Exact Routing in Large Road Networks using Contraction Hierarchies

ROBERT GEISBERGER, PETER SANDERS, DOMINIK SCHULTES
and CHRISTIAN VETTER
Karlsruhe Institute of Technology, 76128 Karlsruhe, Germany

Contraction hierarchies are a simple approach for routing in road networks. Our algorithm calculates exact shortest paths and handles road networks of whole continents. During a preprocessing step, we exploit the inherent hierarchical structure of road networks by adding shortcut edges. A subsequent modified bidirectional Dijkstra algorithm can then find a shortest path visiting only a few hundred nodes. This small search space makes it suitable to implement it on a mobile device. We present a mobile implementation that also handles changes in the road network, like traffic jams, and that allows instantaneous routing without noticeable delay for the user. Also, an algorithm to calculate large distance tables is currently the fastest if based on contraction hierarchies.

Categories and Subject Descriptors: E.1 [**Data Structures**]: Graphs and networks; G.2.2 [**Discrete Mathematics**]: Graph Theory—*Graph algorithms*; I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search—*Graph and tree search strategies*

General Terms: Algorithms, Experimentation, Performance, Theory

Additional Key Words and Phrases: Algorithm engineering, Route planning, Shortest paths

1. INTRODUCTION

Finding optimal routes in road networks is an important problem used in diverse applications, such as car navigation systems, Internet route planners, traffic simulation, or logistics optimisation. Formally, we are given a directed graph $G = (V, E)$ together with an edge weight function $c : E \to \mathbb{R}^+$. Edges represent roads and nodes represent junctions. In practice, c is usually the travel time or some more general cost function highly correlated with travel time. Since the network does not change much over time, it makes sense to *preprocess* this graph in order to speed up subsequent (shortest path) *queries* asking for a minimum weight path from a given source node to a given target node. However, in order to scale to networks with millions of nodes, preprocessing has to be fast and space efficient. In particular, it is not feasible to precompute a complete distance table.

Our motivation for *contraction hierarchies (CH)* was to create an efficient routing algorithm whose simplicity makes it adaptable to a variety of situations. Interestingly, we were able to engineer our simple approach to get one of the most efficient algorithms known today. Our algorithm is currently the fastest for car navigation systems and to calculate large distance tables. It needs less space for preprocessed information than any other scheme and no other approach achieves faster query times with comparable preprocessing time. Subsequent works achieve the fastest query times known and generalise the model. We exploit the observation that road networks have an inherent hierarchy with a few important and many unimportant roads and junctions, i. e., roads and junctions that are only used for local traffic near the source and target of a route. However, it is not necessary to specify road categories in advance. Our algorithm performs the road classification automatically

by evaluating the assigned cost of each road. To show that our algorithm is feasible in practice, we released source-code as open-source [Geisberger et al. 2008a] and we provide a mobile implementation [Vetter 2010] that calculates exact shortest paths within split seconds. We also added features of current commercial systems that can cope with traffic jams or blocked routes.

1.1 Basic Approach

The idea of our algorithm is to remove unimportant nodes from a directed, weighted road network in a way that preserves shortest path distances. This concept is called *node contraction*: deleting a node u and adding shortcut edges (*shortcuts*) [Sanders and Schultes 2005; Goldberg et al. 2006] to preserve shortest path distances between the remaining nodes. The shortcuts bypass node u and represent whole paths. During the preprocessing, we contract one node at a time until the graph is empty. All original edges together with the shortcuts form the result of the preprocessing, a *contraction hierarchy*. Subsequently, nodes removed later will be called *higher up* in the hierarchy. A crucial figure is the number of shortcuts. If it is too large, our algorithm will not be useful because preprocessing time, space consumption and query time will suffer. But due to the inherent hierarchy of road networks, we can keep this figure small by a careful heuristic choice of the order in which the nodes are contracted. Roughly, after contracting a node, the remaining graph should be as sparse as possible. Hence, the *edge difference* – the number of added shortcuts minus number of incident edges – of a contracted node should be small. Further heuristics enforce a uniform contraction everywhere in the graph and try to limit the effort for contraction or subsequent queries.

The concept of node contraction allows an efficient and simple query algorithm. We find a shortest path from source s to target t using a variant of the bidirectional version of Dijkstra's algorithm: Forward search from s is *constrained to upward edges* and backward search from t is *constrained to downward edges*. Because of the shortcuts, both searches will meet on a node u that is highest in the CH on a shortest path between s and t.

We developed two implementations of CHs: one for personal computers and one for mobile devices. The mobile implementation required engineering of an external-memory graph representation to overcome the I/O bandwidth and main memory limitations of those small devices. We divided the graph into several blocks, each containing a subset of the nodes and the corresponding edges. We put particular effort into exploiting the fact that the edges in one block only lead to nodes in a small subset of all blocks; many edges even lead to nodes in the same block.

1.2 Related work

Computing shortest paths in road networks in a well-studied problem, c. f. [Ahuja et al. 1993]. Recently a plethora of faster algorithms (*speedup techniques*) has been developed that are several orders of magnitude faster and can handle much larger graphs than the classic algorithm by [Dijkstra 1959]. We can only give an abridged overview with emphasis on directly related techniques beginning with the closest kin. For a recent overview we refer to [Delling et al. 2009]. Previous heuristic approaches, e. g. [Fu et al. 2006], or speedup techniques based on A*, e. g. [Klunder and Post 2006], are orders of magnitude slower than the best exact methods known

now.

CHs, first introduced by [Geisberger et al. 2008b], are an extreme case of the hierarchies in highway-node routing (HNR) by [Schultes and Sanders 2007] – every node defines its own level of the hierarchy. CHs are nevertheless a new approach in the sense that the node ordering and hierarchy construction algorithms used by HNR are only efficient for a small number of geometrically shrinking levels. We also give a faster and more space efficient query algorithm and improve their dynamization techniques.

The node ordering computed by HNR uses levels acquired by *highway hierarchies* (HHs) by [Sanders and Schultes 2005; 2006]. Our original motivation for CHs was to simplify HNR by obviating the need for another (more complicated) speedup technique (HHs) for node ordering. HHs are constructed by alternating between two subroutines: *Edge removal* is a sophisticated and relatively costly routine that only keeps edges required 'in the middle' of 'long-distance' paths. *Node removal* contracts nodes. In the original paper for undirected HHs of [Sanders and Schultes 2005], node removal only contracted nodes of degree one and two. For directed graphs we needed a more general node ordering criterion for the contraction as described by [Sanders and Schultes 2006]. It turned out that the edge difference is a good way to estimate the cost of contracting a node v. [Goldberg et al. 2007; Bauer and Delling 2009] further refine this method using a priority queue and avoiding parallel edges. All previous approaches to contraction had in common that the average degree of the nodes in the remaining graph would eventually explode. So it looked like an additional technique such as edge removal or reaches would be a necessary ingredient of any high-performance hierarchical routing method. Perhaps the most important result of CHs is that using *only* (a more sophisticated) node contraction, we achieve very good performance.

The fastest speedup technique so far, *transit-node routing* (TNR) by [Bast et al. 2007], offers almost constant query times by reducing most queries to a few table lookups. However, it needs considerably higher preprocessing time and space, is less amenable to dynamisation, and, most importantly it relies on another hierarchical speedup technique for its preprocessing. We will show that using CHs for this purpose leads to improved performance.

A different hierarchical approach is proposed by [Thorup 2004] to answer queries accurate within a factor $(1+\varepsilon)$ in time $O(\log\log(nC)+1/\varepsilon)$ on planar graphs with integer edge weights in a range from 0 to C.

Finally, there is an entirely different family of speedup techniques based on goal-directed routing. In particular, *ALT* (A*, Landmarks, Triangle inequality) [Goldberg and Harrelson 2005] yields strong lower bounds that can direct the search towards the target . It has fast preprocessing but considerable space requirements. *Arc flags (AF)* [Lauther 2004] indicate for each edge into which regions it leads. They give a stronger sense of goal direction than ALT and need less space yet very high preprocessing times. Combination of CHs with goal-directed routing have been systematically studied by [Bauer et al. 2010b]. TNR based on CHs combined with arc flags yields the currently fastest queries. CHs combined with arc flags give another favorable tradeoff between preprocessing time/space and query time. Their experiments suggest that CHs also work for other sparse networks with high locality

such as some transportation networks, or sparse unit-disk graphs. For more dense networks, CHs can be used in an initial contraction phase whereas a goal-directed technique is applied to the resulting core network.

For mobile algorithms, only few academic implementations exist. [Goldberg and Werneck 2005] successfully implemented the ALT algorithm on a Pocket PC. Our implementation, a static version was presented earlier in [Sanders et al. 2008], is more than one magnitude faster and drastically more space efficient. Also the RE algorithm of [Gutman 2004; Goldberg et al. 2007] has been implemented by [Goldberg 2006] on a mobile device, yielding query times of 'a few seconds including path computation and search animation' and requiring '2–3 GB for USA/Europe'. Commercial systems, to the best of our knowledge, do *not* compute exact routes and require several seconds to calculate a route. However a direct quantitative comparison is not possible since they are slowed down due to various reasons like showing a progress bar.

Several aspects of routing in road networks are more or less orthogonal. For example turn restrictions and turn penalties can be modeled using *edge based routing* where nodes represent the starting point of a road segment and edges represent the cost of going from one starting point to a subsequent starting point. [Volker 2008] investigated its effects on speedup techniques.

1.3 Outline

In Section 3 we describe CHs in detail and explain the preprocessing (Sects. 3.1, 3.3) and the query algorithm (Sect. 3.2). Our adaptions for mobile devices are in Section 4. The refined dynamisation technique is described in Section 5 and a variant suitable for the mobile scenario in Section 6.

2. PRELIMINARIES

Dijkstra's algorithm can be used to solve the single-source shortest path problem, i. e., to compute the shortest paths from a single source node s to all other nodes in a given graph. Starting with the source node s as root, Dijkstra's algorithm grows a shortest path tree that contains shortest paths from s to all other nodes. During this process, each node of the graph is either *unreached*, *reached*, or *settled*. A node that already belongs to the tree is *settled*. If a node u is settled, a shortest path P^* from s to u has been found and the distance $d_s(u) = c(P^*)$ is known, where $c(P)$ denotes the sum of the costs of the edges on a path P. A node that is adjacent to a settled node is *reached*. Note that a settled node is also reached. If a node u is reached, a path P from s to u, which might not be the shortest one, has been found and a *tentative distance* $\delta_s(u) = c(P)$ (δ = greek delta) is known. Nodes that are not reached are *unreached*.

