Dynamic Highway-Node Routing[⋆]

Dominik Schultes and Peter Sanders

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

Abstract. We introduce a *dynamic* technique for fast route planning in large road networks. For the first time, it is possible to handle the practically relevant scenarios that arise in present-day navigation systems: When an edge weight changes (e.g., due to a traffic jam), we can update the preprocessed information in 2–40 ms allowing subsequent fast queries in about one millisecond on average. When we want to perform only a single query, we can skip the comparatively expensive update step and directly perform a prudent query that automatically takes the changed situation into account. If the overall cost function changes (e.g., due to a different vehicle type), recomputing the preprocessed information takes typically less than two minutes.

The foundation of our dynamic method is a new static approach that generalises and combines several previous speedup techniques. It has outstandingly low memory requirements of only a few bytes per node.

1 Introduction

Computing fastest routes in road networks is one of the showpieces of real-world applications of algorithmics. In principle we could use Dijkstra's algorithm. But for large road networks this would be far too slow. Therefore, in recent years, there has been considerable interest in speed-up techniques for route planning. For an overview, we refer to [1]. The most successful methods are *static*, i.e., they assume that the network—including its edge weights—does not change. This makes it possible to *preprocess* some information *once and for all* that can be used to accelerate *all* subsequent point-to-point *queries*. Today, the static routing problem in road networks can be regarded as largely solved.

However, real road networks change all the time. In this paper, we address two such *dynamic* scenarios: individual edge weight updates, e.g., due to traffic jams, and switching between different cost functions that take vehicle type, road restrictions, or driver preferences into account.

1.1 Related Work

Bidirectional Search. simultaneously searches forward from s and backwards from t until the search frontiers meet. Many speedup techniques (including ours) use bidirectional search as an ingredient.

[⋆] Partially supported by DFG grant SA 933/1-3.

Separators. Perhaps the most well known property of road networks is that they are almost planar, i.e., techniques developed for planar graphs will often also work for road networks.

Using $O(n \log^2 n)$ space and preprocessing time, query time $O(\sqrt{n} \log n)$ can be achieved [2,3] for directed planar graphs without negative cycles; edge weights can be updated in amortised $O(n^{2/3} \log^{5/3} n)$ time per operation (provided that all edge weights are positive).

A previous practical approach is the *separator-based multi-level method* [4,5,6]. Out of several existing variants, we mainly refer to [5, basic variant]. For a graph $G = (V, E)$ and a node set $V' \subseteq V$, a *shortest-path overlay graph* $G' = (V', E')$ has the property that E' is a minimal set of edges such that each shortest-path distance $d(u, v)$ in G' is equal to the shortest-path distance from u to v in G. In the separator based approach, V' is chosen in such a way that the subgraph induced by $V \setminus V'$ consists of small components of similar size. The overlay graph can be constructed by performing a search in G from each separator node that stops when all neighbouring separator nodes have been found. In a bidirectional query algorithm, the components that contain the source and target nodes are searched considering *all* edges. From the border of these components, i.e., from the separator nodes, however, the search is continued considering only edges of the overlay graph. By recursing on G', this basic idea is generalised to multiple levels.

Bauer [7] observes that if the weight of an edge within some component C changes, we do not have to repeat the complete construction process of G'. It is sufficient to rerun the construction step only from some separator nodes at the boundary of C. No experimental evaluation is given. In a theoretical study on the dynamisation of shortest-path overlay graphs [8], an algorithm is presented that requires $O(|V'|(n+m) \log n)$ preprocessing time and $O(|V'|(n+m))$ space, which seems impractical for large graphs.

Highway Hierarchies. Commercial systems use information on road categories to speed up search. 'Sufficiently far away' from source and target, only 'important' roads are used. This requires manual tuning of the data and a delicate tradeoff between computation speed and suboptimality of the computed routes. In previous papers [9,10] we introduced the idea to *automatically* compute *highway hierarchies* that yield *optimal* routes *uncompromisingly quickly.* The basic idea is to define a neighbourhood for each node to consist of its H closest neighbours. Now an edge (u, v) is a highway edge if there is some shortest path $\langle s, \ldots, u, v, \ldots t\rangle$ such that neither u is in the neighbourhood of t nor v is in the neighbourhood of s. This defines the first level of the highway hierarchy. After contracting the network to remove low degree nodes—obtaining its so-called *core*—, the same procedure (identifying the highway network at the next level followed by contraction) is applied recursively. We obtain a hierarchy. The query algorithm is bidirectional Dijkstra with restrictions on relaxing certain edges. Roughly, far away from source or target, only high level edges need to be considered.

