, a shortest up-down st-path exists that uses no removed arc. We show that a highest shortest up-down st-path has this property. As the metric is customized, we know that a shortest up-down st-path K exists before removing any arcs. If K does not contain an arc (x, y) for which an upper or intermediate triangle $\{x, y, z\}$ exists with $m_P(x, y) = m_P(x, z) + m_P(z, y)$, then there is nothing to show. Otherwise, we modify K by inserting z between x and y. This does not modify the length of K, but we can no longer guarantee that K is an up-down path. If $\{x, y, z\}$ was an intermediate triangle, then K is still an up-down path. However, it is strictly higher, as we added $\pi^{-1}(z)$ into the rank sequence, which is guaranteed to be larger than $\pi^{-1}(x)$. If $\{x, y, z\}$ was an upper triangle, then K is no longer an up-down path. Fortunately, using Lemma 7.4, we can transform K into an up-down path that is no longer and strictly higher. In both cases, the new K is an up-down path or we apply the argument iteratively. As K gets strictly higher in each iteration and the number of up-down paths is finite, we know that we will eventually obtain a shortest up-down st-path where no arc can be removed.

Further, we need to show that if no such triangle exists, then an arc cannot be removed, i.e., we need to show that the only shortest up-down path from x to y is the path consisting only of the (x, y) arc. Assume that no such triangle and a further up-down path Q existed. Q must contain a vertex beside x and y, and all vertices in Q must have the rank of x or higher. Consider the vertex z that comes directly after x in Q. As x is contracted before z and y, an arc between z and y must exist. Therefore, a triangle $\{x, y, z\}$ must exist that is an intermediate triangle if z has a lower rank than y and is an upper triangle if z has a higher rank than y. However, we assumed that no such triangle can exist. We have thus proven that we can remove an arc (x, y) if and only if an upper or intermediate triangle $\{x, y, z\}$ exists with $m_P(x, y) = m_P(x, z) + m_P(z, y)$. □

7.4. Parallelization

The basic customization can be parallelized by processing the arcs (x, y) that depart within a level in parallel. Between levels, we need to synchronize all threads using a barrier. As all threads only write to the arc they are currently assigned to and only read from arcs processed in a strictly lower level, we can thus guarantee that no read/write conflict occurs. Hence, no locks or atomic operations are needed.

On most modern processors, perfect customization can be parallelized analogously to basic customization: we iterate over all arcs departing within a level in parallel and synchronize all threads between levels. For every arc (x, y), we enumerate all upper and intermediate triangles and update $m_P(x, y)$ accordingly.

Correctness of this algorithm is not obvious, because the exact order in which threads are executed influences intermediate results. Consider two threads A and B. Suppose that thread A processes an arc (x, y_A) at the same time as thread B processes another arc (x, y_B). Furthermore, suppose that thread A updates $m_P(x, y_A)$ at the same moment as thread B enumerates an intermediate or upper (with respect to (x, y_B)) triangle $\{x, y_B, y_A\}$. In this situation, it is unclear what value for (x, y_A) thread B will read. However, we will show in the following that our algorithm is correct as long it is guaranteed that thread B will either read the old value or the new value. Then, the end

result within each level is always the same, independent of execution order. Overall correctness follows.

In the proof of Theorem 7.2, we have shown that for every vertex x and arc (x, y_i), either the arc (x, y_i) already has the shortest path distance or an upper or intermediate triangle $\{x, y_i, y_j\}$ exists such that $x \to y_j \to y_i$ is a shortest path. No matter the order in which the threads process the arcs, they do not modify shortest path weights. This implies that the shortest path $x \to y_j \to y_i$ is thus retained, regardless of the execution order. This shortest path is not modified and is guaranteed to exist before any arcs outgoing from the current level are processed. Every thread is thus guaranteed to see it. However, other weights can be modified. Fortunately, this is not a problem as long as we can guarantee that no thread sees a value that is below the corresponding shortest path distance. Therefore, if we can guarantee that thread B either sees the old value or the new value, as is the case on x86 processors, then the algorithm is correct.

Otherwise, if thread B can see some mangled combination of the old value's bits and new value's bits, there are ways to mitigate the problem. To still apply parallelization, however, we would need to use locks or to make sure that all outgoing arcs of x are processed by the same thread.

7.5. Directed Graphs

Up to now, we have focused on customizing undirected graphs. If the input graph G is *directed*, our toolchain works as follows. Based on the *undirected unweighted* graph induced by G, we compute a vertex ordering π (Section 4), build the upward directed Contraction Hierarchy $G_\pi^\wedge$ (Section 5), and optionally perform triangle preprocessing (Section 6). For customization, however, we consider two weights per arc in $G_\pi^\wedge$, one for each direction of travel. One-way streets are modeled by setting the weight corresponding to the forbidden traversal direction to ∞. With respect to π, we define an upward metric m_u and a downward metric m_d on $G_\pi^\wedge$. For each arc $(x, y) \in G$ in the directed input graph with input weight $w(x, y)$, we set $m_u(x, y) = w(x, y)$ if $\pi^{-1}(x) < \pi^{-1}(y)$, i.e., if x is ordered before y; otherwise, we set $m_d(x, y) = w(x, y)$. All other values of m_u and m_d are set to ∞. In other words, each arc $(x, y) \in G_\pi^\wedge$ of the Contraction Hierarchy has upward weight $m_u(x, y) = w(x, y)$ if $(x, y) \in G$, downward weight $m_d(x, y) = w(y, x)$ if $(y, x) \in G$, and ∞ otherwise.

The basic customization considers both metrics m_u and m_d simultaneously. For every lower triangle $\{x, y, z\}$ of (x, y), it sets

$$m_u(x, y) \leftarrow \min\{m_u(x, y), m_d(x, z) + m_u(z, y)\},$$
$$m_d(x, y) \leftarrow \min\{m_d(x, y), m_u(x, z) + m_d(z, y)\}.$$

The perfect customization can be adapted analogously. For every intermediate triangle $\{x, y, z\}$ of (x, y), the perfect customization sets

$$m_u(x, y) \leftarrow \min\{m_u(x, y), m_u(x, z) + m_u(z, y)\},$$
$$m_d(x, y) \leftarrow \min\{m_d(x, y), m_d(x, z) + m_d(z, y)\}.$$

Similarly, for every upper triangle $\{x, y, z\}$ of (x, y), the perfect customization sets

$$m_u(x, y) \leftarrow \min\{m_u(x, y), m_u(x, z) + m_d(z, y)\},$$
$$m_d(x, y) \leftarrow \min\{m_d(x, y), m_d(x, z) + m_u(z, y)\}.$$

The perfect witness search might need to remove an arc only in one direction. It therefore produces, just as in the original Contraction Hierarchies, two search graphs: an upward search graph and a downward search graph. The forward search in the query phase is limited to the upward search graph and the backward search to the downward search graph, just as in the original Contraction Hierarchies. The arc (x, y) is removed from the upward search graph if and only if an intermediate triangle

$\{x, y, z\}$ with $m_u(x, y) = m_u(x, z) + m_u(z, y)$ exists or an upper triangle $\{x, y, z\}$ with $m_u(x, y) = m_u(x, z) + m_d(z, y)$ exists. Analogously, the arc (x, y) is removed from the downward search graph if and only if an intermediate triangle $\{x, y, z\}$ with $m_d(x, y) = m_d(x, z) + m_d(z, y)$ exists or an upper triangle $\{x, y, z\}$ with $m_d(x, y) = m_d(x, z) + m_u(z, y)$ exists.

7.6. Single-Instruction Multiple Data

The weights attached to each arc in the Contraction Hierarchy can be replaced by an interleaved set of k weights by storing for every arc a vector of k elements. Vectors allow us to customize all k metrics in one go, amortizing triangle enumeration time. Additionally, they allow us to use single-instruction multiple data (SIMD) operations. As we use essentially two metrics to enable directed graphs, we can store both of them in a 2-dimensional vector. This allows us to handle both directions in a single processor instruction. Similarly, if we have k directed input weights, we can store them in a $2k$-dimensional vector. (Depending on the width of SIMD registers, we might require more than one SIMD instruction per vector; however, this approach still benefits from amortized triangle enumeration time, which is only done once per arc.)

The processor needs to support component-wise minimum and saturated addition, i.e., $a + b = \text{int}_{\max}$ must hold in the case of an overflow. In the case of directed graphs, it additionally needs to support efficiently swapping neighboring vector components. A current SSE-enabled processor supports all of the necessary operations for 16-bit integer components. For 32-bit, integer saturated addition is missing. There are two possibilities to work around this limitation. The first is to emulate saturated-add using a combination of regular addition, comparison, and blend/if-then-else instruction. The second consists of using 31-bit weights and use $2^{31} - 1$ as value for ∞ instead of $2^{32} - 1$. The algorithm only computes the saturated addition of two weights followed by taking the minimum of the result and some other weight, i.e., if computing $\min(a + b, c)$ for all weights a, b, and c is unproblematic, then the algorithms works correctly. We know that a and b are at most $2^{31} - 1$, and thus their sum is at most $2^{32} - 2$, which fits into a 32-bit integer. In the next step, we know that c is at most $2^{31} - 1$, and thus the resulting minimum is also at most $2^{31} - 1$.

7.7. Partial Updates

Until now, we have only considered computing metrics from scratch. However, in many scenarios this is overkill, as we know that only a few edge weights of the input graph were changed. It is unnecessary to redo all computations in this case. The ideas employed by our algorithm are somewhat similar to those presented in Geisberger et al. [2012], but our situation differs, as we know that we do not have to insert or remove arcs. Denote by $U = \{((x_i, y_i), w_i^{\text{new}})\}$ the set of arcs whose weights should be updated, where (x_i, y_i) is the arc ID and w_i^{new} the new weight. Note that modifying the weight of one arc can trigger further changes. However, these new changes have to be at higher levels. We therefore organize U as a priority queue ordered by the level of x_i. We iteratively remove arcs from the queue and apply the change. If new changes are triggered, we insert these into the queue. The algorithm terminates once the queue is empty.

Denote by (x, y) the arc that was removed from the queue and by w^{new} its new weight and by w^{old} its old weight. We first have to check whether w^{new} can be bypassed using a lower triangle. For this reason, we iterate over all lower triangles $\{x, y, z\}$ of (x, y) and perform $w^{\text{new}} \leftarrow \min\{w^{\text{new}}, m(z, x) + m(z, y)\}$. Furthermore, if $\{x, y\}$ is an edge in the input graph G, we might have overwritten its weight with a shortcut weight, which after the update might not be shorter anymore. Hence, we additionally test that w^{new} is not larger than the input weight. If after both checks $w^{\text{new}} = m(x, y)$ holds, then no change is necessary and no further changes are triggered. If w^{old} and w^{new} differ, we

iterate over all upper triangles $\{x, y, z\}$ of (x, y) and test whether $m(x, z) + w^{\text{old}} = m(y, z)$ holds; if so, the weight of the arc (y, z) must be set to $m(x, z) + w^{\text{new}}$. We add this change to the queue. Analogously, we iterate over all intermediate triangles $\{x, y, z\}$ of (x, y) and queue up a change to (z, y) if $m(x, z) + w^{\text{old}} = m(z, y)$ holds.

How many subsequent changes a single change triggers heavily depends on the metric and can significantly vary. Slightly changing the weight of a dirt road has near to no impact, whereas changing a heavily used highway segment will trigger many changes. In the game setting, such largely varying running times are undesirable, as they lead to lag peaks. We propose to maintain a queue into which all changes are inserted. Every round, a fixed amount of time is spent processing elements from this queue. If time runs out before the queue is emptied, the remaining arcs are processed in the next round. This way costs are amortized resulting in a constant workload per turn. The downside is that as long the queue is not empty, some distance queries will use outdated data. How much time is spent each turn updating the metric determines how long an update needs to be propagated along the whole graph.

8. DISTANCE AND SHORTEST PATH QUERIES

In this section, we describe how to answer distance queries, i.e., we compute the distance in G between two vertices s and t by constructing a shortest up-down st-path in $G^{\wedge}_{\pi}$ given a customized metric. We further describe how to unpack into a shortest path edge sequence in G.

8.1. Basic Query Algorithm

The basic query runs two instances of Dijkstra's algorithm on $G^{\wedge}_{\pi}$ from s and from t. If G is undirected, then both searches use the same metric. Otherwise, if G is directed, the search from s uses the upward metric m_u and the search from t the downward metric m_d. In either case, in contrast to Geisberger et al. [2012], they operate on the same upward search graph $G^{\wedge}_{\pi}$. Once the radius of one of the two searches is larger than the shortest path found so far, we stop the search because we know that no shorter path can exist. We alternate between processing vertices in the forward search and processing vertices in the backward search.

8.2. Stalling

We implemented a basic version of an optimization presented in Geisberger et al. [2012] and Sanders and Schultes [2012] called *stall-on-demand*. The optimization exploits that the shortest strictly upward sv-path in $G^{\wedge}_{\pi}$ can be longer than the shortest sv-path in G^{*}_{π}, which can go up and down arbitrarily. The search from s only finds upward paths, and if we observe that an up-down path exists that is not longer, then we can prune the upward search. Denote by x the vertex removed from the queue. We iterate over all outgoing arcs (x, y) and test whether $d(x) \geq m(x, y) + d(y)$ holds. If it holds for some arc, we prune x by not relaxing its outgoing arcs.

