Customizable Contraction Hierarchies*

Julian Dibbelt, Ben Strasser, and Dorothea Wagner

Karlsruhe Institute of Technology (KIT), Kaiserstruhe 12, 76131 Karlsruhe, Germany
{dibbelt,strasser,dorothea.wagner}@kit.edu

Abstract. We consider the problem of quickly computing shortest paths in weighted graphs given auxiliary data derived in an expensive preprocessing phase. By adding a fast weight-customization phase, we extend Contraction Hierarchies [12] to support the three-phase workflow introduced by Delling et al. [6]. Our Customizable Contraction Hierarchies use nested dissection orders as suggested in [3]. We provide an in-depth experimental analysis on large road and game maps that clearly shows that Customizable Contraction Hierarchies are a very practicable solution in scenarios where edge weights often change.

1 Introduction

Computing optimal routes in road networks has many applications such as navigation devices, logistics, traffic simulation or web-based route planning. 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 [1] for an overview. Among the most successful techniques are Contraction Hierarchies (CH) [12], which have been utilized in many scenarios. However, their preprocessing is in general metric-dependent, e.g., edge weights are needed a-priori. 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 [6], extending the multi-level-overlay MLD techniques of [20,15]. 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.

Besides road networks, game maps are an interesting application where fast shortest path computations are important (c.f. [21]). In real-time strategy games the basic topology of the map often is fixed. However, since buildings are constructed or destroyed, fields are rendered impassable or freed up. Yet, because the *fog of war,* every player has his own knowledge of the map: 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

* Partial support by DFG grant WA654/16-2, EU grant 288094 (eCOMPASS), and Google Focused Research Award.

J. Gudmundsson and J. Katajainen (Eds.): SEA 2014, LNCS 8504, pp. 271–282, 2014.

to land. This results in vastly different, evolving metrics for different unit types per player, making metric-dependent preprocessing difficult to apply.

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 [12]. This approach was proposed by [3], and a preliminary case study can be found in [23]. A similar idea was followed by [9], 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 [18]. 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. Furthermore, note that while customizable speedup techniques for shortest path queries may be a very recent development, the idea to use ND-orders to compute CH-like structures is far older and widely used in the *sparse matrix solving* community. We refer the interested reader to [13,17].

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 [6] 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 [12].

An *extended version* of this paper with more detailed experiments is available at `http://arxiv.org/abs/1402.0402`. It contains concepts that are not strictly necessary for the considered workflow but might be useful for extensions of it.

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*. We consider *simple* graphs that have no loops or multi-edges. We denote by $N(v)$ the set of adjacent vertices of v. 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 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*. *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* $w_d : E \to \mathbb{R}^+$. The *vertex contraction* of v in G consists of removing v and all incident edges and inserting edges between all neighbors if not already present. We iteratively contract all vertices according to their rank resulting in a set Q of additional edges. We direct the edges in E and in Q upward from lower ranks to higher ranks resulting in the upward directed search graph $G^{\wedge}_{\pi}$. The search space $\mathrm{SS}(v)$ of a vertex v is the subgraph of $G^{\wedge}_{\pi}$ 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 from s restricted to $\mathrm{SS}(s)$ and from t restricted to $\mathrm{SS}(t)$ [12]. 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 of minimal rank $\pi^{-1}(v)$. As shown in [3] the set of vertices on the path from v to $\pi(n)$ is the set of vertices in $\mathrm{SS}(v)$. Note that the elimination tree is only defined for undirected unweighted CHs. A *lower triangle* of an arc (x, y) in $G_\pi^\wedge$ 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* a triple such that (x, z) and (y, z) exist. Weights on $G_\pi^\wedge$ are called *metrics*. A weight on G is extended to an *initial metric* on $G_\pi^\wedge$ by assigning ∞ to all non-original arcs. A metric m is called *customized* if for all lower triangles (x, y, z), it holds that $m(x, y) \leq m(z, x) + m(z, y)$.

Note that our CHs build on undirected graphs. To model one-way streets we use two different up- and downward metrics, setting either m_u or m_d to ∞.

