Engineering Fast Route Planning Algorithms[⋆]

Peter Sanders and Dominik Schultes

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

Abstract. Algorithms for route planning in transportation networks have recently undergone a rapid development, leading to methods that are up to one million times faster than Dijkstra's algorithm. We outline ideas, algorithms, implementations, and experimental methods behind this development. We also explain why the story is not over yet because dynamically changing networks, flexible objective functions, and new applications pose a lot of interesting challenges.

1 Introduction

Computing an optimal route in a transportation network between specified source and target nodes is one of the showpieces of real-world applications of algorithmics. We frequently use this functionality when planning trips with cars or public transportation. There are also many applications like logistic planning or traffic simulation that need to solve a huge number of shortest-path queries in transportation networks. In most of this paper we focus on the simplest case, a static *road* network with a fixed cost for each edge. The cost function may be any mix of travel time, distance, toll, energy consumption, scenic value, ... associated with the edges. Some of the techniques described below work best if the cost function is positively correlated with travel time. The task is to compute the costs of optimal paths between arbitrary source-target pairs. Some preprocessing is allowed but it has to be sufficiently fast and space efficient to scale to the road network of a continent.

The main part of this paper is Section 2, which explains the ideas behind several practically successful speedup techniques for static routing in road networks. Section 3 makes an attempt to summarize the development of performance over time. In Section 4 we outline generalizations for public transportation, time-dependent edge weights, outputting optimal paths, and dynamically changing networks. Section 5 describes some experiences we made with implementing route planning algorithms for large networks. Then, Section 6 explains our experimental approach giving some examples by applying it to the algorithms we implemented. We conclude in Section 7 with a discussion of some future challenges.

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

2 Static Routing in Large Road Networks

We consider a directed graph $G = (V, E)$ with n nodes and $m = \Theta(n)$ edges. An edge (u, v) has the nonnegative edge weight $w(u, v)$. A shortest-path query between a source node s and a target node t asks for the minimal weight $d(s, t)$ of any path from s to t. In static routing, the edge weights do not change so that it makes sense to perform some *precomputations*, store their results, and use this information to accelerate the queries. Obviously, there is some tradeoff between query time, preprocessing time, and space for preprocessed information. In particular, for large road networks it would be prohibitive to precompute and store shortest paths between all pairs of nodes.

2.1 "Classical Results"

Dijkstra's Algorithm [1]—the classical algorithm for route planning—maintains an array of *tentative distances* $D[u] \geq d(s, u)$ for each node. The algorithm *visits* (or *settles*) the nodes of the road network in the order of their distance to the source node and maintains the invariant that $D[u] = d(s, u)$ for visited nodes. We call the rank of node u in this order its *Dijkstra rank* $\mathrm{rk}_s(u)$. When a node u is visited, its outgoing edges (u, v) are *relaxed*, i.e., $D[v]$ is set to $\min(D[v], d(s, u) + w(u, v))$. Dijkstra's algorithm terminates when the target node is visited. The size of the search space is $O(n)$ and $n/2$ (nodes) on the average. We will assess the quality of route planning algorithms by looking at their *speedup* compared to Dijkstra's algorithm, i.e., how many times faster they can compute shortest-path distances.

Priority Queues. Dijkstra's algorithm can be implemented using $O(n)$ priority queue operations. In the comparison based model this leads to $O(n \log n)$ execution time. In other models of computation (e.g. [2]) and on the average [3], better bounds exist. However, in practice the impact of priority queues on performance for large road networks is rather limited since cache faults for accessing the graph are usually the main bottleneck. In addition, our experiments indicate that the impact of priority queue implementations diminishes with advanced speedup techniques since these techniques at the same time introduce additional overheads and dramatically reduce the queue sizes.

Bidirectional Search executes Dijkstra's algorithm simultaneously forward from the source and backwards from the target. Once some node has been visited from both directions, the shortest path can be derived from the information already gathered [4]. In a road network, where search spaces will take a roughly circular shape, we can expect a speedup around two —one disk with radius $d(s, t)$ has twice the area of two disks with half the radius. Bidirectional search is important since it can be combined with most other speedup techniques and, more importantly, because it is a necessary ingredient of the most efficient advanced techniques.