If $d(x) > m(x, y) + d(y)$ holds, then pruning is correct because all subpaths of shortest up-down paths must be shortest paths and the upward path ending at x is not shortest path as a shorter up-down path through y exists. We can also prune when $d(x) \geq m(x, y) + d(y)$, but a different argument is needed. To the best of our knowledge, correctness has so far not been proven for the $d(x) = m(x, y) + d(y)$ case. Notice that we do not exploit any special properties of metric-independent orders, and thus our proof works for every Contraction Hierarchy.

THEOREM 8.1. *The upward search can be pruned when $d(x) \geq m(x, y) + d(y)$ holds.*

PROOF. We show that for every pair of vertices s and t, an unprunable, shortest, up-down st-path exists. Our proof relies on Lemma 7.4, which orders paths by height

and states that st-path that are no up-down paths can be transformed into up-down paths that are no longer and strictly higher. We know that some shortest st-path K exists. If K is not pruned, then there is nothing to show. If K is pruned, then there exists a vertex x on K at which the search is pruned. Without loss of generality, we assume that x lies on the upward part of K. Further, there must exist a vertex y and a path Q from s to x going through y such that Q is no longer than the sx-prefix of K. Consider the path R obtained by concatenating Q with the xt-suffix of K. R is by construction not longer than K. If x is the highest vertex on K, then R is an up-down path and R is strictly higher. Otherwise, R is no up-down path, but using Lemma 7.4, R can be transformed into an up-down path that is strictly higher and no longer. In both cases, R is no longer and strictly higher. Either R is unprunable or we apply the argument iteratively. As there are only finitely many up-down paths and each iteration increases the height of R, we eventually end up at an unprunable, shortest, up-down st-path, which concludes the proof. □

8.3. Elimination Tree–Based Query Algorithm

For every vertex, we precompute its parent's vertex ID in the elimination tree in a preprocessing step. This allows us to efficiently enumerate all vertices in SS(s) and SS(t) at query time, increasingly by rank.

We store two tentative distance arrays $d_f(v)$ and $d_b(v)$. Initially, these are all set to ∞. In a first step, we compute the lowest common ancestor (LCA) x of s and t in the elimination tree. We do this by simultaneously enumerating all ancestors of s and t by increasing rank until a common ancestor is found. In a second step, we iterate over all vertices y on the tree path from s to x and relax all forward arcs of such y. In a third step, we do the same for all vertices y from t to x in the backward search. In a final fourth step, we iterate over all vertices y from x to the root r and relax all forward and backward arcs. Further, in the fourth step, we also determine the vertex z that minimizes $d_f(z) + d_b(z)$. A shortest up-down path must exist that goes through z. Knowing z is necessary to determine the shortest path distance and to compute the sequence of arcs that compose the shortest path. In a fifth cleanup step, we iterate over all vertices from s and t to the root r to reset all d_f and d_b to ∞. This fifth step avoids having to spend $O(n)$ running time to initialize all tentative distances to ∞ for each query. Consider the situation depicted in Figure 7. In the first step, the algorithm determines x. In the second step, it relaxes all dotted arcs and the tree arcs departing in the light gray area. In the third step, all dashed arcs and the tree arcs departing in the medium gray area, and in the fourth step, the solid arcs and the remaining tree arcs follow.

The elimination tree query can be combined with the perfect witness search. Before pruning any arc, we compute the elimination tree. We then prune the arcs. It is now possible that a vertex has an ancestor in the tree that is not in its pruned search space. However, we can still guarantee that every vertex in the pruned search space is an ancestor, and this is enough to prove the query correctness. To avoid relaxing the outgoing arcs of an ancestor outside of the search space, we prune vertices whose tentative distance $d_f(x)$ respectively $d_b(x)$ is ∞.

Contrary to the approaches based upon Dijkstra's algorithm, the elimination tree query approach does not need a priority queue. This leads to significantly less work per processed vertex. Unfortunately, the query must always process all vertices in the search space. Luckily, our experiments show that for random queries with s and t sampled uniformly at random the query time ends up being lower for the elimination tree query. If s and t are close in the original graph, i.e., not sampled uniformly at random, then the Dijkstra-based approaches win.

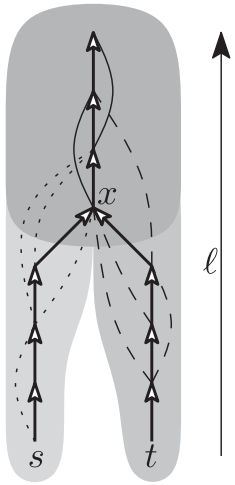


Fig. 7. The union of the dark gray and light gray areas is the search space of s. Analogously, the union of the dark gray and middle gray areas is the search space of t. The dark gray area is the intersection of both search spaces. The dotted arcs start in the search space of s, but not in the search space of t. Analogously, the dashed arcs start in the search space of t, but not in the search space of s. The solid arcs start in the intersection of the two search spaces. The vertex x is the LCA of s and t.

8.4. Path Unpacking

All shortest path queries presented only compute shortest up-down paths. This is enough to determine the distance of a shortest path in the original graph. However, if the sequence of edges that form a shortest path should be computed, then the up-down path must be unpacked. The original Contraction Hierarchy of Geisberger et al. [2012] unpacks an up-down path by storing for every arc (x, y) the vertex z of the lower triangle $\{x, y, z\}$ that caused the weight at $m(x, y)$. This information depends on the metric, and we want to avoid storing additional metric-dependent information. We therefore resort to a different strategy: denote by $p_1 \dots p_k$ the up-down path found by the query. As long as a lower triangle $\{p_i, p_{i+1}, x\}$ of an arc (p_i, p_{i+1}) exists with $m(p_i, p_{i+1}) = m(x, p_i) + m(x, p_{i+1})$, our algorithm inserts the vertex x between p_i and p_{i+1} into the path.

9. EXPERIMENTS

In this section, we present an extensive experimental evaluation of the algorithms introduced and described earlier.

Compiler and machine. We implemented our algorithms in C++, using g++ 4.7.1 with −O3 for compilation. The customization and query experiments were run on a dual eight-core Intel Xeon E5-2670 processor, which is based on the Sandy Bridge architecture, clocked at 2.6GHz, with 64GiB of DDR3-1600 RAM, 20MiB of L3, and 256KiB of L2 cache. The order computation experiments reported later in Table II were run on a single core of an Intel Core i7-2600K processor.

Instances. We evaluate three large instances of practical relevance in detail. In Section 11, we provide summarized experiments on further instances. The sizes of our main test instances are reported in Table I. The DIMACS-Europe graph[3] was provided by PTV[4] for the DIMACS challenge [Demetrescu et al. 2009]. The vertex

[3]Visit www.iti.kit.edu/resources/roadgraphs.php for details on how to acquire this graph.
[4]http://www.ptvgroup.com.

Table I. Benchmark Instances

Instance	Vertices (#)	Arcs (#)	Edges (#)	Symmetric?	Dijkstra (ms)
Karlsruhe	120,412	302,605	154,869	No	6
TheFrozenSea	754,195	5,815,688	2,907,844	Yes	58
Europe	18,010,173	42,188,664	22,211,721	No	1,560

Note: We report the number of vertices and directed arcs, as well as the number of edges in the induced undirected graph. For comparison, we also report the running time of Dijkstra's algorithm (with stop criterion) averaged over 10 000 *st*-queries, where s and t are chosen uniformly at random.

Fig. 8. All vertices in the DIMACS-Europe graph.

positions are depicted in Figure 8. It is the standard benchmarking instance used by road routing papers over the past few years. Note that besides roads, it also contains a few ferries to connect Great Britain and some other islands with the continent. The Europe graph analyzed here is its largest strongly connected component, which is a common method to remove bogus vertices. The numbers in Table I display statistics after computing the strongly connected component. The graph is directed, and we consider two different weights. The first weight is the travel time, and the second weight is the straight-line distance between two vertices on a perfect Earth sphere. Note that in the input data highways are often modeled using only a small number of vertices compared to the streets going through the cities. This differs from other data sources, such as OpenStreetMap (OSM),[5] that have a high number of vertices on highways to model road bends. As demonstrated later in Section 11.1, degree-2 vertices do not hamper the performance of Contraction Hierarchies. The Karlsruhe graph is a subgraph of the PTV graph for a larger region around Karlsruhe. We consider the largest connected component of the graph induced by all vertices with a latitude between 48.3° and 49.2°, and a longitude between 8° and 9°.

The TheFrozenSea graph is based on the largest Star Craft map presented in Sturtevant [2012]. The map is composed of square tiles having at most eight neighbors and distinguishes between walkable and nonwalkable tiles. These are not distributed uniformly but rather form differently sized pockets of freely walkable space alternating with *choke points* of very limited walkable space. The corresponding graph contains for every walkable tile a vertex and for every pair of adjacent walkable tiles an edge. Diagonal edges are weighted by $\sqrt{2}$, whereas horizontal and vertical edges have weight 1. The graph is symmetric, i.e., for each forward arc there is a backward arc, and contains large grid subgraphs.

For comparability with other works, in Table I we report the time needed by Dijkstra's algorithm. Our implementation uses a 4-ary heap. As usual, it is unidirectional and employs a stopping criterion for point-to-point queries. Performance was obtained before reordering vertices in memory.

[5]http://www.openstreetmap.org.

Table II. Orders

Instance	MetDep	Metis	KaHIP
Karlsruhe	4.1	0.5	<1,532
TheFrozenSea	1,280.4	4.7	<22,828
Europe	813.5	131.3	<249,082

Note: Duration of order computation in seconds, without parallelization. KaHIP was parameterized for quality only, disregarding running time (as part of the metric-independent first phase). We are certain that large speedups are possible. However, this is not the focus of this work. See Section 12.1 for a discussion about how good orders could be quickly computed.

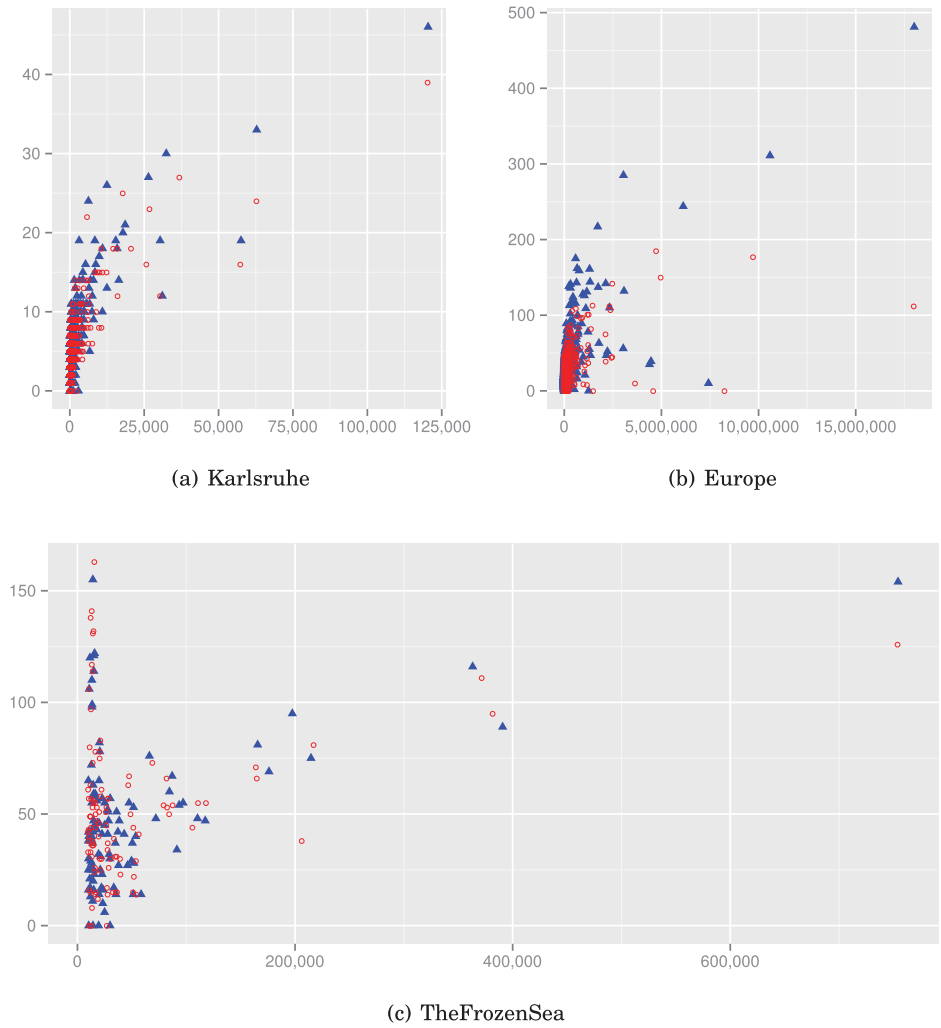
9.1. Orders

We analyze three different vertex orders: (1) the greedy metric-dependent order in the spirit of Geisberger et al. [2012], which we refer to as "MetDep" in the tables; (2) the Metis 5.0.1 graph partitioning package, which contains a tool called `ndmetis` to create ND-orders; and (3) KaHIP 0.61, which provides only graph partitioning tools. We therefore implemented a very basic nested dissection computation on top of it: for every graph, we iteratively compute bisections with different random seeds using the "strong" configuration of KaHIP, until for 10 consecutive runs no better cut is found. We recursively bisect the graph until the parts are too small for KaHIP to handle and assign the order arbitrarily in these small parts. We set the imbalance for KaHIP to 20%. Note that our program is solely tuned for quality, completely disregarding running time. We report the running times only as upper bounds. The running times reported in Table II cannot be used to conclude that the *original* KaHIP package is slow.

