Published online ahead of print April 5, 2012

TRANSPORTATION SCIENCE
Articles in Advance, pp. 1–17
ISSN 0041-1655 (print) | ISSN 1526-5447 (online)

http://dx.doi.org/10.1287/trsc.1110.0401
© 2012 INFORMS

Exact Routing in Large Road Networks Using Contraction Hierarchies

Robert Geisberger, Peter Sanders, Dominik Schultes, Christian Vetter
Karlsruhe Institute of Technology, Department of Informatics, 76128 Karlsruhe, Germany
{geisberger@google.com, sanders@kit.edu, mail@dominik-schultes.de, veaac@gmx.de}

Contraction hierarchies are a simple approach for fast 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 in a fraction of a millisecond, 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.

Key words: route planning; shortest paths; algorithm engineering
History: Received: November 2010; revision received August 2011; accepted: August 2011. Published online in *Articles in Advance*.

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 optimization. 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. Because 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. Somewhat surprisingly, this approach also resulted in one of the most efficient approaches available today. Depending on the desired tradeoff between query time, preprocessing time, and space consumption, CHs are usually the best known approach or used as a component of a more complex algorithm (see §6.7). 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, Geisberger, Sanders, and Schultes (2008) released source code as open source, and Vetter (2010) provided a mobile implementation 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*) 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 because of 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: Both forward search from s and backward search from t relax only edges leading to nodes higher up in the hierarchy. 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 input/output operation (I/O) bandwidth and main memory limitations of those small devices.

We also support dynamic edge weight changes. To reestablish a correct CH on a personal computer, we only need to update affected shortcuts, and recontract nodes that are affected by the changes. On a mobile device, we do not recontract, but we establish a correct query by ensuring that the search can reach affected nodes.

1.2. Related Work

Computing shortest paths in road networks is a well-studied problem, e.g., Ahuja, Magnanti, and Orlin (1993). Recently a plethora of faster algorithms (*speedup techniques*) has been developed, and they 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 the reader to Delling et al. (2009). Previous heuristic approaches, e.g., Fu, Sun, and Rilett (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. In addition, research in transportation science has continued partly unaware of the above developments, e.g., Pijls and Post (2009).

CHs, first introduced by Geisberger et al. (2008), 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). 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 and uses the edge difference to estimate the cost of contracting a node v. Goldberg, Kaplan, and Werneck (2007) and 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 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.

An even faster speedup technique is *transit-node routing* (TNR). The basic idea of TNR is to store an all-to-all lookup table for a set of important *transit nodes* ($\Theta(\sqrt{n})$ many) and, for each node, the minimal set of transit nodes (about 10) that are needed to cover all shortest paths touching a transit node. This essentially reduces long distance routing to a set of table lookups (one for each pair of access nodes at source and destination). However, TNR needs considerably higher preprocessing time and space; is less amenable to dynamization; and, most importantly, relies on another hierarchical speedup technique for its preprocessing. We will show that using CHs for this purpose leads to improved performance.

Recently, it was shown that using even more preprocessing time and space, one can directly use CHs to achieve the currently fastest query times (Abraham et al. 2011). Essentially, this is an extreme case of the many-to-many technique described in §7. The *hublabeling* method explicitly stores the search spaces from all nodes and intersects them for a query.

Finally, there is an entirely different family of speedup techniques based on goal-directed routing. In particular, *ALT* (A^*, *L*andmarks, *T*riangle inequality) by Goldberg and Harrelson (2005) yields strong lower bounds that can direct the search toward the target. It has fast preprocessing but considerable space requirements. *Arc flags* (AF), first introduced by 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). 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 denser 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, Schultes, and Vetter (2008) and is more than one magnitude faster and drastically more space efficient. Also a mobile implementation of the RE algorithm by Goldberg[1] yields 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.

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*, e.g., Winter (2002), 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.

1.3. Outline

In §2 we describe CHs in detail and explain the preprocessing (§§2.1, 2.3) and the query algorithm (§2.2). Our adaptations for mobile devices are in §3. The refined dynamization technique is described in §4 and a variant suitable for the mobile scenario in §5. Section 6 summarizes experiments regarding different variants of CHs and comparisons with other techniques. A few applications of CHs are introduced in §7. The conclusion in §8 summarizes the results and outlines further routes of study. Some important implementation details have been moved to the appendix.

2. Contraction Hierarchies

Before proceeding with the description of contraction hierarchies, we recall that *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* $\delta_s(u) = c(P)$ is known. Nodes that are not reached are *unreached*. *Relaxing* an edge (u, v) means checking whether the path $s - u - v$ improves the tentative distance $\delta_s(u)$.

As introduced in §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 from 1 to n, where $n := |V|$, 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 there must exist a shortest path different from the path $\langle v, u, w\rangle$ via the contracted node. We call such a path a *witness path*. The concept of witness paths is particularly important for the dynamization described in §§4 and 5. Figure 1 shows a completed CH.

2.1. Node Contraction

The most important part of the node contraction is to find witness paths. A simple way to decide whether $\langle v, u, w\rangle$ is the only shortest path is to perform for each node $v \in S$ a forward shortest-path search only using nodes not yet contracted until all nodes in $T\setminus\{v\}$ are settled. Such a search to compute shortest path distances between the neighbors 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. To achieve fast preprocessing we rely on the assumption that local searches are fast because they visit only a tiny fraction of the network. However, this assumption fails when long-distance edges like ferry connections are involved.

