Project Number 288094

eCOMPASS

eCO-friendly urban Multi-modal route PlANning Services for mobile uSers

STREP
Funded by EC, INFSO-G4(ICT for Transport) under FP7

eCOMPASS – TR – 059

Customizable Contraction Hierarchies

JULIAN DIBBELT, BEN STRASSER and DOROTHEA WAGNER

September 2014

A

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, 2) use this auxiliary data to speedup the query phase. Instead, 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, as well as a lightweight phase that adapts the auxiliary data to a specific weight. We achieve this by basing our Customizable Contraction Hierarchies on nested dissection orders. We provide an in-depth experimental analysis on large road and game maps that shows that Customizable Contraction Hierarchies are a very practicable solution in scenarios where edge weights often change.

Categories and Subject Descriptors: []

ACM Reference Format:
ACM J. Exp. Algor. V, N, Article A (January YYYY), 34 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

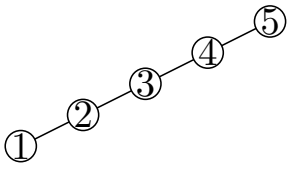
1. INTRODUCTION

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 needs several seconds of running time for a single path query. For practical performance on large road networks, preprocessing techniques that augment the network with auxiliary data in an (expensive) offline phase have proven useful. See [Bast et al. 2014] for an overview. Among the most successful techniques are Contraction Hierarchies (CH) by [Geisberger et al. 2012], which have been utilized in many scenarios. However, their preprocessing is in general metric-dependent, e. g., edge weights (also called the graph metric) need to be known. Substantial changes to the metric, e. g., due to user preferences, may require expensive recomputation. For this reason a Customizable Route Planning (CRP) approach was proposed in [Delling et al. 2011], extending the multi-level-overlay MLD techniques of [Schulz et al. 2000; Holzer et al. 2008]. It works in three phases: In a first expensive phase, auxiliary data is computed that solely exploits the topological structure of the network, disregarding its metric. In a second much less expensive phase, this auxiliary data is *customized* to the specific metric, enabling fast queries in the third phase. In this work we extend CH to support such a three-phase approach.

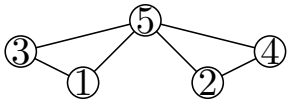
Partial support by DFG grant WA654/16-2 and EU grant 288094 (eCOMPASS) and Google Focused Research Award.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© YYYY ACM 1084-6654/YYYY/01-ARTA $15.00
DOI: http://dx.doi.org/10.1145/0000000.0000000

Game Scenario. Most existing CH 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, since buildings are constructed or destroyed, fields are rendered impassable or freed up. Furthermore, every player has his own knowledge of the map because of features such as *fog of war* and thus has his own metric: A unit must not route around a building that the player has not yet seen. Furthermore, units such as hovercrafts may traverse water and land, while 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, and thus being able to exploit the symmetry of the underlying graph is a useful feature.

Nested Dissection Order. One of the central building blocks of this paper is to use metric-independent nested dissection orders (ND-orders) for CH 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 CHs to engineer subroutines of their customization phase (they also had preliminary experiments on full CH). Worth mentioning are also the works of [Planken et al. 2012]. They consider small graphs of low treewidth and leverage this property to compute good orders and CHs (without explicitly using the term CH). Interestingly, our experiments show that also large road networks have relatively low treewidth. Real world road graphs with vertex counts in the 10^7 have treewidths in the 10^2.



(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

Connection to Sparse Matrix solving. Customizable speedup techniques for shortest path queries are a very recent development but the idea to use ND-orders to compute CH-like structures is far older and widely used in the *sparse matrix solving* community. To the best of our knowledge the idea first appeared in 1973 in [George 1973] and was significantly refined in [Lipton et al. 1979]. Since then there has been a plethora of papers on the topic. Among them are papers showing that computing a CH (without witness search) with a minimum amount of shortcuts is NP-hard [Yannakakis 1981] but fixed parameter tractable in the number of needed shortcuts [Kaplan et al. 1999]. However, note that minimizing the number of shortcuts is not desirable in our situation: Consider for example a path as depicted in Figure 1. Optimizing the CH search space and the number of shortcuts can be competing criteria. The hardness result is therefore not directly applicable to our scenario. In another interesting result it has been shown that for planar graphs the number of arcs in a CH with ND-order 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.

Our Contribution. The main contribution of our work is to show that Customizable Contraction Hierarchies (CCH) solely based on the ND-principle are feasible and practical. Compared to CRP [Delling et al. 2011] we achieve a similar preprocessing–query

tradeoff, albeit with slightly better query performance at slightly slower customization speed (and somewhat more space). Interestingly, for less well-behaved metrics such as travel distance, we achieve query times below the original metric-dependent CH of [Geisberger et al. 2012]. Besides this main results there are number of side results. We show that given a fixed contraction order a metric-independent CH can be constructed in time essentially linear in the Contraction Hierarchy with working space memory linear in the input graph. Our specialized algorithm has better theoretic worst case running times and performs significantly better in experiments than the dynamic adjacency arrays used in [Geisberger et al. 2012]. Another contribution of our work are perfect witness searches. We show that for ND-orders it is possible to construct CHs with a minimum number of arcs in about a minute 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 results in a constant factor approximation for metric-independent CHs on a class of graph with very regular recursive vertex separators. Experimentally we show that road graphs have such a recursive separator structure.

Outline. This work is organized as follows. Section 2 sets necessary notation, while Section 3 discusses metric-dependent orders traditionally used in Contraction Hierarchies (highlighting specifics of our implementation). Next, we discuss metric-independent orders in Section 4, construction of the corresponding CH in Section 5, and a preprocessing step for efficient enumeration of lower arc triangles in Section 6. In terms of the three-phase model, these steps correspond to the first phase: They only depend on the topology and can be preprocessed once–before considering metrics. Then, Section 7 considers the second phase, i. e., the customization of the datastructures w. r. t. to a given metric, while Section 8 describes the third phase: distance queries and path unpacking. Section 9 discusses extensions of the approach for enabling turn restrictions and costs. Finally, in Section 10 we present extensive experiments to evaluate our algorithms, while Section 11 concludes this work and mentions interesting 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 multi-edges. Graphs in this paper are always simple unless noted otherwise (e. g., in parts of Section 5). We denote by $N(v)$ the set of adjacent vertices of v in an undirected graph.

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 $|A|, |B| \leq 2n/3$.

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 *directed upward graph* with respect to a vertex order π, i. e., every edge $\{\pi(i), \pi(j)\}$ with $i < j$ is replace by an arc $(\pi(i), \pi(j))$. Note that all upward directed graphs are acyclic. We denote by $N_u(v)$ the neighbors of v with a higher rank than v and by $N_d(v)$ those 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 $w : E \to \mathbb{R}^+$. With respect to a vertex order π we define an *upward weight* $w_u : E \to \mathbb{R}^+$ and a *downward weight* $w_d : E \to \mathbb{R}^+$. 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 paper. 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 $\mathrm{dist}_G(s,t)$. (We set $\mathrm{dist}_G(s,t) = \infty$ 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 vertices of every acyclic directed 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 directed acyclic graph.

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 refereed to as *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]. We define the *weighted vertex contraction* of a vertex v in the graph G as the operation of removing v and inserting the minimum number of shortcuts among the neighbors of v to obtain a graph G' such that $\mathrm{dist}_G(x,y) = \mathrm{dist}'_G(x,y)$ for all vertices $x \neq v$ and $y \neq v$. To compute G', we iterate over all pairs of neighbors x, y of v increasing by $\mathrm{dist}_G(x,y)$. For each pair we check whether a xy-path of length $\mathrm{dist}_G(x,y)$ exists in $G\backslash\{v\}$, i. e., we check 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 then we skip the pair and do nothing. Otherwise depending on whether an edge $\{x,y\}$ already exists we either decrease its weight to $\mathrm{dist}_G(x,y)$ or insert a shortcut edge with that weight to G. This new shortcut edge is considered in witness searches for subsequent neighbor pairs as part of G. It is important to iterate over the pairs increasing by $\mathrm{dist}_G(x,y)$ 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 then the witness search will not consider them. See Figure 2 for an example. Note that the witness searches are expensive and therefore usually the witness search is aborted after a certain number of steps [Geisberger et al. 2012]. If no witness was found until then, 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.

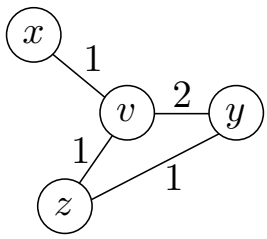


Fig. 2. Contraction of v. If the pair x, y is considered first then a shortcut $\{x,y\}$ with weight 3 is inserted. If the pair x, z is considered first then an edge $\{x,z\}$ with weight 2 is inserted. This shortcut is part of a witness $x \to y \to z$ for the pair x, y. The shortcut $\{x,y\}$ is *not*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

We call a witness search *without* such a one-sided error *perfect*. 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 non-adjacent 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 the $G_{\pi,i}$ form a clique [Fulkerson and Gross 1965]. Such an order is called a *perfect elimination order*.

The elimination tree $T_{G,\pi}$ is a tree directed towards 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.

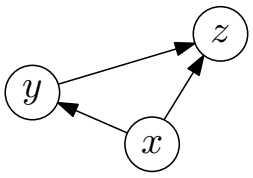


Fig. 3. A triangle in $G^\wedge_\pi$. The triple (y, z, x) is a lower triangle of the arc (y, z). The triple (x, z, y) is an intermediate triangle of the arc (x, z). The triple (x, y, z) is an upper triangle of the arc (x, y).

A *lower triangle* of an arc (x, y) in $G^\wedge_\pi$ is a triple (x, y, z) such that arcs (z, x) and (z, y) exist. Similarly an *intermediate triangle* is a triple such that (x, z) and (z, y) exist and an *upper triangle* is a triple such that (x, z) and (y, z) exist. The situation is illustrated in Figure 3. Recall that arcs in $G^\wedge_\pi$ are directed according to rank and do not necessarily reflect travel direction.

