Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks*

Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling

Universität Karlsruhe (TH), 76128 Karlsruhe, Germany
{robert.geisberger,sanders,schultes,delling}@ira.uka.de

Abstract. We present a route planning technique solely based on the concept of node *contraction*. The nodes are first ordered by 'importance'. A hierarchy is then generated by iteratively *contracting* the least important node. Contracting a node v means replacing shortest paths going through v by *shortcuts*. We obtain a hierarchical query algorithm using bidirectional shortest-path search. The forward search uses only edges leading to more important nodes and the backward search uses only edges coming from more important nodes. For fastest routes in road networks, the graph remains very sparse throughout the contraction process using rather simple heuristics for ordering the nodes. We have five times lower query times than the best previous hierarchical Dijkstra-based speedup techniques and a *negative* space overhead, i.e., the data structure for distance computation needs *less* space than the input graph. CHs can be combined with many other route planning techniques, leading to improved performance for many-to-many routing, transit-node routing, goal-directed routing or mobile and dynamic scenarios.

1 Introduction

Planning optimal routes in road networks has recently attracted considerable interest in algorithm engineering because it is an important application that admits a lot of interesting algorithmic approaches. Many of these techniques exploit the *hierarchical* nature of road networks in some way or another.

Here we present a very simple approach to hierarchical routing. Assume the nodes of a weighted directed graph $G = (V, E)$ are numbered $1..n$ in order of ascending 'importance'. We now construct a hierarchy by *contracting* the nodes in this order. A node v is contracted by removing it from the network in such a way that shortest paths in the remaining *overlay graph* are preserved. This property is achieved by replacing paths of the form $\langle u, v, w\rangle$ by a *shortcut* edge $\langle u, w\rangle$. Note that the shortcut $\langle u, w\rangle$ is only required if $\langle u, v, w\rangle$ is the only shortest path from u to w.

* Partially supported by DFG grant SA 933/1-3, and by the Future and Emerging Technologies Unit of EC (IST priority – 6th FP), under contract no. FP6-021235-2 (project ARRIVAL).

C.C. McGeoch (Ed.): WEA 2008, LNCS 5038, pp. 319–333, 2008.

We shall view the contraction process as a way to add all discovered shortcuts to the edge set E. We obtain a *contraction hierarchy (CH)*. Section 2 gives more details.

In Section 3 we explain how the nodes are ordered. Although 'optimal' node ordering seems a quite difficult problem, already very simple local heuristics turn out to work quite well. The basic idea is to keep the nodes in a priority queue sorted by some estimate of how attractive it is to contract a node. The main ingredient of this heuristic estimate is the *edge difference*: The number of shortcuts introduced when contracting v minus the number of edges incident to v. The intuition behind this is that the contracted graph should have as few edges as possible. Even using only edge difference, quite good CHs are computed. However, further refinements are useful. In particular, it is important to contract nodes 'uniformly'.

For routing, we split the CH (V, E) into an *upward graph* $G_\uparrow := (V, E_\uparrow)$ with $E_\uparrow := \{(u, v) \in E : u < v\}$ and a *downward graph* $G_\downarrow := (V, E_\downarrow)$ *with* $E_\downarrow := \{(u, v) \in E : u > v\}$. For a shortest path query from s to t, we perform a modified bidirectional Dijkstra shortest path search, consisting of a forward search in $G_\uparrow$ and a backward search in $G_\downarrow$. If, and only if, there exists a shortest s-t-path in the original graph, then both search scopes eventually meet at a node v that has the highest order of all nodes in a shortest s-t-path. More details of the query algorithm are given in Section 4. Applications and refinements like dynamic routing (i.e., edge weights are allowed to change), many-to-many routing, and combinations with other speedup techniques can be found in Section 5. Section 6 shows that in many cases, we get significant improvements over previous techniques for large real world inputs. Lessons learned and possible future improvements are summarized in Section 7.

Related Work

Since there has recently been extensive work on speed-up techniques, we can only give a very abridged overview with emphasis on the directly related techniques beginning with the closest kin. For a more detailed overview we refer to [1,2]. CHs are an extreme case of the hierarchies in highway-node routing (HNR) [3,2] – 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 in [3,2] are only efficient for a small number of geometrically shrinking levels. We also give a faster and more space efficient query algorithm using $G_\uparrow$ and $G_\downarrow$.