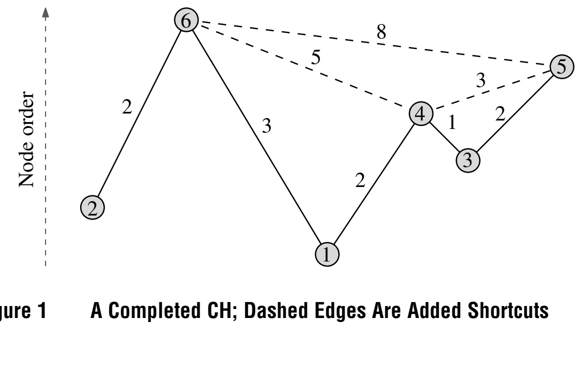

Figure 1 A Completed CH; Dashed Edges Are Added Shortcuts

[1] Goldberg, A (2008). Personal Communication, September 1.

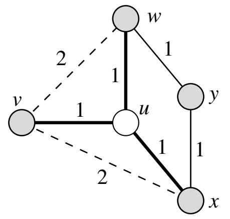

Figure 2 There Is No Witness Path Between the Node Pairs v, w, and v, x so We Need to Add Shortcuts (Dashed) for the Contraction of Node u. The Witness Path $\langle w, y, x\rangle$ Allows us to Omit a Shortcut Between the Node Pair w, x.

We therefore additionally truncate local searches that become too large. Note that this preserves shortest path distances because we only introduce some superfluous shortcuts. Because 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 speed up 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. Geisberger (2008) describes further algorithmic details on how to efficiently implement hop limits.

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 neighbors $x > u$ 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.

2.2. Query

In this section we will introduce our query algorithm and prove its correctness. Recall that we find a shortest path from source s to target t using a variant of the bidirectional version of Dijkstra's algorithm. 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 $G_\downarrow$. The search is stopped when both queues are empty or the smallest priority queue entry exceeds the length of the best path already found. This upper bound is updated whenever the search spaces meet; i.e., both $d_\uparrow[u]$ and $d_\downarrow[u]$ have finite values.

Algorithm 1 Query(s, t)

```
1  $d_\uparrow := \langle\infty, \dots, \infty\rangle$; $d_\uparrow[s] := 0$; $d_\downarrow := \langle\infty, \dots, \infty\rangle$;
      $d_\downarrow[t] := 0$, $d := \infty$;          //tentative distances
2  $Q_\uparrow = \{(0, s)\}$; $Q_\downarrow = \{(0, t)\}$; $r := \uparrow$;
                                          //priority queues
3  while ($Q_\uparrow \neq \varnothing$ or $Q_\downarrow \neq \varnothing$) and
      ($d > \min\{\min Q_\uparrow, \min Q_\downarrow\}$) do
4     if $Q_{\neg r} \neq \varnothing$ then $r := \neg r$;
        //interleave direction, $\neg\uparrow=\downarrow$ and $\neg\downarrow=\uparrow$
5     $(\cdot, u) := Q_r$.deleteMin( );
        $d := \min\{d, d_\uparrow[u] + d_\downarrow[u]\}$;
                      //$u$ is settled and new candidate
6     foreach $e = (u, v) \in E^*$ do
7        if $r(e)$ and $(d_r[u] + c(e) < d_r[v])$ then
                                     //shorter path found
8           $d_r[v] := d_r[u] + c(e)$;
                              //update tentative distance
9           $Q_r$.update($d_r[v]$, $v$);
                                //update priority queue
10 return $d$
```

Theorem 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 because 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 that are *up-down paths* $\langle s = u_0, u_1, \dots, u_p, \dots, u_q = t\rangle$ 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 exists one that is an up-down path.

Give a shortest s-t path $P = \langle s = u_0, u_1, \dots, u_p, \dots, u_q = t\rangle$ 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. $M_P \neq \varnothing$ iff P is not an up-down path, 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

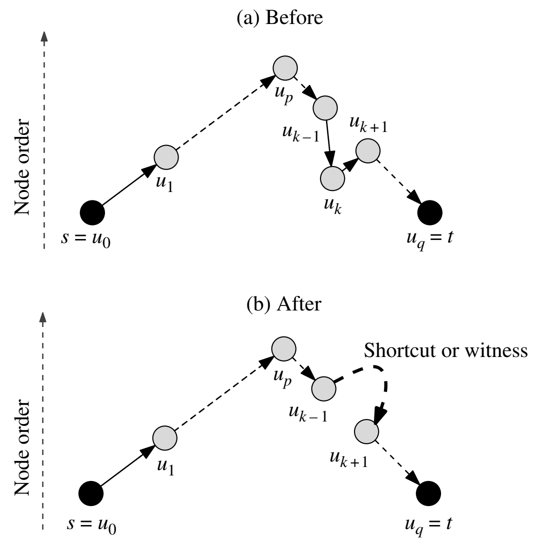

Figure 3 Step of the Correctness Proof to Construct an Up-Down Path

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 = because 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}); see Figure 3(b). The resulting path P' is still a shortest s-t path and $\min M_{P'} > u_k$ or $M_{P'} = \varnothing$. Because $n < \infty$, there must exist a shortest s-t path P'' with $M_{P''} = \varnothing$. Therefore, P'' is the sought after up-down path. □

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). Storing only the *middle node* u with the shortcut (v, w) requires less space but increases the runtime.