Geometric Goal Directed Search (A^*). The intuition behind goal directed search is that shortest paths 'should' lead in the general direction of the target. A^* search [5] achieves this by modifying the weight of edge (u, v) to $w(u, v) - \pi(u) + \pi(v)$ where $\pi(v)$ is a lower bound on $d(v, t)$. Note that this manipulation shortens edges that lead towards the target. Since the added and subtracted *vertex potentials* $\pi(v)$ cancel along any path, this modification of edge weights preserves shortest paths. Moreover, as long as all edge weights remain nonnegative, Dijkstra's algorithm can still be used. The classical way to use A^* for route planning in road maps estimates $d(v, t)$ based on the Euclidean distance between v and t and the average speed of the fastest road anywhere in the network. Since this is a very conservative estimation, the speedup for finding quickest routes is rather small. Goldberg et al. [6] even report a *slow-down* of more than a factor of two since the search space is not significantly reduced but a considerable overhead is added.

Heuristics. In the last decades, commercial navigation systems were developed which had to handle ever more detailed descriptions of road networks on rather low-powered processors. Vendors resolved to heuristics still used today that do not give any performance guarantees: use A^* search with estimates on $d(u, t)$ rather than lower bounds; do not look at 'unimportant' streets, unless you are close to the source or target [7]. The latter heuristic needs careful hand tuning of road classifications to produce reasonable results but yields considerable speedups.

2.2 Exploiting Hierarchy

Small Separators. Road networks are almost planar, i.e., most edges intersect only at nodes. Hence, 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 [8,9] for directed planar graphs without negative cycles. Queries accurate within a factor $(1 + \epsilon)$ can be answered in near constant time using $O((n \log n)/\epsilon)$ space and preprocessing time [10]. Most of these theoretical approaches look difficult to use in practice since they are complicated and need superlinear space.

The first published practical approach to fast route planning [11] uses a set of nodes V_1 whose removal partitions the graph $G = G_0$ into small components. Now consider the *overlay graph* $G_1 = (V_1, E_1)$ where edges in E_1 are *shortcuts* corresponding to shortest paths in G that do not contain nodes from V_1 in their interior. Routing can now be restricted to G_1 and the components containing s and t respectively. This process can be iterated yielding a multi-level method. A limitation of this approach is that the graphs at higher levels become much more dense than the input graphs thus limiting the benefits gained from the hierarchy. Also, computing small separators and shortcuts can become quite costly for large graphs.

Reach-Based Routing. Let $R(v) := \max_{s,t \in V} R_{st}(v)$ denote the *reach* of node v where $R_{st}(v) := \min(d(s,v), d(v,t))$. Gutman [12] observed that a shortest-path search can be stopped at nodes with a reach too small to get to source or target from there. Variants of reach-based routing work with the reach of edges or characterize reach in terms of geometric distance rather than shortest-path distance. The first implementation had disappointing speedups (e.g. compared to [11]) and preprocessing times that would be prohibitive for large networks.

Highway Hierarchies. (HHs) [13,14] group nodes and edges in a hierarchy of levels by alternating between two procedures: Contraction (i.e., node reduction) removes low degree nodes by bypassing them with newly introduced shortcut edges. In particular, all nodes of degree one and two are removed by this process. Edge reduction removes *non-highway edges*, i.e., edges that only appear on shortest paths *close* to source or target. More specifically, every node v has a neighborhood radius $r(v)$ we are free to choose. An edge (u,v) is a highway edge if it belongs to some shortest path P from a node s to a node t such that (u,v) is neither fully contained in the neighborhood of s nor in the neighborhood of t, i.e., $d(s,v) > r(s)$ and $d(u,t) > r(t)$. In all our experiments, neighborhood radii are chosen such that each neighborhood contains a certain number H of nodes. H is a tuning parameter that can be used to control the rate at which the network shrinks. The query algorithm is very similar to bidirectional Dijkstra search with the difference that certain edges need not be expanded when the search is sufficiently far from source or target. HHs were the first speedup technique that could handle the largest available road networks giving query times measured in milliseconds. There are two main reasons for this success: Under the above contraction routines, the road network shrinks in a geometric fashion from level to level and remains sparse and near planar, i.e., levels of the HH are in some sense *self similar*. The other key property is that preprocessing can be done using limited local searches starting from each node. Preprocessing is also the most nontrivial aspect of HHs. In particular, long edges (e.g. long-distance ferry connections) make simple minded approaches far too slow. Instead we use fast heuristics that compute a superset of the set of highway edges.

Routing with HHs is similar to the heuristics used in commercial systems. The crucial difference is that HHs are guaranteed to find the optimal path. This qualitative improvement actually make HHs *much faster* than the heuristics. The latter have to make a precarious compromise between quality and size of the search space that relies on manual classification of the edges into levels of the hierarchy. In contrast, after setting a few quite robust tuning parameters, HH-preprocessing automatically computes a hierarchy aggressively tuned for high performance.