3. CONTRACTION HIERARCHIES

As introduced in Section 1.1, we construct a CH by ordering the nodes and then contracting the nodes in this order. For convenience, we assume that, after node ordering, the nodes are numbered $1..n$, $n := |V|$ in order of ascending importance. A node u is contracted by removing it from the network in such a way that shortest paths in the *remaining graph* are preserved. When we do not add a shortcut, then

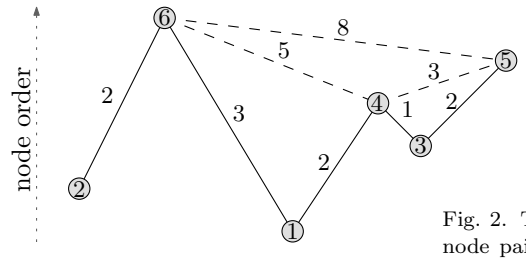


Fig. 1. A completed CH, dashed edges are added shortcuts.

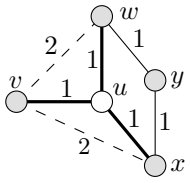


Fig. 2. There is no witness path between the node pairs v, w and v, x so we need to add shortcuts (dashed) for the contraction of node u. The witness path $\langle w, y, x\rangle$ allows us to omit a shortcut between the node pair w, x.

there must exist a shortest path $\langle v, \ldots, w\rangle \neq \langle v, u, w\rangle$, we call such a path *witness path*. The concept of witness paths is particularly important for the dynamization described in Sections 5 and 6. Figure 1 shows a completed CH.

3.1 Node Contraction

The most important part of the node contraction is to find witness paths. A simple way to decide on the uniqueness of $\langle v, u, w\rangle$ is to perform for each node $v \in S$ a forward shortest-path search only using nodes $> u$ until all nodes in $T \setminus \{v\}$ are settled. Such a search to compute shortest path distances between the neighbours is called a *local search*. Let $\delta_v(w)$ be the shortest path distance found by this local search. We add a shortcut if and only if $\delta_v(w) > c(v, u) + c(u, w)$, i. e. if the shortest v-w-path excluding u will be longer, see Figure 2. We can additionally stop the search from a node x when it has reached distance $c(v, u) + \max\{c(u, w) \mid w \in T \setminus \{v\}\}$.

Limit Local Searches. In order to achieve fast preprocessing we rely on the assumption that local searches are fast since they visit only a tiny fraction of the network. However, this assumption fails when long-distance edges like ferry connections are involved. We therefore additionally truncate local searches that become to large. Note that this preserves shortest path distances since we only introduce some superfluous shortcuts. Since additional shortcuts slow down all further processing, the local search limit needs to be carefully selected for maximum performance. We propose two approaches to limit the local searches: a *settled nodes limit* and a *hop limit* that limits the number of edges of witness paths. Limiting the number of settled nodes is simple, but, in our experience, leads to dense remaining graphs and does not speedup the contraction a lot. However, if we only use it to estimate the edge difference and perform the real contraction without a limit, it speeds up the node ordering and yields CHs with fast query times. Hop limits, introduced by [Schultes 2008], provide a better contraction speedup and also adapt to denser remaining graphs. To achieve further speedup, we propose *staged* hop limits: We start contracting nodes with a small hop limit, e. g. one. At some point, we switch to larger hop limits because otherwise the remaining graph will get too dense. We use the average node degree, a measure for density, to trigger these switches.

Fast Local 1-Hop Search. To find witness paths consisting just of one edge, it is sufficient to scan through all outgoing edges of the source node $v \in S$. The 1-hop

search makes sense if lots of edges of the graph are shortest paths, like in a road network. In this case, it allows to contract a significant amount of nodes without too many additional shortcuts being added.

Fast Local 2-Hop Search. We implement a simple variant of the many-to-many shortest paths algorithm of [Knopp et al. 2007]. with each node $x > u$, we associate a bucket $b(x) := \{(w, c(x,w)) \mid w \in T, (x,w) \in E\}$. Computing the non-empty $b(x)$ is done by by scanning the incoming edges of all $w \in T$. For $v \in S$ we then compute $\delta_v(w) := \min\{c(v,x) + c(x,w) \mid (v,x) \in E, (w, c(x,w)) \in b(x)\} \cup \{c(v,w)\}$ by scanning the outgoing edges (v,x) of v and the buckets $b(x)$.

1-Hop Backward Search. To speed up a local search from $v \in S$ with hop limit $a \geq 3$, we first perform a Dijkstra search with $(a-1)$-hop limit resulting in distances $\delta_v(\cdot)$. We improve these to $\delta_v(w) := \min\{\delta_v(w)\} \cup \{\delta_v(x) + c(x,w) \mid (x,w) \in E\}$ by scanning the incoming edges of $w \in T$. The distance limit for the forward search changes, we now stop the search if the last settled node exceeds the distance $c(v,u) + \max\{c(u,w) - \min\{c(x,w) \mid (x,w) \in E\} \mid (u,w) \in E, w \neq v\}$.

On-the-fly Edge Reduction. If the local search is performed by a local Dijkstra search, it computed tentative distances $\delta_v(x)$ to all neighbours $x > u$ of v. We use them to remove superfluous edges $(v,x) \in E$ with $\delta_v(x) < c(v,x)$. This edge reduction is cache efficient and will therefore cause almost no direct overhead but brings potentially faster preprocessing and query times.

3.2 Query

In this section we will introduce our query algorithm and prove its correctness. The query does not relax edges leading to nodes lower than the current node. This property is reflected in the *upward graph* $G_\uparrow := (V, E_\uparrow)$ with $E_\uparrow := \{(u,v) \in E \mid u < v\}$) and, the *downward graph* $G_\downarrow := (V, E_\downarrow)$ with $E_\downarrow := \{(u,v) \in E \mid u > v\}$). We combine them in the *search graph* $G^* = (V, E^*)$ with $\overline{E_\downarrow} := \{(v,u) \mid (u,v) \in E_\downarrow\}$ and $E^* := E_\uparrow \cup \overline{E_\downarrow}$. With each $e \in E^*$, we store a forward and a backward flag such that $\uparrow(e) = \text{true}$ iff $e \in E_\uparrow$ and $\downarrow(e) = \text{true}$ iff $e \in \overline{E_\downarrow}$. Algorithm 1 describes our bidirectional Dijkstra-like query on G^*, essentially a forward search in $G_\uparrow$ and a backward search in $G_\downarrow$.

Algorithm 1: Query(s,t)

```
1  d↑:= ⟨∞,...,∞⟩; d↓:= ⟨∞,...,∞⟩; d:= ∞;      // tentative distances
2  d↑[s]:= 0; d↓[t]:= 0;        // start bidirectional search from s and t
3  Q↑ = {(0,s)}; Q↓ = {(0,t)}; r:= ↑;                  // priority queues
4  while (Q↑ ≠ ∅ or Q↓ ≠ ∅) and (d > min{min Q↑, min Q↓}) do
5      if Q_¬r ≠ ∅ then r:= ¬r;      // interleave direction, ¬↑=↓ and
         ¬↓=↑
6      (·,u):= Q_r.deleteMin();                              // settle u
7      d:= min{d, d↑[u] + d↓[u]};              // u is potential candidate
8      foreach e = (u,v) ∈ E* do                          // relax edges
9          if r(e) and (d_r[u] + c(e) < d_r[v]) then     // shorter path found
10             d_r[v]:= d_r[u] + c(e);              // update tentative distance
11             Q_r.update(d_r[v],v);                   // update priority queue
12 return d;
```

THEOREM 3.1. *Algorithm 1, applied to a CH, returns the correct shortest path distance.*

PROOF. It follows from the definition of a shortcut, that the shortest path distance between s and t in the CH is the same as in the original graph. Every shortest s-t-path in the original graph still exists in the CH but there may be additional shortest s-t-paths. However since we use a modified Dijkstra algorithm that does not relax all incident edges of a settled node, our query algorithm does only find particular ones. It only finds shortest paths of the form (PF) $\langle s = u_0, u_1, \ldots, u_p, \ldots, u_q = t\rangle$ with $p, q \in \mathbb{N}$, $u_i < u_{i+1}$ for $i \in \mathbb{N}, i < p$ and $u_j > u_{j+1}$ for $j \in \mathbb{N}, p \leq j < q$. We will prove that if there exists a shortest s-t-path then there also exists a shortest s-t-path of the form (PF).

Given a shortest s-t-path $P = \langle s = u_0, u_1, \ldots, u_p, \ldots, u_q = t\rangle$ with $p, q \in \mathbb{N}$, $u_p = \max P$. Let $M_P := \{u_k \mid u_{k-1} > u_k < u_{k+1}\}$ denote the set of local minima excluding nodes s, t. $M_P \neq \emptyset$ iff P is not of the form (PF), like in Figure 3(a). Let $u_k := \min M_P$ and consider the two edges $(u_{k-1}, u_k), (u_k, u_{k+1}) \in E$. Both edges already exist at the beginning of the contraction of node u_k. So there is either a witness path $Q = \langle u_{k-1}, \ldots, u_{k+1}\rangle$ consisting of nodes higher than u_k with $c(Q) \leq c(u_{k-1}, u_k) + c(u_k, u_{k+1})$ (even = since P is a shortest path) or a shortcut (u_{k-1}, u_{k+1}) of the same weight is added. So the subpath $\langle u_{k-1}, u_k, u_{k+1}\rangle$ can either be replaced by Q or by the shortcut (u_{k-1}, u_{k+1}). The resulting path P' is still a shortest s-t-path and $\min M_{P'} > u_k$ or $M_{P'} = \emptyset$. Since $n < \infty$, there must exist a shortest s-t-path P'' with $M_{P''} = \emptyset$ of the desired form (PF). □

Outputting Complete Path Descriptions. Algorithm 1 can be extended to return the whole shortest path P. However, P can contain shortcuts. To obtain a shortest path P' in the original graph, we iteratively replace a shortcut (v, w) created from edges (v, u), (u, w) during contraction of u, by these two edges (v, u), (u, w). It runs in $O(|P'|)$ where $|P'|$ is the number of edges in P' when we store pointers to (v, u) and (u, w) in the shortcut (v, w).

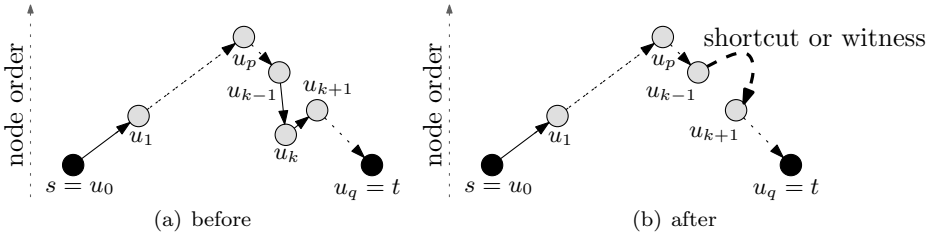


Fig. 3. Step of the correctness proof to construct a s-t-path of the form (PF).