3 Phase 1: Preprocessing the Graph Topology

Metric-Independent Order. To support metric-independence, we use *nested dissection* orders (ND-orders) as suggested in [3]. An order π for G is computed recursively by determining a minimum balanced separator S that splits G into parts induced by the vertex sets A and B. The lowest ranks are assigned to the vertices in A, the next ranks to the vertices in B, and finally the highest ranks are assigned to S. Computing ND-orders requires good graph partitioning, and recent years have seen heuristics that solve the problem very well even for continental road graphs [19,8,7]. Note that these partitioners compute edge cuts. On our instances, a good vertex separator can be derived by arbitrarily picking one incident vertex for each edge.

Theorem 1. *Let G be a graph with recursive balanced separators with $O(n^\alpha)$ vertices. If G has 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 CH in terms of vertices and arcs.*

Proof. (sketch, see extended version) In a lemma the authors of [17] show that a clique exists in the CH of the size of the minimum balanced separator. We observe that this is the top level separator and that it dominates all lower separators.

Constructing the Contraction Hierarchy. While the CH can be constructed given the order and the input graph as described in [12], we improve construction performance by (temporarily) considering all inputs as undirected and unweighted graphs. By designing a specialized *Contraction Graph* datastructure, detailed in the extended version, we significantly accelerate construction time while limiting space use during construction. Denote by n the number of vertices, m the number of edges in G, by m' the number of edges in $G_\pi^\wedge$, and by $\alpha(n)$ the inverse $A(n, n)$ Ackermann function. Our algorithm needs $O(m'\alpha(n))$ time and $O(m)$ space. The core idea, introduced in [14], consists of reducing vertex contraction

to edge contraction, maintaining an independent set of virtually contracted vertices. Also, we build the elimination tree and derive for every vertex its level $\ell(x)$ in $G_\pi^\wedge$.

Memory Order. 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 greatly improves cache locality.

4 Phase 2: Customizing the Metric

In this section, we describe how to customize a metric and parallelize the process. Furthermore, we show how to update the weight of a single arc. A base operation for our algorithms is efficiently enumerating all lower triangles of an arc. It can be implemented using adjacency arrays or accelerated using extra preprocessing.

Basic Triangle Enumeration. Construct an upward and a downward adjacency array for $G_\pi^\wedge$, 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. This approach requires space proportional to the number of arcs in $G_\pi^\wedge$. All upper and intermediate triangles are found by merging $N_u(x)$ and $N_u(y)$ (respectively $N_d(y)$).

Triangle Preprocessing. Instead of merging the neighborhoods, we propose to create an adjacency-array-like 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 in $G_\pi^\wedge$ but allows for faster access.

Metric Customization. Our algorithm iterates over all levels from the bottom to the top. On level i, it iterates (using multiple threads) over all arcs (x, y) with level $\ell(x) = i$. For each such arc (x, y), the algorithm enumerates all lower triangles (x, y, z) and performs $m(x, y) \leftarrow \min\{m(x, y), m(z, x) + m(z, y)\}$. The operation maintains the shortest path structure. Furthermore, the resulting metric is customized by definition. Note that we synchronize the threads between levels. Then, since we only consider lower triangles, no read/write conflicts occurs. Hence, no locks or atomic operations are needed.

Vectorization. A metric can be replaced by an interleaved set of k metrics by replacing every $m(x, y)$ by a vector of k elements. All k metrics are customized in one go, amortizing triangle enumeration time. To customize directed graphs, recall that we first extract upward and downward weights w_u and w_d. These are independently transformed into initial upward and downward metrics m_u and m_d.

However, they must not be customized independently: 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 $m_d(x, y) \leftarrow \min\{m_d(x, y), m_d(z, y)+m_u(z, x)\}$. When using interleaved metrics it is straightforward to use SSE (avoid addition overflow by setting $\infty = \text{int}_{\max}/2$).

Partial Updates. 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. Observe that modifying the weight of one arc can trigger new changes, but only to higher arcs. 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. Let (x, y) be the arc removed from the queue, n its new weight, and o its old weight. We first check if (x, y) can be bypassed via a lower triangle, improving n: We iterate over all lower triangles (x, y, z) and perform $n \leftarrow \min\{n, m(z, x) + m(z, y)\}$. Finally, if $\{x, y\}$ is an edge in the original graph G, we ensure that n is not larger than the original weight. If after both checks $n = m(x, y)$ holds, no change is necessary and no further changes are triggered. Otherwise, we iterate over all upper triangles (x, y, z) and test whether $m(x, z) + o = m(y, z)$ holds. (Note that (y, z, x) is a lower triangle of (y, z).) If so, we add the change $((y, z), m(x, z)+n)$ to the queue. Intermediate triangles are handled analogously.