Table II reports the times needed to compute the orders. Interestingly, Metis is even faster than the metric-dependent greedy vertex ordering strategy. Figure 9 shows the sizes of the computed separators. As expected, KaHIP results in better quality. Moreover, our road graphs have separators that seem to follow a cubic-root law. (Note that a more rigorous complexity analysis as in McGeoch et al. [2002] would be interesting but is not the focus of this work.) On Karlsruhe, the separator sizes steadily decrease from the top level to the bottom level, making Theorem 4.3 directly applicable under the assumption that no significantly better separators exist. The KaHIP separators on the Europe graph have a different structure on the top level. The separators first increase before they get smaller. This is because of the special structure of the European continent. For example, the cut separating Great Britain and Spain from France is far smaller than one would expect for a graph of that size. In the next step, KaHIP cuts Great Britain from Spain, which results in one of the extremely thin cuts observed in the plot. Interestingly, Metis is not able to find these cuts that exploit the continental topology. The game map has a structure that differs from road graphs, as the plots have two peaks. This effect results from the large grid subgraphs. Grids have $\Theta(\sqrt{n})$ separators, whereas at the higher levels the choke points results in separators that approximately follow a cubic-root law. At some point, the bisector has cut all choke points and has to start cutting through the grids. The second peak is at the point where this switch happens.

9.2. Contraction Hierarchy Construction

Table III compares the performance of our specialized contraction graph data structure, described in Section 5, to the dynamic adjacency structure, as used in Geisberger et al. [2012] to compute undirected and unweighted Contraction Hierarchies. We do not



(a) Karlsruhe

(b) Europe

(c) TheFrozenSea

Fig. 9. The amount of vertices in the separator (vertical) versus the number of vertices in the subgraph being bisected (horizontal). We only plot the separators for (sub)graphs of at least 1,000 vertices. The red hollow circles represent KaHIP, and the blue-filled triangles represent Metis.

Table III. Construction of the Contraction Hierarchy

Instance	Dyn. Adj. Array	Contraction Graph
Karlsruhe	0.6	<0.1
TheFrozenSea	490.6	3.8
Europe	305.8	15.5

Note: We report the time in seconds required to compute the arcs in $G_\pi^\wedge$ given a KaHIP ND-order π. No witness search is performed. No weights are assigned.

report numbers for the hash-based approach of Zeitz [2013], as it is fully dominated. Our data structure dramatically improves performance. However, to be fair, our approach cannot immediately be extended to directed or weighted graphs. Fortunately, this is no problem, as we can introduce weights and directions during the customization phase.

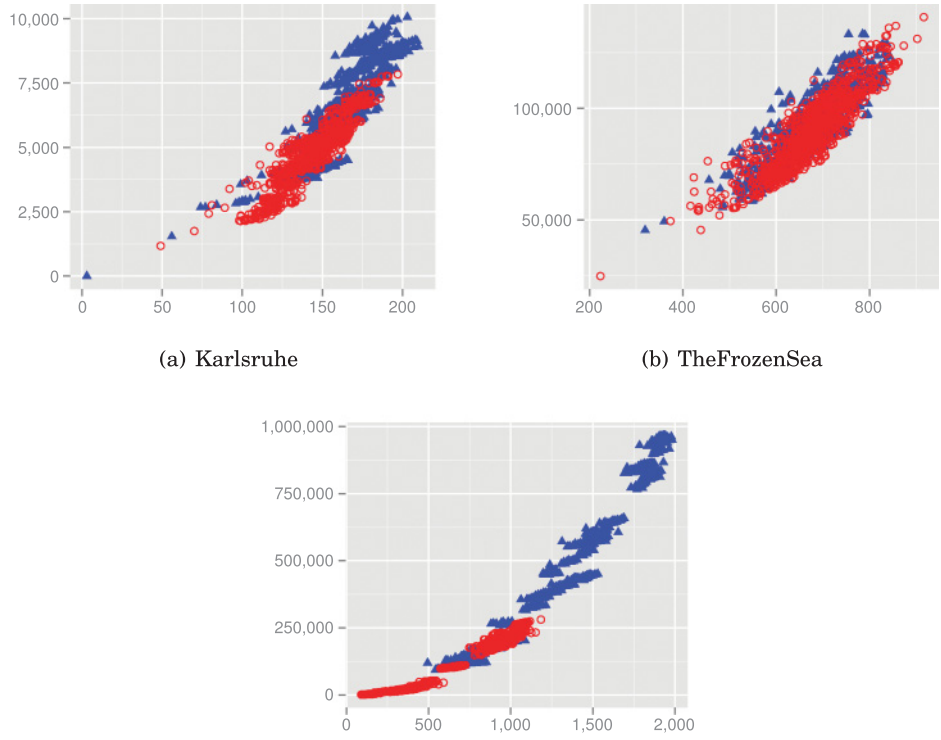
Table IV. Size of the Contraction Hierarchies for Different Instances and Orders

	Order	Witness Search	Arcs (#) [$\cdot 10^3$]		Triangles (#) [$\cdot 10^3$]	Average Upward Search Space Size			
						Unweighted		Weighted	
			Undirected	Upward		Vertices(#)	Arcs (#)	Vertices (#)	Arcs (#)
Karlsruhe	MetDep	None	21,926	17,661	37,439,858	5,870	15,786,622	5,246	11,281,564
		Heuristic	—	244	—	—	—	108	503
		Perfect	—	239	—	—	—	107	498
	Metis	None	478	463	2,590	164	6,579	163	6,411
		Perfect	—	340	—	—	—	152	2,903
	KaHIP	None	528	511	2,207	143	4,723	142	4,544
		Perfect	—	400	—	—	—	136	2,218
TheFrozenSea	MetDep	Heuristic	—	6,400	—	—	—	1,281	13,330
	Metis	None	21,067	21,067	601,846	676	92,144	676	92,144
		Perfect	—	10,296	—	—	—	644	32,106
	KaHIP	None	25,100	25,100	864,041	674	89,567	674	89,567
		Perfect	—	10,162	—	—	—	645	24,782
Europe	MetDep	Heuristic	—	33,912	—	—	—	709	4,808
	Metis	None	70,070	65,546	1,409,250	1,291	464,956	1,289	453,366
		Perfect	—	47,783	—	—	—	1,182	127,588
	KaHIP	None	73,920	69,040	578,248	652	117,406	651	108,121
		Perfect	—	55,657	—	—	—	616	44,677

Note: We report the average number of vertices and arcs reachable in the upward search space of a vertex. This number varies depending on whether a witness search is performed or not. It also varies depending on whether we follow one-way streets in both directions or not. We also report the number of triangles. As an indication for query performance, we report the average search space size in vertices and arcs by sampling the search space of 1,000 random vertices. Metis and KaHIP orders are metric independent. We report resulting figures after applying different variants of witness search. A heuristic witness search is one that exploits the metric in the preprocessing phase. A perfect witness search is described in Section 7.

9.3. Contraction Hierarchy Size

In Table IV, we report the resulting Contraction Hierarchy sizes for various approaches. Computing a Contraction Hierarchy on Europe *without witness search* with the greedy, metric-dependent order is infeasible even using the contraction graph data structure. This is also true if we only want to count the number of arcs. We aborted calculations after several days. However, we can state with certainty that there are at least 1.3×10^{12} arcs in the Contraction Hierarchy, and the maximum upward vertex degree is at least 1.4×10^6. As the original graph has only 4.2×10^7 arcs, it is safe to assume that using this order, it is impossible to achieve a speedup over Dijkstra's algorithm on the input graph. However, at least on the Karlsruhe graph, we can compute the Contraction Hierarchy without witness search and perform a perfect witness search. The numbers show that the heuristic witness search employed by Geisberger et al. [2012] is nearly optimal. Furthermore, the numbers clearly show that using metric-dependent orders in a metric-independent setting, i.e., without witness search, results in unpractical Contraction Hierarchy sizes. However, they also show that a metric-dependent order exploiting the weight structure dominates ND-orders. In Figure 10, we plot the number of arcs in the search space versus the number of vertices. The plots show that the KaHIP order significantly outperforms the Metis order on the road graphs, whereas the situation is a lot less clear on the game map, where the plots suggest nearly a tie. KaHIP only slightly outperforms Metis when using a perfect customization. Table V examines the elimination tree. Most noticeably, it has a relatively small height compared to the number of vertices in G. Note that the height of the elimination tree



(a) Karlsruhe

(b) TheFrozenSea

(c) Europe

Fig. 10. The number of vertices (horizontal) versus the number of arcs (vertical) in the search space of 1,000 random vertices. The red hollow circles represent KaHIP, and the blue-filled triangles represent Metis.

Table V. Elimination Tree Characteristics

Instance	Order	Children (#)		Height		Upper Bound of Treewidth
		Avg.	Max.	Avg.	Max.	
Karlsruhe	Metis	1	5	163.48	211	92
	KaHIP	1	5	142.19	201	72
TheFrozenSea	Metis	1	3	675.61	858	282
	KaHIP	1	3	676.71	949	287
Europe	Metis	1	8	1,283.45	2,017	876
	KaHIP	1	7	654.07	1,232	479

Note: Note that unlike in Table IV, these values are exact and not sampled over a random subset of vertices. We also report upper bounds on the treewidth of the input graphs after dropping the directions of arcs.

corresponds[6] to the number of vertices in the (undirected) search space. As the ratio between the maximum and the average height is only about 2, we know that no special vertex exists that has a search space significantly differing from the numbers shown in Table V.

The treewidth of a graph is a measure widely used in theoretical computer science and thus interesting on its own. The notion of treewidth is deeply coupled with the

[6]The numbers in Tables IV and V deviate a little because the search spaces in the former table are sampled, whereas in the latter we compute precise values.

Table VI. Detailed Analysis of the Size of Contraction Hierarchies after Perfect Witness Search

Instance	Metric	Order	Witness Search	Upward Arcs (#)	Avg. Weighted Upward Search Space	
					Vertices (#)	Arcs (#)
Karlsruhe	Distance	MetDep	None	8,000,880	3,276	4,797,224
			Heuristic	295,759	283	2,881
			Perfect	295,684	281	2,873
		Metis	Perfect	382,905	159	3,641
		KaHIP	Perfect	441,998	141	2,983
	Uniform	MetDep	None	5,705,168	2,887	3,602,407
			Heuristic	272,711	151	808
			Perfect	272,711	151	808
		Metis	Perfect	363,310	153	2,638
		KaHIP	Perfect	426,145	136	2,041
	Random	MetDep	None	6,417,960	3,169	4,257,212
			Heuristic	280,024	160	949
			Perfect	276,742	160	948
		Metis	Perfect	361,964	154	2,800
		KaHIP	perfect	424,999	138	2,093
Europe	Distance	MetDep	Heuristic	39,886,688	4,661	133,151
		Metis	perfect	53,505,231	1,257	178,848
		KaHIP	Perfect	60,692,639	644	62,014

Note: We evaluate uniform, random, and distance weights on the Karlsruhe input graph. Random weights are sampled from [0, 10000]. The distance weight is the straight distance along a perfect Earth sphere's surface. All weights respect one-way streets of the input graph.

notion of chordal supergraphs and vertex separators. Refer to Bodlaender and Koster [2010] for details. The authors show in their Theorem 6 that the maximum upward degree $d_u(v)$ over all vertices v in $G_\pi^\wedge$ is an upper bound to the treewidth of a graph G. This theorem yields a straightforward algorithm that gives us the upper bounds presented in Table V.

Interestingly, these numbers correlate with our other findings: the difference between the bounds on the road graphs reflect that the KaHIP orders are better than Metis orders. On the game map, there is nearly no difference between Metis and KaHIP, which is in accordance with all other performance indicators. The fact that the treewidth grows with the graph size reflects that the running times are not independent of the graph size. These numbers strongly suggest that road graphs are not part of a graph class of constant treewidth. However, fortunately, the treewidth seems to grow sublinearly. Our findings from Figure 9 suggest that assuming a treewidth of $O(\sqrt[3]{n})$ for road graphs of n vertices might come close to reality, although more rigorous analysis as in McGeoch et al. [2002] would be necessary to substantiate this claim.