The node ordering in highway-node routing uses levels computed by *highway hierarchies* (HHs) [4,5,2]. 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 reduction* is a sophisticated and relatively costly routine that only keeps edges required 'in the middle' of 'long-distance' paths. *Node reduction* contracts nodes. In the original paper for undirected HHs [5], node reduction only contracted nodes of degrees one and two, i.e., it removed attached trees and multihop

paths. We originally viewed node contraction as a mere helper for the main workhorse edge reduction. For directed graphs [5], we needed a more general criterion which nodes should be contracted away. It turned out that the edge difference is a good way to estimate the cost of contracting a node v. In [6,7] this method is further refined to use a priority queue and to avoid parallel edges. All previous approaches to contraction had in common that the average degree of the nodes in the overlay graph would eventually explode. So it looked like an additional technique such as edge reduction 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 get very good performance.

The fastest speedup technique so far, *transit-node routing* [8,2], offers a factor up to 40 times better query times than CHs. However, it needs considerably higher preprocessing time and space, is less amenable to dynamization, and, most importantly it relies on another hierarchical speedup technique for its preprocessing. We have preliminary evidence that using CHs for this purpose leads to improved performance.

Finally, there is an entirely different family of speedup techniques based on goal-directed routing. Combination of CHs with goal-directed routing is the subject of another paper [9] that systematically studies such combinations.

2 Contraction

Recall from the introduction that when contracting node v, we are dealing with an overlay graph $G' = (V', E')$ with $V' = v..n$ and an edge set E' that preserves shortest path distances wrt the input graph. In G', we face the following many-to-many shortest-path problem: For each source node $u \in v+1..n$ with $(u,v) \in E'$ and each target node $w \in v+1..n$ with $(v,w) \in E'$, we want to compare the shortest-path distance $d(u,w)$ with the shortcut length $c(u,v)+c(v,w)$ in order to decide whether the shortcut is really needed. A simple way to implement this is to perform a forward shortest-path search in the current overlay graph G' from each source, ignoring node v, until all targets have been found. We can also stop the search from u when it has reached distance $d(u,v) + \max\{c(v,w) : (v,w) \in E'\}$.

Our actual implementation uses a simple, asymmetric form of bidirectional search inspired by [10]: For each target node w we perform a single-hop backward search. For each edge $(x,w) \in E'$ we store a *bucket entry* $(c(x,w), w)$ with node x. This way, forward search from u can be limited to distance

$$c(u,v) + \max_{w:(v,w)\in E'} c(v,w) - \min_{x:(x,w)\in E'} c(x,w) \ .$$

When reaching a node x, we scan its bucket entries. For each entry (C,w), we can infer that there is a path from u to w of length $d(u,x)+C$.

Since exact shortest path search for contraction can be rather expensive, we have implemented two ways to limit the range of searches: We can limit the

number of hops (edges) used in any path $\langle u, \ldots, w \rangle$, and we can limit the total search space size of a forward search. Note that this has no influence on the correctness of subsequent queries in the CH as long as we make sure to always insert a shortcut (u, w) when we have not found a path from u to w witnessing that the shortcut is unnecessary. Also note that for hop limit two, our bidirectional approach obviates a full fledged Dijkstra search. It suffices to scan the edges leaving a source node u.

Let us now focus the discussion on the hop limit. We get a tradeoff between fast contraction 'now' for small hop limits and a more sparse graph with better query time and possible easier contraction 'later' for a large hop limit. In our experiments it turned out, that it makes sense to start with a hop limit as small as one and to later increase it. We switch from one hop limit to the next when the average degree of the overlay graph G' exceeds a specified bound.

3 Node Ordering

As already mentioned in the introduction, our basic approach uses a priority queue whose minimum element contains the node looking most attractive to be contracted next. The priority used is a linear combination of several terms. In addition to the single terms used, the linear coefficients of the different terms are important, some of them can be found in Section 6. In this section we focus on different possible terms. One difficulty with this approach is that when node v is contracted, this might affect the priorities of other nodes. We use several techniques to handle this problem:

- We use *lazy update*, i.e., before actually contracting v, we update its priority. If it now exceeds the priority of the second largest element v', we reinsert v and continue with v'. This process is repeated until a consistent minimum is found. Note that (at least wrt the result of node ordering) lazy update obviates immediate updates when a priority *increases*.
- We recompute the priority of the neighbors of v.
- We periodically reevaluate all priorities and rebuild the priority queue.

Edge Difference. Arguably the most important term is the edge difference. For computing it, node ordering uses the same heuristics for limiting search spaces as are later used in the actual contraction.[1]

Uniformity. Using only the edge difference, one can get quite slow routing. For example, if the the input graph is a path, contraction would produce a linear hierarchy where most queries would again follow paths of linear length. In contrast, if we iteratively contract maximal independent sets, we would get a hierarchy where any query is finished in logarithmic time.

[1] Updating neighbors of contracted nodes and lazy update 'almost' suffice to keep the priorities up to date wrt the edge difference. However, with some highly constructed example, not all priorities are updated in time when the search horizon is limited.

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 two most successful ones. For all measures used here, a large value means that the node is contracted late.