Advanced Reach-Based Routing. It turns out that the preprocessing techniques developed for HHs can be adapted to preprocessing reach information [15]. This makes reach computation faster and more accurate. More importantly, shortcuts make queries more effective by reducing the number of nodes traversed and by reducing the reach-values of the nodes bypassed by shortcuts.

Reach-based routing is slower than HHs both with respect to preprocessing time and query time. However, the latter can be improved by a combination with goal-directed search to a point where both methods have similar performance.

Highway-Node Routing. In [16] we generalize the multi-level routing scheme with overlay graphs so that it works with arbitrary sets of nodes rather than only with separators. This is achieved using a new query algorithm that stalls suboptimal branches of search on lower levels of the hierarchy. By using only *important* nodes for higher levels, we achieve query performance comparable to HHs. Preprocessing is done in two phases. In the first phase, nodes are classified into levels. We currently derive this information from a HH. In the second phase, we recursively compute the shortcuts bottom up. Shortcuts from level ℓ are found by local searches in level $\ell - 1$ starting from nodes in level ℓ. This second phase is very fast and easy to update when edge weights change.

Distance Tables. For HHs the network size shrinks geometrically from level to level. Once a level L has size $\Theta(\sqrt{n})$, we can afford to precompute and store a complete distance table for nodes in level L [14]. Using this table, we can stop a HH search when it has reached level L. To compute the shortest-path distance, it then suffices to lookup all shortest-path distances between nodes entering level L in forward and backward search respectively. Since the number of entrance nodes is not very large, one can achieve a speedup close to two compared to pure HH search.

Transit Node Routing precomputes not only a distance table for important (*transit*) nodes but also all relevant connections between the remaining nodes and the transit nodes [17,18]. Since it turns out that only about ten such *access connections* are needed per node, one can 'almost' reduce routing in large road networks to about 100 table lookups. Interestingly, the difficult queries are now the local ones where the shortest path does not touch any transit node. We solve this problem by introducing several *layers* of transit nodes. Between lower layer transit nodes, only those routes need to be stored that do not touch the higher layers. Transit node routing (e.g., using appropriate levels of a HH for transit node sets) reduces routing times to a few microseconds at the price of preprocessing times an order of magnitude larger than HHs alone.

2.3 Advanced Goal-Directed Search

Edge Labels. The idea behind edge labels is to precompute information for an edge e that specifies a set of nodes $M(e)$ with the property that $M(e)$ is a superset of all nodes that lie on a shortest path starting with e. In an s–t query, an edge e need not be relaxed if $t \notin M(e)$. In [11], $M(e)$ is specified by an *angular range*. More effective is information that can distinguish between long range and short range edges. In [19] many *geometric containers* are evaluated. Very good performance is observed for axis parallel rectangles. A disadvantage of geometric containers is that they require a complete all-pairs shortest-path computation. Faster precomputation is possible by partitioning the graph into k regions that have similar size and only a small number of boundary nodes. Now $M(e)$ is represented as a k-vector of *edge flags* [20,21,22] where flag i indicates whether there is a shortest path containing e that leads to a node in region i. Edge flags can be computed using a single-source shortest-path computation from all boundary nodes of the regions. In [23] a faster variant of the preprocessing algorithm is introduced that takes advantage of the fact that for close boundary nodes the respective shortest-path trees are very similar.

Landmark A^*. Using the triangle inequality, quite strong bounds on shortest-path distances can be obtained by precomputing distances to a set of around 20 *landmark* nodes that are well distributed over the far ends of the network [6,24]. Using reasonable space and much less preprocessing time than for edge labels, these lower bounds yield considerable speedup for route planning.

Precomputed Cluster Distances (PCD). In [25], we give a different way to use precomputed distances for goal-directed search. We partition the network into clusters and then precompute the shortest connection between any pair of clusters U and V, i.e., $\min_{u \in U, v \in V} d(u,v)$. PCDs cannot be used together with A^* search since reduced edge weights can become negative. However, PCDs yield upper and lower bounds for distances that can be used to prune search. This gives speedup comparable to landmark-A^* using less space. Using the many-to-many routing techniques outlined in Section 4, cluster distances can also be computed efficiently.