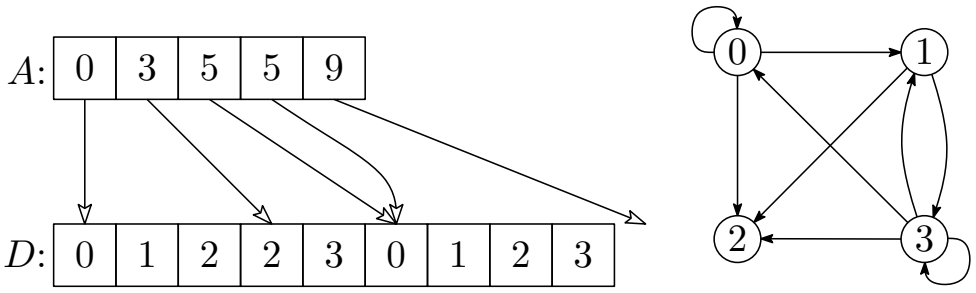
2.1. Metrics

In the following, we denote weights on $G^\wedge_\pi$ as *metrics*. We say that a metric m *respects* a weight w of G if $\mathrm{dist}_G(x, y) = \mathrm{dist}_{G^*_\pi}(x, y)$ for all vertices x and y. Every weight on G can trivially be extended to a w-respecting metric by assigning the weights of w to the original arcs and ∞ to all shortcuts. We refer to this metric as the *w-initial metric*. A metric is called *customized* if for all lower triangles (x, y, z) the *lower triangle inequality* holds, i. e., $m(x, y) \leq m(z, x) + m(z, y)$. Note that the w-initial is w-respecting but it is not customized.

LEMMA 2.1. *Let m be a customized metric on $G^\wedge_\pi$ respecting a weight w on a graph G. For all pairs s and t with $\mathrm{dist}_G(s, t) \neq \infty$ a shortest up-down st-path exists in $G^\wedge_\pi$.*

PROOF. As $\mathrm{dist}_G(s, t) \neq \infty$ a shortest st-path in G must exist. If on $G^\wedge_\pi$ this is not an up-down path, then it must contain a subpath $x \to y \to z$ with $\pi^{-1}(x) > \pi^{-1}(y)$ and $\pi^{-1}(y) < \pi^{-1}(z)$. As y is contracted before x and z an arc (x, y) must exist. As (x, y, z) is a lower triangle and m is customized, we know that removing the vertex y from the path cannot make the path longer. As the path has only finitely many vertices iteratively replacing these lower triangles yields a shortest up-down path after finitely many steps. □

Denote by M_w the set of metrics that respect a weight w and are customized. A metric $m \in M_w$ is *w-maximum* if no other metric exists in M_w that has a higher weight



(a) Adjacency array (b) Represented graph

Fig. 4. The left figure depicts two arrays: The top is the *index array* I and the bottom the *data array* D. The index array has $|V|$+1 entries. The data array has $|A|$ entires. Each entry in I is an index into the data array D. The neighbors of a vertex with ID x have the IDs $D[I[x]], D[I[x]+1], \ldots, D[I[x+1]-1]$.

on some arc. Analogously a w-*minimum* metric is one where no arc weight can be decreased. Note that both of these metrics are unique. Furthermore, a w-minimum metric can be characterized as one where every arc (x, y) has the weight of a shortest xy-path.

2.2. Adjacency Array

An *adjacency array* is a data structure that is used to map IDs onto other objects. As depicted in Figure 4, it consists of two arrays and can be used to store graphs by mapping a vertex ID onto the neighboring vertices' IDs. Note that adjacency arrays also have other applications: For example instead of mapping vertex IDs on the neighboring vertices' IDs, it could map onto the incident arc IDs. Another example would be to map the ID of an arc (x, y) onto vertex IDs z_i such that each (x, y, z_i) forms a lower triangle of (x, y). Adjacency arrays are an omnipresent basic building block in efficient graph algorithms and as such they have many different names. Other names include *compressed row* and *forward star*.

3. METRIC-DEPENDENT ORDERS

Most papers using Contraction Hierarchies use greedy orders in the spirit of [Geisberger et al. 2012]. As the exact details vary from paper to paper, we describe our precise variant in this section. 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 get 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. 2012]. 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 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. For the experiments presented in Section 10, we do not use lazy updates or periodic queue rebuilding as proposed in [Geisberger et al. 2012] as our importance function does not need them.

4. METRIC-INDEPENDENT ORDER

The metric-dependent orders presented in the previous section lead to very good results on road graphs with travel time metric. However, the results for the distance metric are not as good and the orders are completely impracticable to compute Contraction Hierarchies without witness search as our experiments in Section 10. To support metric-independence, we use *nested dissection* orders as suggested in [Bauer et al. 2013] (or ND-orders for short). 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 $\pi(n - |S|) \dots \pi(n)$ 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 graphs 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. 2012; 2011]. This justifies assuming in our particular context that an efficient bisection oracle exists. Note that graph bisectors usually compute edge cuts and not vertex separators. On our instances, a vertex separator is derived by arbitrarily picking for every edge one of its incident vertices. We experimentally examine the performance of nested dissection orders computed by NDMetis [Karypis and Kumar 1999] and KaHIP [Sanders and Schulz 2013] in Section 10. After having obtained the nested dissection 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. 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 $G \in K_\alpha$ a 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]. We provide the full proof for self-containedness.

PROOF. Consider the subgraphs G_i of G^*_π induced by the vertices $\pi(1)\dots\pi(i)$ (not to be confused with the core graphs $G_{\pi,i}$). 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. 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 $u, v \in S'$ there exists a path through A as A is connected. As u and v have the highest ranks on this path (the vertices in A have rank $1\dots i$), there must be and edge $\{u, v\}$ in G^*. 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 a 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. This key observation of this proof is that the top level separator solely dominates the performance. Denote by π the ND-order and by π_{opt} the optimal order. First we show a lower bound on the performance of π_{opt} and then show that π achieves this lower bound showing that π is an $O(1)$-approximation. As the minimum balanced separator has cardinality $\Theta(n^\alpha)$ we know by Lemma 4.2 that at least $n/3$ vertices exist, whose search space in $G^{\wedge}_{\pi_{opt}}$ contains a clique with $\Theta(n^\alpha)$ vertices. Thus the maximum

number of vertices in a search space is $\Omega(n^\alpha)$ as it must contain this clique and as the clique is dense the maximum number of arcs is in $\Omega(n^{2\alpha})$. The average number of vertices is $2/3 \cdot \Omega(0) + 1/3 \cdot \Omega(n) = \Omega(n)$ and as the clique is dense the average number of arcs is in $\Omega(n^{2\alpha})$. From Lemma 4.1 we know that the number of vertices in the maximum search space of $G^\wedge_\pi$ is in $O(n^\alpha)$. A direct consequence is that the average number of vertices is also in $O(n^\alpha)$. In the worst case the search space is dense resulting in $O(n^{2\alpha})$ arcs in the average and the maximum search space. As the derived bounds are tight this shows that π is an $O(1)$-approximation. □

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 [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 graph weights and directed arcs are handled during customization.)

Denote by n the number of vertices, m the number of edges in G, by m' the number of edges in $G^\wedge_\pi$, and by $\alpha(n)$ the inverse $A(n,n)$ Ackermann function. For simplicity we assume that G is connected. Our algorithm enumerates all arcs of $G^\wedge_\pi$ in $O(m'\alpha(n))$ running time and has a memory consumption in $O(m)$ (to store the arcs of $G^\wedge_\pi$, additional space in $O(m')$ 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. We also were not able to find an complexity analysis for the specific variant employed by us. Therefore, in the remainder of this section, we both discuss the approach and present a running time analysis. As a first step, we describe a complex datastructure that supports efficient *edge* contraction and neighborhood enumeration. Then, we show how this datastructure is used to realize a datastructure that supports efficient *vertex* contraction and neighborhood enumeration.

5.1. Technicalities

In the following, we identify vertices with an ID from the range $1 \dots n$. For edges we do not store any IDs. To avoid problems with ID-relabeling, we never remove vertices. That is, contracting a vertex v consists of removing all incident edges and connecting all adjacent vertices, but we do not remove v. After the vertex contraction v has degree 0. Contracting an edge $\{u, v\}$ consists of removing all edges $\{v, w\}$ and adding edges $\{u, w\}$ if necessary. After edge contraction, again, v has degree 0. Note that this makes edge contraction strictly speaking a non-symmetric operation. Enumerating the neighborhood of a vertex v (given by its unique ID) consists of enumerating the IDs of all adjacent vertices exactly once.

5.2. Efficient Edge Contraction

The core idea is to organize contracted vertices in a linked list. Even if G is simple, edge contraction can create unwanted multi edges or loops. We remove these unwanted edges during the enumeration. As an edge contraction does not create news edges we can remove at most as many as there were in G. To efficiently rewire the edges we further need a union find datastructure that introduces some $\alpha(n)$ terms. Our datastructure has an edge contraction in $O(1)$. The enumeration of the neighbors of v needs

$O(d(v)\alpha(n))$ amortized running time. Finally there are global edge removal costs of at most $O(m\alpha(n))$ that do not depend on the operations applied to the datastructure.

We combine an adjacency array, a doubly linked list, a union-find datastructure and a boolean array. The adjacency array initially stores for every vertex in v the IDs of the adjacent vertices in G. The doubly linked list links together the vertices of G that have been contracted. We say that two vertices that are linked together are on the same *ring*. Initially no edges were contracted and therefore all rings only contain a single vertex. The union find datastructure is used to efficiently determine a representative vertex ID for every ring given a vertex of that ring. The boolean array is used to mark vertices and is needed to assure that the neighborhood iteration outputs no vertex twice and that v is not a neighbor of v. Initially all entries are false. After each neighborhood enumeration the entries are reset to false. All vertices in a ring are regarded as having degree 0 with the exception of the representative, which is regarded as incident to all edges incident to the ring.

Contracting an edge is the easy operation. During the enumeration most of the work occurs. To contract an edge $\{u, v\}$ we first check whether u and v are the representatives of their ring. If u or v is not then they have degree 0 and there is nothing to do. We merge the rings of u and v and unite u and v in the union find datastructure and choose either u or v as representative. To enumerate the neighbors of a vertex a we first check whether it is its own representative in the union find datastructure. If it is not then a has degree 0 as the edges have been contracted away. Otherwise we mark a in the boolean array. Next we iterate over all vertex IDs b in the linked ring of a. For every b we iterate over the vertex IDs c in the adjacency array for b. For every c we lookup its representative d in the union find datastructure. If d is not marked in the boolean array we found a new neighbor of a. We output it and mark it. Otherwise we do not output d but remove c from b's adjacency in the array. If this empties the adjacency of b we remove b from a's ring but keep b in the same union as a. After the enumeration we iterate a second time over it to reset the boolean array.