Pruning the Query Search Space. Using the *stall-on-demand* technique by 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 because the computed distance to u is *suboptimal* and we do not relax the edges of u. Moreover, stalling can propagate to additional nodes w in the neighborhood of u, if the path via v to w is shorter than $d_{\uparrow}[w]$. We perform a breath first search 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.

2.3. 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 size of the query search space or the number of shortcuts 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. An exhaustive description of the priority terms is given by Geisberger (2008). 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 neighbors of the contracted node.

Lazy Updates. In general, more than the neighbors are affected, so we update the priority of the top node on the queue before we remove it (*lazy update*). Because 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 occurs 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 important priority term, and we calculate it with a *simulated* contraction of node u.

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 logarithmic time. More generally, it seems to be a good idea to contract nodes everywhere in the graph in a uniform way rather than to 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 Neighbors: Every node has a counter that gets incremented when a neighbor is contracted. 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 $R(v) := \{u \text{ contracted} \mid d(v,u) < \infty, \forall \text{ uncontracted } w\colon\ d(v,u) \le d(w,u)\}$ be the Voronoi-region of an 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 neighboring Voronoi regions will "eat up" $R(v)$.

Cost of Contraction. The most time consuming part of the contraction are the local searches for witness paths. Because their durations vary from node to node, we want to contract "expensive" nodes later in a smaller remaining graph. So we include the number of settled nodes during the local search as a priority term.

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

Generally speaking, one can come up with many heuristic terms, but one 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.

3. Mobile Scenario

Because of the simple query algorithm and the small search space, see §6, 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. We will sketch the modifications here and refer to Sanders, Schultes, and Vetter (2008) for details on algorithms and data structures.

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. Therefore, we need to find a node numbering that reflects locality. We achieve this by combining a topological numbering using depth-first search and a hierarchical renumbering using the CH node order.

Blockwise Representation. Our graph data structure stores per block a subset of nodes together with their incident edges. We distinguish between *internal* edges leading to a node within the same block and the remaining *external* edges. That way, we require less space to store internal edges. Within each block we use the minimal number of bits to store the node and edge attributes.

Storing the Graph Representation. The blocks representing the graph are stored in external memory. In main memory, we manage a cache that can hold a subset of the blocks.

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, because 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 §2.2). We distinguish between internal and external shortcuts, where the middle node is stored in the same block as the shortcut or not. Expanding an external shortcut might require an additional block read. To accelerate the path unpacking, we refine the approach of Delling et al. (2009) to store explicit descriptions of the paths underlying the external shortcuts. A new feature is that we exploit that a shortcut can contain other shortcuts, and we do not need to store its path description again.

4. 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.

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 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 $V\backslash U$ is unnecessary because no new shortcuts have to be added.

Updating Existing Shortcuts. For each changed edge (u, w), we find all shortcuts containing (u, w) to delete them or adjust their weight. Then, 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 in the search graph; see §2.2. Because $u < w$, (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 *depth-first search* 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 $z < y$ (Lines 3–4) or stored in reverse direction with y in case $y < z$ (Lines 5–6); see Figure 4 for an example with $r = \uparrow$.

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 neighbor 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 because they do not invalidate the correctness of the CH, we keep them although they can slow down the query somewhat.

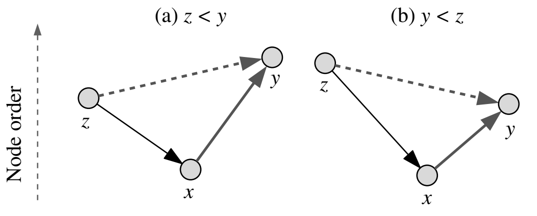

Figure 4 The Shortcut (z, y) Containing Edge (x, y) Is Either Stored (a) at z or (b) in Reverse Direction at y

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 changes, 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 because of 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 because of the changes. If the sum does not exceed $\delta_{\min}$, u has certainly not become a seed node because of lengthened 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; and few new shortcuts have to be added.

5. 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 that affect the queries.

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 update 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.

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 usable 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.

Data Structures: The additional data for each edge, witness data and middle node, is directly stored with the associated edge. We use additional volatile data structures next to our read-only graph to store the changed edges, affected shortcuts, and edges inserted in G^*.

6. 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.

6.1. Experimental Setting

Environment. Our experiments were done on one core of a single AMD Opteron Processor 270 clocked at 2.0 GHz with 8 GB main memory and 2×1 MB 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 MB 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 MB/s though the device limits this to 8 MB/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 optimization 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.

6.2. 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.

6.3. Additional Instances

In addition, we also performed some experiments on two other road networks. A publicly available version of the continental U.S. road network, which has 23,947,347 nodes and 57,708,624 directed edges, was obtained from the TIGER/Line Files. These were provided by the U.S. Census Bureau (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 of 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."

6.4. 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 $rk_s(t) = r$. 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, and 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.

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 neighbor 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.

6.5. 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 (§2.3) and the local search limits for the contraction (§2.1). For our *aggressive variant* we select parameters to minimize the query time and for our *economical variant*, to minimize the product of query time and preprocessing time. Additionally to updating the neighbors, we always use lazy updates because 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 5 shows the development of the average degree during node

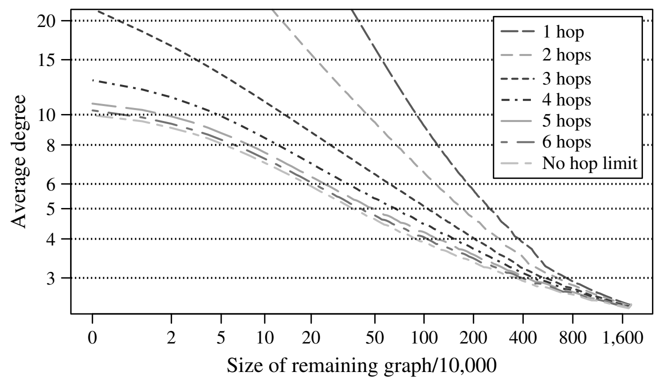

Figure 5 **Average Degree Development for Different Hop Limits on *Europe***

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.