2.4 Combinations

Bidirectional search can be profitably combined with almost all other speedup techniques. Indeed, it is a required ingredient of highway hierarchies, transit and highway-node routing and it gives more than the anticipated factor two for reach-based routing and edge flags. Willhalm et al. have made a systematic comparison of combinations of pre-2004 techniques [26,27]. Landmark A^* harmonizes very well with reach-based routing [15] whereas it gives only a small additional speedup when combined with HHs [28]. The reason is that in HHs, the search cannot be stopped when the search frontiers meet. However, the same approach is very effective at speeding up approximate shortest-path queries.

3 Chronological Summary—The Horse Race

In general it is difficult to compare speedup techniques even when restricting to road networks because there is a complex tradeoff between query time, preprocessing time and space consumption that depends on the network, on the objective function, and on the distribution of queries. Still, we believe that some ranking helps to compare the techniques. To keep things manageable, we will restrict ourselves to average query times for computing optimal travel times in one

Table 1. Chronological development of the fastest speedup techniques. As date for the first publication, we usually give the submission deadline of the respective conference. If available, we always selected measurements for the European road network even if they were conducted after the first publication. Otherwise, we *linearly* extrapolated the preprocessing times to the size of Europe, which can be seen as a *lower bound.* Note that not all speedup techniques have been preprocessed on the same machine.

method	first	date	data	size	space	preproc.	speedup
	pub.	mm/yy	from	$n/10^6$	[B/node]	[min]	
separator multi-level	[11]	04/99	[30]	0.1	?	$>$ 5 400	52
edge flags (basic)	[20]	03/04	[31]	6	13	299	523
landmark A^*	[6]	07/04	[32]	18	72	13	28
edge flags	[21,22]	01/05	[23]	1	141	2 163	1 470
HHs (basic)	[13]	04/05	[13]	18	29	161	2 645
reach + shortc. + A^*	[15]	10/05	[32]	18	82	1 625	1 559
	[32]	08/06	[32]	18	32	144	3 830
HHs	[14]	04/06	[14]	18	27	13	4 002
HHs + dist. tab. (mem)	[14]	04/06	[14]	18	17	55	4 582
HHs + dist. tab.	[14]	04/06	[14]	18	68	15	8 320
HHs + dist. tab. + A^*	[28]	08/06	[28]	18	76	22	11 496
high-perf. multi-level	[33]	06/06	[34]	18	181	11 520	401 109
transit nodes (eco)	[17]	10/06	[17]	18	110	46	471 881
transit nodes (gen)	[17]	10/06	[17]	18	251	164	1 129 143
highway nodes (mem)	[16]	01/07	[16]	18	2	24	4 079

of the largest networks that have been widely used—the road network of (Western) Europe provided by the company PTV AG and also used (in a slightly different version) in the 9th DIMACS Implementation Challenge [29]. We take the liberty to speculate on the performance of some older methods that have never been been run on such large graphs and whose actual implementations might fail when one would attempt it. In Tab. 1 we list speedup techniques in chronological order that are 'best' with respect to speedup for random queries and the largest networks tackled at that point. Sometimes we list variants with slower query times if they are considerably better with respect to space consumption or manageable graph size.

Before [11] the best method would have been a combination of bidirectional search with geometric A^* yielding speedups of 2–3 over unidirectional Dijkstra. The separator-based multi-level method from [11] can be expected to work even for large graphs if implemented carefully. Computing geometric containers [11,19] is still infeasible for large networks. Otherwise, they would achieve much larger speedups then the separator-based multi-level method. So far, computing edge flags has also been too expensive for Europe and the USA but speedups beyond 1 470 have been observed for a graph with one million nodes [23]. Landmark A^* works well for large graphs and achieves average speedup of 28 using reasonable space and preprocessing time [6]. The implementation of HHs [13] was the first that was able to handle Europe and the USA. This implementation wins over all previous methods in almost all aspects. A combination of reach-based routing with landmark A^* [15] achieved better query times for the USA at the price of a considerably higher preprocessing time. At first, that code did not work well on the European network because it is difficult to handle the present long-distance ferry connections, but later it could be considerably improved [32]. By introducing distance tables and numerous other improvements, highway hierarchies took back the lead in query time [14] at the same time using an order of magnitude less preprocessing time than [13]. The cycle of innovation accelerated even further in 2006. Müller [33] aggressively precomputes the pieces of the search space needed for separator-based multi-level routing. At massive expense of space and preprocessing time, this method can achieve speedups around 400 000. (The original implementation cannot directly measure this because it has large overheads for disk access and parsing of XML-data). Independently, transit node routing was developed [17], that lifts the speedup to six orders of magnitude and completely replaces Dijkstra-like search by table lookups. Since transit node routing needs more space and preprocessing time than other methods, the story is not finished yet. For example, [16] achieves speedups comparable to HHs, using only a few bytes per node.