Deleted Neighbors: We count the number of neighbors that have already been contracted. This includes neighbors reached via shortcuts. Obviously, this quantity can be maintained correctly by either lazy update or by updating the neighbors of a contracted node. This heuristics is very simple and can be computed efficiently.

Voronoi Regions: Define the Voronoi-Region $R(v)$ of a node v in an overlay graph as the set of nodes in the input graph that are closer to v than to any other node in the overlay graph. We use the square root of the size of the Voronoi-region as a term in the priority function. By preferably contracting small Voronoi regions, we can hope that the nodes of the overlay graph are spread uniformly over the network. When v is contracted, its neighboring Voronoi regions will 'eat up' $R(v)$. The necessary computations can be made using $O(|R(v)|)$ steps of Dijkstra's algorithm [11]. If we always contract Voronoi regions of size at most a constant times the average region size, we can easily show that the total number of Dijkstra-steps for maintaining the size of the Voronoi regions is $O(n \log n)$, i.e., computing Voronoi regions is reasonably efficient. Since Voronoi regions can only grow, lazy update ensures that the priority queue works correctly wrt this term of the priority function.

There are a number of further, optional parameters of the priority function that turn out to further improve the hierarchy at the cost of increased time for node ordering.

Cost of contraction. A time consuming part of the contraction are the forward shortest-path searches to decide the necessity of shortcuts. So for example, we can use the sum of these search space sizes as a priority term. Note that this quantity can change beyond the direct neighborhood of the contracted node, i.e., our update rules are only heuristics.

Cost of queries. One can try to estimate how contracting nodes affects the size of query search spaces. We have implemented the following simple estimate $Q(v)$ that can be shown to be an upper bound for the number of hops of a path $\langle s, \ldots, v \rangle$ explored during a query: Initially, $Q(v) = 0$. When v is contracted then for each neighbor u of v, $Q(u) := \max(Q(u), Q(v) + 1)$.

Global measures. We can prefer contracting globally unimportant nodes based on some path based centrality measure such as (approximate) betweenness [12] or reach [13,6].

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

the number of actually used terms small, we use the same set of parameters for different inputs, and we make some sensitivity analysis to find out how robust the parameter choices are.

4 Query

In the introduction we have already outlined the basic approach which we shall now describe in more detail. An algorithm that already works quite well performs complete Dijkstra searches from s in $G_\uparrow$ and from t in $G_\downarrow$. We have

Lemma 1. $d(s,t) = \min\{d(s,v) + d(v,t) : v \textit{ is settled in both searches}\}$.

Proof. We only give a proof outline for self-containedness since the CH-query is a special case of the HNR-query for which a detailed yet simple correctness proof is given in [2]. In particular, here we only consider the case where shortest paths are unique.

Let v denote the largest[2] node on the shortest path P from s to t. We first claim that the sequence of prefix maxima[3] of P forms the shortest path from s to v in the upward graph $G_\uparrow$. If $s = v$ there is nothing to prove. Otherwise, consider any pair (u, w) of subsequent prefix maxima in P and the overlay graph $G' = (u..n, E')$ existing at some point during contraction. Since the shortest path from u to w uses only interior nodes smaller than u, and by definition of the properties of an overlay graph, $(u, w) \in E'$ and $c(u, w) = d(u, w)$. Moreover, $u < w$ and hence $(u, w) \in G_\uparrow$. Analogously, the sequence of suffix maxima of P forms the shortest path from v to t in the downward graph. □

There are two refinements to the complete search algorithm (that are also analogous to the HNR-query algorithm [3,2]). The query alternates between forward and backward search. Whenever we settle a node in one direction that is already settled in the other direction, we get a new candidate for a shortest path. Search is aborted in one direction if the smallest element in the queue is at least as large as the best candidate path found so far. This does not affect correctness, since additional settled nodes in this direction cannot possibly contribute to better solutions.

We also prune the search space using the *stall-on-demand* technique: Before a node v is settled at distance $d(v)$ in the forward search, it uses the information available in $G_\downarrow$ to inspect downward edges (w, v) with $w > v$. If $d(w) + c(w, v) < d(v)$, then the search can be stopped (*stalled*) at v with stalling distance $d(w) + c(w, v)$ since the computed distance to v is suboptimal so that a continuation of the search from v would be futile. Such stalled nodes are settled but their incident edges are not relaxed, leading to a considerably smaller search space. Moreover, stalling can propagate to further nodes x in the neighborhood of v, if the path over w in G to x is shorter than the currently found path to x in

[2] Recall that nodes are considered to be numbered during node ordering.

[3] i.e., the sequence of nodes u_i on $P = \langle s = u_1, u_2, \ldots, u_k = t\rangle$ with the property that $u_i > \max\{u_1, u_2, \ldots, u_{i-1}\}$.