Pruning the Query Search Space. using the *stall-on-demand* technique [Schultes and Sanders 2007] reduces the number of settled nodes: Before a node u is settled at distance $d_\uparrow(u)$, we check whether there exists an edge $e = (u, v) \in E^*$ with $\downarrow(e) = \text{true}$ and $d_\uparrow[v] + c(e) < d_\uparrow[u]$. In this case we can *stall* u since the computed distance to u is suboptimal and we do not relax the edges of u which leads to a considerably smaller search space. Moreover, stalling can propagate to additional nodes w in the neighbourhood of u, if the path via v to w is shorter than $d_\uparrow[w]$. We perform a BFS from u using the edges available in G^*, but stop at nodes that are not being stalled. To ensure correctness, we unstall a node u if a shorter path to u than the current one in is found by the regular query algorithm. Stall-on-demand is also applied to the backward search in the same way.

3.3 Node Order Selection

In this section we fill in the remaining details of the node order selection. [Bauer et al. 2010a] show that selecting an optimal node order that minimizes the query search space is NP-hard. Our selection is based on heuristics and the observation is that we do not need to know the complete order of the nodes before we can start contracting nodes: it is sufficient that we know the *next* node to be contracted. The selection of the next node is done using a priority queue. The priority of a node is the linear combination of several *priority terms* and estimates the attractiveness of contracting this node. A priority term is a particular property of the node and can be related to the already contracted nodes and the remaining nodes. Thus the priority terms can change after the contraction of a node and need to be *updated*. To keep the time required for priority updates small, we only update the neighbours of the contracted node.

Lazy Updates. In general, more than the neighbours are affected. So we update the priority of the top node on the queue before we remove it (*lazy update*). Since the node with the *smallest* priority is on top, *increasing* priorities get updated in time. To further improve the node order selection, we also do a complete update of all priorities if there were recently too many lazy updates. A check interval t is given and if more than a certain fraction $a \cdot t$ of successful lazy updates occur during a check interval, the update is triggered. We currently only use $a := 1$.

Edge Difference. Intuitively, the number of edges in the remaining graph should decrease while more and more nodes get contracted. The change in the number of edges caused by a node contraction is called *edge difference*. It is arguably the most

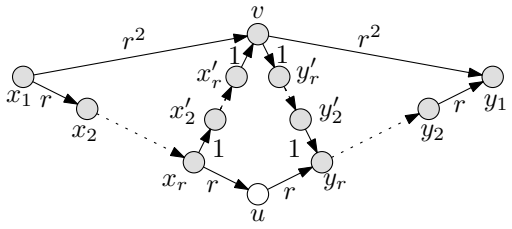


Fig. 4. After the contraction of node u the edge difference of node v in this directed, weighted graph changes.

important priority term and we calculate it with a *simulated* contraction of node u. For our implementation, we used the difference in the space requirements. Note that we store *two* edges (v, w) and (w, v) with the same weight $c(v, w) = c(w, v)$ as only *one* edge with two additional forward and backward flags. We could also use the cardinality difference but this would ignore the space consumption.

Contracting a node u can affect the edge difference of node v that is arbitrarily far away as we show in Figure 4: The contraction of node v will require new shortcut (x_1, y_1), whereas before, this shortcut was not necessary because of the witness path $\langle x_1, x_2, , \ldots, x_r, u, y_r, \ldots, y_2, y_1 \rangle$. However, the neighbours of u are affected the most since they may get new incident edges.

If after the contraction of a node u, the priority of a node v changes that is not a neighbour of u, v may be extracted from the priority queue in a different order than desired. Lemma 3.2 shows that under certain conditions, lazy updates can reestablish the correct order.

LEMMA 3.2. *We contract all nodes in correct order if (a) after the contraction of a node, all of its neighbours are updated, (b) the local searches for witness paths are unlimited, (c) the edge difference is the the only priority term and has a non-negative coefficient and (d) lazy updates are used.*

PROOF. The edge difference of a node v depends only on the incoming and outgoing edges of v and on the existing witness paths. After the contraction of a node u, the edges only change for the neighbours of u. These changes are covered by (a). Therefore only existing witness paths can affect a node v that is not a neighbour of u. Because of (b), we will not find a witness path after the contraction of u, if there previously was no witness path. Thus witness paths can only vanish, leading to an increasing priority (c). Lazy updates (d) will adjust those priorities in time. □

We cannot directly apply Lemma 3.2 to our preprocessing since we have search limits. However, if our limits are sufficiently large, we can expect only few deviations from the projected node order.

Uniformity. Using only the edge difference, one can get quite slow routing. For example, if the input graph is a path, contraction could produce a linear hierarchy where most queries would again follow paths of linear length. Such a situation can happen e. g. in dead-end valleys. In contrast, if we iteratively contract maximal independent sets, we would get a hierarchy where any query is finished in logarith-

mic time. More generally, it seems to be a good idea to contract nodes everywhere in the graph in a uniform way, rather than keep contracting nodes in a small region. We have tried several heuristics for choosing nodes uniformly out of which we present the three most successful ones. For all measures used here, a large value means that the node is contracted late.

Deleted Neighbours: Every node has a counter that gets incremented when a neighbour is contracted. Obviously, this quantity can be maintained correctly by either lazy update or by updating the neighbours of a contracted node. This heuristic is very simple and can be computed efficiently.

Original edges term: For each shortcut we store the number of original edges in the represented path. The original edges term is the sum of the number of original edges of the necessary shortcuts. This increases the space requirements but the term is beneficial, e. g. for path unpacking.

Voronoi Regions: Let $R(v) := \{u \text{ contracted} \mid d(v,u) < \infty, \forall \text{ uncontracted } w : d(v,u) \leq d(w,u)\}$ be the Voronoi-region of a uncontracted node v. We use $\sqrt{|R(v)|}$ as term in the priority function. By arbitrary tie breaking, we ensure that a node is in at most one $R(v)$. Note that in directed graphs, a contracted node may be in no region. When v is contracted, its neighbouring Voronoi regions will 'eat up' $R(v)$. [Maue et al. 2009] describe how the necessary computations can be made using $O(|R(v)|)$ steps of Dijkstra's algorithm. Assuming that we always contract Voronoi regions of size O(average region size), the total number of Dijkstra-steps for maintaining the Voronoi regions is $O(n \log n)$, i. e., computing them is reasonably efficient. Since they can only grow, lazy updates ensures that the priority queue works correctly w.r.t. this term of the priority function.

Cost of Contraction. The most time consuming part of the contraction are the local searches for witness paths. Since their durations vary from node to node, we want to contract 'expensive' nodes later in a smaller remaining graph. For Dijkstra searches, we include the number of settled nodes as priority term, for the fast local 1-hop search we use the number of scanned edges and for the fast local 2-hop search we use the number of bucket entries plus the number of scanned edges during the 1-hop forward search. Perfectly updating the cost of contraction would be difficult since the contraction of any node in a search tree of the local search can affect it.

Cost of Queries. We have implemented the following simple estimate $Q(u)$ that is an upper bound for the number of hops of a path $\langle s, \ldots, u \rangle$ explored during a query: Initially, $Q(u) = 0$. Inductively, when $Q(u)$ is an upper bound and u is contracted then $Q(u) + 1$ is an upper bound for a path from s via u to a neighbour v, so for each neighbour v, we update $Q(v) := \max(Q(v), Q(u) + 1)$.

Lemma 3.3 extends Lemma 3.2 to some of the just presented priority terms; we omit the proof.

LEMMA 3.3. *Lemma 3.2 holds if additionally the uniformity terms and the cost of queries term are used with non-negative coefficients.*

Generally speaking, one can come up with many heuristic terms, but gets an inflation of tuning parameters. Therefore, in the experiments we try to keep their number small, we use the same set of parameters for different inputs, and we make some sensitivity analyses to test their robustness.

4. MOBILE SCENARIO

Because of the simple query algorithm and the small search space, see Section 7, CHs are perfectly suited for mobile devices with slow processors and limited memory. However, some modifications to the original algorithm are necessary to engineer a fast mobile algorithm.

4.1 Locality

Reading data from external memory is the bottleneck of our query application. To get a good performance, we want to arrange the data into blocks and access them blockwise. Obviously, the arrangement should be done in such a way that accessing a single data item from one block typically implies that a lot of data items in the same block have to be accessed in the near future. In other words, we have to exploit locality properties of the data.

In a first level of abstraction, we need to find a node numbering that reflects locality. Therefore, our node numbering will no longer coincidence with our CH node order. The node numbering of real-world road networks sometimes already respects *spatial* locality, i. e., the nodes are numbered somehow by spatial proximity. However, we can do better. We consider the reverse search graph $\overline{G^*} = (V, \overline{E^*} := \{(v,u) \mid (u,v) \in E^*\})$, which is an acyclic graph (as G^*), and compute a topological numbering defined by the finishing times of a depth-first search (DFS).

We furthermore stress the *hierarchical* locality: For a fixed *next-layer fraction* f, we divide the nodes into the group of $(1-f) \cdot |V|$ nodes of smaller importance, and the group of $f \cdot |V|$ nodes of higher importance. Within each group, we keep the relative topological numbering obtained by our modified DFS. We recurse in the second group until all nodes fit into a single block. This *hierarchical renumbering* step is a slightly generalised version of a technique used in [Goldberg et al. 2007]. It is important to note that the resulting numbering still represents a topological order.

Note that a good node numbering has not only the obvious advantage that a loaded block contains a lot of relevant data, but also can be exploited to compress the data effectively.

4.2 Main Data Structure

The starting point for our compact graph data structure is an *adjacency array* representation: All edges (u,v) are kept in a single array, grouped by the source node u. Each edge stores only the target v and its weight. In addition, there is a node array that stores for each node u the index of the first edge (u,v) in the edge array. The end of the edge group of node u is implicitly given by the start of the edge group of u's successor in the node array.

We divide this graph data structure into blocks each of which stores a set of nodes together with their associated edge data, see Figure 5. When encoding the target v of an edge (u,v), we want to exploit the existing locality, i. e., in many cases the difference of the IDs of u and v is quite small and, in particular, u and v often belong to the same block. Therefore, we distinguish between *internal* edges leading to a node within the same block, and the remaining *external* edges. We use a flag to mark external edges. In case of an internal edge, it is sufficient to

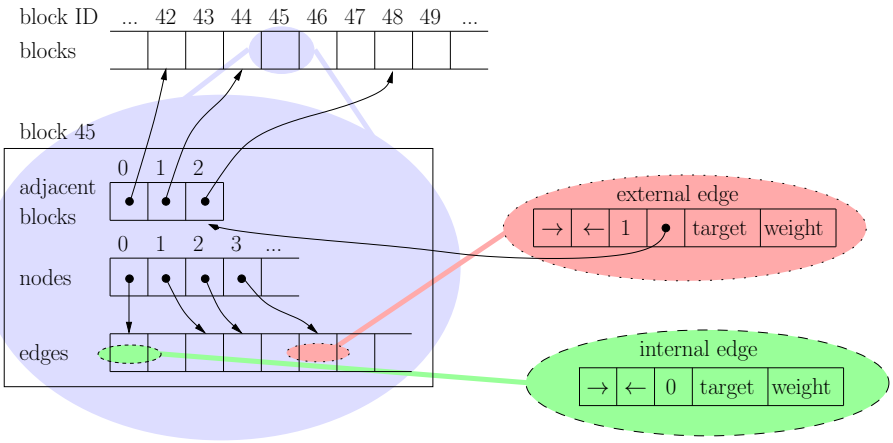


Fig. 5. External-memory graph data structure (without path unpacking information). Each edge stores three flags: a forward flag (→), a backward flag (←), and a flag that indicates whether it is an external edge leading to a node in a different block.

just store the node index within the same block, which requires only a few bits. In case of an external edge, we need the block ID of the target and the node index within the designated block. It can be expected that the number of blocks adjacent to a given block B is rather small, i. e., there are only a few different blocks that contain all the nodes that are adjacent to nodes in B. Thus, it pays to explicitly store the IDs of all adjacent blocks in an array in B. Then, an external edge need not store the full block ID, but it is sufficient to just store the comparatively small block index within the adjacent-blocks array.