5 Phase 3: At Query Time

In this section we describe how to compute a shortest up-down path in $G^{\wedge}_{\pi}$, given a source and target vertex s and t and a customized metric.

Basic and Stalling. The basic query alternates two instances of Dijkstra's algorithm on $G^{\wedge}_{\pi}$ from s and t, maintaining a tentative distance of the shortest path discovered so far (initially ∞). If G is undirected then both searches use the same metric. Otherwise, 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 [12], the searches operate on the same upward search graph $G^{\wedge}_{\pi}$. Each search is stopped once its radius is larger than the tentative distance. Additionally, we evaluate a basic version of the stall-on-demand optimization presented in [12].

Elimination Tree. Inspired by [3], we us the precomputed elimination tree to efficiently enumerate all vertices in SS(s) and SS(t) by increasing rank at query time without using a priority queue. We store two tentative distance arrays $d_f(v)$ and $d_b(v)$. Initially these are all set to ∞. First, we compute the lowest common ancestor (LCA) x of s and t in the elimination tree: We simultaneously enumerate all ancestors of s and t by increasing rank until a common ancestor is found. Second, we iterate over all vertices y on the branch from s to x, relaxing all forward arcs of such y. Third, we do the same for all vertices y from t to x in the backward search. Fourth, we iterate over all vertices y' from x to the root (the top-level vertex), simultaneously relaxing the respective outgoing arcs of each y'

for the forward and backward searches. We further determine the vertex z that minimizes $d_f(z) + d_b(z)$, deriving the path distance and preparing unpacking. We finish by iterating over all vertices from s and t to the root, resetting all distances d_f and d_b to ∞ (which proved cheaper than timestamping).

Path Unpacking. The original CH of [12] unpacks an up-down path by storing for every arc (x, y) the vertex z of the lower triangle (x, y, z) that caused the weight at $m(x, y)$. This information depends on the metric and we want to avoid storing additional metric-dependent information. We therefore resort to a different strategy. Denote by $p_1 \dots p_k$ the up-down path found by the query. As long as a lower triangle (p_i, p_{i+1}, x) 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}.

6 Comparison with CRP

Both CRP and CCH are multi-level overlay techniques in spirit of [20], but CRP typically utilizes less levels. Yet, regarding customization, both approaches clearly converge, since CRP uses further guidance levels and contraction as a subroutine [9]. Note, however, that CRP does not contract the top-level separator. Queries differ more, with CRP running plain Dijkstra on the lowest cells before employing preprocessed data. Note that by exploiting partition information at query time, CRP offers unidirectional queries.

Our triangle preprocessing has similarities with micro and macro code [9]. While their approach does not allow for a—in our context crucial—random access, we can atleast compare space consumption: In that regard, micro code is clearly the most expensive. Macro code, on the other hand, is compactest as the dominating substructure only stores one ID per triangle (instead of our two IDs). However, we enumerate undirected triangles and thus, depending on the instance, may have only half as many triangles.

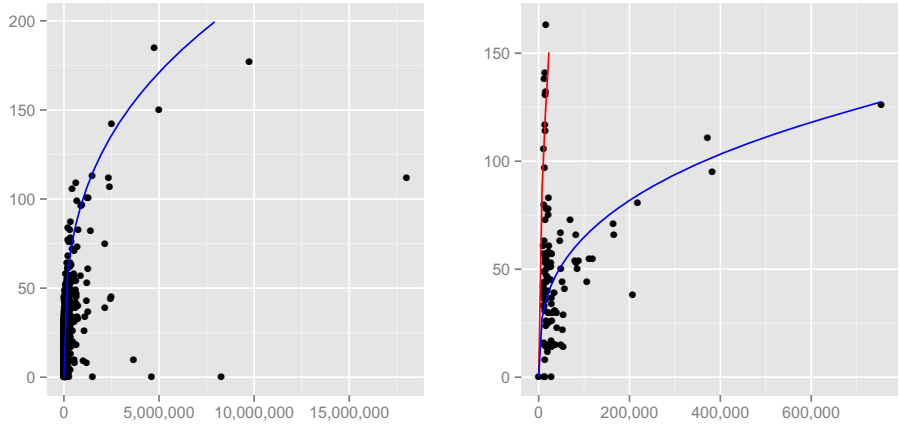
One major advantage of CRP over other techniques is that it works well with turn costs. Since our benchmark instances lack realistic turn cost data (while synthetic data tends to be very simplistic), we deemed it unproper to experimentally evaluate CCH performance on turn costs. However, using a theoretical 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. Note, that the game scenario does not need turn costs.