4 Generalizations

Many-to-Many Routing. In several applications we need complete distance tables between specified sets of source nodes S and target nodes T. For example, in logistics optimization, traffic simulation, and also within preprocessing techniques [25,17]. HHs (and other non-goal-directed bidirectional search methods [11,15,16]) can be adapted in such a way that only a single forward search from each source node and a single backward search from each target node is needed [35]. The basic idea is quite simple: Store the backward search spaces. Arrange them so that each node v stores an array of pairs of the form $(t, d(v,t))$ for all target nodes that have v in their backward search space. When a forward search from s settles a node v, these pairs are scanned and used to update the tentative distance from s to t. This is very efficient because the intersection between any two forward and backward search spaces is only around 100 for HHs and because scanning an array is much faster than priority queue operations and edge relaxations governing the cost of Dijkstra's algorithm. For example, for $|S| = |T| = 10\,000$, the implementation in [35] needs only one minute.

Outputting Paths. The efficiency of many speedup techniques stems from introducing shortcut edges and distance table entries that replace entire paths in the original graph [11,13,15,14,33,17]. A disadvantage of these approaches is that the search will output only a 'summary description' of the optimal path that involves shortcuts. Fortunately, it is quite straightforward to augment the shortcuts with information for unpacking them [28,35,17]. Since we can afford to precompute unpacked representations of the most frequently needed long-distance shortcuts, outputting the path turns out to be up to four times *faster* than just traversing the edges in the original graph.

Flexible Objective Functions. The objective function in road networks depends in a complex way on the vehicle (fast? slow? too heavy for certain bridges?, ...) the behavior and goals of the driver (cost sensitive? thinks he is fast?, ...), the load, and many other aspects. While the appropriate edge weights can be computed from a few basic parameters, it is not feasible to perform preprocessing for all conceivable combinations. Currently, our best answer to this problem is highway-node routing [16]. Assuming that the important nodes are important for any reasonable objective function, only the second phase of preprocessing needs to be repeated. This is an order of magnitude faster than computing a HH.

Dynamization. In online car navigation, we want to take traffic jams etc. into account. On the first glance, this is the death blow for most speedup techniques since even a single traffic jam can invalidate any of the precomputed information. However, we can try to selectively update only the information affected by the traffic jam and/or relevant to the queries at hand. Several solutions are proposed at this conference. Landmark A^* can be dynamized either by noticing that lower bounds remain valid when edge weights can only increase, or by using known dynamic graph algorithms for updating the shortest-path trees from the landmarks [36]. We have developed highway-node routing for this purpose [16] because it allows fast and easy updates (2–40 ms per changed edge weight depending on the importance of the edge).

Public Transportation and Time-Dependent Edge Weights. The standard query in public transportation asks for the *earliest arrival* at the target node t given a departure time and the source node s. This means we are (explicitly or implicitly) searching in a time-dependent network where nodes are some point in space-time. This means that bidirectional search cannot be used out of the box since we do not know the target node in the time-dependent network. This is puzzling because the most successful schemes described above use bidirectional search. This leaves us with the choice to use the most effective unidirectional method, or to somehow make bidirectional search work. An obvious fix is to *guess* the arrival time. This can be done using binary search and there are many ways to tune this (e.g. by interpolation search).

Parallelization. Most preprocessing techniques and many-to-many routing are easy to parallelize [35]. Parallelizing the queries seems difficult and unnecessary (beyond executing forward and backward search in parallel) because the search spaces are already very small when using one of the best available techniques.

5 Implementation

Advanced algorithms for routing in road networks require thousands of lines of well written code and hence require considerable programming skill. In particular, it is not trivial to make the codes work for large networks. Here is an incomplete list of problems and complications that we have seen in routing projects: Graphs have to be glued together from several files. Tools for reading files crash for large graphs. Algorithm library code cannot handle large graphs at all. The code slows down by factor six when switching from a custom graph representation to an algorithm library. 32-bit code will not work. Libraries do not work with 64-bit code.

Our conclusion from these experiences was to design our own graph data structures adapted to the problem at hand. We use C++ with encapsulated abstract data types. Templates and inline functions make this possible without performance penalties.