6.6. Standard Scenario

We start with evolving sets of priority terms and search space limits to get a deeper insight into them; see Table 1. The best performance in every column is bold. 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 because of the additional local searches during the node ordering, particularly the initialization of the priority queue, which 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 neighbors* counter (letter D) accelerates the query time, whose 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 because 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 neighbors term. To significantly decrease the preprocessing time, we introduce a *hop limit* of five to the local searches, leading to a two times better node ordering time. Applying *staged hop limits* (digits 1235) shrinks the preprocessing time below 10 minutes. Replacing the *deleted neighbors* counter by the *original edges* term (letter O) further improve preprocessing, query, and space overhead (Line EOS1235, priority function $= 190 \cdot \mathrm{E} + 600 \cdot \mathrm{O} + \mathrm{S}$). This is our our *economical variant*. Its low preprocessing time of 7.5 minutes and the fast query of about 200 μs provide the best balance. To further decrease the query time, we first exchange the current uniformity term to take the square root of *Voronoi region* sizes (letter V) into account. We combine that with the *original edges* term (letter O) and add a priority term to estimate the *cost of queries* (letter H) (Line EVOSHL, priority function $= 190 \cdot \mathrm{E} + 60 \cdot \mathrm{V} + 70 \cdot \mathrm{O} + \mathrm{S} + 145 \cdot \mathrm{H}$). This leads to our *aggressive variant*, having 29% faster query time then the economical variant and a speedup of about 40,000 compared to 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 §3). Because 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 because 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. Because random queries are unrealistic for large graphs, Figure 6 shows the distributions of query times for various degrees of locality. We see good query performance over all Dijkstra

Table 1 **Performance of Various Node Ordering Heuristics**

Method	Node ordering [s]	Hierarchy construction [s]	Query [μs]	Nodes settled	Nonstalled 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
EDSL	1,414	165	175	399	228	1,335	−1.6
EOSL	1,110	145	222	531	313	1,802	**−3.0**
ECO.							
EDS5	652	99	213	462	256	1,651	−1.1
EDS1235[a]	545	57	223	459	234	1,638	1.6
EOS1235[a]	**451**	**48**	214	487	275	1,684	0.6
AGGR.							
EDSHL	1,648	199	173	385	220	1,378	−1.1
EVSHL	1,627	170	159	368	209	1,181	−1.7
EVOSHL	1,644	165	**152**	**356**	**207**	**1,163**	−2.1

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

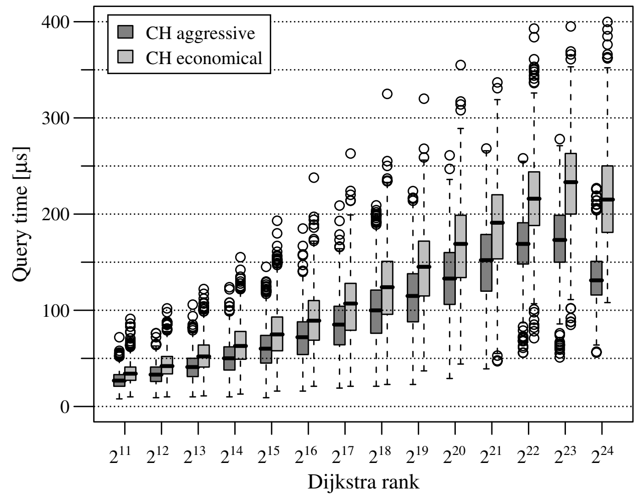

Figure 6 **Performance of Queries, Where the Target Is Chosen by the Dijkstra Rank from the Source**

ranks and small fluctuations. This is further underlined in Figure 7 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 2.4 times the size of the average actual search space sizes (see also Table 1).

6.7. 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. There are many *Pareto-optimal* techniques x; i.e., where for each other technique y, there is a dimension for which x is better than y. However, CHs take a particularly strong position. We compare the most successful speedup techniques in Table 2. A technique is dominated by CHs if CH is better in every dimension. The best performance in every column is bold. For further algorithms, we refer to Delling et al. (2009) and 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 for 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 is TNR, whose precomputation also relies on the CH node order for best results (see §7). The combination of TNR + AF currently provides the best query time.

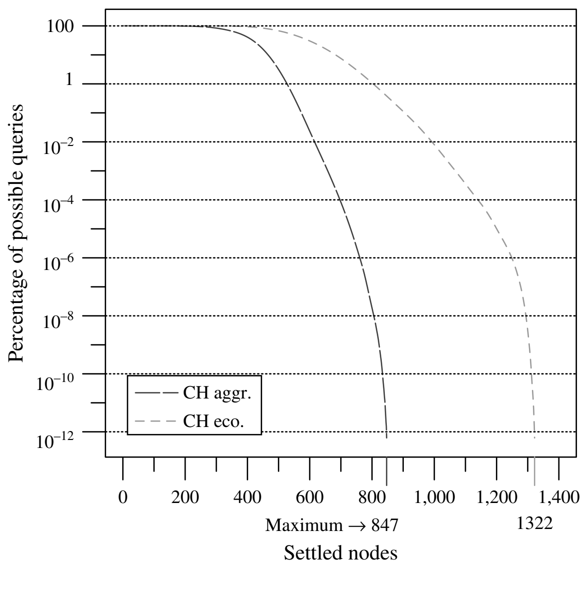

Figure 7 **Upper Bound for the Worst Percentages of Queries**

6.8. 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 (see Table 3). We can expect additional improvements if we were to repeat the parameter search; see §6.5. The *USA* (Tiger) graph shows slightly larger preprocessing times than the *Europe* graph but faster query times. The *New Europe* graph is a larger network thus requiring more preprocessing time. 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 because of availability.