7 Experiments

The experiments were run on a dual 8-core Intel Xeon E5-2670 processor clocked at 2.6 GHz, with 64 GiB of DDR3-1600 RAM, 20 MiB of L3 and 256 KiB of L2 cache (internally called compute11). We use g++ 4.7.1 with -O3.

Table 1. The number of vertices and of directed arcs of the benchmark graphs. We further present the number of edges in the induced undirected graph.

	#Vertices	#Arcs	#Edges	symmetric?
TheFrozenSea	754 195	5 815 688	2 907 844	yes
Europe	18 010 173	42 188 664	22 211 721	no



(a) Eur (b) TFS

Fig. 1. Vertices in the separator (vertical) vs vertices in the subgraph being partitioned (horizontal). The red function is a cubic root ($y = \sqrt[3]{x}$ for Eur and $y = 1.4 \cdot \sqrt[3]{x}$ for TFS) and the blue function is a square root ($y = \sqrt{x}$ for TFS).

We consider two large instances of practical relevance (c.f. Table 1): First, the road network of Europe (Eur in short) as made available by PTV AG for the DIMACS challenge [10]. It is the defacto standard for benchmarking road route planning research. Second, the game map TheFrozenSea (TFS in short), one of the largest of Star Craft maps available at [22]. The map is composed of square tiles, each having at most eight neighbors. Some tiles are non-walkable. Walkable tiles form pockets of free space connected through *choke points* of very limited 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 $\sqrt{2}$, others by 1. The graph is symmetric and contains large grid subgraphs.

ND-Order. To compute a ND-order, we use KaHIP in its "strong" configuration, with imbalance set to 20%. We recursively bisect the graph using edge cuts from which vertex separators are derived. We repeat each cut ten times with different random seeds and pick the smallest one. The resulting separator sizes are illustrated in Figure 1 (in the extended version we also consider Metis [16]). The top-level separators of Europe are curiously small due to the special topology

of the continent (Great Britain, Spain, and France are easily separated). Once all such features are exploited, the cut sizes seem to follow a $\Theta(\sqrt[3]{n})$-law, making the approximation result from Theorem 1 applicable. On TFS, the top-level separators that cut through choke points also follow a $\Theta(\sqrt[3]{n})$-law. The residual subgraphs are grids with $\Theta(\sqrt{n})$ cuts. This explains the two peaks in the plot.

CH Construction. Switching from a dynamic adjacency array to our Contraction Graph approach we decrease construction time on TFS from 490.6s to 3.8s and on Europe from 305.8s to 15.5s. (Be aware that this approach cannot immediately be extended to directed or weighted graphs, that is, without customization.)

CH Size. As a baseline, we were interested in computing a greedy-ordered CH but without witness search. However, this turned out to be infeasible even using the Contraction Graph. We had to abort the calculation after discovering at least one vertex with an upward degree of 1.4×10^6. It is safe to assume that using this order it is impossible to achieve a speedup over Dijkstra's algorithm on the original graph. In Table 2 we report the total number of shortcuts and the search space sizes averaged over 1000 random vertices. The downward-DAG has in essence the same structure as the upward-DAG. Recall that for metric independent CHs both DAGs are the same and therefore it only has to be stored once in memory. For a CH with witness search two separate upward and downward DAGs must be stored. For a ND-order we can perform a perfect witness search (i.e. we remove every arc not part of a shortest path). For a greedy order we only know of the heuristic approach of [12], but on a smaller road instance we were able to show that the difference between perfect and heuristic is very small. The details are in the extended version. Interestingly, the number of vertices in the ND search spaces are smaller than for the greedy order. However, the number of arcs in the ND search spaces is significantly higher, dominating query running time. The witness search decreases the number of arcs for the ND order by a factor of 2 to 4. On game maps the greedy order decreases the number of arcs by another factor of 2 and on Europe even by another factor of 10. Additionally, we counted the number of triangles which is a performance relevant figure for customization: TFS has 864M and Europe has 578M triangles. Precomputing all triangles as suggested requires 6.6GB space on TFS and 4.6GB on Europe. (Using Metis reduces space on TFS to 4.6GB.)

