Car or Public Transport—Two Worlds

Hannah Bast

Max-Planck-Institute for Informatics, Saarbrücken, Germany

Abstract. There are two kinds of people: those who travel by car, and those who use public transport.[1] The topic of this article is to show that the algorithmic problem of computing the fastest way to get from A to B is also surprisingly different on road networks than on public transportation networks.

On road networks, even very large ones like that of the whole of Western Europe, the shortest path from a given source to a given target can be computed in just a few microseconds. Lots of interesting speed-up techniques have been developed to this end, and we will give an overview over the most important ones.

Public transportation networks can be modeled as graphs just like road networks, and most algorithms designed for road networks can be applied for public transportation networks as well. They just happen to perform not nearly as well, and to date we do not know how to route similarly fast on large public transportation networks as we can on large road networks.

The reasons for this are interesting and non-obvious, and it took us a long time to fully comprehend them. Once understood, they are relatively easy to explain, however, and that is what we want to do in this article. Oh, and by the way, happy birthday, Kurt!

1 Introduction

The last five years have seen an exciting surge of research on routing algorithms for large transportation networks. Most of this work has been done on road networks, but some of it was also considering public transportation networks.

Both road networks and public transportation networks can be very naturally modeled as directed graphs. For a road network, each node corresponds to a junction, where two or more road segments meet, and the arcs of the graph correspond to road segments. The cost of an arc is simply the time it takes to travel across the respective road segment. A shortest path in this graph then corresponds to the fastest way to get from a point A to a point B.

Public transportation networks are modeled in a similar way, except that besides the spatial information we also have to deal with time schedules. In the simplest and most natural model, each node corresponds to a departure or arrival

[1] Admittedly, there are a few people using both modes of transportation from time to time, but not that many.

event at a particular station. For example, a node might stand for the event of ICE 500 arriving at Mannheim Hauptbahnhof at 21:24.[2] Arcs between nodes then either correspond to waiting from one event to the next at a particular station, or to taking a particular train (or bus or ...) from one station at a particular time to another station. The cost of an arc is the respective waiting or travel time, so that a shortest path in this graph corresponds to the fastest way to get from a particular station A at a particular time t_A to a particular station B at a particular time t_B.

We will come back to these two models in Section 2, giving slightly more detail and commenting on possible refinements there.

1.1 Dijkstra's Algorithm

The method of choice for computing the shortest path from a give source node to a given target node in a given graph is Dijkstra's algorithm, which dates back to the 1950s [1]. In a nutshell, Dijkstra's algorithm works as follows. Each node is assigned a tentative cost, which is initially 0 for the source node and ∞ for all other nodes in the graph. The algorithm then starts from the source node, and visits all outgoing arcs from there. For each such outgoing arc, it checks whether via this arc it can reach the node at the other side (the so-called *tail*) of the arc at a lower cost than assigned to that node so far. If yes (which is true, in particular, if we reach the node for the first time), its tentative cost is updated to the new, lower cost. This procedure is called *relaxing an arc.* Once all outgoing arcs of a node have been relaxed, that node is called *settled.* In the next round, we pick the node with the smallest tentative cost, which has not been settled so far, and relax its outgoing arcs. We iterate this until the target node is settled.

It is a simple, but non-trivial, elegant three-line proof to show that if the arc costs are non-negative, then once a node is settled, the tentative cost assigned to it at that time is actually the cost of the shortest path from the source to that node. Therefore each node is settled at most once. It is also important to observe that before Dijkstra's algorithm settles the target node, it will have settled (and thus computed the shortest path cost for) all nodes which can be reached from the source at smaller cost. If we color all nodes settled by Dijkstra's algorithm before it reaches the target on a drawing of the road network in the plane, we therefore see a disk-like area around the source node; see Figure 1.

What is the complexity of Dijkstra's algorithm? Each settling of a node requires to find, among the unsettled nodes at that point, that node with the smallest tentative distance. This operation is supported by a data structure called a *priority queue*, and it can be implemented to work in time $O(\log n)$, where n is the number of items in the queue. Relaxing an arc potentially requires to update the tentative cost of a node, and we know that if it is updated it is actually decreased. This operation is therefore called *decrease-key*, and can be supported in amortized constant time, that is, a sequence of m such operations takes $O(m)$

[2] An event the author had the pleasure to witness personally many times over the course of the last year.