6.9. 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, whereas 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 edges.

In the following we use a block size of 4 KB, which was found using experiments with block sizes from

Table 2 **Comparison of Various Speedup Techniques in the Three-Dimensional Space of Preprocessing Time, Space Overhead, and Query Time**

		Preprocessing		Query			
Method	Data from	Time [min]	Overh. [B/n.]	Settled nodes	Time [ms]	Uses CH	Dominated by CH
Dijkstra[a]	Bauer et al. (2010b)	0	0	9,114,385	5,591.6		
Bidir. Dijkstra[a]	Bauer et al. (2010b)	0	0	4,764,110	2,713.2		
Economical CH	This paper	**8**	0.6	487	0.21	✓	
Aggressive CH	This paper	27	−2.1	356	0.15	✓	
ALT, 16 landmarks[b]	Goldberg, Kaplan, and Werneck (2009)	13	70	82,348	120.1		✓
ALT, 64 landmarks[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, Kaplan, and Werneck (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**	965,018	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]2.6 GHz AMD Opteron, SuSE Linux 10.3, 16 GiB of RAM, 2×1 MiB of L2 cache.
[b]2.4 GHz AMD Opteron, Windows Server 2003, 16 GiB of RAM, 2 MiB of L2 cache.
[c]2.2 GHz AMD Opteron, SuSE Linux 9.1, 4 GiB of RAM, 1 MiB of L2 cache.
[d]Dominated by CH + AF of Bauer et al. (2010b).
[e]Uses CHs to compute transit nodes.

1 KB to 64 KB. This block size is optimal with respect to both space consumption and query time. We use a cache size of 64 MB. Additional experiments indicate that reducing it to 32 MB has negligible effect on the performance of warm queries. Even only 256 KB of cache are sufficient to achieve the performance of our cold queries.

Table 4 gives an overview of the external-memory graph representation. The columns 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 percentages 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

Table 3 **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
Query [μs]	152	214	96	180	213	303	1,940	2,276	645	1,857
Nodes settled	356	487	283	526	439	629	1,582	2,216	1,081	3,461
Nonstalled nodes	207	275	157	309	247	351	658	962	485	2,100
Edges relaxed	1,163	1,684	885	1,845	1,732	2,600	15,472	19,227	7,905	27,755
Space overh. [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
Query [μs]	170	238	107	198	243	345	2,206	2,615	721	2,121
Expand path [μs]	323	321	1,105	1,107	972	953	798	792	1,268	1,336
Space overh. [B/node]	6.2	10.3	5.8	7.8	5.6	8.5	10.2	11.7	7.4	8.3
Edges	21	23	21	26	21	24	21	29	22	40
Edges expanded	1,370	1,369	4,548	4,548	4,139	4,136	3,291	3,291	5,128	5,128

Table 4 Building the Graph Representation

	$\lvert V\rvert$ [$\times 10^6$]	$\lvert E\rvert$ [$\times 10^6$]	$\lvert E^*\rvert$ [$\times 10^6$]	#Blocks	#Adj. blocks	Int. edges (%)	Int. shcs. (%)	Ext. shcs. (%)	Time [s]	Space [MB]
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

the search graph is already given), and the total memory consumption including pre-unpacked paths. Building the blocks is very fast and can be done in about two to four 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 the 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 5. On average, a random query has to access 39 blocks in case of the *Europe* 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 is the access 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 nine times faster (0.64 ms)—this is basically the speed difference between the CPUs. The code for the standard scenario (§6.6) is another four times faster (0.15 ms)—this is the overhead because of the compressed data structure. Using the naive data structure in the mobile scenario would likely result in one block access per settled node, resulting in about a seven times larger query time.

Path Unpacking. In Table 6, 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 because 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.

6.10. 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 7 we assess the performance of mobile dynamic CHs. We can restrict ourselves to a comparison with dynamic HNR because Schultes and Sanders (2007) showed that dynamic HNR is better than all previous dynamic speedup techniques. Query times are only given for CHs because 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 1,000 edges. The most important parts

Table 5 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 6 Comparison Between Different Variants of Path Unpacking

	Europe		*USA*		*New Europe*	
	Time [ms]	Space [MB]	Time [ms]	Space [MB]	Time [ms]	Space [MB]
(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

Table 7 Query Performance of the Dynamic Mobile Scenario Depending on the Number of Edge Weight Increases ($\times$10) on Motorways

|Change set|	Affected queries (%)	Search space dynamic CH (dynamic HNR): Touched nodes	Relaxed edges	Cold [ms]	Recompute [ms]	Average #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

of the hierarchy have been distorted by the changes. Dynamic CHs show a behavior similar to dynamic HNR when comparing the search space. They do outperform them in every test though. Figure 8 shows the effect on local queries. 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. 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 substantially more time than the average query.