Table 2. Shortcut count and search space sizes

	Order	Witness search	#Arcs upward	Avg. up SS size vertices	arcs
TFS	Greedy	heuristic	6 399 080	1 281	13 330
	ND	none	25 099 646	674	89 567
		perfect	10 161 889	645	24 782
Europe	Greedy	heuristic	33 911 692	709	4 808
	ND	none	73 920 453	652	117 406
		perfect	55 657 315	616	44 677

Treewidth. As detailed in [4], the treewidth of a graph is a measure deeply coupled with chordal super graphs. The authors show in their Theorem 6 that the maximum upward degree over all vertices in a metric-independent CH of a graph G is an upper bound to the treewidth of G. Using this, we are able to upper bound the treewidth of TFS by 287 and of Europe by 479.

Customization. In Table 3 we report the times needed to customize a metric. Using all optimizations presented we customize Europe in below one second. When amortized[1], we even achieve 0.415ms which is only slightly above the (non-amortized) 0.347ms reported in [9]. (Note that their experiments were run on a different machine with a faster clock but 2×6 instead of 2×8 cores. Also, their implementation is turn-aware, touching more memory, making an exact comparison difficult.) However, their overlays are more space efficient than our metric-independent CHs. The running time of 0.415ms is fast but comes at a high price as many cores are needed. We thus evaluate the time needed for partial updates on a single core. We averaged over 10 000 runs in which we set a random arc in the CH to a random value. The median, average and maximum running times significantly differ. This is because there are a few arcs on highways or choke points that trigger a lot of subsequent changes whereas for most arcs a weight change has nearly no effect.

Table 3. The running times needed to compute or update a metric. The ×4 indicates that 4 interleaved metric pairs are customized in one go.

SSE	precompute triangles?	thread count	TFS time [ms]	Eur time [ms]
no×1	none	1	10.08	10.88
yes×1	none	1	9.34	9.55
yes×1	all	1	3.75	3.22
yes×1	all	16	0.61	0.74
yes×2	all	16	0.76	1.05
yes×4	all	16	1.50	1.66

	Queue removals			time [ms]		
	med.	avg.	max.	med.	avg.	max.
TFS	8	382.4	12035	0.017	2.0	99.2
Eur	1	38.8	10666	0.003	0.2	87.2

Distance Query. In Table 4 we report the performance of our query algorithms, comparing them to the original CH of [12]. We manage to come very close to their query times, but as we also support a fast customization even achieving the same order of magnitude is a major result. We ran 10^6 shortest path distance queries with the source and target vertices picked uniformly at random. The presented times are averaged single core running times without any SSE. The evaluated metric-dependent (greedy order) CH is comparable to [12]. (Because of a slower machine their reported running times are all slightly higher. The only exception is the Europe graph with the distance metric. Here, our measured running time of only 0.540ms is disproportionally faster. We assume that our greedy order is better since we do not use lazy updates, spending more preprocessing time.) As in [12] the stall-on-demand heuristic improves running times by a factor 2 to 5.

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

Table 4. Query running times and explored forward search space sizes

	Metric	Order	Query Type	Settled Vertices	Relaxed Arcs	Time [ms]
TFS	Map-Dist.	Greedy	Basic	1 199	12 692	0.539
		Greedy	Stalling	319	3 459	0.286
		ND	Basic	603	82 824	0.644
		ND	Stalling	560	74 244	0.774
		ND	Tree	674	89 567	0.316
Europe	Travel-Time	Greedy	Basic	546	3 623	0.283
		Greedy	Stalling	113	668	0.107
		ND	Basic	581	107 297	0.810
		ND	Stalling	418	75 694	0.857
		ND	Tree	652	117 406	0.413
	Distance	Greedy	Basic	3 653	104 548	2.662
		Greedy	Stalling	286	7 124	0.540
		ND	Basic	581	107 080	0.867
		ND	Stalling	465	84 718	0.992
		ND	Tree	652	117 406	0.414