Reach-Based Routing [11,12] is a speed-up technique that is similar to highway hierarchies. A dynamic version that handles a set of edge weight changes is presented in [7]. The basic idea is to rerun the construction step only from nodes within a certain area, which has to be identified first. So far, the concept of *shortcuts* [9,12], which is important to get competitive construction and query times, has not been integrated in the dynamic version. No experimental evaluation for the dynamic scenario is given in [7].

Transit Node Routing. [13] is based on the following observation: "When you drive to somewhere 'far away', you will leave your current location via one of only a few 'important' traffic junctions [*transit nodes*]". Distances from each node to all neighbouring transit nodes and between all transit nodes are precomputed so that a non-local shortest-path query can be reduced to a small number of table lookups. Currently, transit node routing provides the best query times on road networks, but so far no dynamic version is available.

Goal Direction. Another interesting property of road networks is that they allow effective goal directed search using $\mathbf{A}^*$ *search* [14]: lower bounds define a vertex potential that directs search towards the target. This approach was shown to be very effective if lower bounds are computed using precomputed shortest-path distances to a carefully selected set of about 20 **L***andmark* nodes [15] using the **T**riangle inequality (ALT). In [15] it is also briefly mentioned that in case of an edge weight *increase*, the query algorithm stays correct even if the landmarks and the landmark distances are not updated. To cope with drastic changes or edge weight decreases, an update of the landmark distances is suggested. In [16], these ideas are pursued leading to an extensive experimental study of landmark-based routing in various dynamic scenarios. For a comparison to our approach, see Section 5.

1.2 Overview and Main Contributions

Our first main contribution is the generalisation of the separator-based multi-level method to *highway-node routing* where overlay graphs are defined using arbitrary node sets $V' \subseteq V$ rather than separators. This requires new algorithms for preprocessing and queries since removing V' will in general *not* partition the graph into small components. Section 2 lies the ground for these new algorithms by systematically investigating the graph theoretical problem of finding all nodes from V' that can be reached on a shortest path from a given node without passing another node from V'. In Section 3 we apply the results to static highway-node routing. The main remaining difficulty is to choose the highway nodes. The idea is that *important* nodes used by many shortest paths will lead to overlay graphs that are more sparse than for the separator-based approach. This will result in faster queries and low space consumption. The intuition behind this idea is that the number of overlay graph edges needed between the separator nodes bordering a region grows quadratically with the number of border nodes (see also [6]). In contrast, important nodes are uniformly distributed over the network and connected to only a small number of nearby important nodes (see also [13]). Indeed, our method is so far the most space-efficient preprocessing technique that allows query times several thousand times faster than Dijkstra's algorithm. A direct comparison to the separator-based variant is difficult since previous papers use comparatively small graphs[1] and it is not clear how the original approach scales to very large graphs.

While there are many ways to choose important nodes, we capitalise on previous results and use *highway hierarchies* to define all required node sets. There is an analogy to *transit node routing* where we also used highway hierarchies to find important nodes.

On the first glance, our approach to highway-node routing looks like a roundabout way to achieve similar results as with the direct application of highway hierarchies. However, by factoring out all the complications of highway hierarchies into a pre-preprocessing step, we simplify data structures, the query algorithm, and, most importantly, *dynamic variants*. The idea is that in practice, a set of nodes important for one weight function will also contain most of the important nodes for another 'reasonable' weight function. The advantage is obvious when the cost function is redefined—all we have to do is to recompute the edge sets of the overlay graphs. Section 4 discusses two variants of the scenario when a few edge weights change: In a server setting, the overlay graphs are updated so that afterwards the static query algorithm will again yield correct results. In a mobile setting, the data structure are not updated. Rather, the query algorithm searches at lower levels of the node hierarchy, (only) where the information at the higher levels might have been compromised by the changed edges.

Together with [16], we are the first to present an approach that tackles such dynamic scenarios and to demonstrate its efficiency in an extensive experimental study using a real-world road network, which is summarised in Section 5.

2 Covering Nodes

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* T 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 $d(s, u) = w(P^*)$ is known. 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*

[1] For a subgraph of the European road network with about 100 000 nodes, [6] gives a preprocessing time of "well over half an hour [plus] several minutes" and a query time 22 times faster than Dijkstra's algorithm. For a comparison, we take a subgraph around Karlsruhe of a very similar size, which we preprocess in seven seconds. Then, we obtain a speedup of 94.

distance $\delta(u) = w(P)$ is known. Reached but not settled nodes are also called *queued*. Nodes that are not reached are *unreached*.