4.3 Building the Graph Representation

Our goal is a graph data structure that occupies as little memory as possible and allows fast data access. We make the following design choices: each block has the same constant size and contains a subset of consecutively numbered nodes together with their incident edges.

All three 'logical' arrays (adjacent blocks, nodes, edges) are stored in a single byte array one after the other, the starting index of each logical array is stored in the header of the block. Within each block we use the minimal number of bits to store the respective attributes. For example, if a block has 42 adjacent blocks, then each external edge (u, v) in this block uses 6 bits to address the adjacent block that contains v.

In general, building the blocks is not trivial since the memory a node and its edges occupy depends on the other nodes in the block. In particular, an internal edge typically occupies less memory than an external edge. Fortunately, we can exploit the fact that we numbered the nodes topologically. When we process the nodes from the lowest numbered one to the highest numbered one, all edges (u, v) point to nodes v that have already been processed. This implies that we already know whether (u, v) is an internal or external edge and, in case of an external edge, we also know the number of nodes in the corresponding block B so that we can

choose the minimal number of bits required to encode the index of node v within the block B. This way, we can easily calculate the memory requirements of the current edge. If all edges of the current node u fit into the current block, the node and its incident edges are added. Otherwise, a new block is started. Note that when we consider to add another node and its edges, we have to account not only for the memory directly used by them, but also for a potential memory increase of the other nodes and edges in the same block: for example, whenever the number of edges in the block exceeds the next power of two, all nodes in the block need an additional bit to store the index of the first outgoing edge.

Since most edge weights in our real-world road networks are rather small and only comparatively few edges, e. g. long shortcuts, are long, we use one bit to distinguish between a long and a short edge; depending on the state of this bit, we use more or less bits to store the weight.

4.4 Storing the Graph Representation

The blocks representing the graph are stored in external memory. In main memory, we manage a cache with a simple least-recently used (LRU) strategy that can hold a subset of the blocks. In the external-memory graph data structure, a node u is identified by its block ID $B(u)$ and the node index $i(u)$ within the block. We need a mapping from the node ID u used in the original graph to the tuple $(B(u), i(u))$. Such a mapping is realised in a simple array, stored in external memory.

We want to access the external memory *read-only* in order to improve the overall performance and to preserve the flash memory, which can get unusable after too many write operations. Therefore we clearly separate the read-only graph data structures from some volatile data structures, in particular the priority queues. We use a hash map to manage pointers from reached nodes to the corresponding entries in the priority queues. Since the search spaces of CHs are so small (a few hundred nodes), it is no problem to keep these data structures in main memory. Note that [Goldberg and Werneck 2005] used a similar distinction between read-only and volatile data structures.

4.5 Path Unpacking Data Structures

The above data structures are sufficient to determine the shortest-path *length*. In order to generate actual driving directions, it must also be possible to generate a description of the shortest path. First of all, since we have changed the node numbering, we need to store for each node its original ID so that we can perform the reverse mapping. Furthermore, we need the functionality to unpack shortcuts. To support a simple recursive unpacking routine, we store the ID of the middle node of each shortcut (see Section 3). We distinguish between internal and external shortcuts (v, w), where the middle node u belongs to the same block as v or not. For an internal shortcut, we store the middle node as index $i(u)$, for an external shortcut, we also have to specify the block $B(u)$.

To accelerate the path unpacking, we refine the approach of [Delling et al. 2006] to store explicit descriptions of the paths underlying some of the shortcuts. Expanding the *external* shortcut (v, w) to the edges (v, u) and (u, w) might require an expensive additional block read. Therefore, it is reasonable to completely pre-unpack all external shortcuts and to store the corresponding node sequences in some additional

data blocks. Instead of the middle node, we store the starting index within these additional data blocks. A new feature is that we exploit the fact that an external shortcut can contain other external shortcuts. We do not store these contained shortcuts explicitly, it is sufficient to just note the correct starting position and a direction flag since contained shortcuts might be filed in the reverse direction. We consider external shortcuts in a descending order of importance of u. A shortcut is unpacked only if it is not contained in an already unpacked shortcut.

5. DYNAMIC SCENARIO

Many applications do not deal with a mere static graph. Small changes take place over time and the CH needs to be updated to remain correct. In such cases, rebuilding the complete CH is often too time-consuming. Here, we present an approach that efficiently processes a small amount of changes in the edge set, i. e. insertion and deletion of edges as well as changes of edge weights.

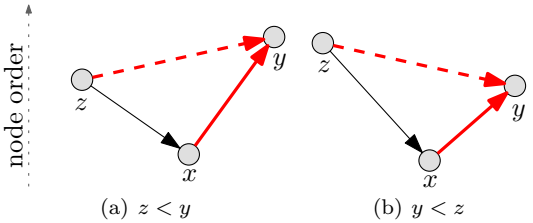
5.1 Processing the Changes

The most time-consuming part of the CH precomputation is the node ordering. Because we only deal with a small amount of edge changes, we keep the original node order. Instead of recontracting of the whole graph, we update existing shortcuts to comply with the changes and then identify the subset U of nodes whose contraction has to be repeated to add new shortcuts. Certainly, the recontraction of the other nodes $V \setminus U$ is unnecessary since no new shortcuts have to be added.

5.2 Updating Existing Shortcuts

For each changed edge (u, w) we find all shortcuts containing (u, w). This way we can delete them or adjust their weight. After this step, only valid shortcuts remain in the graph.

We process all changed edges except new edges, which are never part of an existing shortcut. Let $(u, w) \in E^*$ be an edge or shortcut. Since $u < w$, (u, w) can only become part of other shortcuts if, during the contraction of u, a shortcut $\langle v, u, w\rangle$ is added. This shortcut may be contained in other shortcuts so that a *DFS* from (u, w) (Algorithm 2) will find all shortcuts (z, y) containing (u, w). To identify the shortcuts correctly, we must store the middle node. Also, (z, y) may be either stored in E^* with z in case that $z < y$ (Lines 3–4) or stored in reverse direction with y in case that $y < z$ (Lines 5–6), see Figure 6 for an example with $r = \uparrow$.



(a) $z < y$ (b) $y < z$

Fig. 6. The shortcut (z, y) containing edge (x, y) is (a) either stored at z or (b) in reverse direction at y.

Algorithm 2: LocateShortcutsContaining($(u, w) \in E^*$)

```
1 K:= {}; if ↑(u,w) then K.pushBack(u,w,↑); if ↓(u,w) then
    K.pushBack(u,w,↓);                                   // stack
2 while (x,y,r):= K.popBack() do          // invariant: x < y
3     foreach (x,z) ∈ E*, (z,y) ∈ E* do   // z is neighbour of x
4         if r(z,y) and x is middle node of shortcut (z,y) then
            K.pushBack(z,y,r);                           // z < y
5     foreach (y,z) ∈ E* do                // ¬↑=↓, ¬↓=↑
6         if (¬r)(y,z) and x is middle node of shortcut (y,z) then
            K.pushBack(y,z,¬r);                          // y < z
```

The remaining shortcuts are all valid, though some may have become redundant. They no longer represent a shortest path. But since they do not invalidate the correctness of the CH, we keep them although they can somewhat slow down the query.

5.3 Recontracting Nodes

The changes to the edge set may make new shortcuts necessary. We call a node u *affected* if the contraction of u has to be repeated to add new shortcuts needed for retaining a correct CH. An affected node v is a *seed node* if its recontraction adds new shortcuts even when we only apply the changes to the existing edges. An affected node u is therefore reachable in the search graph from a seed node z. Otherwise u would not be affected at all. When a set of edges change, our strategy is to first find a superset S of the seed nodes and then obtain a superset of the affected nodes by considering all nodes reachable in the search graph from a node in S.

There are two fundamentally different ways in which a node may become a seed node.

(1) We have new or shortened edges (including shortcuts) (u, v). Such edges can be part of new shortcuts. Recontracting $\min\{u, v\}$ can add new shortcuts thus $\min\{u, v\}$ might be a seed node.

(2) Edges (including shortcuts) become longer or are deleted (this is equivalent to becoming longer by ∞). If such edges are part of witnesses found during the contraction of a node u, these witnesses may become invalid. In order to

find u, we precompute for each edge or shortcut e a *seed set* $A_e := \{(u, \delta) \mid$ shortcut $\langle v, u, w \rangle$ was omitted due to witness path $P = \langle v, \ldots, w \rangle$ using e, $\delta := c(\langle v, u, w \rangle) - c(P)\}$. A_e can easily be computed during the original contraction. Note that if an edge e is part of a shortcut f then the nodes in A_e need not be a superset of the nodes in A_f.
To determine the additional seed node we find the smallest delta $\delta_{\min}$ for each u showing up in A_e of a lengthened edge e. Then we add the length increments of all edges e that have u in their A_e and decide whether this sum exceeds $\delta_{\min}$. If the sum exceeds $\delta_{\min}$, u might be a seed node: an eliminated shortcut $\langle v, u, w \rangle$ might be necessary due to the changes. If the sum does not exceed $\delta_{\min}$, u has certainly not become a seed node because of lengthended edges: All witness paths for a shortcut of the form $\langle v, u, w \rangle$ are still shorter.

Having determined S and a superset of U we can now simply repeat the contraction for all these nodes. This limited reconstruction is faster than the normal construction: We contract fewer nodes; a large part of the CH is still intact, thus many correct shortcuts are part of the graph; witness searches benefit from the valid shortcuts; few new shortcuts have to be added.

6. DYNAMIC MOBILE SCENARIO

When dealing with few changes and a limited amount of queries, the conventional dynamic approach has some shortcomings. Even rebuilding only the affected part of the CH takes more time than performing very few queries using a simple Dijkstra's algorithm. In the mobile scenario, we are therefore interested in techniques that only take changes into account which affect the queries.