5.3. Analysis

We first analyze the memory consumption. There is no memory allocation during the algorithm and the sizes of the initial datastructure are dominated by the adjacency array that needs $O(m)$ space. The running time of an edge contraction is in $O(1)$ as all its operations are in $O(1)$. (Note that we are only checking whether u is the representative, we do not actually compute the representative if it is not u.) Analyzing the neighborhood enumeration is more complex. Three key insights are needed: First there are only m initial edges and therefore at most m entries can be deleted. The costs are accounted for in the global $O(m\alpha(n))$ term. The second insight is that as we remove empty adjacencies from the rings a ring never contains more pointers than vertices and therefore the time needed to follow the pointers is dominated by the time spend visiting the vertices. The third insight is that a second enumeration of v cannot find duplicates as they have been removed in the first iterations. Therefore reseting the boolean array is in $O(d(v))$.

5.4. Efficient Vertex Contraction

Based on the efficient edge contraction datastructure described above we design an efficient vertex contraction datastructure. The allowed operations are slightly more restrictive. We require that each enumeration of the neighborhood of v is followed by v's vertex contraction.

Instead of storing the graphs $G_{\pi,i}$ explicitly we store a different graph $G'_{\pi,i}$. We do not replace a contracted vertex v by a clique among its neighbors. Instead we

replace it by a star with a virtual dummy vertex at its center. If a vertex is adjacent to a star center then it is recognized as being adjacent to all vertices in the star. If two star centers become adjacent we merge the stars by contracting the edge between the centers. The complexity of the resulting star is the sum of both original stars. This contrasts with explicitly representing the induced cliques whose complexity would grow super linearly. The idea is illustrated in Figure 5.

Formally the vertices that have non-zero degree in both $G_{\pi,i}$ and $G'_{\pi,i}$ are called *regular* vertices. The additional dummy vertices that have a non-zero degree in $G'_{\pi,i}$ are called *super* vertices. We maintain the invariant that $G'_{\pi,i}$ does not contain two adjacent super vertices. Furthermore, for every edge $\{x, y\}$ in $G_{\pi,i}$ there exists an edge $\{x, y\}$ in $G'_{\pi,i}$ or a path $x \to z \to y$ in $G'_{\pi,i}$ where z is a super vertex and x and y regular vertices.

An alternative way to describe the datastructure is to say that we maintain a graph on which we only perform edge contractions and we maintain an independent set of virtually contracted super vertices.

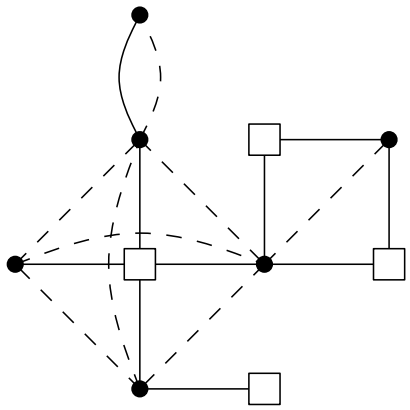


Fig. 5. The dots represent vertices that have non-zero degree in $G_{\pi,i}$ and in $G'_{\pi,i}$. The squares are the additional super vertices in $G'_{\pi,i}$. The solid edges are in $G'_{\pi,i}$ and the dashed ones in $G_{\pi,i}$. Notice how the neighbors of each super vertex in $G'_{\pi,i}$ forms a clique in $G_{\pi,i}$. Furthermore, there are no two adjacent super vertices in $G'_{\pi,i}$, i. e., they form an independent set.

The datastructure only needs to support a single operation: Enumerating the neighbors of an arbitrary vertex x in $G_{\pi,i}$ followed by x's contraction. We actually do it in reversed order : We first contract v and then enumerate the neighbors of the new super vertex v. To contract x we first mark it as super vertex. We then enumerate its neighbor in $G'_{\pi,i}$ to determine all adjacent super vertices y_i. We then contract all edges $\{x, y_i\}$ to assure that x is no longer adjacent to any super vertex. Afterwards all neighbors of x in $G'_{\pi,i}$ are regular and therefore they coincide with those in $G_{\pi,i}$ making the enumeration straightforward.

5.5. Analysis

In addition to the edge contraction datastructure we need a boolean array to mark vertices as super nodes. The additional memory consumption is therefore in $O(n)$ and is negligible. The running time is shown using an amortized analysis. Denote by x the contracted vertex. There are three cost factors: The enumeration of the neighbors $y_1 \dots y_p$ before the contraction, the enumeration of the neighbors $z_1 \dots z_q$ after the contraction, and the contraction of the arcs. The enumeration of each y_i and z_i carries a cost of $\alpha(n)$ resulting from the underlying edge contraction datastructure. Note that while the y_i contain super and regular vertices the z_i contain only regular vertices. As there are at most as many regular y_i vertices as there are z_i vertices we can account for the regular y_i vertices in the costs of the z_i vertices. The remaining y_i are super vertices. Their enumeration is always followed by an edge contraction and therefore we account for their cost in the edge contraction costs. The enumeration costs of the y_i are therefore accounted for. As at most m edges can be contracted their total costs result in a global $O(m\alpha(n))$ term in the running time. As the number of z_i coincides with

the out-degree of x in $G^{\wedge}_{\pi}$ we can account the costs of the z_i to the arcs of $G^{\wedge}_{\pi}$ resulting in $O(m'\alpha(n))$ total costs.

5.6. Obtaining statistics for badly conditioned hierarchies

For every graph G and order π yielding a small m' we efficiently construct and store G^{*}_{π} (and use it for route planning applications). However, even for orders yielding large m', we are interested in the characteristic numbers of G^{*}_{π}(e. g., to exactly quantify the quality (or badness) of an order). We obviously cannot store all arcs. But using the contraction graph datastructure, given enough time, we can count them (recall that our datastructure only requires $O(m)$ space). Furthermore, we can construct the elimination tree of $G^{\wedge}_{\pi}$ and compute the out-degree of all vertices. From these we derive the size of $G^{\wedge}_{\pi}$ (i. e., m') as well as the average and maximum search space size in $G^{\wedge}_{\pi}$.

6. ENUMERATING TRIANGLES

Efficiently enumerating all lower triangles of an arc is an important base operation of the customization and path unpacking algorithms (see Section 7 and 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.

6.1. Basic Triangle Enumeration

Construct an upward and a downward adjacency array for $G^{\wedge}_{\pi}$, where incident arcs are ordered by their head vertex ID. Unlike common practice, we also assign and store arc IDs. (By lexicographically assigning arc IDs we eliminate the need for arc IDs in the upward adjacency array.) Denote by $N_u(v)$ the upward neighborhood of v and by $N_d(v)$ the downward neighborhood. All lower triangles of an arc (x, y) are enumerated by simultaneously scanning $N_d(x)$ and $N_d(y)$ by increasing vertex ID to determine their intersection $N_d(x) \cap N_d(y) = \{z_1 \dots z_k\}$. The lower triangles are all triples (x, y, z_i). The corresponding arc IDs are stored in the adjacency arrays. Similarly intersecting $N_u(x)$ and $N_u(y)$ yields all upper triangles, and intersecting $N_u(x)$ and $N_d(y)$ yields all intermediate triangles. This approach requires space proportional to the number of arcs in $G^{\wedge}_{\pi}$.

6.2. 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) onto the pair of arc ids of (z, x) and (z, y) for every lower triangle (x, y, z). This requires space proportional to the number of triangles t in $G^{\wedge}_{\pi}$ but allows for a very fast access. Analogous structures allow efficient access all upper triangles and all intermediate triangles.

6.3. 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 in $\Theta(n^2)$. It can therefore be prohibitive to store a list of all triangles. We therefore propose a hybrid approach. We only precompute the triangles for the arcs (u, v) where the level of u is below a certain threshold. The threshold is a tuning parameter that trades space for time.

6.4. Comparison with CRP

Our triangle preprocessing has similarities with micro and macro code [Delling and Werneck 2013]. The micro code approach is basically a huge array containing triples

of arc IDs that participate in a triangle. The macro code stores for each vertex v a block that contains an array of incident arc IDs and a matrix of the arcs in the clique that replaces v after its contraction. We compare the space consumption only against a triangle adjacency array that enumerates only lower triangles as this is sufficient for an efficient customization.

The authors of [Delling and Werneck 2013] operate on directed graphs graphs but we operate on undirected graphs. Let t denote the number of undirected triangles and m the number of arcs in $G_\pi^\wedge$. Furthermore, denote by t' the number of directed triangles and by m' the number of arcs used in [Delling and Werneck 2013]. If one-way streets are rare then $m' \approx 2m$ and $t' \approx 2t$.

The micro code approach requires storing $3t' \approx 6t$ arc IDs. Our approach needs to store $2t+m+1$ arc IDs. Estimating the space requirement for the macro code approach is more complex. A lower triangle (x, y, z) is stored in the block of z. Denote by $d(z)$ the degree of z. The block of z needs to store $d^2(z) + d(z) + O(1)$ arc ids (the $O(1)$ data is needed to mark the end of a block). As z participates in $d^2(z)$ many triangles as lowest ranked vertex and every triangle has exactly one lowest ranked vertex we know that $\sum_{z \in V} d^2(z) = t$. Summing over all vertices therefore yields a space requirement of $t' + m' + O(n) = 2t + 2m + O(n)$.

Our approach always outperforms micro code. Furthermore, our approach is slightly more compact than macro code under the assumption that one-way streets are rare. If one-way streets are common then our approach needs at most twice as much data. However, the main advantage of our approach over macro code is that it allows for random access, which is crucial in the algorithms presented in the following sections.