Although speedup techniques developed by algorithmicists come with high level arguments why they should yield optimal paths, few come with a detailed correctness proof.[1] There are plenty of things that can go wrong both with the algorithms and their implementations. For example, we had several cases where the algorithm considered was only correct when all shortest paths are unique. The implementation can help here with extensive consistency checks in assertions and experiments that are always checked against naive implementations, i.e., queries are checked against Dijkstra's algorithm and fast preprocessing algorithms are checked against naive or old implementations. On the long run one also needs a flexible visualization tool that can draw pieces of large graphs, paths, search spaces, and node sets. Since we could not find tools for this purpose that scale to large road networks, we implemented our own system [37].

6 Experiments

Before 2005, speedup techniques were difficult to compare since studies were either based on very small graphs or on proprietary data that could not be used by other groups. In particular, for 'newcomers' it was almost impossible to start working in this field. In [13] we were able to obtain two large road networks for the subcontinents Western Europe and the USA. The European network was made available for scientific use by the company PTV AG. We extracted the USA network from publicly available geographical data [38]. Since then, variants of these graphs have been used for most studies. We view it as likely that the sudden availability of data and the fast rate of innovation since then are not a coincidence. These networks are not directly annotated with edge weights but with lengths and road categories. By setting average speeds for each road category one can obtain realistic estimates of travel time.

Another important issue are which queries should be measured. The obvious choice is to use randomly selected node pairs on the largest available graph.

[1] We are working on one for HHs.

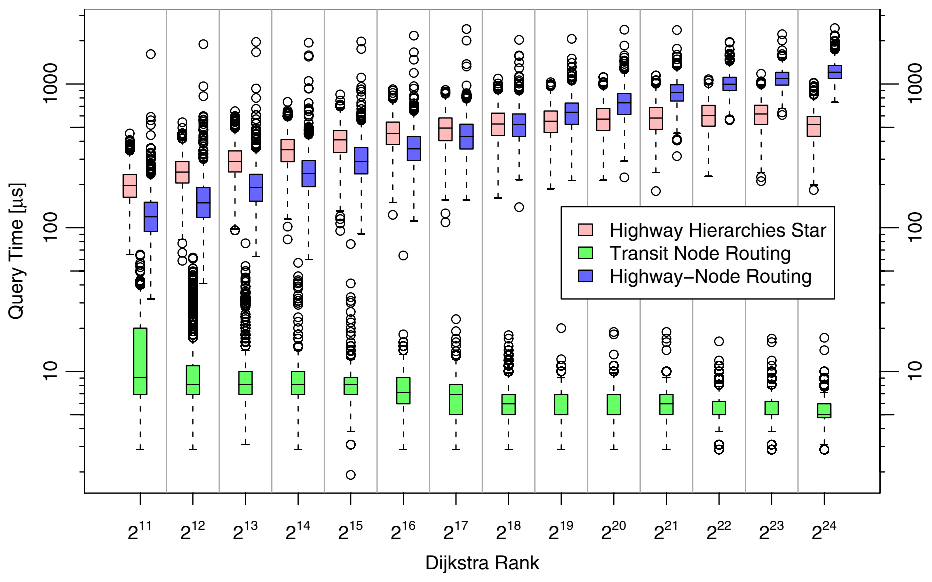

Fig. 1. Query performance of various speedup techniques against Dijkstra rank. The median speedups of HHs and highway-node routing cross at $r = 2^{18}$ since HHs are augmented with distance tables and goal direction. These techniques are particularly effective for large r and could also be adapted to highway-node routing.

Although this is a meaningful number, it is not quite satisfactory since most queries will produce very long paths (thousands of kilometers) that are actually rare in practice. Other studies therefore use random queries on a variety of subgraphs. However, this leads to a plethora of arbitrary choices that make it difficult to compare results. In particular, authors will always be tempted to choose only those subgraphs for which their method performs well.

Sets of real world queries would certainly be interesting, but so far we do not have them and it is also unlikely that a sample taken from one server is actually representative for the entire spectrum of route planning applications. We therefore chose a more systematic approach [13] that has also been adopted in several other studies: We generate a random query with a specified ‘locality’ r by choosing a random starting node s, and a target node t with Dijkstra rank $\text{rk}_s(t) = r$ (i.e., the r-th node visited by a Dijkstra search from s). In our studies, we generate many such queries for each r which is a power of two. We then plot the distribution with median, quartiles, and outliers for each of these values of r. For the European road network, Fig. 1 shows the results for highway hierarchies combined with a distance table and goal-directed search, transit node routing, and highway-node routing. We view it as quite important to give information on the entire distribution since some speedup techniques have large fluctuations in query time.