6.1 Iterative Routing

The most common change in the edge set is increasing the weight of an edge, e. g. introducing a traffic jam. [Schultes 2008] observed that a lengthened or deleted edge can only change the result of an s-t query if it is part of the original shortest s-t path. Therefore this kind of updates can be handled in a way that ensures that only increments important to the query are processed.

We repeat the query until the shortest path does not differ from the shortest path of the previous iteration. Initially we only use all new or shortened edges to determine new seed nodes. Deleted or lengthened edges are only considered if they are on a shortest path of any previous iteration. This can greatly reduce the amount of lengthened and deleted edges that have to be processed. To identify edge changes on the shortest path, we have to unpack it after each query. Usually very few iterations are required to compute the correct result.

This approach is independent of the speedup technique applied and can be used whenever the time to process changes outweighs the time to perform several queries.

6.2 Handling Seed Nodes

In the mobile scenario we cannot afford to recontract nodes. Instead, we ensure that the backward search of the query finds all currently known seed nodes on the shortest path. If the seed nodes are found on the shortest path, no additional shortcuts skipping these nodes are necessary. To achieve this we first determine the

set of nodes reachable from the seed nodes by one or more edges (v, u) in G^* with $\uparrow (v, u) = \text{true}$. We add the edge (u, v) with $\downarrow (u, v) := \text{true}$ to such a node u in G^*, even though $v < u$. This enables the backward search to find the shortest path to any seed node because now every edge useable by the forward search can also be used by the backward search. [Schultes 2008] used a similar technique to make queries unidirectional.

We do not have to completely repeat the search for reachable nodes in G^* in each iteration. It suffices to process the edge changes added in this iteration. Furthermore, if we keep track of all nodes reachable in the last iteration we can prune the search for new nodes quite early on. As a result most of the additional edges get added to G^* in the first two iterations.

6.3 Data Structures

The additional data for each edge, witness data and middle node, is directly stored with the associated edge. To compress the safety of the witness data, we use the same scheme employed to compress the weight of an edge. Furthermore we reduce the size of the seed sets by storing only (u, δ_1) if two elements (u, δ_1), (u, δ_2), $\delta_1 < \delta_2$ are part of the same seed, since we would determine the minimum later on anyway. We also need additional data structures for our read-only graph: Δ_1 (Delta_1) stores all the changes to the edge set. It is mainly used to identify changed edges during the iterative routing, thus only storing deleted and lengthened edges. Δ_2 holds all changes in the CH and is used by the query. It contains all edges inserted in G^*, weight changes and deletions of edges and shortcuts, and new edges. To realise these data structures we employ hash maps. Because the edges inserted in G^* are specific to the query, it pays of to clear Δ_2 afterwards if the next query will be substantially different.

7. EXPERIMENTS

Our programs were written in C++. No libraries, except for the C++ Standard Template Library were used to implement the algorithms. To obtain a robust implementation, we include extensive consistency checks using assertions and perform experiments that are checked against reference implementations, i. e., queries are checked against Dijkstra's algorithm.

7.1 Experimental Setting

Environment. Our experiments were done on one core of a single AMD Opteron Processor 270 clocked at 2.0 GHz with 8 GiB main memory and 2×1 MiB L2 cache, running SuSE Linux 10.3 (kernel 2.6.22). For the mobile scenario, we used a Nokia N800 Internet Tablet, equipped with 128 MiB of RAM and a Texas Instruments OMAP 2420 microprocessor, which features an ARM11 processor running at 400 MHz. We used a SanDisk Extreme III SD flash memory card with a capacity of 2 GB; the manufacturer states a sequential reading speed of 20 MiB/s though the device limits this to 8 MiB/s. The operating system is the Linux-based Maemo 4.1 in the form of Internet Tablet OS2008 4.2008.30-2. The programs were compiled by the GNU C++ compiler 4.2.1 using optimisation level 3.

Instances. For most practical applications, a *travel time* metric is most useful, i. e., the edge weights correspond to an estimate of the travel time that is needed to traverse an edge. In order to compute the edge weights, we assign an average speed to each road category.

7.2 Main Instance

Most of our experiments were done on a road network of Western Europe having 18 029 721 nodes and 42 199 587 directed edges. The countries Austria, Belgium, Denmark, France, Germany, Italy, Luxembourg, the Netherlands, Norway, Portugal, Spain, Sweden, Switzerland, and the UK are represented. We usually refer to it as 'Europe' and it has been made available for scientific use by the company PTV AG. For each edge, its length and its road category are provided. There are four major road categories (motorway, national road, regional road, urban street), which are divided into three subcategories each. In addition, there is one category for forest and gravel roads. The assigned speeds in this order are 130, 120, 110, 100, 90, 80, 70, 60, 50, 40, 30, 20, 10 km/h.

7.3 Additional Instances

In addition, we also performed some experiments on two other road networks. A publicly available version of the US road network (without Alaska and Hawaii) has 23 947 347 nodes and 57 708 624 directed edges that was obtained from the TIGER/Line Files. These were provided by the [U.S. Census Bureau, Washington, DC 2002] (*USA*) and distinguish between four road categories with assigned speeds of 100, 80, 60, 40 km/h. The company ORTEC provided a new version of the European road network (*New Europe*) with 33 726 989 nodes and 75 108 089 directed edges for scientific use. Additionally to *Europe* it covers the Czech Republic, Finland, Hungary, Ireland, Poland, and Slovakia. It distinguishes between motorways, multiple and single lane A and B roads, regional, local and other roads outside/inside cities, delivery roads, pedestrian zones and ferries. We used the rather slow ORTEC car speed profile that assigns speeds 87, 84/77, 73/63, 60/53, 50/40, 37/27, 23/17, 13/10, 8, 5, 2 km/h.

Preliminary Remarks. Unless otherwise stated, the experimental results refer to the scenario where the road network of *Europe* with *travel time* metric is used, and only the shortest-path *length* is computed without outputting the actual route.

When we specify the memory consumption, we usually give the *overhead*, which accounts for the *additional* memory that is needed by our approach compared to a space-efficient *unidirectional* implementation of Dijkstra's algorithm. This overhead is always expressed in 'bytes per node'.

7.4 Methodology

To calculate the average query time, we pick source-target pairs *uniformly at random*. Unless otherwise stated, we perform 100 000 queries.

For use in applications it is unrealistic to assume a uniform distribution of queries in large graphs such as Europe or the USA. Therefore, we also measure *local queries* within the big graphs. We choose random sample points s and for each power of two $r = 2^k$, we use Dijkstra's algorithm to find the node t with Dijkstra rank $\mathrm{rk}_s(t) = r$.

The Dijkstra rank rk_s is the order in which the nodes were settled during the search starting at node s. By plotting the resulting statistics for each value $r = 2^k$, we can see how the performance scales with a natural measure of difficulty of the query. We represent the distributions as a box-and-whiskers plot: each box spreads from the lower to the upper quartile and contains the median, the whiskers extend to the minimum and maximum value omitting outliers, which are plotted individually. Such plots are based on 1 000 random sample points s.

We can obtain a *per-instance worst-case guarantee*, i. e., an upper bound on the search space size for any possible point-to-point query for a given fixed graph G. We do this by executing a forward search and a backward search from each node of G until the priority queue is empty, no abort criterion is applied. This approach was first mentioned in [Sanders and Schultes 2006].

For the mobile scenario, we distinguish between four different query types:

(1) *'cold'*: After each query, clear the cache. This way, we can determine the time that is needed for the first query after the program is started and has an empty cache.

(2) *'warm'*: Perform two experiments with different sets of s-t-pairs in a row. We measure only the second one to determine the average query time when the device has been in use for a while.

(3) *'recompute'*: We have pairs of queries. We only measure the second one and clear the cache after each pair. The second query has the same target node but another source node: We chose a random neighbour of a random node on the shortest path of the first query.

(4) *'w/o I/O'*: Select 100 random source-target pairs. For each pair, repeat the same query 101 times; ignore the first iteration when measuring the running time. This way, we obtain a benchmark for the actual processing speed of the device when no I/O operations are performed.

For practical scenarios, the first and the third query type are most relevant; The second query time is closest to the situation reported in related work.

7.5 Parameters

Despite the simplicity of the description of CHs, there are many parameters, i. e., the coefficients of the priority terms in the priority function for the node ordering (Section 3.3) and the local search limits for the contraction (Section 3.1). For our *aggressive variant* we select parameters to minimise the query time and for our *economical variant* to minimise the product of query time and preprocessing time. Additionally to updating the neighbors, we always use lazy updates since they decrease the query time more than they increase the preprocessing time, the differences are around 15–20%. Our parameters have been determined by a manual, systematic coordinate search. Figure 7 shows the development of the average degree during node contraction for different hop limits. We see that for hop limits below four, the average degree eventually explodes. We choose limits for the average degree that switch to a larger hop limit before this explosion. We refer to [Geisberger 2008] for an in-depth description including a sensitivity analysis.

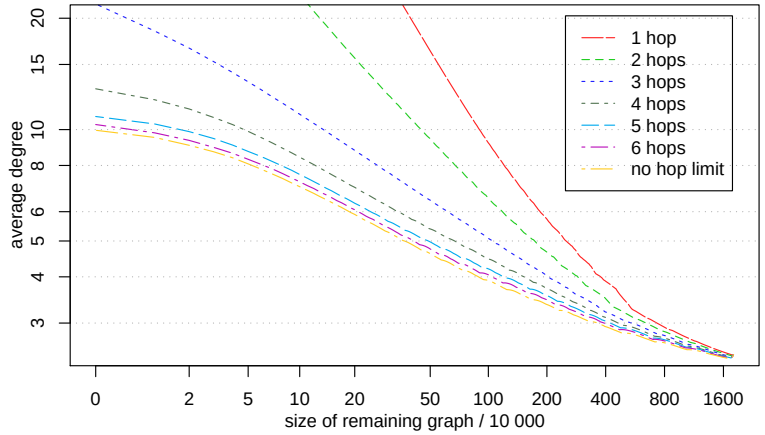


Fig. 7. Average degree development for different hop limits on Europe.