In Table VI, we evaluate the witness search performances for different metrics. It turns out that the distance metric is the most difficult one of the tested metrics. That the distance metric is more difficult than the travel time metric is well known. However, it surprised us that uniform and random metrics are easier than the distance metric. We suppose that the random metric contains a few very long arcs that are nearly never used. These could just as well be removed from the graph, resulting in a thinner graph with nearly the same shortest path structure. The Contraction Hierarchy of a thinner graph with a similar shortest path structure naturally has a smaller size. To explain why the uniform metric behaves more similar to the travel time metric than to the distance metric, we have to realize that on our data source, highways do not have many degree-2 vertices in the input graph. Highways are therefore also preferred by

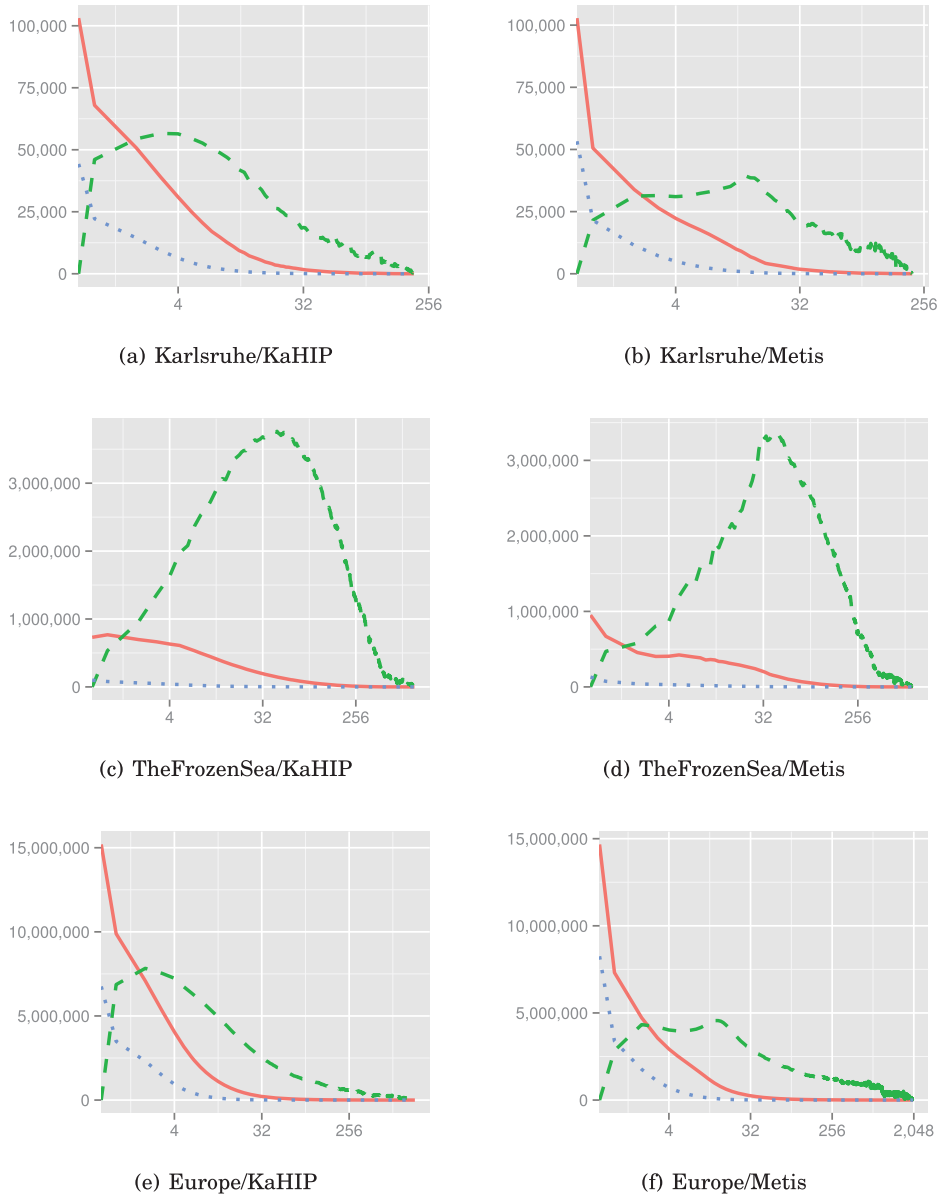
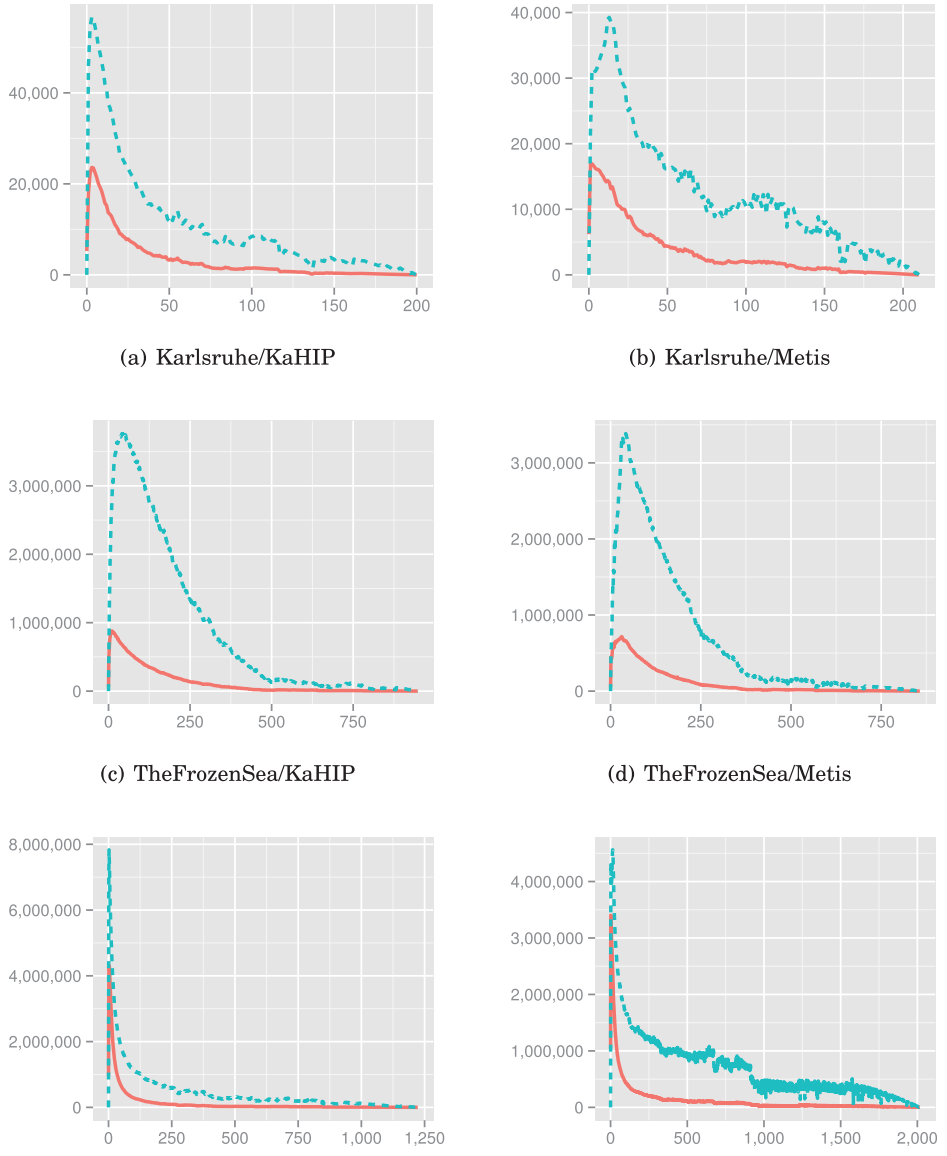


Fig. 11. The number of vertices (y-axis) per level (x-axis) is represented by the blue dotted line. The number of arcs departing in each level is represented by the red solid line, and the number of lower triangles in each level is represented by the green dashed line. Warning: In contrast to Figure 12, these figures have a logarithmic x-scale.

the uniform metric. We expect an instance with more degree-2 vertices on highways to behave differently. Interestingly, the heuristic witness search is perfect for a uniform metric. We expect this effect to disappear on larger graphs.

Recall that a Contraction Hierarchy is a DAG, and in DAGs, each vertex can be assigned a level. If a vertex can be placed in several levels, we put it in the lowest level. Figure 11 illustrates the amount of vertices and arcs in each level of a Contraction Hierarchy. The many highly ranked extremely thin levels are a result of the top-level



(a) Karlsruhe/KaHIP (b) Karlsruhe/Metis

(c) TheFrozenSea/KaHIP (d) TheFrozenSea/Metis

(e) Europe/KaHIP (f) Europe/Metis

Fig. 12. The number of lower triangles (y-axis) per level (x-axis) is represented by the blue dashed line, and the time needed to enumerate all of them per level is represented by the red solid line. The time unit is 100ns. If the time curve thus rises to 1,000,000 on the plot, the algorithm needs 0.1 seconds. Warning: In contrast to Figure 11, these figures do not have a logarithmic x-scale.

separator clique: inside a clique, every vertex must be on its own level. A few big separators therefore significantly increase the level count.

9.4. Triangle Enumeration

We first evaluate the running time of the adjacency-array–based triangle enumeration algorithm. Figure 12 clearly shows that most time is spent enumerating the triangles of the lower levels. This justifies our suggestion to only precompute the triangles for the

Table VII. Precomputed Triangles

		Karlsruhe		TheFrozenSea		Europe	
		Metis	KaHIP	Metis	KaHIP	Metis	KaHIP
Full	Triangles (#) [10^3]	2,590	2,207	601,846	864,041	1,409,250	578,247
	Contraction Hierarchy arcs (#) [10^2]	478	528	21,067	25,100	70,070	73,920
	Memory [MB]	22	19	4,672	6,688	11,019	4,694
Partial	Threshold level	16	11	51	54	42	17
	Triangles (#) [10^3]	507	512	126,750	172,240	147,620	92,144
	Contraction Hierarchy arcs (#) [10^3]	367	393	13,954	15,996	58,259	59,282
	Memory [MB]	5	5	1,020	1,375	1,348	929
	Enum.time [%]	33	32	33	33	32	33

Note: As shown in Section 6, the memory needed is proportional to $2t + m + 1$, where t is the triangle count and m is the number of arcs in the Contraction Hierarchy. We use 4-byte integers. We report t and m for precomputing all levels ("full") and all levels below a reasonable threshold level ("partial"). We further indicate the percentage of total unaccelerated enumeration time spent below the given threshold level. We chose the threshold level such that this factor is about 33%.

Table VIII. Basic Customization Performance

				Karlsruhe		TheFrozenSea		Europe	
SSE	Pre. Trian.	Thr. (#)	Metric Pairs (#)	Metis Time (s)	KaHIP Time (s)	Metis Time (s)	KaHIP Time (s)	Metis Time (s)	KaHIP Time (s)
No	None	1	1	0.0567	0.0468	7.88	10.08	21.90	10.88
Yes	None	1	1	0.0513	0.0427	7.33	9.34	19.91	9.55
Yes	All	1	1	0.0094	0.0091	3.74	3.75	7.32	3.22
Yes	All	16	1	0.0034	0.0035	0.45	0.61	1.03	0.74
Yes	All	16	2	0.0035	0.0033	0.66	0.76	1.34	1.05
Yes	All	16	4	0.0040	0.0048	1.19	1.50	2.80	1.66

Note: The input graphs are assumed to be directed, i.e., separate upward and downward metrics are used. We show the impact of enabling SSE, precomputing triangles (Pre. Trian.), multithreading (Thr. (#)), and customizing several metric pairs at once.

lower levels, as these are the levels where the optimization is most effective. However, precomputing more levels does not hurt if enough memory is available. We propose to determine the threshold level up to which triangles are precomputed based on the size of the available unoccupied memory. On modern server machines such as our benchmarking machine, there is enough memory to precompute all levels. The memory consumption is summarized in Table VII. However, note that precomputing all triangles is prohibitive in the game scenario, as less available memory should be expected.

9.5. Customization

In Table VIII, we report the times needed to compute a customized metric using the basic customization algorithm. A first observation is that on the road graphs, the KaHIP order leads to a faster customization, whereas on the game map, Metis dominates. Using all optimizations presented, we customize Europe in less than 1 second. When amortized,[7] we even achieve 415ms, which is only slightly greater than the nonamortized 347ms reported in Delling and Werneck [2013] for CRP. Note that their experiments were run on a different machine with a faster clock but 2×6 instead of 2×8 cores while using a turn-aware data structure, making an exact comparison difficult.

[7]We refer to a server scenario of multiple active users that require simultaneous customization, e.g., due to traffic updates.

Table IX. Detailed Basic Customization Performance on TheFrozenSea

Undirected	Metrics (#)	Metric Bits	SSE	Thr. (#)	Pre. Trian.	Customization Time (s)	Amortized Time (s)
No	2 (up & down)	32	No	1	None	7.88	7.88
Yes	1	32	No	1	None	6.65	6.65
Yes	4	32	No	1	None	9.36	2.34
Yes	4	32	Yes	1	None	8.51	2.13
Yes	8	16	Yes	1	None	8.52	1.06
Yes	8	16	Yes	2	None	5.00	0.63
Yes	8	16	Yes	2	All	2.16	0.27
Yes	8	16	Yes	16	All	0.63	0.08

Note: We show the impact of exploiting undirectedness, customizing several metrics at once, reducing the bitwidth of the metric, enabling SSE, multithreading (Thr. (#)), and precomputing triangles (Pre. Trian.). Note that the order in which improvements are investigated is different from Table VIII. Also note that results are based on the Metis order, as Table VIII shows that KaHIP is outperformed.

Previous works have tried to accelerate the preprocessing phase of the original two-phase Contraction Hierarchy to the point that it can be used in a similar scenario as our technique. A fast preprocessing phase can be viewed as form of customization phase. In Geisberger et al. [2012], a sequential preprocessing time of 451 seconds was reported. This compares best to our 9.5-second sequential customization time. Note that the machine on which the 451 seconds were measured is slower than our machine. However, the gap in performance is large enough to conclude that we achieve a significant speedup. Furthermore, Abraham et al. [2012b] report a Contraction Hierarchy preprocessing time of 2 minutes when parallelized on 12 cores. This compares best against our 415ms parallelized amortized customization time. Although the machine used in Abraham et al. [2012b] is slightly older and slower than our machine and the number of cores differs (12 vs. 16), again, the performance gap is large enough to safely conclude that a significant speedup is present. Besides these differences, both Contraction Hierarchy preprocessing experiments were only performed for travel time weights. To the best of our knowledge, nobody has been able to match performance achieved for travel time weights for less well behaved weights, such as travel distance. For example, Contraction Hierarchy preprocessing times reported in Geisberger et al. [2012] show at least a factor of 2 difference in performance on distance over travel time metric on any of the considered benchmarks. This contrasts with CCH, for which we can prove that basic customization and elimination tree query performance are completely independent of the metric considered.