$G_\uparrow$. We perform a local BFS from v using the edges available in $G_\uparrow$ or $G_\downarrow$.[4] The search stops at nodes that are not being stalled. To ensure correctness, we unstall a node x if a shorter path in $G_\uparrow$ to x than the current one in $G_\uparrow$ is found. Stall-on-demand is also applied to the backward search in the same way.

The graphs $G_\uparrow$ and $G_\downarrow$ can be stored in one data structure, using two direction flags for each edge to indicate whether it belongs to $G_\uparrow$ or $G_\downarrow$. Irrespective of the direction flags, each edge (u, v) is stored only once, namely at the smaller node, which complies with the requirements of both forward and backward search (including the stall-on-demand technique). In particular, this also applies to undirected edges $\{u, v\}$ with the same weight in both directions. In contrast, an efficient implementation of Dijkstra's (even unidirectional) algorithm needs to store such undirected edges $\{u, v\}$ both at u and v. This is the reason why we may need less space than Dijkstra's algorithm for the original graph, even though we have to insert shortcuts.

Outputting Paths. As all routing techniques that use shortcuts, we need a way to unpack them in order to obtain a shortest path in the input graph. This is particularly simple for CHs since each shortcut (u, w) bypasses exactly one node v. We therefore obtain a simple recursive unpacking routine. In order to implement this efficiently, we need to store v together with the shortcut somewhere. Note that this information is *not* easily obtained just from $G_\uparrow$ or $G_\downarrow$, i.e., our observation that we may need less space than the input graph only holds when path unpacking is not required.

5 Applications

Changing all Edge Weights. In CHs we can distinguish between two main phases of preprocessing, node ordering and hierarchy construction. Similar to highway-node routing, we do not have to redo node ordering when the edge weights change – for example when we switch from driving times for a fast car to a slow truck. Hierarchy construction ensures correctness for *all* node orderings. We will see that the resulting hierarchies are almost as good as hierarchies where node ordering has been repeated. The intuition behind this is that most important nodes remain important even if the actual edge weights change – both sports cars and trucks are fastest on the motorway.

Changing some Edge Weights. Since CHs are a special case of HNR [3,2], we can also adopt the successful approaches used there for routing in presence of some changed edges (e.g., due to traffic jams).

Many-to-Many Routing. In [10] we developed an algorithm based on highway hierarchies that finds all shortest path distances between a set S of source nodes and a set T of target nodes. The idea is to perform only $|T|$ backward searches,

[4] We also have a version that additionally exploits the parent pointers of the shortest path tree. This slightly decreases search space but slightly *increases* query time.

store the resulting search spaces appropriately and then to perform $|S|$ forward searches that use the stored information on the backward searches to find the shortest path distances. As explained in [2], this works for a large family of non-goal-directed hierarchical routing techniques including highway-node routing and reach-based routing [13,6]. CHs are particularly well suited for many-to-many routing because they have very small search spaces and because for the backward search spaces *we only need to store nodes that are not stalled*.

Distance Oracles for Replacing Large Distance Tables. CH search-spaces are so small that we can drop the distance tables computed by many-to-many routing and instead store the search spaces from S and T as arrays of node-distance pairs sorted by node-id. Then an s-t query amounts to intersecting the search spaces for s and t and computing the minimum resulting distance. This intersection operation is similar to binary merging and thus runs very fast and cache efficiently.

Transit-Node Routing. Transit-node routing [8,2] is currently the fastest static routing technique available. Its main disadvantage compared to simpler techniques is that it needs considerably more preprocessing time. The preprocessing for transit-node routing is essentially a generalization of many-to-many routing. Hence, we can also do preprocessing using CHs and expect to obtain an improvement. We can use the nodes designated as most important by node ordering to define the sets of transit nodes. The edge difference criterion used by node ordering might help to identify transit-node sets that imply small sets of access nodes.

Combination with Other Speedup Techniques. There are interesting synergies between hierarchical speedup techniques and goal-directed methods such as landmark A^* [6] or arc flags [14,15]. Goal-directed techniques become cheaper in terms of preprocessing time and space if they are only applied to a *core* obtained after some contraction [16,17,6,7]. Since CHs are a fast, flexible, effective, and very fine-grained approach to this contraction, they seem best suited for this. The resulting overall query time is often better than any of the techniques alone. For example, an integration of CHs and arc-flags is so fast that it almost achieves the query times of transit-node routing using less space [9]. Another interesting example is SHARC-routing [7] which applies a sophisticated, multi-level variant of arc-flags to an network enriched with shortcuts. This has the advantage that it yields a unidirectional, very simple query algorithm that takes hierarchy into account indirectly via the arc flags.