Problem Definition. During a Dijkstra search from s, we say that a settled node u is *covered* by a node set V' if there is at least one node $v \in V'$ on the path from the root s to u. A queued node is *covered* if its tentative parent is covered. The current partial shortest-path tree T is *covered* if all currently queued nodes are covered. All nodes $v \in V' \cap T$ that have no parent in T that is covered are *covering nodes*, forming the set $\mathcal{C}_G(V', s)$.

The crucial subroutine of all algorithms in the subsequent sections takes a graph G, a node set V', and a root s and determines all covering nodes $\mathcal{C}_G(V', s)$. We distinguish between four different ways of doing this.

Conservative Approach. The *conservative* variant (Fig. 1 (a)) works in the obvious way: a search from s is stopped as soon as the current partial shortest-path tree T is covered. Then, it is straightforward to read off all covering nodes. However, if the partial shortest-path tree contains one path that is not covered for a long time, the tree can get very big even though all other branches might have been covered very early. In our application, this is a critical issue due to long-distance ferry connections.

Aggressive Approach. As an overreaction to the above observation, we might want to define an *aggressive* variant that does not continue the search from any covering node, i.e., some branches might be terminated early, while only the non-covered paths are followed further on. Unfortunately, this provokes two problems. First, we can no longer guarantee that T contains only shortest paths. As a consequence, we get a superset $\overline{\mathcal{C}}_G(V', s)$ of the covering nodes, which still can be used to obtain correct results. However, the performance will be impaired. In Section 3, we will explain how to reduce a given superset rather efficiently in order to obtain the exact covering node set. Second, the tree T can get even bigger since the search might continue *around* the covering nodes where we pruned the search.[2] In our example (Fig. 1 (b)), the search is pruned at u so that v is reached using a much longer path that leads around u. As a consequence, w is superfluously marked as a covering node.

Stall-in-Advance Technique. If we decide not to prune the search immediately, but to go on 'for a while' in order to *stall* other branches, we obtain a compromise between the conservative and the aggressive variant, which we call *stall-in-advance*. One heuristic we use prunes the search at node z when the path explored from s to z contains p nodes of V' for some tuning parameter p. Note that for $p := 1$, the stall-in-advance variant corresponds to the aggressive variant. In our example (Fig. 1 (c)), we use $p := 2$. Therefore, the search is pruned

[2] Note that the query algorithm of the separator-based approach virtually uses the aggressive variant to compute covering nodes. This is reasonable since the search can never 'escape' the component where it started.

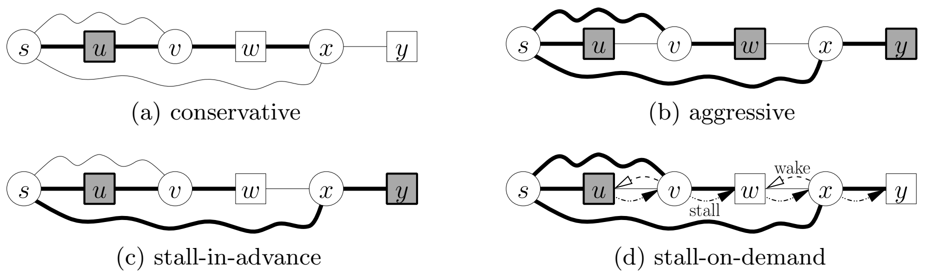

Fig. 1. Simple example for the computation of covering nodes. We assume that all edges have weight 1 except for the edges (s, v) and (s, x), which have weight 10. In each case, the search process is started from s. The set V' consists of all nodes that are represented by a square. Thick edges belong to the search tree T. Nodes that belong to the computed superset $\overline{\mathcal{C}}_G(V', s)$ of the covering nodes are highlighted in grey. Note that the actual covering node set contains only one node, namely u.

not until w is settled. This stalls the edge (s, v) and, in contrast to (b), the node v is covered. Still, the search is pruned too early so that the edge (s, x) is used to settle x.