Unfortunately, the optimizations illustrated in Table VIII are pretty far from what is possible with the hardware normally available in a game scenario. Regular PCs do not have 16 cores, and one cannot clutter up the whole RAM with several gigabytes of precomputed triangles. We therefore ran additional experiments with different parameters and report the results in Table IX. The experiments show that it is possible to fully customize TheFrozenSea in an amortized[8] time of 1.06 seconds without precomputing triangles or using multiple cores. However, a whole second is still too slow to be usable, as graphics, network, and game logic also require resources.

We therefore evaluated the time needed by partial updates as described in Section 7.7. We report our results in Table X, also for the road networks. The median, average, and maximum running times significantly differ. There are a few arcs that trigger a lot of subsequent changes, whereas for most arcs a weight change has nearly

[8]We refer to a multiplayer scenario, e.g., where fog of war requires player-specific simultaneous customization.

Table X. Partial Update Performance

		Arcs Removed from Queue			Partial Update Time (ms)		
		Med.	Avg.	Max.	Med.	Avg.	Max.
Karlsruhe	Metis	2	3.5	857	0.001	0.003	0.9
	KaHIP	3	3.7	466	0.001	0.002	1.0
TheFrozenSea	Metis	6	311.7	14,494	0.008	1.412	100.2
	KaHIP	6	343.1	19,417	0.008	1.490	164.6
Europe	Metis	2	10.2	14,188	0.005	0.052	134.6
	KaHIP	3	9.8	8,202	0.008	0.045	81.0

Note: We report time required in milliseconds and number of arcs changed for partial metric updates. We report median, average, and maximum over 10,000 runs. In each run, we change the upward and the downward weight of a single random arc in G to random values in $[0, 10^5]$. The metric is reset to the initial state between runs. Timings are sequential without SSE. No triangles were precomputed.

Table XI. Perfect Customization

		Karlsruhe		TheFrozenSea		Europe	
Thr. (#)	Pre. Trian.	Metis	KaHIP	Metis	KaHIP	Metis	KaHIP
1	None	0.15	0.13	30.54	33.76	67.01	32.96
16	None	0.03	0.02	3.26	4.37	14.41	5.47
1	All	0.05	0.05	8.95	12.51	23.93	10.75
16	All	0.01	0.01	1.93	2.29	3.50	2.35

Note: We report the time required to turn an initial metric into a perfect metric. Runtime is given in seconds, without the use of SSE.

no effect. The explanation is that highway arcs and choke point arcs are part of many shortest paths, and thus updating such an arc triggers significantly more changes. On the Europe road network, the maximum observed time for a partial CCH update (81.0ms) is similiar to CRP (73.77ms), but the average time is much lower for CCH (0.045ms) than for CRP (17.94ms) (compare to Delling et al. [2015]).

Finally, we report the running times of the perfect customization algorithm in Table XI. The required running time is about three times the running time needed by the basic customization. Recall that the basic customization in essence enumerates all lower triangles, i.e., it visits every triangle once, whereas the perfect customization also enumerates all intermediate and upper triangles, i.e., it visits every triangle three times.

9.6. Query Performance

We experimentally evaluated the running times of the query algorithms. For this, we ran 10^6 shortest path distance queries with the source and target vertices picked uniformly at random. The presented times are averaged running times on a single core without SSE.

In Table XII and XIII, we compare query performance. The MetDep+w variant uses a metric-dependent order and a nonperfect witness search in the spirit of Geisberger et al. [2012]. The details are described in Section 3. The Metis-w and KaHIP-w variants use a metric-independent order computed by Metis or KaHIP. Only a basic customization was performed, i.e., no witness search was performed. The Metis+w and KaHIP+w variants use the same metric-independent order, but a perfect customization followed by a perfect witness search was performed. We evaluate three query variants. The Basic variant uses a bidirectional variant of Dijkstra's algorithm with stopping criterion. The Stalling variant additionally uses the stall-on-demand optimization as described in Section 8.2. Finally, we also evaluate the elimination tree query and refer to it as Tree. This query requires the existence of an elimination tree of low depth and is therefore not

Table XII. Query Performance in Microseconds and the Search Space Visited, Averaged over 10^6 Queries with Source and Target Vertices Picked Uniformly at Random

Instance	Metric	Variant	Algorithm	Visited Upward Search Space		Stalling		Time
				Vertices (#)	Arcs (#)	Vertices (#)	Arcs (#)	(μs)
Karlsruhe	Travel Time	MetDep+w	Basic	81	370	—	—	17
			Stalling	43	182	167	227	16
		Metis-w	Basic	138	5,594	—	—	62
			Stalling	104	4,027	32	4,278	67
			Tree	164	6,579	—	—	33
		KaHIP-w	Basic	120	4,024	—	—	48
			Stalling	93	3,051	26	3,244	55
			Tree	143	4,723	—	—	25
		Metis+w	Basic	127	2,432	—	—	32
			Stalling	104	2,043	19	2,146	41
			Tree	164	2,882	—	—	17
		KaHIP+w	Basic	114	1,919	—	—	27
			Stalling	93	1,611	18	1,691	35
			Tree	143	2,198	—	—	14
	Distance	MetDep+w	Basic	208	1,978	—	—	57
			Stalling	70	559	46	759	35
		Metis-w	Basic	142	5,725	—	—	65
			Stalling	115	4,594	26	4,804	75
			Tree	164	6,579	—	—	33
		KaHIP-w	Basic	123	4,117	—	—	50
			Stalling	106	3,480	17	3,564	59
			Tree	143	4,723	—	—	26
		Metis+w	Basic	138	3,221	—	—	39
			Stalling	115	2,757	21	2,867	50
			Tree	164	3,604	—	—	21
		KaHIP+w	Basic	122	2,626	—	—	32
			Stalling	106	2,302	14	2,350	43
			Tree	143	2,956	—	—	17
TheFrozenSea	Map Distance	MetDep+w	Basic	1,199	12,692	—	—	539
			Stalling	319	3,460	197	4,345	286
		Metis-w	Basic	610	81,909	—	—	608
			Stalling	578	78,655	24	79,166	837
			Tree	676	92,144	—	—	317
		KaHIP-w	Basic	603	82,824	—	—	644
			Stalling	560	74,244	50	74,895	774
			Tree	674	89,567	—	—	316
		Metis+w	Basic	567	28,746	—	—	243
			Stalling	474	25,041	86	25,445	333
			Tree	676	31,883	—	—	120
		KaHIP+w	Basic	578	22,803	—	—	203
			Stalling	475	19,978	81	20,138	276
			Tree	674	24,670	—	—	106

Note: We use 'visited" to differentiate from the maximum reachable search space given in Table IV. If applicable, we additionally report the number of vertices stalled, as well as the number of arcs touched during the stalling test. Note that stalled vertices are not counted as visited. All reported vertex and arc counts only refer to the forward search. We evaluate several algorithmic variants. Each variant is composed of an input graph, a contraction order, and whether a witness search is used. "+w" means that a witness search is used, whereas "-w" means that no witness search is used. "MetDep+w" corresponds to the original Contraction Hierarchies. The metrics used for "-w" are directed and maximum. Results for Karlsruhe and the TheFrozenSea instances are in this table. Table XIII contains results for Europe.

Table XIII. Query Performance (Continuation of Table XII)

Instance	Metric	Variant	Algorithm	Visited Upward Search Space		Stalling		Time
				Vertices (#)	Arcs (#)	Vertices (#)	Arcs (#)	(μs)
Europe	Travel Time	MetDep+w	Basic	546	3,623	—	—	283
			Stalling	113	668	75	911	107
		Metis-w	Basic	1,126	405,367	—	—	2,838
			Stalling	719	241,820	398	268,499	2,602
			Tree	1,291	464,956	—	—	1,496
		KaHIP-w	Basic	581	107,297	—	—	810
			Stalling	418	75,694	152	77,871	857
			Tree	652	117,406	—	—	413
		Metis+w	Basic	1,026	110,590	—	—	731
			Stalling	716	83,047	271	89,444	951
			Tree	1,291	126,403	—	—	398
		KaHIP+w	Basic	549	41,410	—	—	305
			Stalling	418	33,078	117	34,614	425
			Tree	652	45,587	—	—	161
	Distance	MetDep+w	Basic	3,653	104,548	—	—	2,662
			Stalling	286	7,124	426	11,500	540
		Metis-w	Basic	1,128	410,985	—	—	3,087
			Stalling	831	291,545	293	308,632	3,128
			Tree	1,291	464,956	—	—	1,520
		KaHIP-w	Basic	584	108,039	—	—	867
			Stalling	468	85,422	113	87,315	1,000
			Tree	652	117,406	—	—	426
		Metis+w	Basic	1,085	157,400	—	—	1,075
			Stalling	823	124,472	247	127,523	1,400
			Tree	1,291	177,513	—	—	557
		KaHIP+w	Basic	575	56,386	—	—	425
			Stalling	467	46,657	101	47,920	578
			Tree	652	61,714	—	—	214

available for metric-dependent orders. We ran our experiments on all three of our main benchmark instances. Experiments on additional instances are available in Section 11. For both road graphs, we evaluate the travel time and distance variants. We report the average running time needed to perform a distance query, i.e., we do not unpack the paths. We further report the average number of "visited" vertices in the forward search. For the "basic" and "stalling" queries, these are the vertices removed from the queue. For the "tree" query, we regard every ancestor as "visited." The numbers for the backward search are analogous and therefore not reported. We also report the average number of arcs relaxed in forward search of each query variant. Finally, we also report the average number of vertices stalled and the average number of arcs that need to be tested in the stalling test. Note that a stalled vertex is not counted as "visited."

An important detail necessary to reproduce these results consists of reordering the vertex IDs according to the contraction order. Preliminary experiments showed that this reordering results in better cache behavior and a speedup of about 2 to 3 because much query time is spent on the top-most clique, and this order ensures that these vertices appear adjacent in memory.

As already observed by the original Contraction Hierarchy authors, we confirm that the stall-on-demand heuristic improves running times by a factor of 2 to 5 compared to the basic algorithm for Greedy+w. Interestingly, this is not the case with any variant

using a metric-independent order. This can be explained by the density of the search spaces. Whereas the number of vertices in the search spaces are comparable between metric-independent orders and metric-dependent order, the number of arcs are not comparable and thus metric-independent search spaces are denser. As a consequence, we need to test significantly more arcs in the stalling test, which makes the test more expensive, and therefore the additional time spent in the test does not make up for the time economized in the actual search. We thus conclude that stall-on-demand is not useful when using metric-independent orders.

The comparison between the elimination tree query and the basic query is quite interesting. The elimination tree query always explores the whole search space. In contrast to the basic query, it does not have a stopping criterion. However, the elimination tree query does not require a priority queue. It thus performs less work per vertex and arc than the basic query. Our experiments show that the basic query always explores large parts of the search space regardless of the stopping criterion. The elimination tree query therefore does not visit significantly more vertices. A consequence of this effect is that the time spent in the priority queue outweighs the additional time necessary to explore the remainder of the search space. The elimination tree query is therefore always the fastest among the three query types when using metric-independent orders. Combining a perfect witness search with the elimination tree query results in the fastest queries for metric-independent orders. However, the perfect witness search results in three times higher customization times. Whether it is superior therefore depends on the specific application and the specific trade-off between customization and query running time needed.

The orders computed by KaHIP are nearly always significantly better than those produced by Metis. However, significantly more running time must be invested in the preprocessing phase to obtain these better orders. It therefore depends on the situation as to which order is better. If the running time of the preprocessing phase is relevant, then Metis seems to strike a very good balance between all criteria. However, if the graph topology is fixed, as we expect it to be, then the flexibility gained by using Metis is not worth the price. Interestingly, on the game map KaHIP and Metis seem to be very close in terms of search space size. The difference is only apparent when using the perfect customization. For a setup with basic customization, the two orders are nearly indistinguishable.

On travel time, the metric-dependent orders outperform the metric-independent orders. However, it is very interesting how close the query times actually are. On the Europe graph, the basic query visits about the same number of vertices, regardless of whether a metric-dependent or KaHIP order is used. The real difference lies in the number of arcs that need to be relaxed. This number is significantly higher with metric-independent orders. However, the effect this has on the actual running times is comparatively slim. Using KaHIP without perfect witness search results in an elimination tree query that is only about four times slower than using the stalling query combined with metric-dependent orders. If a perfect witness search is used then, the gap is below a factor of 2. Further, the metric-dependent orders only win because of the stall-on-demand optimization. The KaHIP order combined with perfect customization *outperforms* the basic query combined with metric-dependent orders.