Recall that in addition to the obvious modifications we also reordered the vertices in memory according to the nested dissection order for the metric-independent CH. Preliminary experiments have shown that this has a measurable effect on cache performance and results in 2 to 3 times faster query times. We do not reorder the vertices for the metric-dependent CHs to remain comparable to [12] and because it is a lot less clear what a good cache-friendly order is. For the basic CCH query (ND order) on Europe with travel time metric, the number of settled vertices only increases by a small factor compared to metric-dependent case. On TFS or Europe with distance metric this difference is even smaller. However, the number of relaxed arcs is significantly larger. As this increases the costs of the stalling test, stall-on-demand does not pay off anymore. We observe that the basic query algorithm (despite the use of a stopping criterion) still visits large portions of the possible search space. In contrast, the elimination tree based query visits slightly more vertices. However, it does not use a priority queue and needs therefore less time per vertex. Furthermore, for the elimination tree based algorithm the code paths do not depend on the metric. Hence, query times are completely independent of the metric as can be seen by comparing the tree ND query times for travel time metric to distance metric. In contrast, for the basic ND query algorithms the metric still has a slight influence on the performance. A possible downside of the elimination tree query currently is that local queries are not faster than global queries. However, one might be able to exploit LCA information as a locality filter, switching to the basic query if sufficiently local.

A stalling query on the metric-dependent CH with travel time is on Europe about a factor of 5 faster than the elimination tree query. However for the distance metric the order is inversed and our CHs are even faster by a factor of about 20%. We see three factors contributing here: 1) The distance metric is a comparatively hard metric, 2) the in-memory ID reordering, 3) the lack of a priority queue.

CRP without turn costs on Europe needs 0.72ms with travel time and 0.79ms with distance [6]. However, they parallelize by running the forward and the backward searches in different threads. This parallelization can be added to our query but even without it our query outperforms their query times. Table 5 summarizes the situation.

Table 5. Comparison with CRP/MLD. CCH hits a Pareto-point.

[ms]	Cust.	Query
CRP	0.347	0.72
CCH	0.415	0.41

Finally we measured the time needed to fully unpack the paths. For travel time on Europe the time is 0.25ms (and 0.27ms for TFS), which is below a query time. For Europe with distance the time is 0.52ms, which is slightly higher than query time, as paths tend to contain more arcs because the input graph is modeled a lot less fine grained on the highways used by the travel metric. In [12] they report full path extraction costs of 0.323ms for travel time on Europe but with precomputed middle nodes. This is slightly faster when accounting for the processor differences. However, we are positive that we could achieve similar performance exploiting preprocessed triangle information for path unpacking.

8 Conclusions

We have extended CHs 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.

Future Work. While a graph topology with small cuts is one of the main driving force behind the running time performance of CHs it is clear from Table 2 that better metric-dependent orders can be constructed by exploiting additional travel time specific properties. We would like to further investigate this gap. Better ND-orders directly result in better CHs and thus further partitioning research is useful. Further investigation into algorithms explicitly exploiting treewidth [5,18] might be promising. Also, determining the precise treewidth could prove useful. Revisiting all of the existing CH extensions to see which can profit from an ND-order is worthwhile. An interesting candidate are Time-Dependent CHs [2] where computing a good metric-dependent order has proven relatively expensive.

Acknowledgements We would like to thank Ignaz Rutter and Tim Zeitz for very inspiring conversations on the topic.