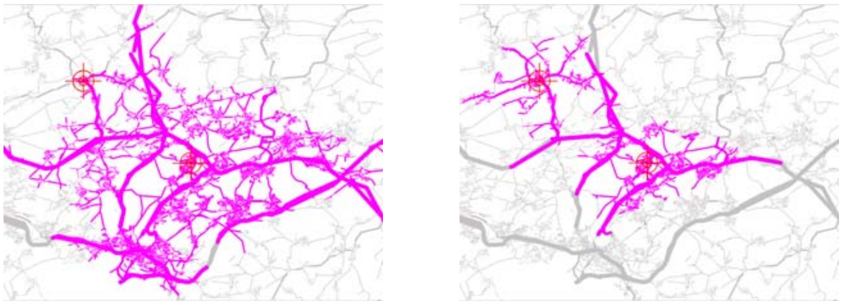


Fig. 1. The search space of Dijkstra's algorithm on a road network, for a given source and target. Left: the original algorithm. Right: its bidirectional version.

time. The complexity of Dijkstra's algorithm is thus $O(n \cdot \log n + m)$, where n is the number of nodes settled before the target node, and m is the sum of the out-degrees of these nodes. In any case, n is bounded by the total number of nodes in the graph, and m by the total number of arcs, and if source and target are far apart, these bounds are actually tight within a small constant factor.

Interestingly, even 50 years after its invention, it is still not known whether Dijkstra's algorithm is theoretically optimal, or whether an algorithm exists that solves the shortest path problem in linear time $O(n + m)$. For our application, this question is academic, however, since even in the best case, each node and each arc would have to be visited at least once, and that alone is very expensive when the network is very large.

For example, consider the road network of the whole of Western Europe. This can be modeled by a graph with about 20 million nodes and about 50 million arcs. It is hard to make the operations involved in settling a node faster than 100 nanoseconds on a standard PC (that is about the time it takes to read a single cache line, or the time for a single cache miss). But even for such a highly tuned implementation of Dijkstra's algorithm, settling all nodes would take on the order of seconds.

In public transportation networks we have yet more nodes. The local public transportation network of Berlin-Brandenburg alone has around 4 million departure and arrival events. Extrapolating this to the whole of Europe (we don't have the actual data yet, so we can only guess) would give a graph with hundreds of millions of nodes.

2 Models Again

In our introduction above, we already gave a brief description of how to model both road networks and public transportation networks as directed graphs. We here recall these descriptions, and talk about a few more relevant details and possible refinements.

2.1 Road Networks

There isn't much to add to the description for road networks; it really is that simple. We have an arc for each road segment[3], a node for each junction of two or more segments, the cost of the arc is the time it takes to travel along that arc, and the goal is to compute the shortest path from a given source node to a given target node.

A number of recent works have addressed a *time-dependent* variant of this problem, where an arc cost is not just a scalar value, but a piece-wise linear function that maps each possible arrival time at the head of the arc to a travel cost [2] [3]. A simple variant of Dijkstra's algorithm can solve this problem as well. As we will see in the next section, public transportation networks can also be modeled by time-dependent graphs.

2.2 Public Transportation Networks

Let us recall the simple model from the introduction. We have a node for each departure and arrival event, nodes are grouped by stations, and arcs are either waiting arcs (between two nodes of the same station) or transit arcs (between two nodes from different stations).

This modeling leaves out the important issue of *transfer safety buffers and costs*: a change of vehicle takes time, and we want to penalize paths with many changes of vehicle—two issues that do not arise in road networks. A simple and natural way to model this, is by having *two* nodes for each arrival or departure event, which represent the state of being *on board* a vehicle and *at the station*, respectively, at the respective station and time.

In its simplest form, a query is given by a source station, an earliest departure time at that station, and a target station. More realistically, however, source and target are not stations, but *geographic locations*, from which we first have to walk to nearby stations. This is important especially in municipal areas, where it is not at all clear which station is the best to walk to first, and it really is (and hence should be made) part of the routing problem to identify the best such station. We then effectively have *sets* of source and target stations.

Note that this is not an issue in road networks, because these are typically so dense that without significant loss of quality in the results, we can simply *snap* to the nearest road segment or junction when source and target are given as geographic locations.

The model we described so far is known as the *time-expanded* model. As an alternative, we can also represent public transportation networks in the *time-dependent* model described in the previous subsection. Simply have one node per station, and the arc cost of getting from the station at the head of the arc at time x to the tail of the arc is $d - x + t$, where $d \geq x$ is the next departure of

[3] A long, curvy piece of road is typically approximated by a sequence of straight-line segments. However, this is done for the purpose of realistic rendering of the network, and is irrelevant for solving the shortest path problem. In fact, the first thing an efficient algorithm would do is contract such sequences to one arc again.

a vehicle to the tail station, and t its travel time. This indeed yields piece-wise linear arc costs.