Stall-on-Demand Technique. In the stall-in-advance variant, relaxing an edge leaving a covered node is based on the 'hope' that this might stall another branch. However, our heuristic is not perfect, i.e., some edges are relaxed in vain, while other edges which would have been able to stall other branches, are not relaxed. Since we are not able to make the perfect decision in advance, we introduce a fourth variant, namely *stall-on-demand*. It is an extension of the aggressive variant, i.e., at first, edges leaving a covered node are not relaxed. However, if such a node u is reached later via another path, it is *woken up* and a breadth-first search (BFS) is performed from that node: an adjacent node v that has already been reached by the main search is inserted into the BFS queue if we can prove that the best path P found so far is suboptimal. This is certainly the case if the path from s via u to v is shorter than P. All nodes encountered during the BFS are marked as *stalled*. The main search is pruned at stalled nodes. Furthermore, stalled nodes are never marked as covering nodes. The stalling process cannot invalidate the correctness since only nodes are stalled that otherwise would contribute to suboptimal paths. In our example (Fig. 1 (d)), the search is pruned at u. When v is settled, we assume that the edge (v, w) is relaxed first. Then, the edge (v, u) wakes the node u up. A stalling process (a BFS search) is started from u. The nodes v and w are marked as stalled. When w is settled, its outgoing edges are not relaxed. Similarly, the edge (x, w) wakes the stalled node w and another stalling process is performed.

3 Static Highway-Node Routing

Multi-Level Overlay Graph. For given *highway-node sets* $V =: V_0 \supseteq V_1 \supseteq \ldots \supseteq V_L$, we give a definition of the *multi-level overlay graph* $\mathcal{G} = (G_0, G_1, \ldots, G_L)$

that is almost equivalent to the definition in [5]: $G_0 := G$ and for each $\ell > 0$, we have $G_\ell := (V_\ell, E_\ell)$ with $E_\ell := \{(s,t) \in V_\ell \times V_\ell \mid \exists$ shortest path $P = \langle s, u_1, u_2, \ldots, u_k, t\rangle$ in $G_{\ell-1}$ s.t. $\forall i : u_i \notin V_\ell\}$.

Node Selection. We can choose any highway-node sets to get a correct procedure. However, this choice has a big impact on preprocessing and query performance. Roughly speaking, a node that lies on many shortest paths should belong to the node set of a high level. In a first implementation, we use the set of level-ℓ core nodes of the highway hierarchy of G as highway-node set V_ℓ. In other words, we let the construction procedure of the highway hierarchies decide the importance of the nodes.

Construction. The multi-level overlay graph is constructed in a bottom-up fashion. In order to construct level $\ell > 0$, we determine for each node $s \in V_\ell$ its covering node set $\mathcal{C} := \mathcal{C}_{G_{\ell-1}}(V_\ell \setminus \{s\}, s)$. For each $u \in \mathcal{C}$, we add an edge with weight $d(s,u)$ to E_ℓ. The stall-in-advance and the stall-on-demand variant introduced in Section 2 are efficient algorithms for computing a *superset* of $\mathcal{C}$, implying that they add some superfluous edges. Optionally, we can apply the following *reduction step* to eliminate those edges: for each node $u \in V_\ell$, we perform a search in G_ℓ (instead of $G_{\ell-1}$) until all adjacent nodes have been settled. For any node v that has been settled via a path that consists of more than one edge, we can remove the edge (u,v) since a (better) alternative that does not require this edge has been found.

Query. The query algorithm is a symmetric bidirectional procedure so that it suffices to describe the forward search, which works in a bottom-up fashion. For conciseness, we only describe the variant based on stall-on-demand. We perform a modified Dijkstra search from s in $(V, E_0 \cup \cdots \cup E_L)$. From a node whose highest level is i, only edges in $E_i \cup \cdots \cup E_L$ are relaxed. When an edge in $E_0 \cup \cdots \cup E_{i-1}$ reaches a node v in V_i, v is 'woken up' and performs a BFS in the shortest path tree to stall nodes that are not on shortest paths.

Forward and backward search are interleaved. We keep track of a tentative shortest-path length resulting from nodes that have been settled in both search directions. We abort the forward (backward) search when all keys in the forward (backward) priority queue are greater than the tentative shortest-path length.

4 Dynamic Highway-Node Routing

When a small set of edges change their weight, we can distinguish between a *server scenario* and a *mobile scenario*: In the former, a server has to react to incoming events by updating its data structures so that *any* point-to-point query can be answered correctly; in the latter, a mobile device has to react to incoming events by (re)computing a *single* point-to-point query taking into account the new situation. In the server scenario, it pays to invest some time to perform the update operation since a lot of queries depend on it. In the mobile scenario, we do not want to waste time for updating parts of the graph that are irrelevant to the current query.

Server Scenario. Similar to an exchange of the cost function, when a single or a few edge weights change, we keep the highway-node sets and update only the overlay graphs. In this case, however, we do not have to repeat the complete construction from scratch, but it is sufficient to perform the construction step only from nodes that might be affected by the change. Certainly, a node v whose partial shortest-path tree of the initial construction did not contain any node u of a modified edge (u, x) is *not* affected: if we repeated the construction step from v, we would get exactly the same partial shortest-path tree and, consequently, the same result.