Table I. Performance of various node ordering heuristics. Terms of the priority function: E=edge difference, D=deleted neighbours, S=search space size, W=relative betweenness, V=$\sqrt{\text{Voronoi region size}}$, L=limit search space on weight calculation (1 000 settled nodes), Q=upper bound on edges in search paths, O=original edges term. Digits denote hop limits for testing shortcuts. The best performance in every column is bold.

	method	node ordering [s]	hierarchy construction [s]	query [μs]	nodes settled	non-stalled nodes	edges relaxed	space overhead [B/node]
	E	13 010	1 739	670	1 791	1 127	4 999	-0.6
	ES	5 355	123	245	614	366	1 803	-2.5
	ESL	1 158	123	292	758	465	2 169	-2.5
	ED	7 746	1 062	183	403	236	1 454	-1.3
	EDL	2 071	576	187	418	243	1 483	-1.3
	EDSL	1 414	165	175	399	228	1 335	-1.6
	EO	5 979	758	250	617	395	2 119	**-3.1**
	EOL	1 274	319	245	604	383	2 119	**-3.1**
	EOSL	1 110	145	222	531	313	1 802	-3.0
ECONOMICAL	ED5	634	98	224	470	250	1 674	-0.6
ECONOMICAL	EDS5	652	99	213	462	256	1 651	-1.1
ECONOMICAL	EDS1235[a]	545	57	223	459	234	1 638	1.6
ECONOMICAL	EDSQ1235[a]	591	64	211	440	236	1 621	2.0
ECONOMICAL	EDOSQ1235[a]	555	59	198	435	241	1 540	1.5
ECONOMICAL	EDOS1235[a]	498	53	200	438	239	1 514	1.1
ECONOMICAL	EOS1235[a]	**451**	**48**	214	487	275	1 684	0.6
AGGR.	EDSQL	1 648	199	173	385	220	1 378	-1.1
AGGR.	EVSQL	1 627	170	159	368	209	1 181	-1.7
AGGR.	EVOSQL	1 644	165	**152**	**356**	**207**	**1 163**	-2.1

[a] hop@degree limit: 1@3.3, 2@10, 3@10, 5

7.6 Standard Scenario

We start with evolving sets of priority terms and search space limits to get a deeper insight into them, see Table I.

Using the edge difference (letter E) as the sole priority term yields a CH that already answers a query in less than 1 ms. However, the preprocessing time is still too large. Also regarding the cost of contraction as a priority term (letter S) results in more than two times better node ordering time and 14 times better hierarchy construction time. The imbalance between the improvement of those two parts is due to the additional local searches during the node ordering, particularly the initialisation of the priority queue takes more than 30 minutes. So we limit the local searches (letter L), improving the node ordering time by an additional factor of four.

Adding the deleted neighbours counter (letter D) accelerates the query, the average is below 200 μs. The algorithm in Line EDSL is a simple combination with improved preprocessing, fast query times and *negative* space overhead for shortest path distance calculation. We can achieve negative space overhead since in a CH we need to store an edge only with one of its endpoints, even if the edge is bidirected. Using the original edges term (letter O) as uniformity term decreases preprocessing time and space but increases query time compared to the deleted neighbours term. To significantly decrease the preprocessing time, we introduce a hop limit to the local searches (digit 5) leading to a two times better node ordering time. Applying staged hop limits (digits 1235) shrinks the preprocessing time below 10 minutes. The original edges term (letter O) can further improve preprocessing, query and space overhead. After removing the contracted neighbours counter (Line EOS1235, priority function $= 190 \cdot \mathrm{E} + 600 \cdot \mathrm{O} + \mathrm{S}$), we get our *economical variant*. Its low preprocessing time of 7.5 minutes and the fast query time of about 200 μs provide the best balance. To further decrease the query time, we first exchange the current uniformity term for Voronoi regions (letter V) combined with the original edges term (letter O), and add a priority term to estimate the cost of queries (letter Q) to decrease the query time (Line EVOSQL, priority function $= 190 \cdot \mathrm{E} + 60 \cdot \mathrm{V} + 70 \cdot \mathrm{O} + \mathrm{S} + 145 \cdot \mathrm{Q}$). This leads to our *aggressive variant* having 29% faster query time then the economical variant, and a speedup of 40 000 compared Dijkstra's algorithm. However we need to invest more time into preprocessing.

Outputting Complete Path Descriptions. needs an average of 323 μs for the aggressive variant and 321 μs for the economical variant. These unpacking times are the fastest we have seen when no completely unpacked representations of shortcuts are used (see Section 4.5). Since we store the middle node, the hierarchy construction and query time increases up to 13% and the space overhead reaches 6.2 B/node and 10.3 B/node respectively. The comparatively large space overhead is owed to the fact that we even use 12 B/node for non-shortcuts. If we would implement a more sophisticated version with only 8 B/edge for non-shortcuts, we would achieve a space overhead of only 1.2 B/node for the aggressive variant.

Local Queries. Since random queries are unrealistic for large graphs, Figure 8 shows the distributions of query times for various degrees of locality. We see good query performance over all Dijkstra ranks and small fluctuations. This is further underlined in Figure 9 where we give upper bounds for the search space size of *all* $n \times n$ possible queries. We see a superexponential decay of the probability to observe a certain search space size and a maximal search space size bound less than

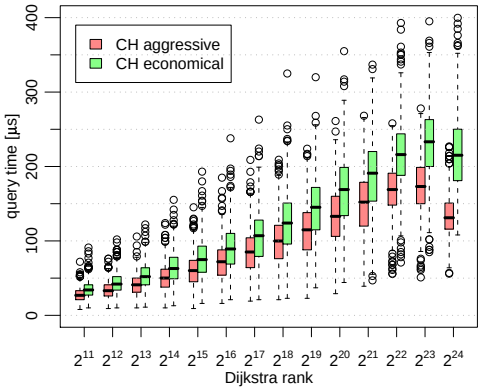


Fig. 8. Performance of queries, where the target is chosen by the Dijkstra rank from the source.

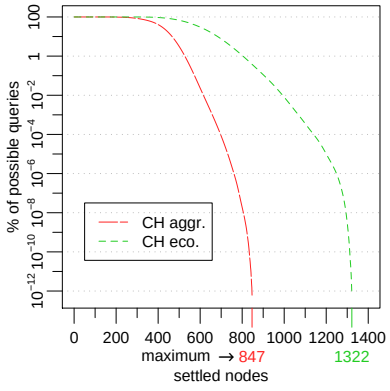


Fig. 9. Upper bound for the worst percentages of queries.

2.4 times the size of the average actual searchspace sizes (see also Table I).

7.7 Comparisons

Speedup techniques for routing in road networks are usually compared against each other in the three-dimensional space of preprocessing time, space overhead and query time. Although there are many Pareto-optimal techniques (techniques being better than any other technique for at least one dimension), CHs take a particularly strong position. We compare the most successful speedup techniques in Table II. For further algorithms we refer to [Delling et al. 2009; Bauer et al. 2010b]. We took the timings from several other papers that used roughly the same hardware, so we can only do a rough comparison. Still, all techniques either use CHs or are clearly dominated by a technique using CHs. CHs provide the lowest space overhead, even lower than Dijkstra's algorithm, and the fastest preprocessing times, except Dijkstra's algorithm. There are two kinds of algorithms with faster query times. First, there are combinations of hierarchical techniques and goal-directed techniques, of which the most successful ones are all based on CHs. Second, transit-node routing (TNR), whose precomputation also relies on the CH node order for best results (see Section 8). The combination of TNR + AF provides currently the best query time.

[RG: fix positions of tables and figures]

7.8 Other Inputs

We also tested our aggressive and economical variant on different road networks and metrics to examine the robustness of CHs using the same parameters as for the Europe road network. We can expect additional improvements if we were to repeat the parameter search. The USA (Tiger) graph shows slightly larger preprocessing times than the Europe graph but faster query times, those effects are already known from HNR. The New Europe graph is a larger network thus requiring more preprocessing time. The hierarchy construction time for the aggressive variant is more than a factor 2 larger than expected. That is because the limit on the settled nodes for the local searches during the priority calculation is too restrictive and does

Table II. Comparison of various speed-up techniques in the three-dimensional space of preprocessing time, space overhead and query time. A technique is dominated by CHs if CHs is better in every dimension. The best performance in every column is bold.

		prepro.		query			dom.
		time	overh.	settled	time	uses	by
method	data from	[min]	[B/n.]	nodes	[ms]	CH	CH
Dijkstra[a]	[Bauer et al. 2010b]	0	0	9.11 M	5 591		
bidir. Dijkstra[a]	[Bauer et al. 2010b]	0	0	4.76 M	2 713		
eco. CH	this paper	**8**	0.6	487	0.21	✓	
aggr. CH	this paper	27	-2.1	356	0.15	✓	
ALT-16[b]	[Goldberg et al. 2009]	13	70	82 348	120.1		✓
ALT-64[a]	[Delling and Wagner 2007]	68	512	25 234	19.6		✓
AF[c]	[Hilger et al. 2009]	2 156	25	1 593	1.1		✓
REAL[b]	[Goldberg et al. 2009]	103	36	610	0.91		✓
HH	[Schultes 2008]	13	48	709	0.61		✓
HNR	[Schultes 2008]	15	2.4	981	0.85		✓
SHARC[a]	[Bauer and Delling 2009]	81	14.5	654	0.29		✓
bidir. SHARC[a]	[Bauer and Delling 2009]	158	21.0	125	0.065		✓[d]
CALT[a]	[Bauer et al. 2010b]	11	15.4	1 394	1.34		✓
eco. CH+AF[a]	[Bauer et al. 2010b]	32	0.0	111	0.044	✓	
gen. CH+AF[a]	[Bauer et al. 2010b]	99	12	**45**	0.017	✓	
partial CH[a]	[Bauer et al. 2010b]	15	**-2.9**	965 k	53.63	✓	
TNR	this paper	46	193	N/A	0.0033	✓[e]	
TNR+AF	[Bauer et al. 2010b]	229	321	N/A	**0.0019**	✓[e]	

[a] 2.6GHz AMD Opteron, SuSE Linux 10.3, 16GiB of RAM, 2×1MiB of L2 cache.
[b] 2.4GHz AMD Opteron, Windows Server 2003, 16GiB of RAM, 2MiB of L2 cache.
[c] 2.2GHz AMD Opteron, SuSE Linux 9.1, 4GiB of RAM, 1MiB of L2 cache.
[d] Dominated by CH+AF [Bauer et al. 2010b].
[e] Uses CHs to compute transit nodes.

not punish searches enough which have e. g. more than 2 million settled nodes in an unlimited local search. The additional time for contracting such nodes is not significant for node ordering, but for hierarchy construction. When we increase the settled nodes limit to 3 000 nodes, it yields a construction time of 259 s, and even reduces the node ordering to 2 047 s. For the distance metric, where each edge represents the driving distance, there are no real fast routes that could be preferred over other slower routes. It is less clear how to identify important nodes and more shortcuts are necessary. Note that the experiments on the distance metric of Europe were performed on a subgraph, the largest strongly connected component consisting of 18 010 173 nodes and 42 188 664 edges due to availability.

7.9 Mobile Scenario

Unless otherwise stated, our experiments refer to the case that the path-unpacking data structures exist, but are not used and 1 000 queries are performed. Instead of giving the space overhead, the space consumption includes the graph itself. Note that the query times always include the time needed to map the original source and target IDs to the corresponding block IDs and node indices, while figures on the memory consumption do not include the space needed for the mapping. The space consumption for the mapping is excluded because in most practical applications more sophisticated mappings are needed: For example street names are mapped to

Table III. Performance of different graphs and metrics.