7. CUSTOMIZATION

In this section, we describe how to transform a w-initial metric m_0 into a w-maximum metric m_1. In a second step we transform m_1 into a w-minimum metric m_2. Based on m_2, it is possible to construct a weighted contraction hierarchy with perfect witness search. We also discuss how to apply multi-threading and single instruction multiple data (SIMD) instructions. Furthermore, we show how to update a metric if only the weights of a few edges change.

7.1. Maximum Metric

We want to turn an initial metric m_0 into a customized one m_1. For this we first copy m_0 into m_1 and then modify m_1 as following: Our algorithm iterates over all vertex levels $\ell(x)$ in $G_\pi^\wedge$ from the lowest level upward. On level i, it iterates (using multiple threads) over all arcs (x, y) with $\ell(x) = i$. Between each level all threads must be synchronized. For each such arc (x, y), the algorithm enumerates all lower triangles (x, y, z) and performs $m_1(x, y) \leftarrow \min\{m_1(x, y), m_1(z, x) + m_1(z, y)\}$, i. e., it makes sure that the lower triangle inequality holds. The resulting metric still respects w as we only set weights $m_1(x, y)$ to the distances of xy-paths. Note that this xy-path is not necessarily the shortest and thus the resulting metric is not necessarily minimum. Furthermore, by definition m_1 is customized. The metric is w-maximum, because increasing the weight of a shortcut (x, y) would violate the lower triangle equality of some lower triangle of (x, y). As all threads only write to the arc they are assigned to and only read from arcs processed in a strictly lower level we can guarantee that no read/write conflicts occurs. Hence, no locks or atomic operations are needed.

7.2. Minimum Metric and Perfect Witness Search

Suppose m_1 is already customized. We want to turn it into a w-minimum metric m_2. Recall that a w-minimum metric is a metric where every arc (x, y) has the weight of a

shortest xy-path. As a side-product our algorithm marks all arcs that a perfect witness-search would remove. We first describe what our algorithm does and afterwards why it is correct. We first copy m_1 over into m_2 and then modify m_2. The algorithm iterates over all levels downward starting at the top-most level. It then iterates over all arcs (x, y) with $\ell(x) = i$. On most processor architectures the algorithm can iterate over the arcs of a level in parallel as long as it synchronizes between levels. However, this depends on the exact details of how write-conflicts are resolved. In some cases a different strategy is needed to enable parallelization. We postpone the details to the end of this subsection. For every arc (x, y) our algorithm enumerates all upper triangles (x, y, z) and if $m_2(x, z) + m_2(y, z) \leq m_2(x, y)$ it sets $m_2(x, y) \leftarrow m_2(x, z) + m_2(y, z)$ and marks (x, y) for removal. Analogously it iterates over all intermediate triangles (x, y, z) and if $m_2(x, z) + m_2(z, y) \leq m_2(x, y)$ it sets $m_2(x, y) \leftarrow m_2(x, z) + m_2(z, y)$ and marks (x, y) for removal. Notice that we mark the arcs for removal even if both sides are equal. The order in which the intermediate and upper triangles for one specific arc are enumerated does not matter. The resulting metric is w-minimum. The arcs marked for removal are exactly those that a perfect witness search would prune.

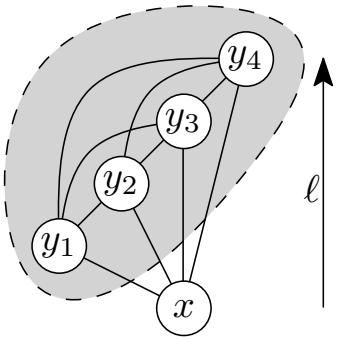


Fig. 6. The vertices $y_1 \dots y_4$ denote the upper neighborhood $N_u(x)$ of x. They form a clique (the gray area) because x was contracted first. As $\ell(x) < \ell(y_j)$ for every j we know by the induction hypothesis that the arcs in this clique are weighted by shortest path distances. We therefore 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.

It remains to show that the algorithm is correct. We have to show that after the algorithm has finished processing a vertex x all of its outgoing arcs are weighted by the shortest path distance. We prove this by induction of the levels over the processed vertices. The top-most vertex is the only vertex in the top level. It does not have any outgoing arcs and thus the algorithm does not have to do anything. 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 detailed in Figure 6 we know that vertices in $N_u(x)$ form a clique weighted by shortest paths. Pick some arbitrary outgoing arc (x, y_j). Either it already has the shortest path weight and there is nothing left to do or a shortest path through some vertex y_k in $N_u(x)$ must exist. As we know that (y_j, y_k) is a shortest path we know that $x \to y_k \to y_j$ is also a shortest path. What our algorithm does is enumerate the paths for every possible y_k. The upper triangles correspond to paths with $\ell(y_k) > \ell(y_j)$ and the intermediate triangles to paths with $\ell(y_k) < \ell(y_j)$. Our algorithm marks an arc (x, y) for removal if an xy-up-down-path exists that has the same length or is shorter and does not use (x, y). As only the existence of a shortest xy-up-down-path is needed for correctness we can not remove additional arcs. Further for all st-pairs a shortest up-down st-path exists and thus the shortest path queries are correct. The the witness search is thus perfect.

As already hinted it is less clear how to parallelize the operations within a level than it is for a plain customization. Consider the following situation: Thread A processes arc (x, y_A) at the same time as thread B processes the arc (x, y_B). Notice that (x, y_A) and (x, y_B) are both outgoing arcs of the same vertex x. Suppose that thread A updates the weight at (x, y_A) at the same moment as thread B enumerates the (x, y_B, y_A) triangle. In this situation it unclear what value thread B will see. However our algorithm is correct as long it is guaranteed that thread B will either see the old value or the new value. The new value must be smaller than the old one and therefore only an addi-

tional shortest path can have been created by thread A. If thread B sees the new value then it will see an additional shortest path. If it does not then it sees the old shortest path that has the same length and goes through some different y_j. Which shortest path thread B sees does not matter as all of them have the same length and seeing one is enough. Further seeing multiple shortest paths is not harmful. The algorithm is non-deterministic but the results is always correct. On most processor architectures (including x86) it is guaranteed that 32-bit-aligned 32-bit writes have the required property. However, if the weights have 64-bits then this property might not be given as the compiler might generate two consecutive 32-bit writes to memory. If the processor used does not have the necessary write-conflict resolution then the algorithm should iterate in parallel over all vertices in a level in parallel and each thread iterates sequentially over all outgoing arcs. This approach guarantees that all operations that might conflict are performed sequentially and does not need locks or atomic operations.

7.3. Directed Graphs and Single Instruction Multiple Data

A metric can be replaced by an interleaved set of k metrics by replacing every $m(x, y)$ by a vector of k elements. This allows us to customize all k metrics in one go, amortizing triangle enumeration time. A further advantage is that the customization can be accelerated using single instruction multiple data (SIMD) operations to combine the metric vectors. The processor needs to support component-wise minimum and saturated addition (i.e. $a + b = \text{int}_{\max}$ in case of overflow).

Up to now we have focused on customizing undirected graphs. If the graph is directed then we use two metrics: an upward metric m_u and a downward metric m_d. It is natural to store these two metrics interleaved. For correctness it is important to customize both metrics simultaneously because the data they convey must be interweaved. For every lower triangle (x, y, z) we set $m_u(x, y) \leftarrow \min\{m_u(x, y), m_d(z, x) + m_u(z, y)\}$ and $m_d(x, y) \leftarrow \min\{m_d(x, y), m_u(z, x)+m_d(z, y)\}$. The perfect customization can be adapted analogously. We can use single SIMD-operations to process the upward and downward metrics in parallel given that the processor is capable of permuting vector components efficiently.

A current SSE-enabled processor supports all 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 $2^{31} - 1$. On many game graphs with uniform weights the graph diameter often fits into 16-bits and thus the 16-bit-saturated-add instruction provided by SSE is useful there.

7.4. 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 $U = \{((x_i, y_i), n_i)\}$ the set of arcs whose weights should be updated where

(x_i, y_i) is the arc ID and n_i the new weight. Note that modifying the weight of one arc can trigger new 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 n its new weight and by o its old weight. We first have to check whether n can be bypassed using a lower triangle. For this reason we iterate over all lower triangles (x, y, z) a perform $n \leftarrow \min\{n, m(z, x) + m(z, y)\}$. Furthermore, if $\{x, y\}$ was an edge in the original graph G we have to make sure that n is not larger than the original weight. If after both checks $n = m(x, y)$ holds then no change is necessary and no further changes are triggered. If o and n differ we iterate over all upper triangles (x, y, z) and test whether $m(x, z) + o = m(y, z)$ holds and if so the weight of the arc (y, z) must be set to $m(x, z) + n$. We add this change to the queue. Analogously we iterate over all intermediate triangles (x, y, z) and queue up a change to (z, y) if $m(x, z) + o = m(z, y)$ 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 QUERY

In this section we describe how to compute a shortest up-down path in $G_\pi^\wedge$ between two vertices s and t given a customized metric and how to unpack into a shortest path edge sequence in G.

8.1. Basic

The basic query runs two instances of Dijkstra's algorithm on $G_\pi^\wedge$ 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 $G_\pi^\wedge$. In [Geisberger et al. 2012] different search graphs are used for the upward and downward search. 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] called stall-on-demand. The optimization exploits that the shortest strictly upward sv-path in $G_\pi^\wedge$ can be bigger than the shortest sv-path (which can also go down). The search from s only finds upward paths and if we observe that a shorter up-down path exists then we can prune the 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 then an up-down path $s \rightsquigarrow y \rightarrow x$ exists that is no longer

than the shortest strictly upward sx path. This allows us to prune x by not relaxing its outgoing arcs.

8.3. Elimination Tree

We precompute for every vertex its parent's vertex ID in the elimination tree in a preprocessing step. This allows us to efficiently enumerate all vertices in $\mathrm{SS}(s)$ and $\mathrm{SS}(t)$ at query time. The vertices are enumerated increasing 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 lightgray area. In the third step all dashed arcs and the tree arcs departing in the middlegray area and in the fourth step the solid arcs and the remaining tree arcs follow.

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 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 Dijkstra-based approaches win.