References

1. Bast, H., Delling, D., Goldberg, A.V., Müller–Hannemann, M., Pajor, T., Sanders, P., Wagner, D., Werneck, R.F.: Route planning in transportation networks. In: Technical Report MSR-TR-2014-4. Microsoft Research, Mountain View (2014)
2. Batz, G.V., Geisberger, R., Sanders, P., Vetter, C.: Minimum time-dependent travel times with contraction hierarchies. ACM J. Exp. Algorithmics 18, 1–43 (2013)

3. Bauer, R., Columbus, T., Rutter, I., Wagner, D.: Search-space size in contraction hierarchies. In: Fomin, F.V., Freivalds, R., Kwiatkowska, M., Peleg, D. (eds.) ICALP 2013, Part I. LNCS, vol. 7965, pp. 93–104. Springer, Heidelberg (2013)
4. Bodlaender, H.L., Koster, A.M.C.A.: Treewidth computations I. Upper bounds. Inform. and Comput. 208, 259–275 (2010)
5. Chaudhuri, S., Zaroliagis, C.: Shortest paths in digraphs of small treewidth. Part I: Sequential algorithms. Algorithmica 27, 212–226 (2000)
6. Delling, D., Goldberg, A.V., Pajor, T., Werneck, R.F.: Customizable route planning. In: Pardalos, P.M., Rebennack, S. (eds.) SEA 2011. LNCS, vol. 6630, pp. 376–387. Springer, Heidelberg (2011)
7. Delling, D., Goldberg, A.V., Razenshteyn, I., Werneck, R.F.: Graph partitioning with natural cuts. In: 2011 IEEE International Parallel & Distributed Processing Symposium (IPDPS 2011), pp. 1135–1146. IEEE Computer Society, Los Alamitos (2011)
8. Delling, D., Goldberg, A.V., Razenshteyn, I., Werneck, R.F.: Exact combinatorial branch-and-bound for graph bisection. In: Bader, D.A., Mutzel, P. (eds.) ALENEX 2012, pp. 30–44. SIAM, Philadelphia (2012)
9. Delling, D., Werneck, R.F.: Faster customization of road networks. In: Bonifaci, V., Demetrescu, C., Marchetti-Spaccamela, A. (eds.) SEA 2013. LNCS, vol. 7933, pp. 30–42. Springer, Heidelberg (2013)
10. Demetrescu, C., Goldberg, A.V., Johnson, D.S. (eds.): The Shortest Path Problem: Ninth DIMACS Implementation Challenge. American Mathematical Society, Rhode Island (2009)
11. Fulkerson, D.R., Gross, O.A.: Incidence matrices and interval graphs. Pacific J. Math. 15, 835–855 (1965)
12. Geisberger, R., Sanders, P., Schultes, D., Vetter, C.: Exact routing in large road networks using contraction hierarchies. Transportation Science 46, 388–404 (2012)
13. George, A.: Nested dissection of a regular finite element mesh. SIAM J. Numer. Anal. 10, 345–363 (1973)
14. George, A., Liu, J.W.: A quotient graph model for symmetric factorization. In: Duff, I.S., Stewart, G.W. (eds.) Sparse Matrix Proceedings, pp. 154–175. SIAM, Philadelphia (1978)
15. Holzer, M., Schulz, F., Wagner, D.: Engineering multilevel overlay graphs for shortest-path queries. ACM J. Exp. Algorithmics 13, 1–26 (2008)
16. Karypis, G., Kumar, V.: A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM J. Sci. Comput. 20, 359–392 (1999)
17. Lipton, R.J., Rose, D.J., Tarjan, R.: Generalized nested dissection. SIAM J. Numer. Anal. 16, 346–358 (1979)
18. Planken, L., de Weerdt, M., van Krogt, R.: Computing all-pairs shortest paths by leveraging low treewidth. J. Artificial Intelligence Res. 43, 353–388 (2012)
19. Sanders, P., Schulz, C.: Think locally, act globally: Highly balanced graph partitioning. In: Bonifaci, V., Demetrescu, C., Marchetti-Spaccamela, A. (eds.) SEA 2013. LNCS, vol. 7933, pp. 164–175. Springer, Heidelberg (2013)
20. Schulz, F., Wagner, D., Weihe, K.: Dijkstra's algorithm on-line: An empirical case study from public railroad transport. ACM J. Exp. Algorithmics 5, 1–23 (2000)
21. Storandt, S.: Contraction hierarchies on grid graphs. In: Timm, I.J., Thimm, M. (eds.) KI 2013. LNCS, vol. 8077, pp. 236–247. Springer, Heidelberg (2013)
22. Sturtevant, N.: Benchmarks for grid-based pathfinding. Transactions on Computational Intelligence and AI in Games (2012)
23. Zeitz, T.: Weak contraction hierarchies work! Bachelor thesis. KIT, Karlsruhe (2013)