Asymptotically, the two models do not differ, since a time-aware Dijkstra computation on the time-dependent graph essentially performs the same sequence of operations as an ordinary Dijkstra computation on the time-expanded graph. In practice, a carefully tuned implementation of the time-dependent model can give improvements of a factor of 10 and more over the time-expanded model, but this difference vanishes as soon as realistic features like transfer costs are taken into account. For details on the comparison between the two models, see [4].

2.3 Multi-criteria Cost and Traffic Days

A non-trivial model extension that makes sense for road networks, but is almost mandatory for public transportation networks is to consider *multi-criteria cost functions*. For example, users are typically interested in both travel time and the number of transfers but often not both of them can be minimized at the same time: there may be a connection that takes two hours and does not require any transfers, and there may be a connection which takes only one and a half hours but requires two transfers. Some users will prefer the faster one, and some will prefer the no-transfer one, and so we should (compute and) present both.

Another practical issue that significantly complicates routing on public transportation networks are *traffic days*: certain connections operate only on certain days and not on others.

With respect to their algorithmic solution, both issues are closely related in that they mean that each node in the graph is no longer labeled by only a single cost but by a whole *set of incomparable costs* instead. But again, Dijkstra's algorithm can be easily extended to also deal with this situation. The items in the queue are now individual cost labels (of which a single node can have several), and when settling a cost label, we relax each arc of the node to which the label belongs as before except that we now have to consider the new cost together with *all* the costs of the tail node of the arc, and discard those costs which are no longer optimal.

Obviously, the complexity of relaxing an arc now depends on the number of incomparable costs at the tail of the arc, and, in principle, this number could grow very large. However, we and others have found that with a cost function modeling traveling time and transfer costs, and considering traffic days over periods of a few weeks, the average number of incomparable costs per node is a small constant, and the running time of Dijkstra's algorithm adapted to deal with multiple costs per node lies about a factor of 10 over that of an ordinary Dijkstra computation [5].

2.4 Computing Costs Versus Computing Actual Paths

In the next section we will often tacitly assume that all we want to compute is the *cost* of a shortest path. It indeed typically holds that once we can compute costs fast, we can also compute paths reasonably fast. A very simple, generic way goes as follows. Start at the source node. For each adjacent node v, compute the cost of a shortest path from v to the target, and add the cost of the arc from the source to v. The adjacent node with the smallest such sum lies on the (or rather: a) shortest path. Pick that node, and do the same thing from there. Iterate.

This generic way requires $d \cdot l$ cost computations, where d is the average degree of the nodes on the shortest path, and l is the length of the path. For any of the algorithms mentioned in the following, there are approach-specific ways to do much better this, but we will not get into the details in this paper. The bottom line to remember is that once we can compute costs fast, we can also compute the paths itself fast.

3 Tricks of the Trade . . . and Why and When They Work

With the models all set, the rest of the paper is now essentially a list of the most relevant and effective "tricks of the trade" that have been developed for the speeding up of shortest path queries on transportation networks, in particular from the last decade. Most of these tricks have been invented and applied for road networks first, and were only later transferred to public transportation networks (with, as we will see, limited success so far).

The structure of each of the following subsections is as follows: give a short description of the "trick", explain why it works well for road networks, and then say what the problems are when applying it to public transportation networks. Wherever possible, we will roughly quantify the performance gain in terms of asymptotic complexity and / or actual running times, and refer to the respective papers for the detailed experiments.

As a side effect, this section will also be giving an overview of all the fascinating recent work on routing in transportation networks. This overview is by no means complete, however, since we focus on those tricks which turned out to be most successful, and in each case mention only the one or two most representative works using that trick. For a more complete survey of recent techniques on routing in road networks, see, for example, [6]. For an account of routing algorithms for public transportation networks, see, for example, [7].