It is well-known that metric-dependent Contraction Hierarchies work significantly better with the travel-time metric than with other less well behaved metrics such as the geographic distance. For such metrics, the KaHIP order outperforms the metric-dependent orders. For example, the basic query with perfect customization visits fewer vertices and fewer arcs. This is very surprising, especially considering that the metric-dependent orders that we computed are better than those reported in Geisberger et al. [2012], i.e., the gap with respect to the original implementation is even larger. However, combining the stalling query with metric-dependent orders

Table XIV. Detailed Elimination Tree Performance without Perfect Witness Search

			Distance Query				Path	
			LCA (μs)	Reset (μs)	Arc relax (μs)	Total (μs)	Unpack (μs)	Length (vert.)
Karlsruhe	Travel time	Metis	0.6	0.8	31.3	33.0	20.5	189.6
		KaHIP	0.6	1.4	23.1	25.2	18.6	
	Distance	Metis	0.6	0.8	31.5	33.2	27.4	249.4
		KaHIP	0.6	1.4	23.5	25.7	24.7	
TheFrozenSea	Map distance	Metis	2.7	3.1	310.1	316.5	220.0	596.3
		KaHIP	3.0	3.2	308.7	315.5	270.8	
Europe	Travel time	Metis	4.6	19.0	1,471.2	1,496.3	323.9	1,390.6
		KaHIP	3.4	9.9	399.4	413.3	252.7	
	Distance	Metis	4.7	19.0	1,494.5	1,519.9	608.8	3,111.0
		KaHIP	3.6	10.0	411.6	425.8	524.1	

Note: We report running time in microseconds for the elimination tree–based query algorithms. We report the time needed to compute the LCA, the time needed to reset the tentative distances, the time needed to relax the arcs, the total time of a distance query, and the time needed for full path unpacking as well as the average number of vertices on such a path that is metric dependent.

yields the smallest number of visited vertices and relaxed arcs. Unfortunately, combining the stalling query with metric-independent orders does not yield the same benefit and even makes the query running times worse. Fortunately, the metric-independent orders can be combined with the elimination tree query. As a result, the fastest variant is the combination of KaHIP order, perfect witness search, and elimination tree query, which is more than a factor of 2 faster than stalling with the metric-dependent order. Interestingly, the latter is even beaten when no perfect witness search is performed but with a significantly lower margin.

A huge advantage of metric-independent orders compared to metric-dependent orders is that the resulting Contraction Hierarchy performs equally well regardless of the weights of the input graph. The combination of metric-independent order, elimination tree query, and basic customization results in a setup where the order in which the vertices are visited and the order in which the arcs are relaxed during the query execution do not even depend on the weights of the input graph. It is thus impossible to construct a metric, where this setup performs badly. This contrasts with the Contraction Hierarchy of Geisberger et al. [2012], whose performance varies significantly depending on the input metric.

In Table XIV, we give a more in-depth experimental analysis of the elimination tree query algorithm without perfect witness search. We break the running times down into the time needed to compute the least common ancestor (LCA), the time needed to reset the tentative distances, and the time needed to relax all arcs. We further report the total distance query time, which is in essence the sum of the former three. We additionally report the time needed to unpack the full path. Our experiments show that the arc-relaxation phase clearly dominates the running times. It is therefore not useful to further optimize the LCA computation or to accelerate tentative distance resetting using, e.g., timestamps. We only report path unpacking performance without precomputed lower triangles. Using them would result in a further speedup with a similar speed-memory trade-off, as already discussed for customization.

9.7. Comparison with Related Work

We conclude our experimental analysis on the DIMACS-Europe road network with a final comparison of related techniques, as shown in Table XV. For Contraction Hierarchies, we report results based on implementations by Geisberger et al. [2012], Delling et al. [2015], and ourselves, covering different trade-offs in terms of preprocessing versus query speed. More precisely, we observe that our own Contraction Hierarchy

Table XV. Comparison with Related Work on the DIMACS-Europe Instance with the Travel Time and Distance Metrics

					Metric-Dep. Prepro.		Queries	
Algorithm	Implementation	Machine	Metric	Turn-Aware	Time (s) (Threads (#)		Search Space (Vertices (#))	Time (μs) (Threads (#))
Contraction Hierarchy	Geisberger et al. [2012]	Opt 270	Time	○	1,809	(1)	356	152 (1)
Contraction Hierarchy	Geisberger et al. [2012]	Opt 270	Dist	○	5,723	(1)	1,582	1,940 (1)
Contraction Hierarchy	Geisberger et al. [2012]	E5-2670	Time	○	1,075.88	(1)	353	91 (1)
Contraction Hierarchy	Geisberger et al. [2012]	E5-2670	Dist	○	3,547.44	(1)	1,714	1,135 (1)
Contraction Hierarchy	Our	E5-2670	Time	○	813.53	(1)	375	107 (1)
Contraction Hierarchy	our	E5-2670	Dist	○	9,390.32	(1)	1,422	540 (1)
Contraction Hierarchy	Delling et al. [2015]	X5680	Time	○	109	(12)	280	110 (1)
Contraction Hierarchy	Delling et al. [2015]	X5680	Dist	○	726	(12)	858	870 (1)
CRP	Delling et al. [2015]	X5680	Time	•	0.37	(12)	2,766	1,650 (1)
CRP	Delling et al. [2015]	X5680	Dist	•	0.37	(12)	2,942	1,910 (1)
CRP	Delling et al. [2011a]	i7 920	Time	○	4.7	(4)	3,828	720 (2)
CRP	Delling et al. [2011a]	i7 920	Dist	○	4.7	(4)	4,033	790 (2)
CCH	Our	E5-2670	Time	○	0.74	(16)	1,303	413 (1)
CCH	Our	E5-2670	Dist	○	0.74	(16)	1,303	426 (1)
CCH+a	Our	E5-2670	Time	○	0.42	(16)	1,303	416 (1)
CCH+a	Our	E5-2670	Dist	○	0.42	(16)	1,303	421 (1)
CCH+w	Our	E5-2670	Time	○	2.35	(16)	1,303	161 (1)
CCH+w	Our	E5-2670	Dist	○	2.35	(16)	1,303	214 (1)

Note: We compare our approaches—CCH, CCH with amortized customization (CCH+a), and CCH with perfect witness search (CCH+w)—to different CRP and Contraction Hierarchy implementations from the literature. We report performance of the metric-dependent fraction of overall preprocessing, i.e., vertex ordering and contraction time for Contraction Hierarchy and customization time for CRP and CCH. We further report average query search space, including stalled vertices for Contraction Hierarchy (which might not be included in the Contraction Hierarchy figures taken from Delling et al. [2015]). We finally report running time in microseconds. If parallelized, the number of threads used is noted in parentheses. Since the Contraction Hierarchy performance in Geisberger et al. [2012] was evaluated on a 10-year-old machine (AMD Opteron 270), we obtained the source code and reran experiments on our hardware (Intel Xeon E5-2670) for better comparability. In addition, note that the latest CRP implementation by Delling et al. [2015], evaluated on an Intel Xeon X5680, is turn-aware (•), i.e., it uses turn tables (set to zero in the reported experiments); we therefore additionally take results from Delling et al. [2011a] obtained on an Intel Core-i7 920, which uses a turn-unaware implementation but parallelizes queries.

implementation (used for detailed analysis and comparison in Sections 9.1 through 9.6) has slightly slower queries on the travel time metric but a factor of 2.1 faster queries on the distance metric, at the cost of higher preprocessing time. Recall from Section 3 that we employ a different vertex priority function and no lazy updates. For CRP, we report results from Delling et al. [2011a, 2015].

Traditional, metric-dependent Contraction Hierarchy offers the fastest query time (91μs on our machine), but it incurs substantial metric-dependent preprocessing costs, even when parallelized (109 seconds, 12 cores). Furthermore, Contraction Hierarchy performance is very sensible regarding the metrics used: for the distance metric, preprocessing time increases by factor of 3.2 to 11.5 and query time by factor of 4.9 to 12.8.

In contrast to traditional Contraction Hierarchy, CCHs by design achieve a performance trade-off with much lower metric-dependent preprocessing costs, similar to CRP. Accounting for differences in hardware, CCH basic customization time is about a factor of 2 to 3 slower than CRP customization, but still well below a second. On the other hand, CCH query performance is factor of 2 to 4 faster than CRP, both in terms of search space as well as query time (even when accounting for differences due to turn-aware implementation and hardware used). Overall, CCH is more robust with respect to the metric than CRP. By design, CCH customization processes the same sequence of lower triangles for any metric, whereas the CCH elimination tree query (given a fixed source and target) processes the same sequence of vertices and arcs for any metric—unless, of course, we employ perfect witness search (CCH+w) (see later discussion).

Table XVI. Time in Seconds to Compute Minimum Degree (MinDeg) and Minimum Shortcut (MinArc) Orders

	Karlsruhe	TheFrozenSea	Europe
MinDeg	1.7	67	250
MinArc	2.1	6,907	30,220

The CRP implementation of Delling et al. [2015] uses SSE to achieve its customization time of 0.37 seconds. In a server scenario where customization is run for many users concurrently, e.g., to customize a stream of traffic updates for all active users at once, we propose amortizing the triangle enumeration time as described in Section 7.6. By using SSE and processing metrics for four users at once, this amortized customization (CCH+a, 0.42 seconds) can almost close the gap to CRP customization performance. Refer to Table VIII for other configurations.

Most interestingly, in terms of query performance on travel distance, CCH outperforms even the best Contraction Hierarchy result. For even better CCH query performance, we may employ perfect customization and witness search (CCH+w). It increases customization time by factor of 3.2 (enumerating all lower, intermediate, and upper triangles), but enables a CCH query variant that, while still visiting all vertices in the elimination tree, needs to consider far fewer arcs (compare to Table XIII). Therefore, CCH+w further improves CCH query performance by a factor of 1.9 for the distance metric and a factor of 2.6 for the time metric. With 161μs for travel time, CCH+w query performance is almost as good as the best Contraction Hierarchy result of 91μs.

10. FURTHER METRIC-INDEPENDENT ORDERING STRATEGIES

So far, we have only discussed metric-independent orders based on nested dissection. For completeness, we consider two other metric-independent orders in the following.

In the context of sparse matrix factorization, a common approach is the *minimum degree heuristic* (MinDeg). To the best of our knowledge, the first variant of this ordering heuristic was described in Tinney and Walker [1967], but we refer to George and Liu [1989] for more details. The basic idea is simple: iteratively contract a vertex with a minimum degree in the core. Note the difference to sorting vertices by degree in the input graph, which does not work well on road networks [Delling et al. 2014].

A variant of the minimum degree heuristic was proposed in Delling et al. [2015]. The idea is also simple: iteratively contract a vertex that adds the least number of arcs to the chordal supergraph (MinArc), using degree in the core to break ties. However, Delling et al. [2015] already observed that MinArc orders can be improved when augmented with partitioning information from what they refer to as guidance levels. Their reported experimental results are only with respect to these hybrid orders.

We implemented both MinDeg and MinArc in the straightforward way using a priority queue of vertices ordered by the respective weighting function. Table XVI shows the resulting order computation times on our three main instances. Note that at least for MinDeg, more sophisticated strategies exist [George and Liu 1989] that might be faster. Nonetheless, MinArc is significantly slower than MinDeg because it simulates the contraction of every vertex in the graph, including those that yield a high number of shortcuts but are only contracted in the end, once their degree has already decreased due to the graph shrinking.

More importantly, Table XVII[9] reports performance indicators for both MinDeg and MinArc in comparison to Metis and KaHIP. Recall that the number of triangles determines customization running times, the number of arcs in the Contraction Hierarchy

[9]The KaHIP and Metis numbers slightly differ from those in Table IV, where they were only sampled over 1,000 random search spaces.

Table XVII. Further Ordering Strategies: Minimum Degree (MinDeg) and Minimum Shortcut (MinArc)

Graph	Order	Triangles (#) [$\cdot 10^6$]	Upper Treewidth Bound	Arcs in Contraction Hierarchy (#) [$\cdot 10^3$]	Search Space Vertices (#) Avg.	Vertices (#) Max.	Arcs (#) [$\cdot 10^3$] Avg.	Arcs (#) [$\cdot 10^3$] Max.
Karlsruhe	MinDeg	2.37	94	423	244.6	369	11.9	16.2
	MinArc	1.63	75	393	222.4	386	9.2	16.0
	Metis	2.59	92	478	163.5	211	6.5	10.0
	KaHIP	2.21	72	528	142.2	201	4.7	7.9
TheFrozenSea	MinDeg	1,123	500	25,698	1,462.1	2,351	351	502
	MinArc	769	303	22,554	1,192.1	2,034	200	336
	Metis	601	282	21,067	675.6	858	92	135
	KaHIP	864	287	25,100	676.7	949	90	146
Europe	MinDeg	1,800	938	64,313	2,348.0	3,719	1,052	1,494
	MinArc	767	599	56,948	1,815.4	3,256	552	889
	Metis	1,409	876	70,070	1,283.5	2,017	462	967
	KaHIP	578	479	73,920	638.6	1,224	114	284

Table XVIII. Size of the DIMACS-Europe Instance Compared to the OSM-Europe Instance

Instance	Vertices (#)	Arcs (#)	Degree 1 Vertices (#)	Degree 2 Vertices (#)	Degree >2 Vertices (#)
DIMACS-Eur	18M	42M	4M (22%)	2M (11%)	11M (61%)
OSM-Eur	174M	348M	8M (5%)	143M (82%)	23M (13%)

is proportional to the memory consumption if precomputed triangles are not used, and the number of arcs in the search space gives a good indication of query performance. MinArc nearly always dominates MinDeg with respect to every criterion except computation time. Although both can be computed faster than the KaHIP-based orders, Metis is still fastest. Similarly, both MinArc and MinDeg result in lower memory consumption than KaHIP-based orders, but not lower than Metis on the TheFrozenSea instance. Upper treewidth bounds from KaHIP-based orders are consistently better than from MinDeg or MinArc.