		TRAVEL TIME						DISTANCE			
		Europe		USA Tiger		New Europe		Europe		USA Tiger	
		aggr.	eco.	aggr.	eco.	aggr.	eco.	aggr.	eco.	aggr.	eco.
	node ordering [s]	1 644	451	1 684	626	2 420	657	5 459	2 853	3 586	1 775
LENGTH	hier. construction [s]	165	48	181	61	646	72	264	137	255	113
LENGTH	query [μs]	152	214	96	180	213	303	1 940	2 276	645	1 857
LENGTH	nodes settled	356	487	283	526	439	629	1 582	2 216	1 081	3 461
LENGTH	non-stalled nodes	207	275	157	309	247	351	658	962	485	2 100
LENGTH	edges relaxed	1 163	1 684	885	1 845	1 732	2 600	15 472	19 227	7 905	27 755
LENGTH	space ov. [B/node]	-2.1	0.6	-2.6	-1.3	-2.0	-0.3	0.6	1.5	-1.5	-0.9
PATH	hier. construction [s]	176	54	191	68	673	82	287	152	269	122
PATH	query [μs]	170	238	107	198	243	345	2 206	2 615	721	2 121
PATH	expand path [μs]	323	321	1 105	1 107	972	953	798	792	1 268	1 336
PATH	space ov. [B/node]	6.2	10.3	5.8	7.8	5.6	8.5	10.2	11.7	7.4	8.3
PATH	edges	21	23	21	26	21	24	21	29	22	40
PATH	edges expanded	1 370	1 369	4 548	4 548	4 139	4 136	3 291	3 291	5 128	5 128

Table IV. Building the graph representation. We give the number of nodes, the number of edges in the original graph and in the search graph, the number of graph-data blocks (without counting the blocks that contain pre-unpacked paths), the average number of adjacent blocks per block, the numbers of internal edges, internal shortcuts and external shortcuts as percentage of the total number of edges, the time needed to pre-unpack the external shortcuts and to build the external-memory graph representation (provided that the search graph is already given), and the total memory consumption including pre-unpacked paths.

	$\lvert V\rvert$ [$\times 10^6$]	$\lvert E\rvert$ [$\times 10^6$]	$\lvert E^*\rvert$ [$\times 10^6$]	#blocks	#adj. blocks	int. edges	int. shcs.	ext. shcs.	time [s]	space [MiB]
Europe	18.0	42.2	36.9	52 107	9.1	70.6%	32.2%	7.7%	123	275
USA	23.9	57.7	49.4	80 099	8.4	69.2%	33.7%	8.0%	186	400
New Europe	33.7	75.1	65.7	103 371	8.3	70.3%	32.7%	7.5%	210	548

edges.

In the following we use a block size of 4 KiB, that was found using experiments with block sizes from 1 KiB to 64 KiB. This block size is optimal with respect to both space consumption and query time. We use a cache size of 64 MiB. Additional experiments indicate that reducing it to 32 MiB has negligible effect on the performance of 'warm'-queries. Even only 256 KiB of cache are sufficient to achieve the performance of our 'cold' queries. Finally, we use a value of 1/16 for the next-layer fraction from Section 4. This minimises query time and has only a small detrimental effect on the space consumption for which even smaller values would be better.

Table IV gives an overview of the external-memory graph representation. Building the blocks is very fast and can be done in about 2–4 minutes. Although the given memory consumption already covers everything that is needed to obtain very fast query times (including path unpacking), we need 30% *less* space than the original graph would occupy in a standard adjacency-array representation in case of Europe. Most of the savings come from using less bits than the naive representation, but we also save space because CHs need to store bidirectional edges only at one of their end points.

The results for the four query types are represented in Table V. On average, a

Table V. Query performance for four different query types.

		cold		warm		recompute		w/o I/O
	settled nodes	blocks read	time [ms]	blocks read	time [ms]	blocks read	time [ms]	time [ms]
Europe	280	39.2	56.3	3.6	10.5	7.9	14.3	5.8
USA	223	30.1	43.6	4.4	9.8	6.1	13.1	4.1
New Europe	351	44.5	65.2	4.6	15.8	8.5	17.2	8.8

Table VI. Comparison between different variants of path unpacking.

	Europe		USA		New Europe	
	time [ms]	space [MiB]	time [ms]	space [MiB]	time [ms]	space [MiB]
(a) no path data	45.7	140	35.9	213	52.1	257
(b) length only	56.3	203	43.6	312	65.2	403
(c) first edge	56.4	203	43.8	312	65.3	403
(d) complete path	341.7	203	691.3	312	517.9	403
(e) compl. path (fast)	73.1	275	65.6	400	88.7	548

random query has to access 39 blocks in case of the European road network. When the cache has been warmed-up, most blocks (in particular the ones that contain very important nodes) reside in the cache so that on average less than four blocks have to be fetched from external memory. This yields a very good query time of 10.5 ms. Recomputing the optimal path using the same target, but a different source node can be done in 14.3 ms. As expected, the bottleneck of our application are the accesses to the external memory: if all blocks had been preloaded, a shortest-path computation would take only about 5.8 ms instead of the 56.3 ms that include the I/O operations. For comparison, on a PC (our 2 GHz Opteron), the same code runs about 9 times faster (0.64 ms) – this is basically the speed difference between the CPUs. The code for the standard scenario is another four times faster (0.15 ms) – this is the overhead due to the compressed data structure. Using the naive data structure in the mobile scenario would likely result in 1 block access per settled node, resulting in about 7 times larger query time.

Path Unpacking. In Table VI, we compare five different variants of path (not-)unpacking, using the first query type ('cold') in each case. First (a), we store no path data at all. This makes the query very fast since more nodes fit into a single block. However, with this variant, we can only compute the shortest-path length. For all other variants, we also store the middle nodes of the shortcuts in the data blocks. This slows down the query even if we do not use the additional data (b). After having computed the shortest-path length, getting the very first edge of the path (which is useful to generate the very first driving direction) is almost for free (c). Computing the complete path takes considerably longer if we do not use pre-unpacked path data (d). Pre-unpacked paths (e) somewhat increase the memory requirements, but greatly improve the running times. Note that almost half of the pre-unpacked paths are contained in other pre-unpacked paths so that they require no additional space.

Table VII. Query performance of the dynamic mobile scenario depending on the number of edge weight increases (× 10) on motorways. The column 'affected queries' gives the percentage of queries whose shortest-path is affected by the changes. Also, we give the number of average iterations for the cold case.

		search space						
	affected	dynamic CH (dynamic HNR)				cold	recompute	average
|change set|	queries	touched nodes		relaxed edges		[ms]	[ms]	#iterations
1	0.4 %	349	(1 337)	1 190	(9 416)	94.4	23.2	1.0
10	5.7 %	397	(1 546)	1 320	(10 584)	134.1	23.6	1.1
100	40.0 %	1 311	(3 249)	4 130	(19 726)	184.5	30.4	1.4
1 000	83.7 %	6 573	(19 790)	23 459	(95 341)	698.6	74.0	2.7
10 000	97.9 %	70 179	(396 380)	297 539	(1 609 505)	14 871.4	930.5	7.9

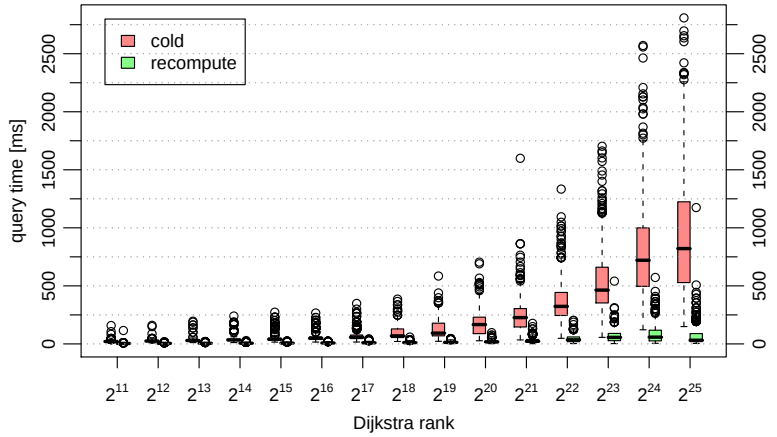


Fig. 10. Local queries after 1 000 changed edges.

7.10 Mobile Dynamic Scenario

We use the same settings as in the mobile scenario. Because our algorithm needs to unpack the shortest path for every query, we decided to use pre-unpacked paths. In Table VII we compare the performance of the mobile dynamic CH to the dynamic HNR. Query times are only given for the CH since there exists no mobile implementation of HNR. We only changed motorways because lower ranked street categories had little to no impact on the query performance. Changing only one edge yields a lower performance compared to the mobile scenario. This is essentially caused by the additional data that is stored in the graph. For a small amount of random changes, the query times scale quite well, but they get out of hand when changing more than 1000 edges. The most important parts of the hierarchy have been distorted by the changes. The dynamic CH show a behaviour similar to dynamic HNR when comparing the search space. They do outperform them in every test though. Figure 10 shows the effect on local queries. Long-distance queries are disproportionately affected: The shortest path consists mostly of important edges and therefore more changes must be taken into account. Because of this our algorithm needs more iterations for them. Furthermore, some queries are especially affected by the changes and need way more time than the average query.

8. APPLICATIONS

Many-to-Many Shortest Paths. Instead of a point-to-point query, the goal of many-to-many routing is to find *all* distances between a given set S of source nodes and set T of target nodes. [Knopp et al. 2007] developed an efficient algorithm based on HHs to compute them. The idea is to perform only $|T|$ backward searches, store the resulting search spaces appropriately and then perform $|S|$ forward searches that use the stored information on the backward searches to find the shortest path distances. This works for any routing technique based on non-goal-directed bidirectional search. CHs are particularly well suited because they have very small search spaces and because for the backward search spaces *we only need to store nodes that are not stalled*.

For our experiments, we do not use the aggressive variant but the method EVSQL from Table I because it shows slightly better performance. In Table VIII we show that CHs are more than two times faster than HNR, for small instances the factor is not as large.

Table VIII. Computing $|S| \times |S|$ distance tables using CHs and HNR. The times for HNR are due to [Schultes 2008] using an older compiler version that generates slightly slower code. All times are given in seconds.

$\|S\|$	100	500	1 000	5 000	10 000	20 000
CH	0.4	0.5	0.6	3.3	10.2	36.6
HNR	0.4	0.8	1.4	8.5	23.2	75.1

Transit-Node Routing. We employ the method of [Geisberger 2008] to use the nodes designated the most important by the node ordering to define the sets of transit nodes. Compared to generous TNR based on HHs by [Schultes 2008], CHs improve preprocessing time from 75 $\rightarrow$ 46 min, query time from 4.3 $\rightarrow$ 3.3 μs and space consumption from 247 $\rightarrow$ 193 B/node. We have not yet implemented a preprocessing completely based on CHs, so that it is too early to judge the whole effect of CHs on preprocessing time but we hope for additional improvements.