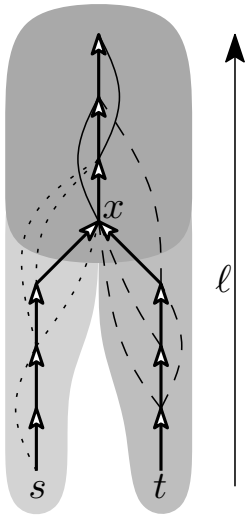


Fig. 7. The union of the darkgray and lightgray areas is the search space of s. Analogously the union of the darkgray and middlegray areas is the search space of t. The darkgray area is the intersection of both search spaces. The dotted arcs start in the search space of s but not in the search space 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 in 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 CH 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:

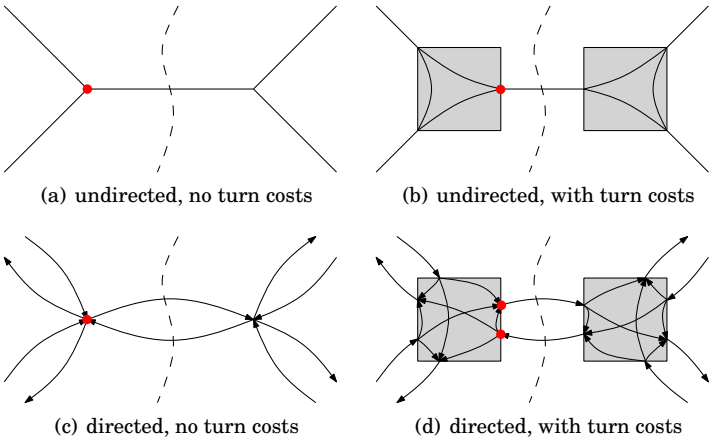


Fig. 8. Expanded turn models for combinations of directed and undirected, with and without turn costs. The dashed line represents the edge cut found by the bisector. The red dots represent the vertices in the derived vertex separator. The gray rectangle marks the boundaries of the turn gadgets.

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) exists with $m(p_i, p_{i+1}) = m(x, p_i) + m(x, p_{i+1})$ insert the vertex x between p_i and p_{i+1}. For minimum metrics also intermediate and upper triangles have to be considered.

9. A WORD ABOUT TURN COSTS

In practical road route planners (but not in the game scenario) it is important to be able to penalize or forbid turns.

Since our benchmark instances lack realistic turn cost data (while synthetic data tends to be very simplistic), we deemed it improper to experimentally evaluate CCH performance on turn costs. However, in this section using a theoretical argument we predict that turn costs have no major impact: They can be incorporated by adding turn cliques to the graph. Small edge cuts in the original graph correspond to small cuts in the turn-aware graph. Analyzing the exact growth of cuts (in the extended version), we conclude that the impact on search space size is at most a factor of 2 to 4. Practical performance might be better.

A straightforward implementation expands the graph by inserting turn clique gadgets as depicted in Figure 8. Note that many of these cliques will have the same weights and therefore a more compact representation that shares this information between cliques might be preferable in practice as described in [Delling and Werneck 2013; Geisberger and Vetter 2011]. However, the form in which these cliques are represented is only constant-tuning with respect to query times. In a real-world implementation constant-tuning is certainly important but it does not make the difference between being practicable and infeasible. In this section we solely want to establish that our approach is feasible and thus ignore these constants.

If the graph is undirected then turn costs can be added by replacing each vertex of degree d with a complete graph K_d as depicted in the Figures 8(a) and 8(b). If the graph is directed then the situation is slightly more complex as depicted in the difference between the Figures 8(c) and 8(d). A vertex of degree d is replaced by a directed $K_{d,d}$ complete bipartite graph. We refer to the vertices that only have incoming arcs inside the gadget as *exit* vertices and to the other vertices are *entry* vertices.

Recall that we determine our ordered by first computing an edge cut and then deriving a vertex separator from it. The first important observation is that a balanced edge cut in the unexpanded graph induces a balanced edge cut in the expanded graph. The second central observation needed is the same as the one used in the proof of Theorem 4.3: The performance is dominated by the size of the top level vertex separator. Suppose that the dashed cut represented in Figure 8 is the cut from which the top level vertex separator is derived. Denote by n the size of the vertex separator in the graph without turn costs. In the undirected case the size of this vertex separator does not increase as can be seen by comparing 8(a) to 8(b). We therefore expect the query running time performance of the CH to be mostly independent of whether turn costs are used or not. In the directed case the size of the derived top level vertex separator is doubled as can be seen by comparing 8(c) to 8(d). The top level clique in the CH is thus a K_{2n}. The number of arcs in the search spaces therefore increases by a factor of

$$\frac{|K_{2n}|}{|K_n|} = \frac{1/2(2n-1)2n}{1/2(n-1)n} \to 4$$

for n tending towards ∞. If you implement Customizable Contraction Hierarchies precisely as described so far then the search space sizes will indeed increase by this factor 4 in terms of arcs. However, one can do better. As can be seen in Figure 8(d) it is possible to assure that half of the separator vertices in the turn clique are entry vertices while the other vertices are exit vertices. Arcs between two entry vertices or two exit vertices are guaranteed to have a weight of ∞ in both directions for any metric (that respects the direction of the directed input graph). These arcs may therefore be removed from the CH. Instead of a top level K_{2n} complete clique, a complete bipartite clique $K_{n,n}$ is thus sufficient. The number of arcs is therefore only expected to increase by a factor of

$$\frac{|K_{n,n}|}{|K_n|} = \frac{n^2}{1/2(n-1)n} \to 2$$

for n tending towards ∞. To exploit this observation, we propose the following approach: First construct the CH without removing any arcs. Then, still during the metric-independent phase, "customize" it with a metric where all arcs going the wrong way through a one-way street have weight ∞ (and all others have finite weight, e. g., 0). Finally, remove from the CH all arcs that have both an upward and a downward weight of ∞ in the CH. The customization works without modifications. If the elimination-tree query should be used then it is important to construct the elimination tree before removing the arcs.

We conclude that while turn costs do not come for free, they most likely do not represent a major obstacle.

10. EXPERIMENTS

In this section we present our careful and extensive experimental evaluation of the algorithms introduced and described before.

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-CPU 8-core Intel Xeon E5-2670 processor (Sandy Bridge architecture) clocked at 2.6 GHz, with 64 GiB of DDR3-1600 RAM, 20 MiB of L3 and 256 KiB of L2 cache. The order computation experiments (see Table II) were run on a single core of an Intel Core i7-2600K CPU processor.

Table I. Instances. We report the number of vertices and of directed arcs of the benchmark graphs. We further present the number of edges in the induced undirected graph.

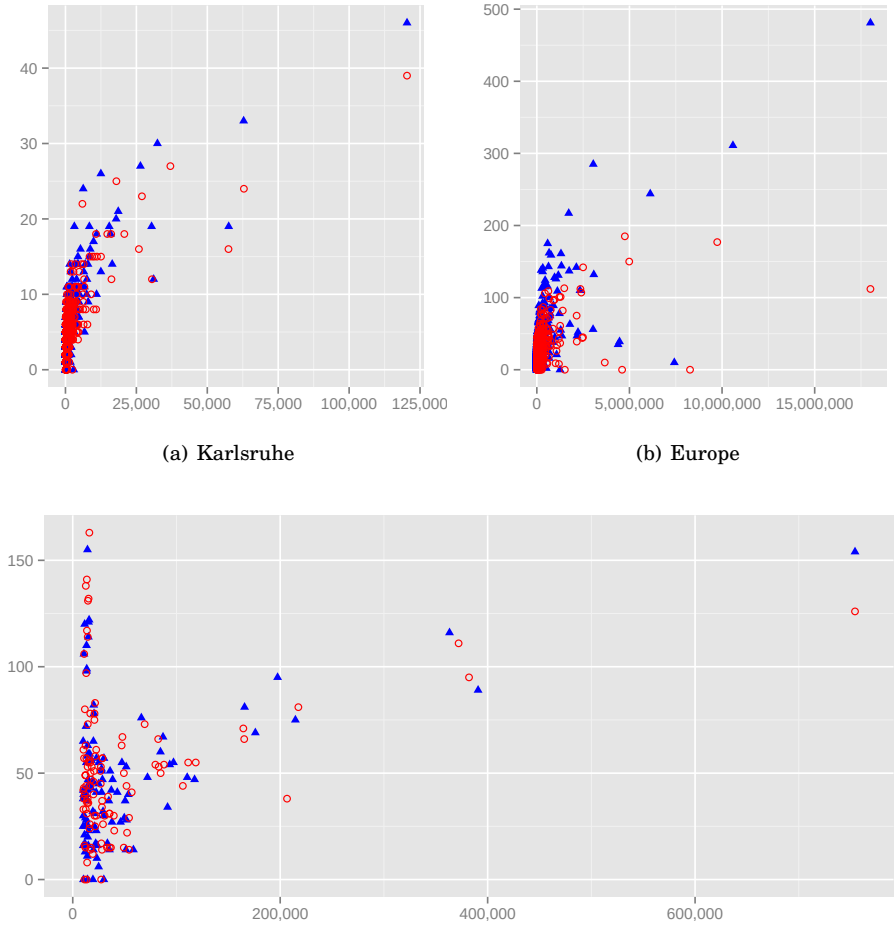
Instance	# Vertices	# Arcs	# Edges	Symmetric?
Karlsruhe	120 412	302 605	154 869	no
TheFrozenSea	754 195	5 815 688	2 907 844	yes
Europe	18 010 173	42 188 664	22 211 721	no

Table II. Orders. Duration of order computation in seconds. No parallelization was used.

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

Fig. 9. All vertices in the PTV-Europe graph.

Instances. We consider three large instances of practical relevance (see Table I): The Europe graph was made available by PTV[1] for the DIMACS challenge [Demetrescu et al. 2009]. The vertex positions are depicted in Figure 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 (a common method to remove bogus vertices). It 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[2]) that sometimes have a high number of vertices on highways to model road bends. It is clear that degree 2 vertices do not hamper the performance of CHs but for some of our side-experiments this difference might be relevant. 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 non-walkable 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 $\sqrt{2}$, while 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. Table I reports the instance sizes.



(a) Karlsruhe

(b) Europe

(c) TheFrozenSea

Fig. 10. The amount of vertices in the separator (vertical) vs the number of vertices in the subgraph being bisected (horizontal). We only plot the separators for (sub)graphs of at least 1000 vertices. The red hollow circles is KaHIP and the blue filled triangles is Metis.