During the first construction (and all subsequent update operations), we manage sets A^ℓ_u of nodes whose level-ℓ preprocessing might be affected when an outgoing edge of u changes: when a level-ℓ construction step from some node v is performed, for each node u in the partial shortest-path tree[3], we add v to A^ℓ_u. Note that these sets can be stored explicitly (as we do it in our current implementation) or we could store a superset, e.g., by some kind of *geometric container* (a disk, for instance). Figure 2 contains the pseudo-code of the update algorithm.

```
**input:** set of edges $E^m$ with modified weight;
define set of modified nodes: $V^m_0 := \{u \mid (u,v) \in E^m\}$;
**foreach** level $\ell, 1 \le \ell \le L$, **do**
    $V^m_\ell := \emptyset$; $R_\ell := \bigcup_{u \in V^m_{\ell-1}} A^\ell_u$;
    **foreach** node $v \in R_\ell$ **do**
        repeat construction step from $v$;
        if something changes, put $v$ to $V^m_\ell$;
```

Fig. 2. The update algorithm that deals with a set of edge weight changes

Mobile Scenario. In the mobile scenario, we only determine the sets of potentially unreliable nodes by using a fast variant of the update algorithm (Fig. 2), where from the last two lines only the "put v to V^m_ℓ" is kept. (Note that in particular the construction step is *not* repeated.) Then, for each node $u \in V$, we define the *reliable level* $r(u) := \min\{i - 1 \mid u \in R_i\}$ with $\min \emptyset := \infty$. In order to get correct results without updating the data structures, the query algorithm has to be modified. First, we do not relax any edge (u, v) that has been created

[3] When the stall-in-advance technique is used, some nodes are only added to the tree to potentially stall other branches. Upon completion of the construction step, we can identify nodes that have been added in vain, i.e., that were not able to stall other branches. Those nodes had no actual influence on the construction and, thus, can be ignored at this point.

during the construction of some level $> r(u)$. Second, if the search at some node u has already reached a level $\ell > r(u)$, then the search at this node is *downgraded* to level $r(u)$. In other words, if we arrive at some node from which we would have to repeat the construction step, we do not use potentially unreliable edges, but continue the search in a sufficiently low level to ensure that the correct path can be found.

Note that the update procedure, which is also used to determine the sets of potentially unreliable nodes, is performed in the forward direction. Its results cannot be directly applied to the backward direction of the query. It is simple to adjust the first modification to this situation (by considering $r(v)$ instead of $r(u)$ when dealing with an edge (u, v)). Adjusting the second modification would be more difficult, but fortunately we can completely omit it for the backward direction. As a consequence, the search process becomes asymmetric. While the forward search is continued in lower levels whenever it is necessary, the backward search is never downgraded. If 'in doubt', the backward search stops and waits for the forward search to finish the job.

5 Experiments

Environment and Instances. The 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 10.0 (kernel 2.6.13). The program was compiled by the GNU C++ compiler 4.0.2 using optimisation level 3.

We deal with the road network of Western Europe[4], which has been made available for scientific use by the company PTV AG. It consists of 18 029 721 nodes and 42 199 587 directed edges. The original graph contains for each edge a length and a road category. 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. We assign average speeds (130, 120, . . ., 10 km/h) to the road categories, compute for each edge the average travel time, and use it as weight. We call this our *default* speed profile. Experiments which we did on a US and Canadian road network of roughly the same size (provided by PTV as well) show exactly the same relative behaviour as in [10], namely that it is slightly more difficult to handle North America than Europe (e.g., 20% slower query times). Due to space constraints, we give detailed results only for Europe.

For now, we report the times needed to compute the shortest-path distance between two nodes without outputting the actual route. Note that we could also output full path descriptions using the techniques from [17], expecting a similar performance as in [17]. The query times are averages based on 10 000 randomly chosen (s, t)-pairs. In addition to providing average values, we use the methodology from [9] in order to plot query times against the 'distance' of the target from the source, where in this context, the *Dijkstra rank* is used as a measure of distance: for a fixed source s, the Dijkstra rank of a node t

[4] 14 countries: at, be, ch, de, dk, es, fr, it, lu, nl, no, pt, se, uk.