Contraction of Other Graph Families. Node contraction works very well on road networks with travel time edge weights, as only few shortcuts need to be added during contraction. We observe that we need to add about as much shortcuts as there are edges in the input graph. However, other graph families may require much more shortcuts. [Bauer et al. 2010b] showed that it is a good idea to stop contraction at some point and solely rely on goal-directed techniques for the *core* of remaining nodes.

9. DISCUSSION

The key features of CHs are their *simple concept* and their *practicability*. The simple query algorithm together with the highly engineered preprocessing form an efficient basis for many hierarchical routing methods in road networks. We have currently the fastest hierarchical, Dijkstra based routing algorithm with preprocessing times of a few *minutes* and query times of a few hundred *microseconds*. Additionally, our algorithm is the fastest implementation for the calculation of large distance tables

and is the preferred hierarchical method to use in combination with goal-direction when low preprocessing and query times are desired. Our mobile implementation is, as far as we know, the first implementation of an *exact* route planning algorithm on a *mobile device* that answers queries in a road network of a whole continent *instantaneously*, i. e., with a delay that is virtually not observable for a human user. Furthermore, our algorithm is *simple* to implement on a mobile device, our graph representation is comparatively *small* (only a few hundred megabytes), and we efficiently handle increases of edge-weights, e. g. caused by traffic jams. These facts suggest an application of our implementation in car navigation systems.

9.1 Ensuing Work

Contraction hierarchies also build the basis for routing algorithms beyond a single static edge weight function. [Batz et al. 2009] successfully adapted CHs to *time-dependent* road networks, where the travel time depends on the departure time. [Geisberger 2009] researched CHs on time-dependent *timetable networks*. And [Geisberger et al. 2010] extend CHs to the *flexible scenario* with two edge weight functions, where the query returns the shortest path for a fixed ratio between both functions. This ratio is fixed separately before each query, but after preprocessing. Flexible edge restrictions for CHs have been researched by [Rice and Tsotras 2010]. A fast algorithm to solve the one-to-all shortest path problem was presented by [Delling et al. 2010]. It processes a CH using a GPU to compute all distances within a few milliseconds. [Abraham et al. 2010a] studied the computation of a alternative routes and also considered CHs for a fast computation. A first attempt grasp the theoretical performance of shortest-path speed-up techniques, including CHs, was published by [Abraham et al. 2010b]. [RG: Anything on labelset-algorithm? How to reference labelset-algorithm without having a techreport from microsoft? It will take some more weeks until they have on.]

REFERENCES

Abraham, I., Delling, D., Goldberg, A. V., and Werneck, R. F. 2010a. Alternative Routes in Road Networks. In *Proceedings of the 9th Symposium on Experimental Algorithms (SEA)*, P. Festa, Ed. Lecture Notes in Computer Science, vol. 6049. Springer, 23–34.

Abraham, I., Fiat, A., Goldberg, A. V., and Werneck, R. F. 2010b. Highway Dimension, Shortest Paths, and Provably Efficient Algorithms. In *Proceedings of the 21st ACM–SIAM Symposium on Discrete Algorithms (SODA)*, M. Charikar, Ed. SIAM, 782–793.

Ahuja, R. K., Magnanti, T. L., and Orlin, J. B. 1993. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall.

Bast, H., Funke, S., Sanders, P., and Schultes, D. 2007. Fast Routing in Road Networks with Transit Nodes. *Science 316,* 5824, 566.

Batz, G. V., Delling, D., Sanders, P., and Vetter, C. 2009. Time-Dependent Contraction

Hierarchies. In *Proceedings of the 11th Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 97–105.

Bauer, R., Columbus, T., Katz, B., Krug, M., and Wagner, D. 2010a. Preprocessing Speed-Up Techniques is Hard. In *Proceedings of the 7th Conference on Algorithms and Complexity (CIAC)*. Lecture Notes in Computer Science, vol. 6078. Springer, 359–370.

Bauer, R. and Delling, D. 2009. SHARC: Fast and Robust Unidirectional Routing. *ACM Journal of Experimental Algorithmics 14,* 2.4 (August), 1–29. Special Section on Selected Papers from ALENEX 2008.

Bauer, R., Delling, D., Sanders, P., Schieferdecker, D., Schultes, D., and Wagner, D. 2010b. Combining Hierarchical and Goal-Directed Speed-Up Techniques for Dijkstra's Algorithm. *ACM Journal of Experimental Algorithmics 15,* 2.3 (January), 1–31. Special Section devoted to WEA'08.

Delling, D., Goldberg, A. V., Nowatzyk, A., and Werneck, R. F. 2010. PHAST: Hardware-Accelerated Shortest Path Trees. Tech. Rep. MSR-TR-2010-125, Microsoft Research.

Delling, D., Sanders, P., Schultes, D., and Wagner, D. 2006. Highway Hierarchies Star. In *9th DIMACS Implementation Challenge – Shortest Paths*, C. Demetrescu, A. V. Goldberg, and D. S. Johnson, Eds.

Delling, D., Sanders, P., Schultes, D., and Wagner, D. 2009. Engineering Route Planning Algorithms. In *Algorithmics of Large and Complex Networks*, J. Lerner, D. Wagner, and K. A. Zweig, Eds. Lecture Notes in Computer Science, vol. 5515. Springer, 117–139.

Delling, D. and Wagner, D. 2007. Landmark-Based Routing in Dynamic Graphs. See Demetrescu [2007], 52–65.

Demetrescu, C., Ed. 2007. *Proceedings of the 6th Workshop on Experimental Algorithms (WEA)*. Lecture Notes in Computer Science, vol. 4525. Springer.

Demetrescu, C., Goldberg, A. V., and Johnson, D. S., Eds. 2009. *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*. DIMACS Book, vol. 74. American Mathematical Society.

Dijkstra, E. W. 1959. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik 1*.

Fu, L., Sun, D., and Rilett, L. R. 2006. Heuristic shortest path algorithms for transportation applications: State of the art. *Computers & Operations Research 33,* 11, 3324–3343.

Geisberger, R. 2008. Contraction Hierarchies. M.S. thesis, Universität Karlsruhe (TH), Fakultät für Informatik. `http://algo2.iti.kit.edu/documents/routeplanning/geisberger_dipl.pdf`.

Geisberger, R. 2009. Contraction of Timetable Networks with Realistic Transfers. Tech. rep., ITI Sanders, Faculty of Informatics, Universität Karlsruhe (TH).

Geisberger, R., Kobitzsch, M., and Sanders, P. 2010. Route Planning with Flexible Objective Functions. In *Proceedings of the 12th Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 124–137.

Geisberger, R., Sanders, P., and Schultes, D. 2008a. Contraction hierarchies source code. `http://algo2.iti.kit.edu/routeplanning.php`.

Geisberger, R., Sanders, P., Schultes, D., and Delling, D. 2008b. Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. In *Proceedings of the 7th Workshop on Experimental Algorithms (WEA)*, C. C. McGeoch, Ed. Lecture Notes in Computer Science, vol. 5038. Springer, 319–333.

Goldberg, A. 2006. personal communication.

Goldberg, A. V. and Harrelson, C. 2005. Computing the Shortest Path: A* Search Meets Graph Theory. In *Proceedings of the 16th ACM–SIAM Symposium on Discrete Algorithms (SODA)*. SIAM, 156–165.

Goldberg, A. V., Kaplan, H., and Werneck, R. F. 2006. Reach for A*: Efficient Point-to-Point Shortest Path Algorithms. In *Proceedings of the 8th Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 129–143.

Goldberg, A. V., Kaplan, H., and Werneck, R. F. 2007. Better Landmarks Within Reach. See Demetrescu [2007], 38–51.

Goldberg, A. V., Kaplan, H., and Werneck, R. F. 2009. Reach for A*: Shortest Path Algorithms with Preprocessing. See Demetrescu et al. [2009], 93–139.

Goldberg, A. V. and Werneck, R. F. 2005. Computing Point-to-Point Shortest Paths from External Memory. In *Proceedings of the 7th Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 26–40.

Gutman, R. J. 2004. Reach-Based Routing: A New Approach to Shortest Path Algorithms Optimized for Road Networks. In *Proceedings of the 6th Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 100–111.

Hilger, M., Köhler, E., Möhring, R. H., and Schilling, H. 2009. Fast Point-to-Point Shortest Path Computations with Arc-Flags. See Demetrescu et al. [2009], 41–72.

Klunder, G. A. and Post, H. N. 2006. The Shortest Path Problem on Large-Scale Real-Road Networks. *Networks 48,* 4, 182–194.

Knopp, S., Sanders, P., Schultes, D., Schulz, F., and Wagner, D. 2007. Computing Many-to-Many Shortest Paths Using Highway Hierarchies. In *Proceedings of the 9th Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 36–45.

Lauther, U. 2004. An Extremely Fast, Exact Algorithm for Finding Shortest Paths in Static Networks with Geographical Background. In *Geoinformation und Mobilität – von der Forschung zur praktischen Anwendung*. Vol. 22. IfGI prints, 219–230.

Maue, J., Sanders, P., and Matijevic, D. 2009. Goal-Directed Shortest-Path Queries Using Precomputed Cluster Distances. *ACM Journal of Experimental Algorithmics 14*, 3.2:1–3.2:27.

Rice, M. and Tsotras, V. J. 2010. Graph indexing of road networks for shortest path queries with label restrictions. *Proc. VLDB Endow. 4*, 69–80.

Sanders, P. and Schultes, D. 2005. Highway Hierarchies Hasten Exact Shortest Path Queries. In *Proceedings of the 13th Annual European Symposium on Algorithms (ESA)*. Lecture Notes in Computer Science, vol. 3669. Springer, 568–579.

Sanders, P. and Schultes, D. 2006. Engineering Highway Hierarchies. In *Proceedings of the 14th Annual European Symposium on Algorithms (ESA)*. Lecture Notes in Computer Science, vol. 4168. Springer, 804–816.

Sanders, P., Schultes, D., and Vetter, C. 2008. Mobile Route Planning. In *Proceedings of the 16th Annual European Symposium on Algorithms (ESA)*. Lecture Notes in Computer Science, vol. 5193. Springer, 732–743.

Schultes, D. 2008. Route Planning in Road Networks. Ph.D. thesis, Universität Karlsruhe (TH), Fakultät für Informatik. `http://algo2.iti.uka.de/schultes/hwy/schultes_diss.pdf`.

Schultes, D. and Sanders, P. 2007. Dynamic Highway-Node Routing. See Demetrescu [2007], 66–79.

Thorup, M. 2004. Compact Oracles for Reachability and Approximate Distances in Planar Digraphs . *Journal of the ACM 51,* 6, 993–1024.

U.S. Census Bureau, Washington, DC. 2002. UA Census 2000 TIGER/Line Files. `http://www.census.gov/geo/www/tiger/tigerua/ua_tgr2k.html`.

Vetter, C. 2010. Monav. `http://code.google.com/p/monav/`.

Volker, L. 2008. Route Planning in Road Networks with Turn Costs. Student Research Project. `http://algo2.iti.uni-karlsruhe.de/documents/routeplanning/volker_sa.pdf`.