Perhaps most importantly, not all graph families are as well behaved as road networks with travel time weights with respect to contraction. So it sometimes seems to be the best idea to stop contraction at some point and solely rely on goal-directed techniques for the core [9].

Node contraction started out as an ingredient of highway hierarchies (HHs). It would be interesting to see how good HHs would perform if we would reintegrate CHs into HHs. We could expect a more sparse network in the upper levels but

also a more complicated, less focused query algorithm. Our guess would be that for road networks, we cannot expect an additional improvement but perhaps we should keep this approach in mind for network where contraction does not work so well.

Similarly, we could integrate CHs with reach-based routing [13,6]. CHs could contribute the shortcuts to be used, possibly simplifying the reach approximations during preprocessing. During the query, we could use reach values to prune the search additionally.

Implementation on Mobile Devices. Due to its small memory overhead and search space, CHs are a good starting point for route planning on mobile devices. This is the subject of a separate paper [18].

6 Experiments

Environment. Experiments have been 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). The program was compiled by the GNU C++ compiler 4.2.1 using optimization level 3.

Test Instances. Our experiments in this section have been done on a road network of Western Europe[5] with 18 029 721 nodes and 42 199 587 directed edges, which has been made available for scientific use by the company PTV AG. For each edge, its length and one out of 13 road categories (e.g., motorway, national road, regional road, urban street) is provided so that an expected travel time can be derived, which we use as edge weight. Results for other test instances can be found in the full paper.

Different Variants. Although the basic idea of CHs is simple, we have many tuning parameters that should be set carefully and we should verify that these choices are robust in the sense that they work reasonably well for different instances. Therefore, we build up the system incrementally. Tab. 1 shows the most fundamental performance parameters for a number of increasingly sophisticated variants. For comparison, we add the times for the fastest variant of highway-node routing (HNR) from [3] using the same system environment. Note that this version of HNR outperforms all previous speedup techniques with comparable preprocessing time so that focusing on HNR is meaningful.

Already using only the edge difference we obtain query times better than HNR. However, the preprocessing time and space is quite large. Just adding the uniformity parameter based on number of deleted neighbors (Line ED), we obtain more than four times better query time than HNR. The time for hierarchy construction becomes better than HNR once we take the search space size into

[5] Austria, Belgium, Denmark, France, Germany, Italy, Luxembourg, the Netherlands, Norway, Portugal, Spain, Sweden, Switzerland, and the UK.

Table 1. Performance of various node ordering heuristics. Terms of the priority function: E=edge difference, D=deleted neighbors, S=search space size, W=relative betweenness, V=$\sqrt{\text{Voronoi region size}}$, L=limit search space on weight calculation, Q=upper bound on edges in search paths. Digits denote hop limits for testing shortcuts. Space overhead is wrt an adjacency array for *bidirectional* Dijkstra that stores each directed edge at both endpoints. The bottom line shows the performance for highway-node routing using the code from [3].

method	node ordering [s]	hierarchy construction [s]	query [μs]	nodes settled	non-stalled nodes	edges relaxed	space overhead [B/node]
E	13010	1739	670	1791	1127	4999	-1.6
ED	7746	1062	183	403	236	1454	-2.3
ES	5355	123	245	614	366	1803	**-3.5**
ESL	1158	123	292	758	465	2169	**-3.5**
EDL	2071	576	187	418	243	1483	-2.3
EDSL	1414	165	175	399	228	1335	-2.6
ED5	634	98	224	470	250	1674	-1.6
EDS5	652	99	213	462	256	1651	-2.1
EDS1235	**545**	**57**	223	459	234	1638	0.6
EDSQ1235	591	64	211	440	236	1621	1.0
EDSQL	1648	199	173	385	220	1378	-2.1
EVSQL	1627	170	159	368	209	1181	-2.7
EDSQWL	1629	199	163	372	218	1293	-2.5
EVSQWL	1734	180	**154**	**359**	**208**	**1159**	-3.0
HNR	594	203	802	957	630	7561	9.5

account (letter S). This also improves node ordering if we limit the size of a local search (letter L).

To improve the preprocessing times, it helps to limit the number of hops in the searches during preprocessing and to take search space sizes for contraction into account. Figure 1 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 sufficiently before this explosion.[6] Interestingly, this also further improves query time. The algorithm in Line EDS1235 of Tab. 1 outperforms HNR in all respects and with a wide margin with respect to query time and hierarchy construction[7] time. As explained in Section 5, the latter time is particularly interesting when we want to exchange the edge weight function. We use this variant as our main *economical*[8] variant for further experiments.

[6] $1 \to 2$ hops @ degree 3.3, $2 \to 3$ @ 10, $3 \to 5$ @ 10. After switching to hop limit 3, we remove all edges e for which there is a witness with at most 3 edges that e is not a shortest path. This reduces the average degree and leaves some time before we have to switch to hop limit 5.