In some cases, e.g., for HHs, it is also possible to compute good upper bounds on the search space size of *all* queries that can ever happen for a given graph [14]. We view this as a quite good surrogate for the absence of meaningful worst case upper bounds that would apply to all conceivable networks.

7 Conclusions and Open Problems

Speedup techniques for routing in static road networks have made tremendous progress in the last few years. Were it not for challenging applications such as logistics planning and traffic simulation, we could even say that the methods available now are *too* fast since other overheads like displaying routes or transmitting them over the network are the bottleneck once the query time is below a few milliseconds.

A major challenge is to close the gap to theory, e.g., by giving meaningful characterizations of 'well-behaved' networks that allow provably good worst-case bounds. In particular, we would like to know for which networks the existing techniques will also work, e.g., for communication networks, VLSI design, social networks, computer games, graphs derived from geometric routing problems, . . .

Even routing techniques themselves are not quite finished yet. For example, we can look at better ways to select transit and highway nodes. We could also try to integrate edge labels with hierarchical routing schemes so that hierarchies help to approximate edge labels that in turn allow strong goal direction for queries.

Perhaps the main academic challenge is to go beyond static point-to-point routing. Public transportation and road networks with *time-dependent* travel times are an obvious generalization that should also be combined with updates of edge weights due to traffic jams. Further beyond that, we want multi-criteria optimization for individual paths and we want to compute social optima and Nash-equilibria taking the entire traffic in an area into account.

The main practical issue is how to transfer the academic results into applications. Many details have to be taken care of, like turn penalties, implementations on mobile devices, user specific objective functions, and compatibility with existing parts of the applications. The difficulty here is not so much scientific but one of finding the right approach to cooperation between academia and industry.

Acknowledgements

We would like to thank our coauthors on route planning, Holger Bast, Daniel Delling, Stefan Funke, Sebastian Knopp, Domagoj Matijevic, Jens Maue, Frank Schulz, and Dorothea Wagner for their valuable contributions. We also had many interesting discussions with Rob van den Berg, Sebastian Egner, Andrew Goldberg, Joaquim Gromicho, Martin Holzer, Stefan Hug, Ali Nowbakht Irani, Ekkehard Köhler, Ulrich Lauther, Ramon Lentink, Rolf Möhring, Kirill Müller, Matthias Müller-Hannemann, Paul Perdon, Heiko Schilling, Mikkel Thorup, Jaques Verrier, Peter Vortisch, Renato Werneck, and Thomas Willhalm.

References

1. Dijkstra, E.W.: A note on two problems in connexion with graphs. Numerische Mathematik 1, 269–271 (1959)
2. Thorup, M.: Integer priority queues with decrease key in constant time and the single source shortest paths problem. In: 35th ACM Symposium on Theory of Computing. pp. 149–158 (2003)
3. Meyer, U.: Single-source shortest-paths on arbitrary directed graphs in linear average-case time. In: 12th Symposium on Discrete Algorithms. pp. 797–806 (2001)
4. Dantzig, G.B.: Linear Programming and Extensions. Princeton University Press, Princeton (1962)
5. 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)
6. Goldberg, A.V., Harrelson, C.: Computing the shortest path: A^* meets graph theory. In: 16th ACM-SIAM Symposium on Discrete Algorithms, pp. 156–165. ACM Press, New York (2005)
7. Ishikawa, K., Ogawa, M., Azume, S., Ito, T.: Map Navigation Software of the Electro Multivision of the '91 Toyota Soarer. In: IEEE Int. Conf. Vehicle Navig. Inform. Syst. pp. 463–473 (1991)
8. Fakcharoenphol, J., Rao, S.: Planar graphs, negative weight edges, shortest paths, and near linear time. J. Comput. Syst. Sci. 72(5), 868–889 (2006)
9. Klein, P.: Multiple-source shortest paths in planar graphs. In: 16th ACM-SIAM Symposium on Discrete Algorithms, SIAM, pp. 146–155 (2005)
10. Thorup, M.: Compact oracles for reachability and approximate distances in planar digraphs. In: 42nd IEEE Symposium on Foundations of Computer Science. pp. 242–251 (2001)
11. Schulz, F., Wagner, D., Weihe, K.: Dijkstra's algorithm on-line: An empirical case study from public railroad transport. In: Vitter, J.S., Zaroliagis, C.D. (eds.) WAE 1999. LNCS, vol. 1668, pp. 110–123. Springer, Heidelberg (1999)
12. 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)
13. 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)
14. Sanders, P., Schultes, D.: Engineering highway hierarchies. In: Azar, Y., Erlebach, T. (eds.) ESA 2006. LNCS, vol. 4168, pp. 804–816. Springer, Heidelberg (2006)
15. Goldberg, A., Kaplan, H., Werneck, R.: Reach for A^*: Efficient point-to-point shortest path algorithms. In: Workshop on Algorithm Engineering & Experiments, Miami (2006) 129–143
16. Schultes, D., Sanders, P.: Dynamic highway-node routing. In: 6th Workshop on Experimental Algorithms (2007)
17. Bast, H., Funke, S., Matijevic, D., Sanders, P., Schultes, D.: In: transit to constant time shortest-path queries in road networks. In: Workshop on Algorithm Engineering and Experiments (2007)
18. Bast, H., Funke, S., Sanders, P., Schultes, D.: Fast routing in road networks with transit nodes. Science (2007) to appear
19. Wagner, D., Willhalm, T.: Geometric speed-up techniques for finding shortest paths in large sparse graphs. In: Di Battista, G., Zwick, U. (eds.) ESA 2003. LNCS, vol. 2832, pp. 776–787. Springer, Heidelberg (2003)