Note, however, that at least for our work, customization and query performance are most important. In both aspects, MinDeg and MinArc are clearly dominated by Metis on the large game map and by KaHIP-based order on the large road network. Therefore, we did not further consider MinDeg and MinArc orderings in our experiments.

11. FURTHER INSTANCES

11.1. OpenStreetMap-Based Road Graphs

OSM is a very popular collaborative effort to create a map of the world. From this huge data source, very large road graphs can be extracted, which are very detailed depending on the exact region considered. Using the data provided by GeoFabrik[10] and the tools provided by OSRM,[11] we extracted a road graph of Europe and report its size in Table XVIII. The exact graph is available in DIMACS format on our Web site.[12] The geographic region corresponding to the graph is depicted in Figure 13. Note that compared to the DIMACS-Europe graph, our OSM-Europe graph also contains Eastern Europe and Turkey. The graph's east border ends at the east border of Turkey and then goes upward, cutting through Russia. On the other hand, the DIMACS-Europe graph stops at the Germany–Poland border.

[10] http://download.geofabrik.de/.
[11] http://project-osrm.org/.
[12] http://i11/www.iti.kit.edu/resources/roadgraphs.php.

Fig. 13. Comparison of European instances from DIMACS and OSM.

Table XIX. Contraction Hierarchy Sizes for OSM-Europe

	Order	Vertices	Arcs
Input Graph Size	—	174M	348M
Search Graph Size	Metis	174M	400M
	KaHIP	174M	434M
Avg. Search Space	Metis	1,312	495,930
	KaHIP	678	119,295

Note: The search space sizes were obtained by randomly sampling 10,000 vertices uniformly at random.

At first glance, the DIMACS-Europe graph looks drastically smaller, at least in terms of vertex count. However, this is very misleading. A peculiarity of OSM is that the road graphs have a huge number of degree-2 vertices. These vertices are used to encode the curvature of a road. This information is needed to correctly represent a road graph on a map, but not necessarily for routing. However, most other data sources, including the one on which the DIMACS graph is based, encode this information as arc attributes and thus have fewer degree-2 vertices. Accelerating shortest path computations on graphs with a huge number of vertices of degree 1 or 2 is significantly easier relative to the graph size. One reason is that Dijkstra's algorithm cannot exploit the abundance of low-degree vertices. Dijkstra's algorithm with stopping criterion needs an average of 27 seconds for an *st*-query, with s and t picked uniformly at random on the OSM-Europe graph. This contrasts with the DIMACS-Europe graph, where only 1.6 seconds are needed. A slower baseline obviously leads to larger speedups. Table XVIII shows that the difference between the two Europe graphs in terms of vertex count is significantly smaller when discarding degree-1 and degree-2 vertices. In fact, relative to their geographical region's area, the two graphs seem to be approximately comparable in size.

We computed contraction orders for OSM-Europe. The sizes of the resulting Contraction Hierarchies are reported in Table XIX. These sizes can be compared to the "undirected" numbers of Table IV. We did not perform experiments with a perfect witness search. Metis ordered the vertices within 29 minutes, whereas the KaHIP-based ordering algorithm needed slightly less than 3 weeks. However, as already discussed in detail, we did not optimize the latter for speed, and therefore one must *not* conclude from this experiment that KaHIP is slow. The Contraction Hierarchies for OSM-based graphs are significantly larger. The DIMACS-Europe Contraction Hierarchy only contains 70M arcs for Metis, whereas the OSM-Europe Contraction Hierarchy contains 400M arcs for Metis. However, this is due to the huge amount of low-degree vertices in the input. On the DIMACS graph, the size increase compared to the number of input arcs is $70\,\mathrm{M}/42\,\mathrm{M} = 1.67$, whereas for the OSM-based graph, the size increase is only $400\,\mathrm{M}/348\,\mathrm{M} = 1.15$. This effect can be explained by considering what happens when

Table XX. Customization Performance on OSM-Europe

	Thr. (#)	Triangle Space (GB)	Customization Time (s)
Metis	1	—	43.1
	16	—	5.3
	1	16.0	17.3
	16	16.0	2.1
KaHIP	1	—	30.6
	16	—	3.4
	1	7.2	11.4
	16	7.2	1.7

Note: We vary the number of threads and whether precomputed triangles are used. SSE is enabled, running times are nonamortized, and no perfect customization was performed.

Table XXI. Query Performance on OSM Europe Averaged over 10,000 Random *st*-Pairs Chosen Uniformly at Random

		Query Time (ms)
Dijkstra	Metis	3.7
Based	KaHIP	1.0
Elimination	Metis	1.7
tree	KaHIP	0.5

contracting a graph consisting of a single path. In the input graph, every vertex, except the endpoints, has two outgoing arcs—one in each direction. As long as the endpoints are contracted last, every vertex, except the endpoints, in the resulting Contraction Hierarchy search graph also has degree 2. Thus, there is no size increase. As the OSM-based graph has many degree-2 vertices, this effect dominates and explains the comparatively small size increase. The search space sizes are nearly identical. For example, the KaHIP search space contains 117K arcs for the DIMACS-Europe graph, whereas it contains 119K arcs for the OSM-Europe graph. This effect is explained by the fact that both data sources correspond to almost the same geographical region. The mountains and rivers are thus in the same locations, and the number of roads through these geographic obstacles are the same in both graphs, i.e., both graphs have very similar recursive separators. The small size increase is explained by the fact that the OSM-based graph also includes Eastern Europe. The customization times are reported in Table XX. As the OSM-based graph has more arcs, the customization times are higher on that graph. On the DIMACS-Europe graph, 0.61 seconds are needed, whereas 1.7 seconds are needed on OSM-Europe for the KaHIP order and 16 threads, which is a surprisingly small gap considering the differences in input sizes. Eliminating the degree-2 vertices from the input should further narrow this gap. As the search space sizes are very similar, it is not surprising that the query running times reported in Table XXI are nearly identical.

11.2. Further DIMACS Instances

During the DIMACS challenge on shortest path [Demetrescu et al. 2009], several benchmark instances were made available. Among them is the Europe instance used throughout our in-depth experiments in previous sections. Besides this instance, a set of graphs representing the road network of the USA was published. In Table XXII, we report experiments for these additional DIMACS road graphs. Other than the DIMACS-Europe instance, these USA instances originate from the U.S. Census Bureau. Note that the USA instances have some known data quality issues: the graphs are generally undirected (no one-way streets), and highways are sometimes not connected at state borders. The DIMACS-Europe comes from another data source and does not have these limitations. This is why we focus on the Europe instance in the main part of our evaluation.

However, as the graphs are undirected, we can evaluate the impact that using a single undirected metric has on customization running times compared to using two directed metrics (as used on DIMACS-Europe). Experiments using a single metric are

Table XXII. Instance Sizes and Experimental Results for the Additional DIMACS Road Graphs

Graph	Vertices [$\cdot 10^3$]	Arcs [$\cdot 10^3$]	Contraction Hierarchy Arcs [$\cdot 10^3$]	Customization (ms) 1 Thread Uni	1 Thread Bi	16 Threads Uni	16 Threads Bi	Dijkstra (μs)	Contraction Hierarchy Query (μs) Uni	Bi
NY	264	730	1,547	46	52	11	12	16,303	34	34
BAY	321	795	1,334	29	39	7	8	17,964	20	20
COL	436	1,042	1,692	40	51	12	12	25,505	35	41
FLA	1,070	2,688	4,239	93	117	25	32	63,497	30	26
NW	1,208	2,821	4,266	88	110	24	31	73,045	27	27
NE	1,524	3,868	6,871	195	255	54	57	96,628	68	64
CAL	1,891	4,630	7,587	195	250	61	64	114,047	43	43
LKS	2,758	6,795	12,829	478	646	75	87	175,084	138	149
E	3,599	8,708	14,169	395	514	85	96	233,511	86	88
W	6,262	15,120	24,115	682	894	121	132	425,244	82	84
CTR	14,082	33,867	57,222	2,656	3,592	392	416	1,050,314	276	285
USA	23,947	57,709	97,902	3,617	7,184	698	979	1,883,053	264	286

Note: The instances are weighted by travel time. They are undirected, i.e., no one-way streets exist and the weight of an arc corresponds to its backward arc's weight. The Uni numbers exploit this and have one weight per Contraction Hierarchy arc, whereas the Bi numbers do not and have two weights per Contraction Hierarchy arc. We report the number of vertices, directed arcs after removing multiarcs, arcs in the Contraction Hierarchy search graph, the time needed to do a full nonamortized customization with 1 and 16 threads, the average running time of Dijkstra's algorithm with stopping criterion, and the average running time of an elimination tree distance query. The order were computed with KaHIP. We averaged for 10,000 queries, where s and t were uniformly at random. We only do a basic customization and no perfect customization.

Table XXIII. Instance Sizes and Experimental Results for the Additional Game-Based Graphs

Graph	Vertices [$\cdot 10^3$]	Edges [$\cdot 10^3$]	Contraction Hierarchy Arcs [$\cdot 10^3$]	Metis [s]	Customize (ms) 1 Thread	Customize (ms) 16 Threads	Dijkstra (μs)	Contraction Hierarchy Query (μs)
16room_005	231	838	2,959	1.196	359	41	10,913	15
AcrosstheCape	392	1,534	12,632	2.452	4,789	563	23,609	239
blastedlands	131	507	3,740	0.896	1,250	144	6,075	304
maze512-4-3	209	686	1,641	0.996	138	35	7,810	6
ost100d	137	531	3,722	0.880	1,076	124	5,607	116
random512-35-8	161	421	1,805	0.988	469	39	7,422	223
random512-40-8	114	280	797	0.684	115	16	4,856	41

Note: We report the number of vertices, undirected edges, arcs in the Contraction Hierarchy search graph, the running time Metis needed to compute the order, the time needed to do a full nonamortized customization with 1 and 16 threads using an undirected weight, the average running time of Dijkstra's algorithm with stopping criterion and the average running time of an elimination tree distance query. We averaged for 10,000 queries where s and t were uniformly at random.

marked with "Uni" in the table, whereas the experiments with two metrics are marked with "Bi." The query running times are very similar. This is not surprising, as the number of relaxed arcs does not depend on whether one or two weights are used. For larger graphs, there is a slightly larger difference in running times. We believe that this is a cache effect. As the Bi variant has twice as many weights, less arcs fit into the L3 cache. For the smaller graphs, this effect does not occur because the higher Contraction Hierarchy levels occupy less memory than the cache's size, and thus doubling the memory consumption is nonproblematic.

The difference in customization times between the two variants is larger. The number of enumerated triangles is the same, but twice as many instructions are executed per triangle. We would thus expect a factor of 2 difference in the customization running times. However, this factor is only observed on the largest instance. On all smaller

instances, the gap is significantly smaller. Again, this is most likely the result of cache effects.

11.3. Further Game Instances

Besides our main game benchmark instance TheFrozenSea, the benchmark dataset of Sturtevant [2012] contains a large variety of different game maps. To demonstrate that our technique also works on other game maps, we ran our experiments on a selection of different graphs from the set. "16room_005" is a synthetic map with many rooms in grid shape that are connected through small doors. "AcrosstheCape" is another Star Craft map that is sometimes used as benchmark instance. "blastedlands" originates from WarCraft 3 and is the largest map in that set in terms of vertices. "maze512-4-3" is a synthetic map that consists of a random maze with corridors that are four fields wide. "ost100d" is the largest map from the Dragon Age Origins map set. "random512-35-8" and "random512-40-8" are synthetic maps that contain random obstacles. The difference between them is the amount of space covered by obstacles. The Web site[13] from which the data originates includes pictures depicting each instance.

All experiments were run using a single undirected metric with 32 bits per weight. The customization running times are nonamortized. We did not perform experiments with a perfect witness search.

All additional game-based instances have fewer vertices than TheFrozenSea. Further, the Contraction Hierarchy query is the slowest on TheFrozenSea at 316μs. Interestingly, a full customization is slower on AcrosstheCape than on TheFrozenSea by about a factor of 2. This is most likely due to slight differences in the structures of the maps. However, we believe that it is safe to conclude from the experiments that our technique works across a wide range of maps.

12. CONCLUSIONS

We have extended Contraction Hierarchies to a three-phase customization approach and demonstrated in an extensive experimental evaluation that our CCHs approach is practicable and efficient not only on real-world road graphs but also on game maps. We have proposed new algorithms that improve on the state of the art for nearly all stages of the toolchain: using our contraction graph data structure, a metric-independent Contraction Hierarchy can be constructed faster than with the established approach based on dynamic arrays. We have shown that the customization phase is essentially a triangle enumeration algorithm. We have provided two variants of the customization. The basic variant yields faster customization running times, whereas perfect customization and witness search compute Contraction Hierarchies with a provable minimum number of shortcuts within seconds given a metric-independent vertex order. We proposed an elimination tree–based query that unlike previous approaches is not based on Dijkstra's algorithm and thus does not use a priority queue. This results in significantly lower overhead per visited arc, enabling faster queries. Finally, our extensive experimental analysis shed some light onto the inner workings of Contraction Hierarchies.