[7] There is a version of HNR in [3] with about two times faster hierarchy construction but with slower queries and more space consumption.

[8] Coefficients for priority: E=190, D=120, S=1.

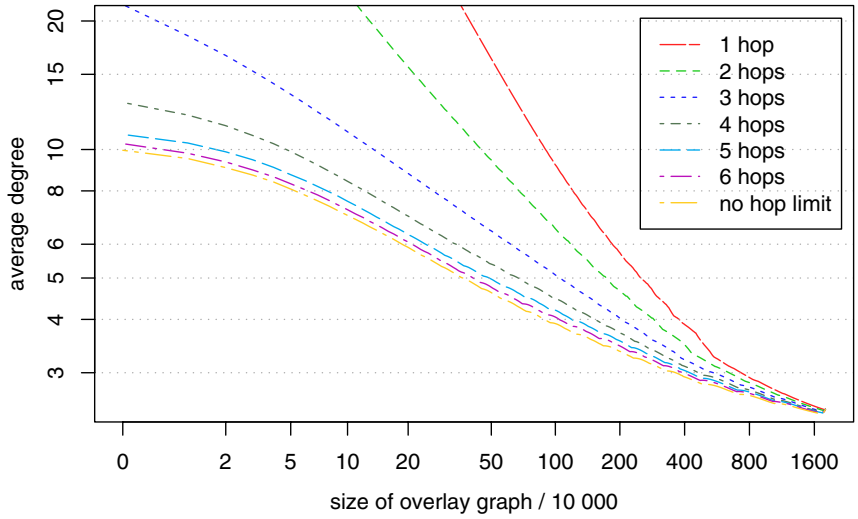

Fig. 1. Average degree development for different hop limits

By investing more preprocessing time, we can further improve the query performance. We abandon hop limitations and take the path-length estimate $Q(v)$ into account. The resulting algorithm, Line EVSQL in Tab. 1, is used as our *aggressive*[9] variant for further experiments. Using betweenness[10] approximations (letter W) can improve the query time by additional 3%.[11] It is interesting to compare different indicators for query performance between aggressive CH and HNR. CHs are 5 times faster although the number of settled nodes is only 2.6 times smaller. This is in part due to a simpler data structure[12] and in part due to a far larger improvement (factor 6.4) wrt the number of relaxed edges. For many-to-many routing, we are mostly interested in the number of non-stalled nodes, which make the bucket-scan operations more expensive. In this respect, CHs are a factor 3 better.

Local Queries. Since random queries are unrealistic for large graphs, Fig. 2 shows the distributions of query times for various degrees of locality [4]. We see a uniform improvement over HNR and small fluctuations in query time. This is further underlined in Fig. 3 where we give upper bounds for the search space size of *all* $n \times n$ possible queries (see [5] for the algorithm). We see a superexponential decay of the probability to observe a certain search-space size and maximal search-space size bound less than 2.5 times the size of the average actual search-space sizes (see also Tab. 1).

[9] Coefficients for priority: E=190, V=60, S=1, Q=145, L=1000.
[10] The execution times for betweenness approximation [12] are not included in Tab. 1.
[11] Preliminary experiments with reach-approximations were not successful.
[12] The HNR implementation from [3] has to compare level information to find out which edges should be relaxed.

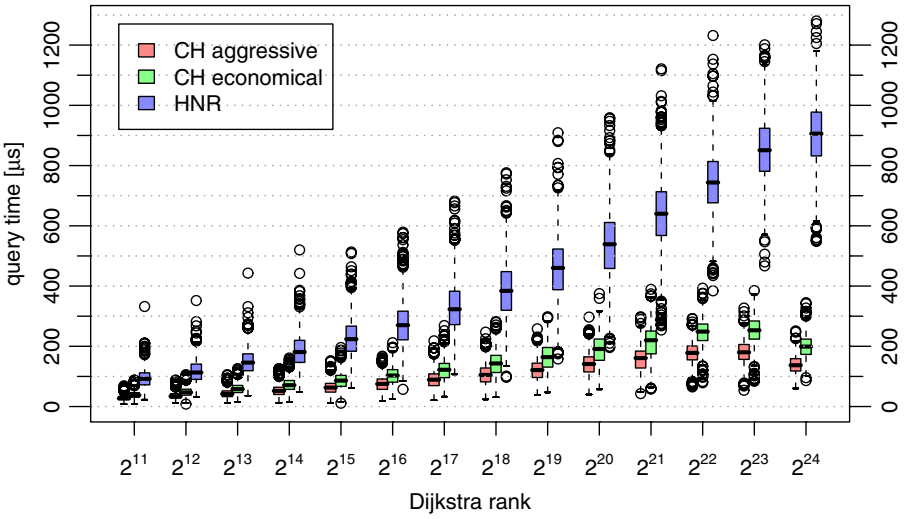

Fig. 2. Local queries, box-and-whisker plot [19]: each box spreads from the lower to the upper quartile and contains the median, the whiskers extend to the minimum and maximum value omitting outliers, which are plotted individually. The queries generated for x-value r are random s–t-queries under the constraint that t is the r-th node visited by Dijkstra's algorithm (see also [4]).