7. 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. Replacing HHs by CHs improves the performance, as Table 8 shows. For our experiments, we do not use the aggressive variant but the method EVSHL from Table 1 because it displays slightly better performance.

Table 8 Time in Seconds Required to Compute a $|S| \times |S|$ Distance Tables. The Times for HH and HNR Are from Schultes (2008). The Times for Dijkstra's Algorithm Are Extrapolated from Table 2.

$\|S\|$	100	500	1,000	5,000	10,000	20,000
Dijkstra	559.1	2,796.5	5,591.0	27,955.0	55,910.0	111,820.0
HH	0.6	1.7	3.3	26.3	76.6	247.7
HNR	0.4	0.8	1.4	8.5	23.2	75.1
CH	0.4	0.5	0.6	3.3	10.2	36.6

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

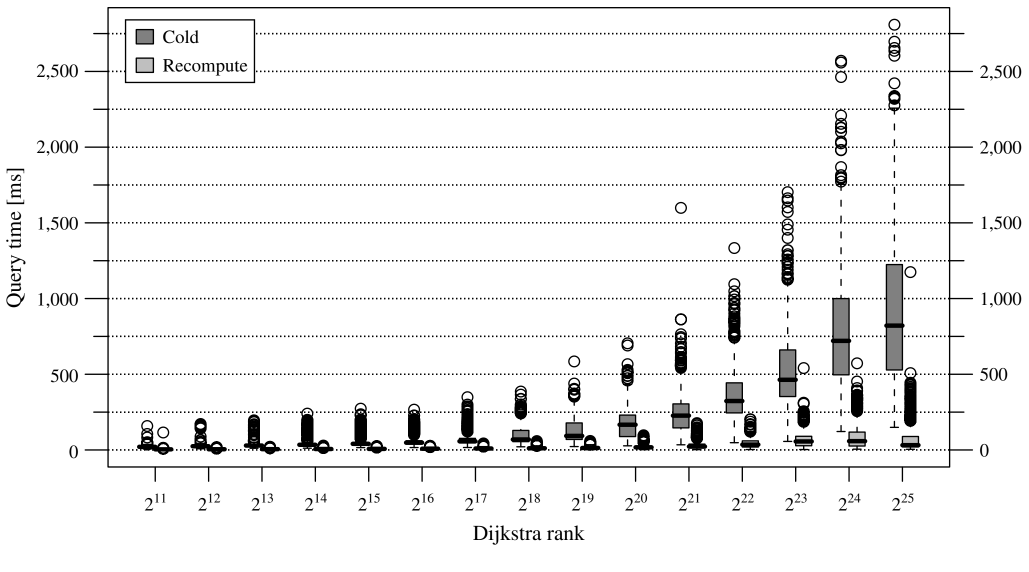

Figure 8 Local Queries After 1,000 Changed Edges

$4.3 \rightarrow 3.3$ μs and space consumption from $247 \rightarrow 193$ B/node. We have not yet implemented a preprocessing completely based on CHs, so it is too early to judge the whole effect of CHs on preprocessing time, but we hope for additional improvements.

8. Conclusion

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.

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. Vetter (2009) provides a parallel version, and Kieritz et al. (2010) a distributed version. Geisberger (2010) researched CHs on time-dependent *timetable networks*. And Geisberger, Kobitzsch, and Sanders (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 alternative routes and used CHs for a fast computation. A first attempt to grasp the theoretical performance of shortest path speedup techniques, including CHs, was published by Abraham et al. (2010b).

Acknowledgments
In part, the idea to use only node contraction for preprocessing was developed together with Daniel Delling. Partially supported by DFG grant SA 933/5-1.

Appendix A. Fast Local Search

The local search with hop limit described in §2.1 can be accelerated by certain measures.

Fast Local One-Hop Search. To find witness paths consisting of just one edge, it is sufficient to scan through all outgoing edges of the source node $v \in S$. The one-hop search makes sense if lots of edges of the graph are shortest paths, as 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 Two-Hop Search. We implement a simple variant of the many-to-many shortest paths algorithm of Knopp et al. (2007). We associate a bucket $b(x) := \{(w, c(x,w)) \mid w \in T, (x,w) \in E\}$ with each node $x > u$. Computing the nonempty $b(x)$ 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 $b(x)$.

One-Hop Backward Search. To speed up a local search from $v \in S$ with hop limit $a \geq 3$, we first perform a Dijkstra algorithm with $(a-1)$ hop limit resulting in distances $\delta_v(\cdot)$. Then 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, and we now stop the search if the last settled node exceeds the distance $c(v,u) + \max\{c(u,w) - \min\{c(x,w) \mid (x,w) \in E\} \mid (u,w) \in E, w \neq v\}$.

Appendix B. Node Order Selection

More details on the effect of the priority terms introduced in §2.3 are given below.

Edge Difference. For our implementation of the edge difference, 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 B.1: The contraction of node v will require a 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 neighbors of u are affected the most because 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 neighbor of u, v may be extracted from the priority queue in a different order than desired. Lemma 1 shows that under certain conditions, lazy updates can reestablish the correct order.