10.1. Orders

We analyze three different vertex orders: 1) The greedy order is an order in the spirit of [Geisberger et al. 2012]. 2) The Metis graph partitioning package contains a tool called `ndmetis` to create ND-orders. 3) KaHIP provides just graph partitioning tools. As far as we know tools to directly compute ND-orders are planned by the authors but not yet finished. We therefore implemented a very basic program on top of it. For every graph we compute 10 bisections with different random seeds using the "strong" configuration. 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

[1] http://www.ptvgroup.com
[2] http://www.openstreetmap.org

Table III. Construction of the Contraction Hierarchy. We report the time in seconds required to compute the arcs in $G^{\wedge}_{\pi}$ given a KaHIP ND-order π. No witness search is performed. No weights are assigned (yet).

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

for KaHIP to 20%. Note that our program is purely tuned for quality. It is certainly possible to trade much speed for a negligible (or even no) decrease in quality. Table II reports the times needed to compute the orders. Interestingly, Metis outperforms even the greedy strategy. Figure 10 shows the sizes of the computed separators. As expected KaHIP results in better quality. The road graphs seem to have separators following a $\Theta(\sqrt[3]{n})$-law. 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. The grids have $\Theta(\sqrt{n})$ separators, whereas at the higher levels the choke points results in separators that approximately follow a $\Theta(\sqrt[3]{n})$-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.

10.2. CH Construction

Table III compares the performance of our specialized Contraction Graph datastructure (see Section 5) to the dynamic adjacency structure (see [Geisberger et al. 2012]) to compute undirected and unweighted CHs. We do not report numbers for the hash-based approach (see [Zeitz 2013]) as it is fully dominated. Our datastructure dramatically improves performance (recall from Section 5 that it also requires less memory). However to be fair, our approach cannot immediately be extended to directed or weighted graphs (i. e., without employing customization).

10.3. CH Size

In Table IV we report the resulting CH sizes for various approaches. Computing a CH on Europe *without witness search* with the greedy order is infeasible even using the Contraction Graph datastructure. This is even true if we only want to count the number of arcs: We aborted calculations after several days. We can however say with certainty that there are at least 1.3×10^{12} arcs in the CH 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 compared to Dijkstra's algorithm on the input graph. However, on the Karlsruhe graph we can actually compute the CH 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 greedy orders in a metric-independent setting (i. e., without witness search) results in unpractical CH sizes. However they also show that a greedy order exploiting the weight struc-

Table IV. Size of the Contraction Hierarchies for different instances and orders. We report the number of undirected as well as upward directed arcs of the CH, as well as the number of supporting lower triangles. As an indication for query performance, we report the average search space size in vertices and arcs (both metric-independent undirected and upward weighted), by sampling the search space of 1000 random vertices. Metis and KaHIP orders are metric-independent. Greedy orders are metric-dependent. 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.

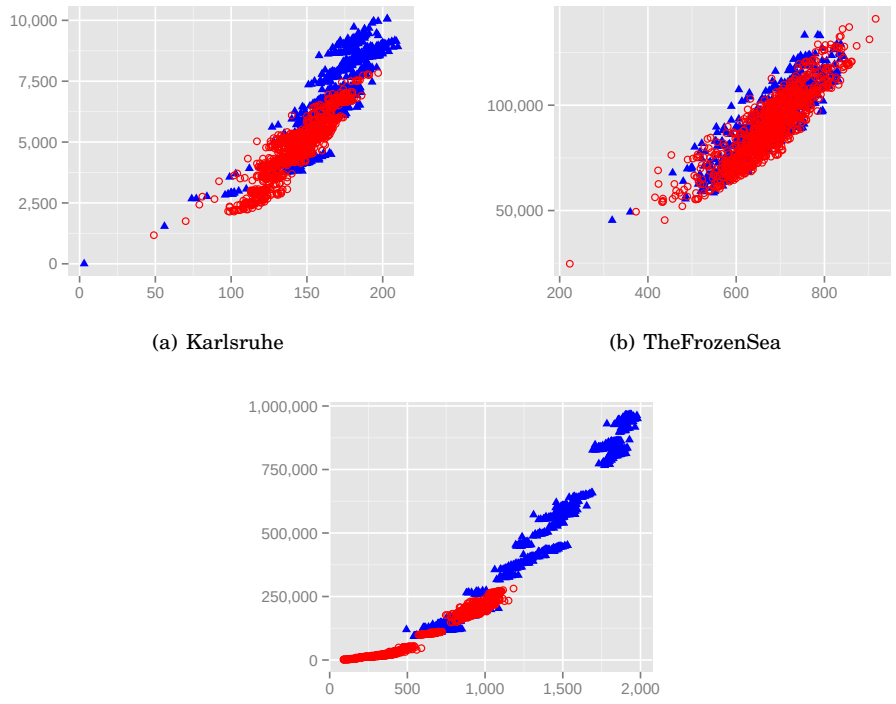
						Average search space size			
			# Arcs $[\cdot 10^3]$		# Triangles	undirected		upward	
	Order	Witness search	undir.	upward	$[\cdot 10^3]$	# Vertices	# Arcs	# Vertices	# Arcs
Karlsruhe	Greedy	none	21 926	17 661	37 439 858	5 870	15 786 622	5 246	11 281 564
		heuristic	—	244	—	—	—	108	503
		perfect	—	239	—	—	—	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	Greedy	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	Greedy	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

Table V. Elimination tree characteristics. 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 (undirected) input graphs.

		# Children		Height		Treewidth
Instance	Order	avg.	max.	avg.	max.	(upper bound)
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	1283.45	2017	876
	KaHIP	1	7	654.07	1232	479

ture dominates ND-orders (for a more detailed discussion see below). In Figure 11 we plot the number of arcs in the search space vs 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 a tie. Table V examines the elimination tree. Note that the height of the elimination tree corresponds[3] 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 spe-

[3]The numbers in Table IV and Table V deviate a little because the search spaces in the former table are sampled while in the latter we compute precise values.



(a) Karlsruhe

(b) TheFrozenSea

(c) Europe

Fig. 11. The number of vertices (horizontal) vs the number of arcs (vertical) in the search space of 1000 random vertices. The red hollow circles is KaHIP and the blue filled triangles is Metis.

cial vertex exists that has a search space significantly differing from the the numbers shown in Table V. The elimination tree has a relatively small height compared to the number of vertices in G (in particular, it is not just a path).

The treewidth of a graph is a measure widely used in theoretical computer science. Many NP-hard problems have been shown to be solvable in polynomial time on graphs of bounded treewidth. The notion of treewidth is deeply coupled with the notion of chordal super graphs and vertex separators. See [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 (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 grows sublinearly. Our findings from Figure 10 suggest that assuming a $O(\sqrt[3]{n})$ treewidth for road graphs of n vertices might come close to reality. Further investigation into algorithms explicitly exploiting treewidth might be promising. The works of [Chaudhuri

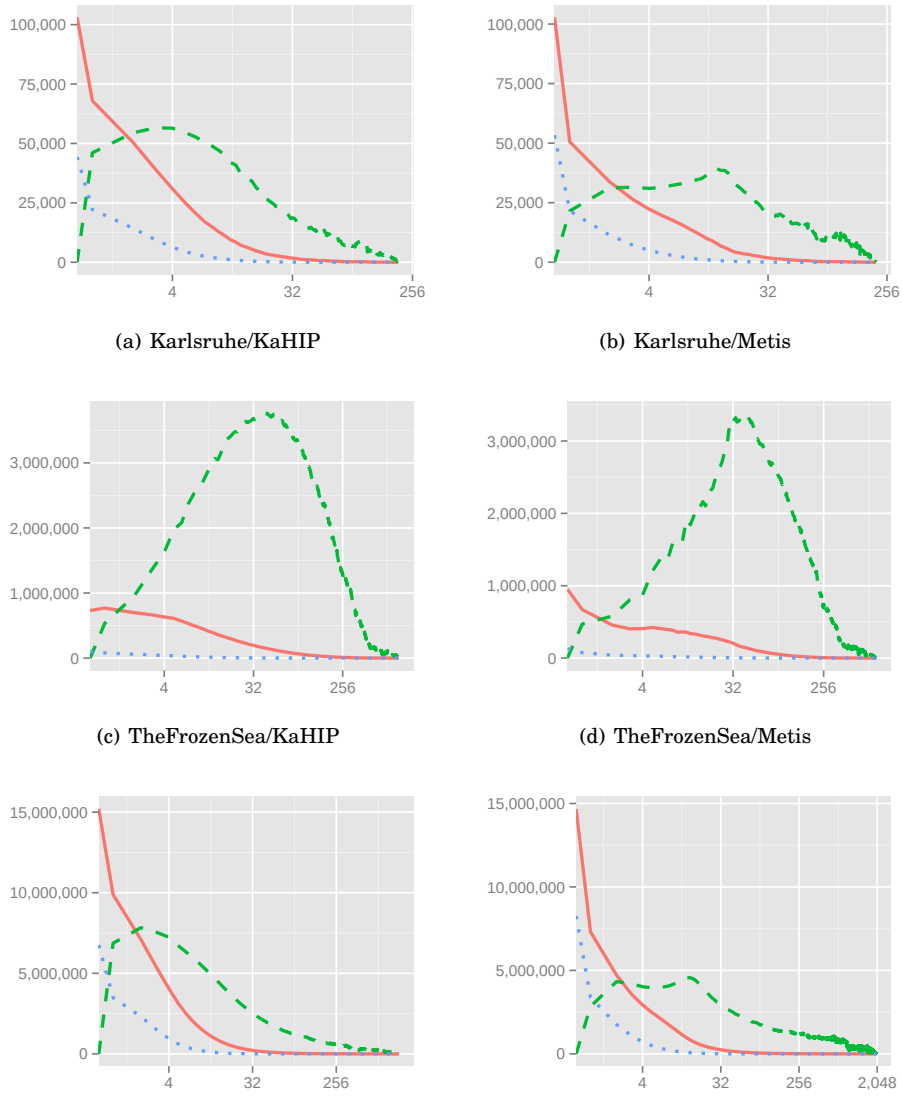
Table VI. Detailed analysis of the size of CHs. We evaluate uniform, random and distance weights on the Karlsruhe input graph. Random weights are sampled from $[0, 10000]$. The distance weight is the straight distance along a perfect Earth sphere's surface. All weights respect one-way streets of the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

Instance	Metric	Order	Witness search	# Upward arcs	Avg. upward search space	
					# Vertices	# Arcs
Karlsruhe	Distance	Greedy	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	Greedy	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	Greedy	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	Greedy	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

and Zaroliagis 2000; Planken et al. 2012] seem like a good start. Also, determining the precise treewidth could prove useful. However note, that an order inducing a minimum treewidth is not necessarily an optimal CH order as the path example of Figure 1 shows.