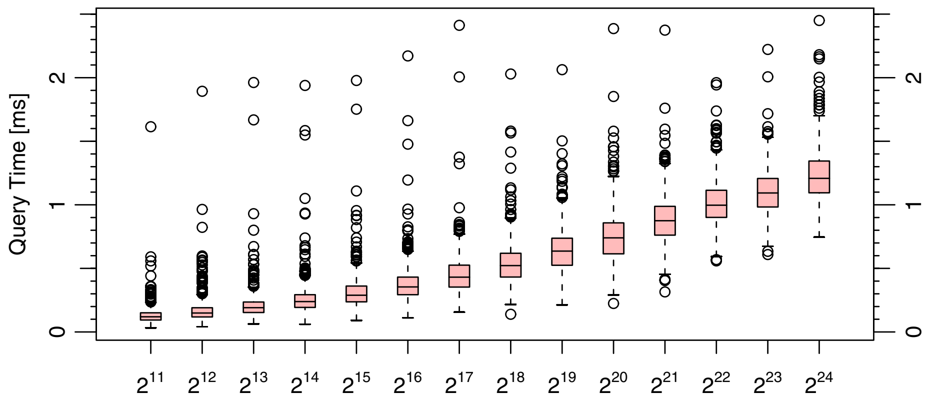

Fig. 3. Query performance against Dijkstra rank for the default speed profile, with edge reduction. Each box represents the three quartiles [18, box-and-whisker plot].

is the rank w.r.t. the order which Dijkstra's algorithm settles the nodes in. Such plots are based on 1 000 random source nodes. After performing a lot of preliminary experiments, we decided to apply the stall-in-advance technique to the construction and update process (with $p := 1$ for the construction of level 1 and $p := 5$ for all other levels) and the stall-on-demand technique to the query.

Highway Hierarchy Construction. In order to determine the highway node sets, we construct seven levels of the highway hierarchy using our default speed profile and neighbourhood size[5] $H = 70$. This can be done in 16 minutes. For *all* further experiments, these highway-node sets are used.

Static Scenario. The first data column of Tab. 1 contains the construction time of the multi-level overlay graph and the average query performance for the default speed profile. Figure 3 shows the query performance against the Dijkstra rank. The disk space overhead of the static variant is 8 bytes per node to store the additional edges of the multi-level overlay graph and the level data associated with the nodes. Note that this overhead can be further reduced to as little as 2.0 bytes per node yielding query times of 1.55 ms (Tab. 4). The total disk space[6] of 32 bytes per node also includes the original edges and a mapping from original to internal node IDs (that is needed since the nodes are reordered by level).

Changing the Cost Function. In addition to our default speed profile, Tab. 1 also gives the construction and query times for a few other selected speed profiles (which have been provided by the company PTV AG) using the same highway-node sets. Note that for most road categories, our profile is slightly faster than

[5] For details on the highway hierarchy construction and the definition of *neighbourhood size*, we refer to [9,10].

[6] The main memory usage is somewhat higher. However, we cannot give exact numbers for the static variant since our implementation does not allow to switch off the dynamic data structures.

Table 1. Construction time of the overlay graphs and query performance for different speed profiles using the same highway-node sets. For the default speed profile, we also give results for the case that the edge reduction step (Section 3) is applied.

speed profile	default	(reduced)	fast car	slow car	slow truck	distance
constr. [min]	1:40	(3:04)	1:41	1:39	1:36	3:56
query [ms]	1.17	(1.12)	1.20	1.28	1.50	35.62
#settled nodes	1 414	(1 382)	1 444	1 507	1 667	7 057

Table 2. Update times per changed edge [ms] for different road types and different update types: add a traffic jam (+), cancel a traffic jam (−), block a road (∞), and multiply the weight by 10 (×). Due to space constraints, some columns are omitted.

	any road type				motorway				national		regional		urban	
|change set|	+	−	∞	×	+	−	∞	×	+	∞	+	∞	+	∞
1	2.7	2.5	2.8	2.6	40.0	40.0	40.1	37.3	19.9	20.3	8.4	8.6	2.1	2.1
1000	2.4	2.3	2.4	2.4	8.4	8.1	8.3	8.1	7.1	7.1	5.3	5.3	2.0	2.0

Table 3. Query performance depending on the number of edge weight changes (select only motorways, multiply weight by 10). For $\leq$ 100 changes, 100 different edge sets are considered; for $\geq$ 1 000 changes, we deal only with one set. For each set, 1 000 queries are performed. We give the average percentage of queries whose shortest-path length is affected by the changes, the average number of settled nodes (also relative to zero changes), and the average query time, broken down into the init phase where the reliable levels are determined and the search phase.