20. 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)
21. 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 (2005)
22. Möhring, R.H., Schilling, H., Schütz, B., Wagner, D., Willhalm, T.: Partitioning graphs to speed up Dijkstra's algorithm. In: 4th International Workshop on Efficient and Experimental Algorithms. pp. 189–202 (2005)
23. Köhler, E., Möhring, R.H., Schilling, H.: Fast point-to-point shortest path computations with arc-flags. In: 9th DIMACS Implementation Challenge [29] (2006)
24. Goldberg, A.V., Werneck, R.F.: An efficient external memory shortest path algorithm. In: Workshop on Algorithm Engineering and Experimentation. pp. 26–40 (2005)
25. Maue, J., Sanders, P., Matijevic, D.: Goal directed shortest path queries using Precomputed Cluster Distances. In: Àlvarez, C., Serna, M. (eds.) WEA 2006. LNCS, vol. 4007, pp. 316–328. Springer, Heidelberg (2006)
26. Holzer, M., Schulz, F., Willhalm, T.: Combining speed-up techniques for shortest-path computations. In: Ribeiro, C.C., Martins, S.L. (eds.) WEA 2004. LNCS, vol. 3059, pp. 269–284. Springer, Heidelberg (2004)
27. Willhalm, T.: Engineering Shortest Path and Layout Algorithms for Large Graphs. PhD thesis, Universität Karlsruhe (TH), Fakultät für Informatik (2005)
28. Delling, D., Sanders, P., Schultes, D., Wagner, D.: Highway hierarchies star. In: 9th DIMACS Implementation Challenge [29] (2006)
29. 9th DIMACS Implementation Challenge: Shortest Paths. http://www.dis.uniroma1.it/~challenge9 (2006)
30. 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)
31. Lauther, U.: An experimental evaluation of point-to-point shortest path calculation on roadnetworks with precalculated edge-flags. In: 9th DIMACS Implementation Challenge [29] (2006)
32. Goldberg, A.V., Kaplan, H., Werneck, R.F.: Better landmarks within reach. In: 9th DIMACS Implementation Challenge [29] (2006)
33. Müller, K.: Design and implementation of an efficient hierarchical speed-up technique for computation of exact shortest paths in graphs. Master's thesis, Universtät Karlsruhe supervised by Delling, D., Holzer, M., Schulz, F., Wagner, D.: (2006)
34. Delling, D., Holzer, M., Müller, K., Schulz, F., Wagner, D.: High-performance multi-level graphs. In: 9th DIMACS Implementation Challenge [29] (2006)
35. 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)
36. Delling, D., Wagner, D.: Landmark-based routing in dynamic graphs. In: 6th Workshop on Experimental Algorithms (2007)
37. Bingmann, T.: Visualisierung sehr großer Graphen. Student Research Project, Universität Karlsruhe, supervised by Sanders, P., Schultes, D.: (2006)
38. U.S. Census Bureau, Washington, DC: UA Census 2000 TIGER/Line Files. http://www.census.gov/geo/www/tiger/tigerua/ua_tgr2k.html (2002)