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 CH 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 highways do not have many degree 2 vertices in the input graph. (Note that for different data sources this assumption might not hold.) Highways are therefore also preferred by the uniform metric. We expect a more an instance with more degree 2 nodes 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 CH 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 12 illustrates the amount of vertices and arcs in each level of a CH. The many highly ranked extremely thin levels are a result of the top level separator clique: Inside a clique every vertex must be on its own level. A few big separators therefore significantly increase the level count.

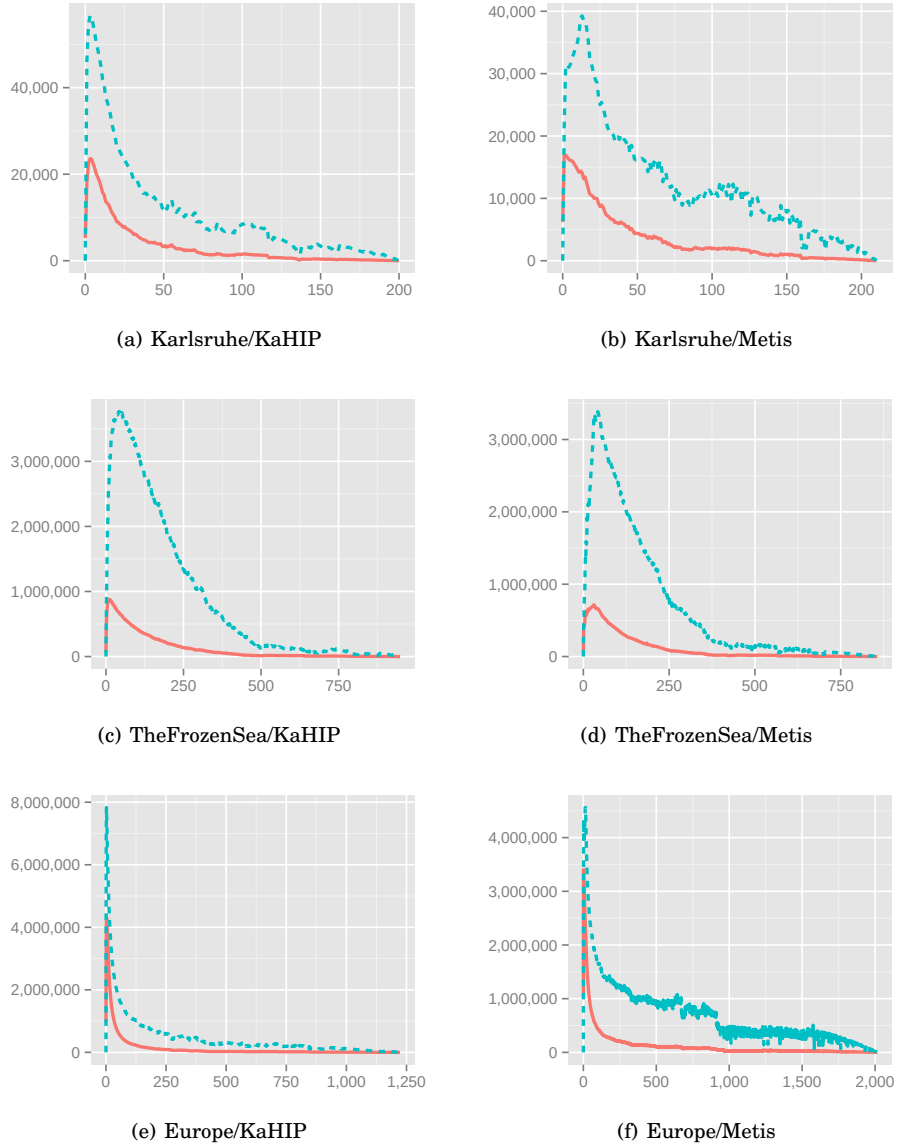


(a) Karlsruhe/KaHIP (b) Karlsruhe/Metis

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

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

Fig. 12. The number of vertices per level (blue dotted line), arcs departing in each level (red solid line) and lower triangles in each level (green dashed line). Warning: In contrast to Figure 13 these figures have a logarithmic x-scale.



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

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

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

Fig. 13. The number of lower triangles per level (blue dashed line) and the time needed to enumerate all of them per level (red solid line). The time unit is 100 nanoseconds. If the time curve thus rises to 1 000 000 on the plot the algorithm needs 0.1 seconds. Warning: In contrast to Figure 12 these figures do not have a logarithmic x-scale.

10.4. Triangle Enumeration

We first evaluate the running time of the adjacency-array-based triangle enumeration algorithm. Figure 13 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 lower levels as these are the levels were the optimization is most effective. How-

Table VII. Precomputed triangles. As show in Section 6 the memory needed is proportional to $2t + m + 1$, where t is the triangle count and m the number of arcs in the CH. 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 how much percent of the total unaccelerated enumeration time is spent below the given threshold level. We chose the threshold level such that this factor is about $33\,\%$.

		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
	# CH arcs [10^3]	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
	# CH 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

Table VIII. Customization performance. We report the time needed to compute a maximum customized metric given an initial pair of upward and downward metrics. We show the impact of enabling SSE, precomputing triangles (Pre. trian.), multi-threading (# Thr.), and customizing several metric pairs at once.

				Karlsruhe		TheFrozenSea		Europe	
SSE	Pre. trian.	# Thr.	# Metrics 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

ever, 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.

10.5. Customization

In Table VIII we report the times needed to compute a maximum metric given an initial one. 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 below one second. When amortized[4], we even achieve 415 ms which is only slightly above the (non-amortized) 347 ms 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 and use a turn-aware data structure making an exact comparison difficult.)

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 GB of precomputed triangles. We therefore ran additionally experiments with different parameters

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

Table IX. Detailed customization performance on TheFrozenSea. We report the time needed to compute a maximum customized metric from an initial metric. We show the impact of exploiting undirectedness, customizing several metrics at once, reducing the bitwidth of the metric, enabling SSE, multi-threading (# 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.

Undirected	# Metrics	Metric bits	SSE	# Threads	Precomputed triangles	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

Table X. Partial update performance. We report time required in milliseconds and number of arcs changed for partial metric updates. We report median, average and maximum over 10000 runs. In each run we change the upward and the downward weight of a single random arc in $G^{\wedge}_{\pi}$ (the arc is not necessarily in G) to random values in $[0, 10^5]$. The metric is reset to initial state between runs. Timings are sequential without SSE. No triangles were precomputed.

		Arcs removed from queue			Partial update time [ms]		
		med.	avg.	max.	med.	avg.	max.
Karlsruhe	Metis	1	4.1	442	0.001	0.004	0.9
	KaHIP	1	3.7	354	0.001	0.003	1.0
TheFrozenSea	Metis	8	395.7	15023	0.021	2.2	91.4
	KaHIP	8	382.4	12035	0.017	2.0	99.2
Europe	Metis	1	89.3	16997	0.003	1.0	219.3
	KaHIP	1	38.8	10666	0.003	0.2	87.2

and report the results in Table IX. The experiments show that it is possible to fully customize TheFrozenSea in an amortized[5] time of 1.06s without precomputing triangles or using multiple cores. However a whole second is still too slow to be usable (with graphics, network and game logic also requiring resources). We therefore evaluated the time needed by partial updates as described in Section 7.4. We report our results in Table X. 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 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. Interestingly in the worst observed case, using the KaHIP order triggers less changes on TheFrozenSea graph than using the Metis order but an update needs more time. The reason for this is that the KaHIP order results in significantly more triangles and thus the work per arc is higher than what is needed with the Metis order.

For completeness we report the running times of the perfect customizations in Table XI. Note that a perfect customization is not a necessary step of our proposed tool chain. Hence, optimizing this code path had a low priority.

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

Table XI. Perfect Customization. We report the time required to turn an initial metric into a perfect metric. Runtime is given in seconds, without use of SSE or triangle precomputation.

	Karlsruhe		TheFrozenSea		Europe	
# Thr.	Metis	KaHIP	Metis	KaHIP	Metis	KaHIP
1	0.15	0.13	30.54	33.76	67.01	32.96
16	0.03	0.02	3.26	4.37	14.41	5.47

10.6. Distance Query

We experimentally evaluated the running times of the queries algorithms. For this we ran 10^6 shortest path distance queries with the source and target vertices picked uniformly at random. (For Europe + Distance we only ran 10^4 queries.) The presented times are averaged running times on a single core without any SSE.

In Table XII we compare the query running times of weighted CHs with Customizable CHs (CCHs). To construct the weighted CHs we used a (non-perfect) witness search whereas no witness search was used for the metric-independent CHs. We further reordered the vertices in the metric-independent CHs by ND-order. Preliminary experiments showed that this reordering results in better cache behavior and a speed-up of about 2 to 3 because much query time is spent on the topmost clique. We evaluate the basic query, the stall-on-demand optimization, and the elimination-tree based query. Note that the latter only works for metric-independent CHs (as the metric-independent search spaces of weighted CHs get huge).

In comparison to the numbers reported in the original CH paper [Geisberger et al. 2012] our running times for weighted CHs tend to be slightly faster. However, our machine is faster which should explain most differences. The only exception is the Europe graph with the distance metric. Here, our measured running time of only 0.540 ms is disproportionately faster. We suppose that the reason is that our order is better as we do not use lazy update and thus have a higher preprocessing time. As already observed by the original authors we confirm that the stall-on-demand heuristic improves running times by a factor 2 to 5 compared to the basic algorithm on weighted CHs. When using ND-order the stalling query is however slower: The search spaces of weighted CHs are spare whereas in the metric-independent case they are dense. This significantly increases the number of arcs that must be tested in the stalling test and explains why stalling is not useful.