12.1. Future Work

Good separators are the foundation of CCHs: finding better separators directly improves both customization as well as query performance. For our purposes, the time required to compute good separators was of no primary concern (we do it once per graph). Hence, *our* nested dissection implementation *based* on KaHIP [Sanders and

[13] http://www.movingai.com/benchmarks/.

Schulz 2013] was not optimized for speed but rather to demonstrate that good separators exist.

For some applications, this may be too slow. However, since we performed the experiments reported in this article, significant improvements have been made in this domain. The works of Wegner [2014] lay the foundations of a well-implemented KaHIP-based algorithm. Schild and Sommer [2015] introduce a new and surprisingly simple road graph bisection algorithm called *Inertial Flow*. FlowCutter [Hamann and Strasser 2015], available as a preprint, computes the best metric-independent contraction orders that we have observed so far, much faster than the nested dissection implementation presented in this work. This results in decreased query and customization times and reduced memory consumption for CCHs.

Our experiments suggest that even metric-dependent Contraction Hierarchies implicitly exploit the existence of small graph cuts. However, this does not seem to be the only exploited feature, as metric-independent orders behave differently when it comes to details. For example, the stall-on-demand optimization only works with metric-dependent orders, for both the travel time and distance metrics, but not for the metric-independent orders. Further investigations into this effect might yield additional valuable insights. A good starting point could be the theoretical works of Abraham et al. [2010].

Further investigation into algorithms explicitly exploiting treewidth [Chaudhuri and Zaroliagis 2000; Planken et al. 2012] seems promising. Additionally, determining the precise treewidth of road networks could prove useful.

In practice, route planning services consider several additional real-world constraints beyond the scope of this article. For example, these include turn costs and restrictions, historic traffic data for rush hours, and range constraints due to limited electric vehicle batteries. For turns, we assume that CCHs are well applicable, as the size of graph cuts cannot grow arbitrarily when turn expanding the road graph. We are interested in further in-depth experimental analysis of such scenarios. Moreover, although we considered point-to-point queries in this article, other types are also of practical importance, e.g., one-to-all, many-to-many, and k-nearest-neighbor queries. These have been evaluated for traditional (greedy) Contraction Hierarchy and related hierarchical techniques by Abraham et al. [2012a], Delling et al. [2011c, 2013], Geisberger et al. [2010, 2012], and Knopp et al. [2007]. Since CCH still yields a Contraction Hierarchy search space, algorithmic results directly carry over. Similarly, CCH could be applied to the computation of alternative routes as in Abraham et al. [2013b] and Kobitzsch et al. [2013].

ACKNOWLEDGMENTS

We would like to thank Ignaz Rutter and Tim Zeitz for very inspiring conversations.

REFERENCES

Ittai Abraham, Daniel Delling, Amos Fiat, Andrew V. Goldberg, and Renato F. Werneck. 2012a. HLDB: Location-based services in databases. In *Proceedings of the 20th ACM SIGSPATIAL International Symposium on Advances in Geographic Information Systems (GIS'12)*. ACM, New York, NY, 339–348.

Ittai Abraham, Daniel Delling, Amos Fiat, Andrew V. Goldberg, and Renato F. Werneck. 2013a. Highway dimension and provably efficient shortest path algorithms. In *Proceedings of the 21st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'10)*. 782–793.

Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. 2012b. Hierarchical hub labelings for shortest paths. In *Algorithms—ESA 2012*. Lecture Notes in Computer Science, Vol. 7501. Springer, 24–35.

Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. 2013b. Alternative routes in road networks. *ACM Journal of Experimental Algorithmics* 18, 1, 1–17.

Ittai Abraham, Amos Fiat, Andrew V. Goldberg, and Renato F. Werneck. 2010. Highway dimension, shortest paths, and provably efficient algorithms. In *Proceedings of the 21st Annual ACM–SIAM Symposium on Discrete Algorithms (SODA'10)*. 782–793.

Hannah Bast, Daniel Delling, Andrew V. Goldberg, Matthias Müller–Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F. Werneck. 2015. *Route Planning in Transportation Networks*. Technical Report. arXiv:1504.05140. http://arxiv.org/abs/1504.05140.

Reinhard Bauer, Tobias Columbus, Bastian Katz, Marcus Krug, and Dorothea Wagner. 2010. Preprocessing speed-up techniques is hard. In *Algorithms and Complexity*. Lecture Notes in Computer Science, Vol. 6078. Springer, 259–370.

Reinhard Bauer, Tobias Columbus, Ignaz Rutter, and Dorothea Wagner. 2013. Search-space size in contraction hierarchies In *Automata, Languages, and Programming*. Lecture Notes in Computer Science, Vol. 7965. Springer, 93–104.

Reinhard Bauer, Gianlorenzo D'Angelo, Daniel Delling, Andrea Schumm, and Dorothea Wagner. 2012. The shortcut problem—complexity and algorithms. *Journal of Graph Algorithms and Applications* 16, 2, 447–481. http://jgaa.info/16/270.html.

Hans L. Bodlaender. 1993. A tourist guide through treewidth. *Acta Cybernetica* 11, 1–21.

Hans L. Bodlaender. 2007. Treewidth: Structure and algorithms. In *Structural Information and Communication Complexity*. Lecture Notes in Computer Science, Vol. 4474. Spriniger, 11–25.

Hans L. Bodlaender and Arie M. C. A. Koster. 2010. Treewidth computations I. upper bounds. *Information and Computation* 208, 3, 259–275.

Soma Chaudhuri and Christos Zaroliagis. 2000. Shortest paths in digraphs of small treewidth. Part I: Sequential algorithms. *Algorithmica* 27, 3, 212–226.

Daniel Delling, Andrew V. Goldberg, Andreas Nowatzyk, and Renato F. Werneck. 2013. PHAST: Hardware-accelerated shortest path trees. *Journal of Parallel and Distributed Computing* 73, 7, 940–952. http://www.sciencedirect.com/science/article/pii/S074373151200041X.

Daniel Delling, Andrew V. Goldberg, Thomas Pajor, and Renato F. Werneck. 2011a. Customizable route planning. In *Proceedings of the 10th International Symposium on Experimental Algorithms (SEA'11)*. 376–387.

Daniel Delling, Andrew V. Goldberg, Thomas Pajor, and Renato F. Werneck. 2014. Robust distance queries on massive networks. In *Algorithms—ESA 2014*. Lecture Notes in Computer Science, Vol. 8737. Springer, 321–333.

Daniel Delling, Andrew V. Goldberg, Thomas Pajor, and Renato F. Werneck. 2015. Customizable route planning in road networks. *Transportation Science*. Published online May 22, 2015. http://dx.doi.org/10.1287/trsc.2014.0579.

Daniel Delling, Andrew V. Goldberg, Ilya Razenshteyn, and Renato F. Werneck. 2011b. Graph partitioning with natural cuts. In *Proceedings of the 25th International Parallel and Distributed Processing Symposium (IPDPS'11)*. IEEE, Los Alamitos, CA, 1135–1146.

Daniel Delling, Andrew V. Goldberg, Ilya Razenshteyn, and Renato F. Werneck. 2012. Exact combinatorial branch-and-bound for graph bisection. In *Proceedings of the 14th Meeting on Algorithm Engineering and Experiments (ALENEX'12)*. 30–44.

Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. 2011c. Faster batched shortest paths in road networks. In *Proceedings of the 11th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'11)*. 52–63.

Daniel Delling and Renato F. Werneck. 2013. Faster customization of road networks. In *Experimental Algorithms*. Lecture Notes in Computer Science, Vol. 7933. Springer, 30–42.

Camil Demetrescu, Andrew V. Goldberg, and David S. Johnson (Eds.). 2009. *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*. DIMACS Book, Vol. 74. American Mathematical Society.

Edsger W. Dijkstra. 1959. A note on two problems in connexion with graphs. *Numerische Mathematik* 1, 269–271.

D. R. Fulkerson and O. A. Gross. 1965. Incidence matrices and interval graphs. *Pacific Journal of Mathematics* 15, 3, 835–855.

Robert Geisberger, Dennis Luxen, Peter Sanders, Sabine Neubauer, and Lars Volker. 2010. Fast detour computation for ride sharing. In *Proceedings of the 10th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'10)*. 88–99.

Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. 2008. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *Experimental Algorithms*. Lecture Notes in Computer Science, Vol. 5038. Springer, 319–333.

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

Alan George. 1973. Nested dissection of a regular finite element mesh. *SIAM Journal on Numerical Analysis* 10, 2, 345–363.

Alan George and Joseph W. Liu. 1978. A quotient graph model for symmetric factorization. In *Sparse Matrix Proceedings*, I. S. Duff and G. W. Stewart (Eds.). SIAM, Philadelphia, PA, 154–175.

Alan George and Joseph W. Liu. 1989. The evolution of the minimum degree ordering algorithm. *SIAM Review* 31, 1, 1–19.

John R. Gilbert and Robert Tarjan. 1986. The analysis of a nested dissection algorithm. *Numerische Mathematik* 50, 4, 377–404.

Michael Hamann and Ben Strasser. 2015. *Graph Bisection with Pareto-Optimization*. Technical Report. arXiv:1504.03812. http://arxiv.org/abs/1504.03812.

Martin Holzer, Frank Schulz, and Dorothea Wagner. 2008. Engineering multilevel overlay graphs for shortest-path queries. *ACM Journal of Experimental Algorithmics* 13, Article No. 5.

Haim Kaplan, Ron Shamir, and Robert Tarjan. 1999. Tractability of parameterized completion problems on chordal, strongly chordal, and proper interval graphs. *SIAM Journal on Computing* 28, 5, 1906–1922.

George Karypis and Vipin Kumar. 1999. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing* 20, 1, 359–392. http://dx.doi.org/10.1137/S1064827595287997.

Sebastian Knopp, Peter Sanders, Dominik Schultes, Frank Schulz, and Dorothea Wagner. 2007. Computing many-to-many shortest paths using highway hierarchies. In *Proceedings of the 9th Workshop on Algorithm Engineering and Experiments (ALENEX'07)*. 36–45.

Moritz Kobitzsch, Marcel Radermacher, and Dennis Schieferdecker. 2013. Evolution and evaluation of the penalty method for alternative graphs. In *Proceedings of the 13th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'13)*. 94–107. http://drops.dagstuhl.de/opus/volltexte/2013/4247.

Richard J. Lipton, Donald J. Rose, and Robert Tarjan. 1979. Generalized nested dissection. *SIAM Journal on Numerical Analysis* 16, 2, 346–358.

Catherine C. McGeoch, Peter Sanders, Rudolf Fleischer, Paul R. Cohen, and Doina Precup. 2002. Using finite experiments to study asymptotic performance. In *Experimental Algorithmics—From Algorithm Design to Robust and Efficient Software*. Lecture Notes in Computer Science, Vol. 2547. Springer, 93–126. http://dx.doi.org/10.1007/3-540-36383-1_5.

Nikola Milosavljević. 2012. On optimal preprocessing for contraction hierarchies. In *Proceedings of the 5th ACM SIGSPATIAL International Workshop on Computational Transportation Science*. ACM, New York, NY, 33–38.

Léon Planken, Mathijs de Weerdt, and Roman van Krogt. 2012. Computing all-pairs shortest paths by leveraging low treewidth. *Journal of Artificial Intelligence Research* 43, 1, 353–388.

Alex Pothen. 1988. *The Complexity of Optimal Elimination Trees*. Technical Report. Pennsylvania State University.

Michael Rice and Vassilis Tsotras. 2010. Graph indexing of road networks for shortest path queries with label restrictions. *Proceedings of the VLDB Endowment* 4, 2, 69–80.

Peter Sanders and Dominik Schultes. 2012. Engineering highway hierarchies. *ACM Journal of Experimental Algorithmics* 17, 1, 1–40.

Peter Sanders and Christian Schulz. 2013. Think locally, act globally: Highly balanced graph partitioning. In *Experimental Algorithms*. Lecture Notes in Computer Science, Vol. 7933. Springer, 164–175.

Aaron Schild and Christian Sommer. 2015. On balanced separators in road networks. In *Experimental Algorithms*. Lecture Notes in Computer Science, Vol. 9125. Springer, 286–297.

Frank Schulz, Dorothea Wagner, and Karsten Weihe. 2000. Dijkstra's algorithm on-line: An empirical case study from public railroad transport. *ACM Journal of Experimental Algorithmics* 5, 12, 1–23.

Sabine Storandt. 2013. Contraction hierarchies on grid graphs. In *KI 2013: Advances in Artificial Intelligence*. Lecture Notes in Computer Science, Vol. 8077. Springer, 236–247.

Nathan Sturtevant. 2012. Benchmarks for grid-based pathfinding. *Transactions on Computational Intelligence and AI in Games* 4, 2, 144–148.

William F. Tinney and J. W. Walker. 1967. Direct solutions of sparse network equations by optimally ordered triangular factorization. *Proceedings of the IEEE* 55, 11, 1801–1809.

Michael Wegner. 2014. *Finding Small Node Separators*. Bachelor's Thesis. Karlsruhe Institute of Technology.

Fang Wei. 2010. TEDI: Efficient shortest path query answering on graphs. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD'10)*. ACM, New York, NY.

Mihalis Yannakakis. 1981. Computing the minimum fill-in is NP-complete. *SIAM Journal on Algebraic and Discrete Methods* 2, 1, 77–79.

Tim Zeitz. 2013. *Weak Contraction Hierarchies Work!* Bachelor's Thesis. Karlsruhe Institute of Technology.

Received September 2014; revised October 2015; accepted November 2015