Unpacking Paths needs an average of 317 μs for the aggressive variant and 332 μs for the economical variant. The difference between the two variants is bigger for the space overhead which is 5.8 B/node and 10.8 respectively. Among the path unpacking times we have seen, this is only outperformed by the fastest variant for highway hierarchies in [5] that explicitly stores completely unpacked representations of the most important shortcuts. Note that this optimization works for any shortcut-based speedup technique including CHs.

Many-to-Many Routing for a random 10000×10000 table using the aggressive variant needs 10.2 s. This is about six times faster than the highway-hierarchy-based code from [10] and more than twice as fast as the HNR-based implementation from [2]. Our current implementation of many-to-many routing does not (yet) use the asymmetry between forward and backward search that has proved useful in [10,2]. Hence, we can expect further improvements.

Exchanging the Edge Weight Function. The table below shows the hierarchy construction time and query time using our economical variant for different speed profiles which come from the company PTV (see also [3]). The times in brackets refer to the case when node ordering was done with the same speed profile and the main times are for the case that node ordering was done for our default speed profile.

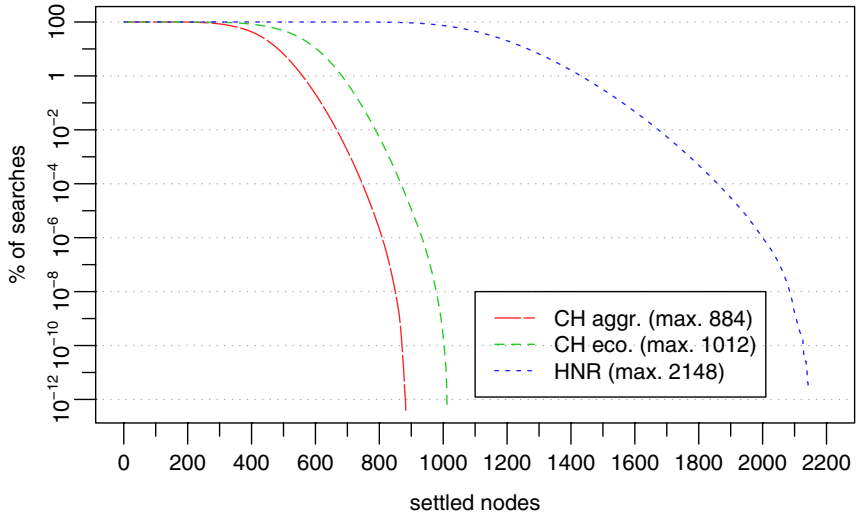

Fig. 3. Upper bound on search space in settled nodes for the worst percentages of queries

	default	fast car	slow car	slow truck
hier. construction [s]	57	80 (62)	82 (63)	88 (65)
query [μs]	223	208 (211)	232 (243)	294 (291)

We can see that preprocessing time goes up by about 30 %. Query times are about the same. Query performance decreases with the speed of the vehicle since the hierarchy induced by fast streets gets less pronounced.

Transit-Node Routing. We used the node ordering with the aggressive variant of CHs to determine the transit-node sets for the implementation from [2]. As we hoped for, this resulted in a reduced number of access nodes, which in turn results in better query time (4.3 → 3.4 μs) and lower space consumption (247 → 204 Byte/node), compared to [2]. Preliminary experiments suggest that we get further improvements with an additional term for node ordering that takes into account the number of edges of the input graph that make up a shortcut. We have not yet implemented a CH-based preprocessing so that it is too early to judge the effect of CHs on preprocessing time. It is quite likely however, that we will also see an improvement in preprocessing time.

7 Conclusions

CHs are a simple and efficient basis for many hierarchical routing methods in road networks. The experiments in [9] suggest that CHs also work well for other sparse networks with high locality such as transportation networks, or sparse

unit-disk graphs. For more dense networks, CHs can be used for an initial contraction phase whereas a goal-directed technique is applied to the resulting core network.

Several further improvements might be possible. The performance of node ordering is so far only slightly better than the HH based method used in [3] for HNR. One reason is that we perform many similar searches that might be saved if we would reuse search spaces. The main problem with reuse is that storing search spaces would cost a lot of space. But perhaps we can partition large networks into smaller networks; perform the node ordering separately for each subnetwork; and only then merge the pieces into a global order. To a lesser extend such an optimization might also accelerate hierarchy construction. As a side effect we might also obtain a way to update the search space sizes of all nodes affected by a node contraction.