	affected	#settled nodes		query time [ms]		
|change set|	queries	absolute	relative	init	search	total
1	0.6 %	2 347	(1.7)	0.3	2.0	2.3
10	6.3 %	8 294	(5.9)	1.9	7.2	9.1
100	41.3 %	43 042	(30.4)	10.6	36.9	47.5
1 000	82.6 %	200 465	(141.8)	62.0	181.9	243.9
10 000	97.5 %	645 579	(456.6)	309.9	627.1	937.0

PTV's fast car profile. The last speed profile ('distance') virtually corresponds to a distance metric since for each road type the same constant speed is assumed. The performance in case of the three PTV travel time profiles is quite close to the performance for the default profile. Hence, we can switch between these profiles without recomputing the highway-node sets. The constant speed profile is a rather difficult case. Still, it would not completely fail, although the performance gets considerably worse. We assume that any other 'reasonable' cost function would rank somewhere between our default and the constant profile.

Updating a Few Edge Weights (Server Scenario). In the dynamic scenario, we need additional space to manage the affected node sets A_u^ℓ. Furthermore, the edge reduction step is not yet supported in the dynamic case so that the total disk space usage increases to 56 bytes per node. In contrast to the static

Table 4. Comparison between pure highway hierarchies [17], three variants of highway-node routing (HNR), and dynamic ALT-16 [16]. 'Space' denotes the average disk space *overhead*. We give execution times for both a complete recomputation using a similar cost function and an update of a single motorway edge multiplying its weight by 10. Furthermore, we give search space sizes after 10 and 1 000 edge weight changes (motorway, $\times 10$) for the mobile scenario. Time measurements in parentheses have been obtained on a similar, but not identical machine.

	preprocessing		static queries		updates		dynamic queries	
	time	space	time	#settled	compl.	single	#settled nodes	
method	[min]	[B/node]	[ms]	nodes	[min]	[ms]	10 chgs.	1000 chgs.
HH pure	17	28	1.16	1 662	17	–	–	–
StHNR	19	8	1.12	1 382	3	–	–	–
StHNR mem	24	2	1.55	2 453	8	–	–	–
DynHNR	18	32	1.17	1 414	2	37	8 294	200 465
DynALT-16	(85)	128	(53.6)	74 441	(6)	(2 036)	75 501	255 754

variant, the main memory usage is considerably higher than the disk space usage (around a factor of two) mainly because the dynamic data structures maintain vacancies that might be filled during future update operations.

We can expect different performances when updating very important roads (like motorways) or very unimportant ones (like urban streets, which are usually only relevant to very few connections). Therefore, for each of the four major road categories, we pick 1 000 edges at random. In addition, we randomly pick 1 000 edges irrespective of the road type. For each of these edge sets, we consider four types of updates: first, we add a traffic jam to each edge (by increasing the weight by 30 minutes); second, we cancel all traffic jams (by setting the original weights); third, we block all edges (by increasing the weights by 100 hours, which virtually corresponds to 'infinity' in our scenario); fourth, we multiply the weights by 10 in order to allow comparisons to [16]. For each of these cases, Tab. 2 gives the average update time per changed edge. We distinguish between two change set sizes: dealing with only one change at a time and processing 1 000 changes simultaneously.

As expected, the performance depends mainly on the selected edge and hardly on the type of update. The average execution times for a single update operation range between 40 ms (for motorways) and 2 ms (for urban streets). Usually, an update of a motorway edge requires updates of most levels of the overlay graph, while the effects of an urban-street update are limited to the lowest levels. We get a better performance when several changes are processed at once: for example, 1 000 random motorway segments can be updated in about 8 seconds. Note that such an update operation will be even more efficient when the involved edges belong to the same local area (instead of being randomly spread), which might be a common case in real-world applications.

Updating a Few Edge Weights (Mobile Scenario). Table 3 shows for the most difficult case (updating motorways) that using our modified query algorithm we can omit the comparatively expensive update operation and still get acceptable execution times, at least if only a moderate amount of edge weight changes occur. Additional experiments have confirmed that, similar to the results in Tab. 2, the performance does not depend on the update type (add 30 minutes, multiply by 10, ...), but on the edge type (motorway, urban street, ...) and, of course, on the number of updates.