Lemma 1. *We contract all nodes in correct order if* (a) *after the contraction of a node, all of its neighbors are updated,* (b) *the local searches for witness paths are unlimited,* (c) *the edge difference is the the only priority term and has a nonnegative coefficient, and* (d) *lazy updates are used.*

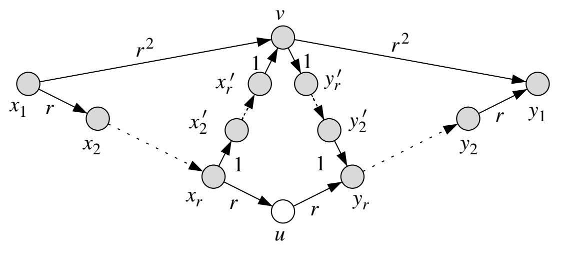

Figure B.1 After the Contraction of Node u the Edge Difference of Node v in this Directed, Weighted Graph Changes

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 neighbors of u. These changes are covered by (a). Therefore, only existing witness paths can affect a node v that is not a neighbor 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 1 to our preprocessing because we have search limits. However, if our limits are sufficiently large, we can expect only few deviations from the projected node order.

Uniformity Deleted Neighbors: This quantity can be maintained correctly by either lazy update or by updating the neighbors of a contracted node.

Voronoi Regions: When v is contracted, its neighboring Voronoi regions will "eat up" its Voronoi region $R(v)$. Maue, Sanders, and Matijevic (2006) 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. Because they can only grow, lazy updates ensure that the priority queue works correctly w.r.t. this term of the priority function.

Cost of Contraction. For Dijkstra searches, we include the number of settled nodes as priority term, for the fast local one-hop search we use the number of scanned edges, and for the fast local two-hop search we use the number of bucket entries plus the number of scanned edges during the one-hop forward search. Perfectly updating the cost of contraction would be difficult because the contraction of any node in a search tree of the local search can affect it.

Lemma 2 extends Lemma 1 to additional priority terms; we omit the proof.

Lemma 2. *Lemma* 1 *holds if additionally the uniformity terms and the cost of queries term are used with nonnegative coefficients*.

Appendix C. Mobile Dynamic 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 because 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. Data structure 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 realize these data structures we employ hash maps. Because the edges inserted in G^* are specific to the query, it pays off to clear Δ_2 afterward if the next query will be substantially different.

References

Abraham, I., D. Delling, A. V. Goldberg, R. F. Werneck. 2010a. Alternative routes in road networks. P. Festa, ed. *Proc. 9th Sympos. Experiment. Algorithms* (*SEA*), Vol. 6049, Lecture Notes Comput. Sci., Springer, Berlin/Heidelberg, 23–34.

Abraham, I., D. Delling, A. V. Goldberg, R. F. Werneck. 2011. A hub-based labeling algorithm for shortest paths in road networks. P. M. Pardalos, S. Rebennack, eds. *Proc. 10th Sympos. Experiment. Algorithms* (*SEA*), Vol. 6630, Lecture Notes Comput. Sci., Springer, Berlin/Heidelberg, 230–241.

Abraham, I., A. Fiat, A. V. Goldberg, R. F. Werneck. 2010b. Highway dimension, shortest paths, and provably efficient algorithms. M. Charikar, ed. *Proc. 21st ACM–SIAM Sympos. Discrete Algorithms* (*SODA*). SIAM, Philadelphia, 782–793.

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

Batz, G. V., D. Delling, P. Sanders, C. Vetter. 2009. Time-dependent contraction hierarchies. *Proc. 11th Workshop on Algorithm Engrg. Experiments* (*ALENEX*). SIAM, Philadelphia, 97–105.

Bauer, R., D. Delling. 2009. SHARC: Fast and robust unidirectional routing. *ACM J. Experiment. Algorithmics* **14**(December) Article 4.

Bauer, R., T. Columbus, B. Katz, M. Krug, D. Wagner. 2010a. Preprocessing speed-up techniques is hard. *Proc. 7th Conf. Algorithms and Complexity* (*CIAC*), Vol. 6078, Lecture Notes Comput. Sci., Springer, Berlin/Heidelberg, 359–370.

Bauer, R., D. Delling, P. Sanders, D. Schieferdecker, D. Schultes, D. Wagner. 2010b. Combining hierarchical and goal-directed speed-up techniques for Dijkstra's algorithm. *ACM J. Experiment. Algorithmics* **15**(March) Article 2.3.

Delling, D., D. Wagner. 2007. Landmark-based routing in dynamic graphs. C. Demetrescu, ed. *Proc. 6th Workshop Experiment. Algorithms* (*WEA*), Vol. 4525, Lecture Notes Comput. Sci., Springer, Berlin/Heidelberg, 52–65.

Delling, D., A. V. Goldberg, A. Nowatzyk, R. F. Werneck. 2010. PHAST: Hardware-accelerated shortest path trees. Technical report MSR-TR-2010-125, Microsoft Research, Palo Alto, CA.

Delling, D., P. Sanders, D. Schultes, D. Wagner. 2009. Engineering route planning algorithms. J. Lerner, D. Wagner, K. A. Zweig, eds. *Algorithmics of Large and Complex Networks*, Springer-Verlag, Berlin/Heidelberg, 117–139.

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

Fu, L., D. Sun, L. R. Rilett. 2006. Heuristic shortest path algorithms for transportation applications: State of the art. *Comput. Oper. Res.* **33**(11) 3324–3343.

Geisberger, R. 2008. Contraction hierarchies. Master's thesis, Universität Karlsruhe (TH), Fakultät für Informatik, http://algo2.iti.kit.edu/documents/routeplanning/geisberger_dipl.pdf.