Although we have established that uniformity is important for good node ordering, it is not so clear whether the two uniformity measures we have introduced are the final word. In particular, the right measure might depend on the application. For example, our current code for transit-node routing uses a geometric locality filter and hence it might be good if the uniformity measure would take geometry into account.

We have already demonstrated that CHs yield improved preprocessing times when changing the entire cost function. We still have to try how well the dynamization techniques for changing few edge weights from [3,2] translate.

Last but not least, we are now developing a method for fast routing in road networks with time-dependent edge weights. We hope that the simplicity and efficiency of CHs will give us a good starting point for this challenging task. The good performance of CHs for (unrolled) transportation networks observed in [9] may be an indicator that this will work well.

References

1. Sanders, P., Schultes, D.: Engineering fast route planning algorithms. In: Demetrescu, C. (ed.) WEA 2007. LNCS, vol. 4525, pp. 23–36. Springer, Heidelberg (2007)
2. Schultes, D.: Route Planning in Road Networks. PhD thesis (2008)
3. Schultes, D., Sanders, P.: Dynamic highway-node routing. In: Demetrescu, C. (ed.) WEA 2007. LNCS, vol. 4525, pp. 66–79. Springer, Heidelberg (2007)
4. Sanders, P., Schultes, D.: Highway hierarchies hasten exact shortest path queries. In: Brodal, G.S., Leonardi, S. (eds.) ESA 2005. LNCS, vol. 3669, pp. 568–579. Springer, Heidelberg (2005)
5. Sanders, P., Schultes, D.: Engineering highway hierarchies. In: Azar, Y., Erlebach, T. (eds.) ESA 2006. LNCS, vol. 4168, pp. 804–816. Springer, Heidelberg (2006)
6. Goldberg, A.V., Kaplan, H., Werneck, R.F.: Better landmarks within reach. In: Demetrescu, C. (ed.) WEA 2007. LNCS, vol. 4525, pp. 38–51. Springer, Heidelberg (2007)
7. Bauer, R., Delling, D.: SHARC: Fast and robust unidirectional routing. In: Workshop on Algorithm Engineering and Experiments (ALENEX) (2008)
8. Bast, H., Funke, S., Sanders, P., Schultes, D.: Fast routing in road networks with transit nodes. Science 316(5824), 566 (2007)

9. Bauer, R., Delling, D., Sanders, P., Schieferdecker, D., Schultes, D., Wagner, D.: Combining hierarchical and goal-directed speed-up techniques for Dijkstra's algorithm. In: 7th Workshop on Experimental Algorithms (WEA) (2008)
10. Knopp, S., Sanders, P., Schultes, D., Schulz, F., Wagner, D.: Computing many-to-many shortest paths using highway hierarchies. In: Workshop on Algorithm Engineering and Experiments (ALENEX) (2007)
11. Maue, J., Sanders, P., Matijevic, D.: Goal directed shortest path queries using Precomputed Cluster Distances. In: Àlvarez, C., Serna, M.J. (eds.) WEA 2006. LNCS, vol. 4007, pp. 316–328. Springer, Heidelberg (2006)
12. Geisberger, R., Sanders, P., Schultes, D.: Better approximation of betweenness centrality. In: Workshop on Algorithm Engineering and Experiments (ALENEX) (2008)
13. Gutman, R.: Reach-based routing: A new approach to shortest path algorithms optimized for road networks. In: Workshop on Algorithm Engineering and Experiments (ALENEX)., pp. 100–111 (2004)
14. Lauther, U.: 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, pp. 219–230. IfGI prints, Institut für Geoinformatik, Münster (2004)
15. Köhler, E., Möhring, R.H., Schilling, H.: Acceleration of shortest path and constrained shortest path computation. In: 4th International Workshop on Efficient and Experimental Algorithms (WEA) (2005)
16. Delling, D., Sanders, P., Schultes, D., Wagner, D.: Highway hierarchies star. In: 9th DIMACS Implementation Challenge [20] (2006)
17. Goldberg, A.V., Kaplan, H., Werneck, R.F.: Better landmarks within reach. In: 9th DIMACS Implementation Challenge [20] (2006)
18. Sanders, P., Schultes, D., Vetter, C.: Mobile Route Planning (2008) in preparation, `http://algo2.iti.uka.de/schultes/hwy/`
19. R Development Core Team: R: A Language and Environment for Statistical Computing (2004), `http://www.r-project.org`
20. 9th DIMACS Implementation Challenge: Shortest Paths (2006), `http://www.dis.uniroma1.it/~challenge9/`