Comparisons. Highway-node routing has similar preprocessing and query times as pure highway hierarchies [17, w/o distance table], but (in the static case) a significantly smaller memory overhead. Table 4 gives detailed numbers, and it also contains a comparison to the dynamic ALT approach [16] with 16 landmarks. We can conclude that as a stand-alone method, highway-node routing is (clearly) superior to dynamic ALT w.r.t. all studied aspects.[7]

6 Conclusion

Combining and considerably extending ideas of several previous static point-to-point route planning techniques yields a new static approach with extremely low space requirements, fast preprocessing, and very good query times. More important and innovative, however, is the fact that the approach can be extended to work in dynamic scenarios: we can efficiently react to events like traffic jams. Furthermore, we deal with the case that different cost functions are handled.

There is room to improve the performance both at the implementation and at the algorithmic level. In particular, better ways to select the highway-node sets might be found. The memory consumption of the dynamic variant can be considerably reduced by using a more space-efficient representation of the affected node sets. There is also room for additional functionality. For instance, we can adapt the algorithm from [19] to work with the new approach in order to support *many-to-many* shortest-path computations for exchangeable cost functions. An extension to a *time-dependent* scenario—where the edge weights depend on the time of day according to some function known in advance—is an important open problem.

References

1. Sanders, P., Schultes, D.: Engineering fast route planning algorithms. In: 6th Workshop on Experimental Algorithms (2007)
2. Fakcharoenphol, J., Rao, S.: Planar graphs, negative weight edges, shortest paths, and near linear time. J. Comput. Syst. Sci. 72(5), 868–889 (2006)
3. Klein, P.: Multiple-source shortest paths in planar graphs. In: 16th ACM-SIAM Symposium on Discrete Algorithms, SIAM, pp. 146–155 (2005)

[7] Note that our comparison concentrates on only one variant of dynamic ALT: different landmark sets can yield different tradeoffs. Also, better results can be expected when a lot of very small changes are involved. Moreover, dynamic ALT can turn out to be very useful in combination with other dynamic speedup techniques yet to come.

4. Schulz, F., Wagner, D., Weihe, K.: Dijkstra's algorithm on-line: an empirical case study from public railroad transport. ACM Journal of Experimental Algorithmics 5, 12 (2000)
5. Holzer, M., Schulz, F., Wagner, D.: Engineering Multi-Level Overlay Graphs for Shortest-Path Queries. In: Workshop on Algorithm Engineering and Experiments. In: Proceedings in Applied Mathematics. SIAM pp. vol. 129, pp. 156–170 (2006)
6. Holzer, M., Schulz, F., Wagner, D.: Engineering multi-level overlay graphs for shortest-path queries. invited for ACM Journal of Experimental Algorithmics (special issue Alenex 2006) (2007)
7. Bauer, R.: Dynamic speed-up techniques for Dijkstra's algorithm. Diploma Thesis, Universität Karlsruhe (TH) (2006)
8. Bruera, F., Cicerone, S., D'Angelo, G., Di Stefano, G., Frigioni, D.: On the dynamization of shortest path overlay graphs. Technical Report 0026, ARRIVAL (2006)
9. 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)
10. Sanders, P., Schultes, D.: Engineering highway hierarchies. In: Azar, Y., Erlebach, T. (eds.) ESA 2006. LNCS, vol. 4168, pp. 804–816. Springer, Heidelberg (2006)
11. Gutman, R.: Reach-based routing: A new approach to shortest path algorithms optimized for road networks. In: 6th Workshop on Algorithm Engineering and Experiments. pp. 100–111 (2004)
12. Goldberg, A., Kaplan, H., Werneck, R.: Reach for A^*: Efficient point-to-point shortest path algorithms. In: Workshop on Algorithm Engineering & Experiments, Miami pp. 129–143 (2006)
13. Bast, H., Funke, S., Matijevic, D., Sanders, P., Schultes, D.: Intransit to constant time shortest-path queries in road networks. In: Workshop on Algorithm Engineering and Experiments (2007)
14. Hart, P.E., Nilsson, N.J., Raphael, B.: A formal basis for the heuristic determination of minimum cost paths. IEEE Transactions on System Science and Cybernetics 4(2), 100–107 (1968)
15. Goldberg, A.V., Harrelson, C.: Computing the shortest path: A^* meets graph theory. Technical Report MSR-TR-2004-24, Microsoft Research (2004)
16. Delling, D., Wagner, D.: Landmark-based routing in dynamic graphs. In: 6th Workshop on Experimental Algorithms (2007)
17. Delling, D., Sanders, P., Schultes, D., Wagner, D.: Highway hierarchies star. In: 9th DIMACS Implementation Challenge, `http://www.dis.uniroma1.it/~challenge9/` (2006)
18. R Development Core Team: R: A Language and Environment for Statistical Computing. `http://www.r-project.org` (2004)
19. 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 (2007)