Geisberger, R. 2010. Contraction of timetable networks with realistic transfers. P. Festa, ed. *Proc. 9th Sympos. Experiment. Algorithms* (*SEA*), Vol. 6049, Lecture Notes Comput. Sci., Springer, Berlin/Heidelberg, 71–82.

Geisberger, R., M. Kobitzsch, P. Sanders. 2010. Route planning with flexible objective functions. *Proc. 12th Workshop on Algorithm Engrg. Experiments* (*ALENEX*). SIAM, Philadelphia, 124–137.

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

Geisberger, R., P. Sanders, D. Schultes, D. Delling. 2008. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. C. C. McGeoch, ed. *Proc. 7th Workshop Experiment. Algorithms* (*WEA*), Vol. 5038, Lecture Notes Comput. Sci., Springer, Berlin/Heidelberg, 319–333.

Goldberg, A. V., C. Harrelson. 2005. Computing the shortest path: A^* search meets graph theory. *Proc. 16th ACM–SIAM Sympos. Discrete Algorithms* (*SODA*). SIAM, Philadelphia, 156–165.

Goldberg, A. V., R. F. Werneck. 2005. Computing point-to-point shortest paths from external memory. *Proc. 7th Workshop on Algorithm Engrg. Experiments* (*ALENEX*). SIAM, Philadelphia, 26–40.

Goldberg, A. V., H. Kaplan, R. F. Werneck. 2007. Better landmarks within reach. C. Demetrescu, ed. *Proc. 6th Workshop Experiment. Algorithms* (*WEA*), Vol. 4525, Lecture Notes Comput. Sci., Springer-Verlag, Berlin/Heidelberg, 38–51.

Goldberg, A. V., H. Kaplan, R. F. Werneck. 2009. Reach for A^*: Shortest path algorithms with preprocessing. C. Demetrescu, A. V. Goldberg, D. S. Johnson, eds. *The Shortest Path Problem*: *Ninth DIMACS Implementation Challenge*, Vol. 74, DIMACS: Series in Discrete Mathematics and Theoretical Computer Science, American Mathematical Society, Providence, RI, 93–139.

Hilger, M., E. Köhler, R. H. Möhring, H. Schilling. 2009. Fast point-to-point shortest path computations with arc-flags. C. Demetrescu, A. V. Goldberg, D. S. Johnson, eds. *The Shortest Path Problem*: *Ninth DIMACS Implementation Challenge*, Vol. 74, DIMACS: Series in Discrete Mathematics and Theoretical Computer Science, American Mathematical Society, Providence, RI, 41–72.

Kieritz, T., D. Luxen, P. Sanders, C. Vetter. 2010. Distributed time-dependent contraction hierarchies. P. Festa, ed. *Proc. 9th sympos. Experiment. Algorithms* (*SEA*), Vol. 6049, Lecture Notes Comput. Sci., Springer, Berlin/Heidelberg, 83–93.

Klunder, G. A., H. N. Post. 2006. The shortest path problem on large-scale real-road networks. *Networks* **48**(4) 182–194.

Knopp, S., P. Sanders, D. Schultes, F. Schulz, D. Wagner. 2007. Computing many-to-many shortest paths using highway hierarchies. *Proc. 9th Workshop on Algorithm Engrg. Experiments* (*ALENEX*). SIAM, Philadelphia, 36–45.

Lauther, U. 2004. An extremely fast, exact algorithm for finding shortest paths in static networks with geographical background. *Geoinformation und Mobilität—von Der Forschung Zur Praktischen Anwendung*, Vol. 22. IfGI Prints, Münster, 219–230.

Maue, J., P. Sanders, D. Matijevic. 2006. Goal directed shortest path queries using precomputed cluster distances. *Proc. 5th Workshop on Experiment. Algorithms* (*WEA*), Vol. 4007, Lecture Notes Comput. Sci., Springer, Berlin/Heidelberg, 316–328.

Pijls, W., H. N. Post. 2009. A new bidirectional search algorithm with shortened postprocessing. *Eur. J. Oper. Res.* **198**(2) 363–369.

Rice, M., V. J. Tsotras. 2010. Graph indexing of road networks for shortest path queries with label restrictions. *Proc. VLDB Endowment* **4**(2) 69–80.

Sanders, P., D. Schultes. 2005. Highway hierarchies hasten exact shortest path queries. *Proc. 13th Annual Eur. Sympos. Algorithms* (*ESA*), Vol. 3669, Lecture Notes Comput. Sci., Springer, Berlin/Heidelberg, 568–579.

Sanders, P., D. Schultes, C. Vetter. 2008. Mobile route planning. *Proc. 16th Annual Eur. Sympos. Algorithms* (*ESA*), Vol. 5193, Lecture Notes Comput. Sci., Springer, Berlin/Heidelberg, 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., P. Sanders. 2007. Dynamic highway-node routing. C. Demetrescu, ed. *Proc. 6th Workshop Experiment. Algorithms* (*WEA*), Vol. 4525, Lecture Notes Comput. Sci., Springer, Berlin/Heidelberg, 66–79.

U.S. Census Bureau. 2002. U.S. Census 2000 TIGER/Line Files. Washington DC. Accessed March 1, 2005, http://www.census.gov/geo/www/tiger/tigerua/ua_tgr2k.html.

Vetter, C. 2009. Parallel time-dependent contraction hierarchies. Student research project, Universität Karlsruhe, Fakultät für Informatik, http://algo2.iti.kit.edu/download/vetter_sa.pdf.

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

Winter, S. 2002. Modeling costs of turns in route planning. *GeoInformatica* **6**(4) 345–361.