For the metric-independent CHs the basic query algorithm (i. e., bidirectional search with stopping criterion) visits large portions of the search space, as can be seen by comparing the search space sizes from Table IV with the numbers reported in Table XII. For this reason, it pays off to use the elimination tree based query algorithm. It always visits the whole search space but as we see these are only slightly more vertices. However, it does not need a priority queue and therefore spends less time per vertex. Another advantage of the elimination tree based algorithm is that the code paths do not depend on the metric. This means that query times are completely independent of the metric as can be seen by comparing the running times of the travel time metric to the distance metric. For the basic query algorithms the metric has a slight influence on the performance. A stalling query on the weighted CH with travel time is on Europe about a factor of 5 faster than the elimination tree based algorithm. However for the distance metric this is no longer the case. Here, the metric-independent elimination tree based approach is even faster by about 20% because of the lack of priority queue.

In Table XIII, we give a more in-depth experimental analysis of the elimination tree query algorithm. We break the running times up into the time needed to compute the least common ancestor (LCA), the time needed to reset the tentative distances and the

Table XII. Contraction Hierarchies query performance. We report the query time *in microseconds* as well as the search space visited (we use visited to differentiate from the maximum reachable search space given in Table IV). For query algorithms that use stalling, we additionally report the number of vertices stalled after queue removal, as well as the number of arcs touched during the stalling test. Note that the search space figures do not contain such stalled vertices. 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 (non-perfect) witness search is used, whereas "-w" means that no witness search is used. "greedy+w" corresponds to the original CHs. The metrics used for the non-greedy CHs are directed and maximum.

				Visited search space		Stalling		Time
Instance	Metric	Variant	Algorithm	# Vertices	# Arcs	# Vertices	# Arcs	[µs]
Karlsruhe	Travel-Time	Greedy+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
	Distance	Greedy+w	Basic	208	1978	—	—	57
			Stalling	69.5	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
TheFrozenSea	Map-Distance	Greedy+w	Basic	1 199	12 692	—	—	539
			Stalling	319	3 459.3	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
Europe	Travel-Time	Greedy+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
	Distance	Greedy+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

Table XIII. Detailed elimination tree performance. 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 (which is metric-dependent).

			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	1471.2	1496.3	323.9	1390.6
		KaHIP	3.4	9.9	399.4	413.3	252.7	
	Distance	Metis	4.7	19.0	1494.5	1519.9	608.8	3111.0
		KaHIP	3.6	10.0	411.6	425.8	524.1	

time needed to relax all arcs. We further report the total distance query time (which is in essence the sum of the former three) and 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.

Fair query time comparisons with CRP [Delling et al. 2014] are difficult because they nearly only report turn-aware query running times, whereas the graphs we tested do not use turns. As far as we are aware, non-turn-aware query performance was only published in [Delling et al. 2011], but here queries were parallelized using two cores: The forward and backward searches are run in parallel. The authors report queries in 0.72 ms for travel time and 0.79 ms for distance metric on Europe. This is slower than our sequential query times of 0.41 ms and 0.43 ms, respectively. (Note that these experiments were run on a slightly different machine than ours.)

We have shown in Table VIII that ND-orders can be combined with perfect witness search to get CHs of smaller search spaces. This could be exploited to achieve (even) faster query times as the number of arcs decrease by a factor $\approx$2 on road and $\approx$4 on game maps. As the elimination-tree query spends nearly all of its time visiting arcs we expect its running time to go down by about the same factor. However, a perfect customization is slower by a factor of $\approx$3 (c. f. Table XI). In total, combining ND-orders and perfect witness search yields another Pareto-optimal trade-off between customization time and query time, but one that is less comparable to CRP.

11. CONCLUSIONS

We have extended Contraction Hierarchies to a three phase customization approach and demonstrated that the approach is practicable and efficient not only on real world road graphs but also on game maps. Furthermore, we have performed an extensive experimental analysis of its performance that hopefully sheds some light onto the inner workings of Contraction Hierarchies.

11.1. Future Work

While a graph topology with small cuts is one of the main driving force behind the running time performance of Contraction Hierarchies, it is clear from Table IV that better metric-dependent orders can be constructed by exploiting additional travel time metric specific properties. While the works of [Abraham et al. 2010] explain this effect to some extend we believe that it is worthwhile to further investigate this gap between metric-dependent and metric-independent orders.

Better nested dissection orders directly increase the performance of the introduced Customizable Contraction Hierarchies. Research aiming at providing better vertex orders or proving that the existing orders are close to optimal seems useful. Revisiting all of the existing Contraction Hierarchy extensions to see which can profit form a metric-independent vertex order or can be made customizable seems worthwhile. An obvious candidate are Time-Dependent Contraction Hierarchies [Batz et al. 2013] where computing a good metric-dependent order has proven relatively expensive.

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

REFERENCES

Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. 2012. Hierarchical Hub Labelings for Shortest Paths. In *Proceedings of the 20th Annual European Symposium on Algorithms (ESA'12) (Lecture Notes in Computer Science)*, Vol. 7501. Springer, 24–35.

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)*. SIAM, 782–793.

Hannah Bast, Daniel Delling, Andrew V. Goldberg, Matthias Müller–Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F. Werneck. 2014. *Route Planning in Transportation Networks*. Technical Report MSR-TR-2014-4. Microsoft Research. http://research.microsoft.com/apps/pubs/?id=207102

Gernot Veit Batz, Robert Geisberger, Peter Sanders, and Christian Vetter. 2013. Minimum Time-Dependent Travel Times with Contraction Hierarchies. *ACM Journal of Experimental Algorithmics* 18, 1.4 (April 2013), 1–43.

Reinhard Bauer, Tobias Columbus, Ignaz Rutter, and Dorothea Wagner. 2013. Search-Space Size in Contraction Hierarchies. In *Proceedings of the 40th International Colloquium on Automata, Languages, and Programming (ICALP'13) (Lecture Notes in Computer Science)*, Vol. 7965. Springer, 93–104.

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

Soma Chaudhuri and Christos Zaroliagis. 2000. Shortest Paths in Digraphs of Small Treewidth. Part I: Sequential Algorithms. *Algorithmica* (2000).

Daniel Delling, Andrew V. Goldberg, Thomas Pajor, and Renato F. Werneck. 2011. Customizable Route Planning. In *Proceedings of the 10th International Symposium on Experimental Algorithms (SEA'11) (Lecture Notes in Computer Science)*, Vol. 6630. Springer, 376–387.

Daniel Delling, Andrew V. Goldberg, Thomas Pajor, and Renato F. Werneck. 2014. Customizable Route Planning in Road Networks. *Transportation Science* (2014). http://research.microsoft.com/apps/pubs/?id=198358 accepted for publication.

Daniel Delling, Andrew V. Goldberg, Ilya Razenshteyn, and Renato F. Werneck. 2011. Graph Partitioning with Natural Cuts. In *25th International Parallel and Distributed Processing Symposium (IPDPS'11)*. IEEE Computer Society, 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)*. SIAM, 30–44.

Daniel Delling and Renato F. Werneck. 2013. Faster Customization of Road Networks. In *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13) (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. *Numer. Math.* 1 (1959), 269–271.

Delbert R. Fulkerson and O. A. Gross. 1965. Incidence Matrices and Interval Graphs. *Pacific J. Math.* 15, 3 (1965), 835–855.

Robert Geisberger, Peter Sanders, Dominik Schultes, and Christian Vetter. 2012. Exact Routing in Large Road Networks Using Contraction Hierarchies. *Transportation Science* 46, 3 (August 2012), 388–404.

Robert Geisberger and Christian Vetter. 2011. Efficient Routing in Road Networks with Turn Costs. In *Proceedings of the 10th International Symposium on Experimental Algorithms (SEA'11) (Lecture Notes in Computer Science)*, Vol. 6630. Springer, 100–111.

Alan George. 1973. Nested Dissection of a Regular Finite Element Mesh. *SIAM J. Numer. Anal.* (1973).

Alan George and Joseph W. Liu. 1978. A Quotient Graph Model for Symmetric Factorization. In *Sparse Matrix Proceedings*. SIAM.

John R. Gilbert and Robert Tarjan. 1986. The analysis of a nested dissection algorithm. *Numer. Math.* (1986).

Martin Holzer, Frank Schulz, and Dorothea Wagner. 2008. Engineering Multilevel Overlay Graphs for Shortest-Path Queries. *ACM Journal of Experimental Algorithmics* 13, 2.5 (December 2008), 1–26.

Haim Kaplan, Ron Shamir, and Robert Tarjan. 1999. Tractability of Parameterized Completion Problems on Chordal, Strongly Chordal, and Proper Interval Graphs. *SIAM J. Comput.* (1999).

George Karypis and Vipin Kumar. 1999. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing* 20, 1 (1999), 359–392. http://dx.doi.org/10.1137/S1064827595287997

Richard J. Lipton, Donald J. Rose, and Robert Tarjan. 1979. Generalized Nested Dissection. *SIAM J. Numer. Anal.* 16, 2 (April 1979), 346–358.

Léon Planken, Mathijs de Weerdt, and Roman van Krogt. 2012. Computing All-pairs Shortest Paths by Leveraging Low Treewidth. *Journal of Artificial Intelligence Research* (2012).

Peter Sanders and Christian Schulz. 2013. Think Locally, Act Globally: Highly Balanced Graph Partitioning. In *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13) (Lecture Notes in Computer Science)*, Vol. 7933. Springer, 164–175.

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 (2000), 1–23.

Sabine Storandt. 2013. Contraction Hierarchies on Grid Graphs. In *Proceedings of the 36rd Annual German Conference on Advances in Artificial Intelligence (Lecture Notes in Computer Science)*. Springer.

Nathan Sturtevant. 2012. Benchmarks for Grid-Based Pathfinding. *Transactions on Computational Intelligence and AI in Games* (2012).

Mihalis Yannakakis. 1981. Computing the minimum fill-in is NP-complete. *SIAM J. Algebraic Discrete Methods* (1981).

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