3.1 Bidirectional Search

A very simple idea to improve over the plain Dijkstra algorithm is to simultaneously search from the source and target node at the same time, until "the two search frontiers meet". More precisely, we maintain two priority queues, one for the search from the source, as for the ordinary Dijkstra, and one for the backward search from the target, which is just a forward search in the reversed graph, that is, the graph where each arc (u, v) is replaced by (v, u). In each round, we settle the node with the smallest *overall* tentative cost, that is, from the source or to the target; for this, a simple comparison of the minima of the two priority queues suffices. Once we settle a node in one queue that is already settled in the other queue, we get the first tentative cost of a shortest path. To guarantee optimality, we have to continue until the sum of the tentative costs of the current minima of the queues is above the current tentative shortest path cost (which then is indeed the cost of the shortest path).

Speaking in terms of Figure 1, bidirectional Dijkstra reduces the search from a single disk with radius r, to two discs with radius $r/2$. That is, the search space (and hence the query processing time), halves. This by itself is not a big improvement, but as we will see in the following subsections, bidirectional search turns out to be a key ingredient in other, more sophisticated speed-up techniques.

In public transportation networks bidirectional search is more complicated, since we know the target *station*, but not the particular *node* at that station at which we are going to arrive. In fact, finding that node is a significant part of the problem we want to solve in the first place. What we can so, however, is to search backwards from the *set of all nodes* at that station. The backward search would then compute, for each node that it settles, the cost of the path to the earliest node of the station which it can reach. Combined with other techniques this becomes yet more complicated, but by itself is not one of the main obstacles.

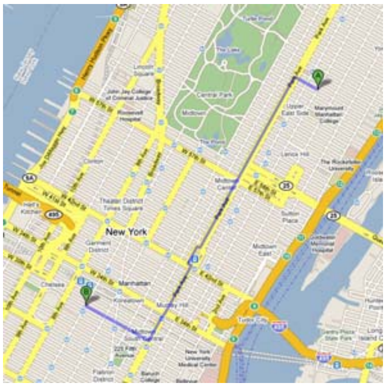
Summary: Bidirectional search by itself is not very effective, but is an important ingredient in more sophisticated techniques. In public transportation networks, we need to search backwards from a whole set of potential target nodes, which makes things more complicated.

3.2 Hierarchy

Most navigation devices in public use nowadays implement a variant of the following simple routing heuristic. Roads have different *levels of importance*: for example, in the road map of Manhattan in Figure 2 (left), we see white (small) roads, yellow (national) roads, and orange roads (motorways). A simple heuristic is then to do a bidirectional search, that takes into account *all* the roads in close proximity to the source and target, but once a certain distance from the source or target is reached, considers only yellow and orange roads, and at a certain even larger distance from the source or target considers only the orange roads. For an appropriate definition of "close proximity" and "certain distance" most shortest paths indeed have that property, like the path in the Figure 2.

This heuristic very significantly reduces the number of nodes that have to be settled and arcs that have to be relaxed, however, at the price of a certain loss of exactness. In the seminal works of [8] and [9] this heuristic has been turned into an exact algorithm, by actually *computing* a level of importance for each arc (which intuitively correspond to the road colors in Figure 2, but algorithmically have nothing to do with them). On road networks both precomputation and query times are very fast. With the latest version of their algorithm, the importance levels can be computed in about 15 minutes for the complete road network of Western Europe, with subsequent query times on the order of 1 millisecond. The method was, quite appropriately, named *highway hierarchies.*

On public transportation networks, even if we leave the complications of bidirectional search described in Section 3.1 aside, experimental studies ([10] and also our own) have shown that the speed-ups obtained are much less dramatic



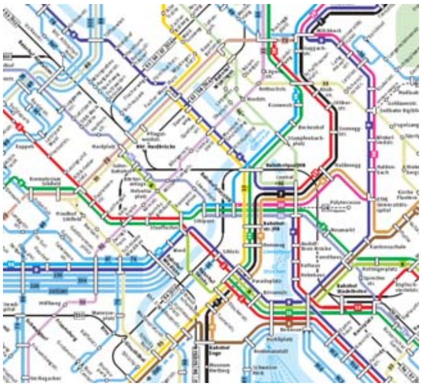


Fig. 2. Left: a shortest path in the road network of Manhattan. Right: a section of the tram + bus network of Zurich.

than for road networks. On large municipal areas, query processing times can even be *worse* than for a well-tuned implementation of Dijkstra's algorithm.

The main reason for this disappointing performance is actually easy to understand. Look at the tram + bus network of Zurich in Figure 2 (right), and think of a few random queries and their solutions. You will find that there is *hardly any hierarchy.* Intuitively, all the trams and buses are equally important, and exactly which tram or bus is chosen for a given query depends more on how well the schedules of the various lines match, than on some connection being more important than others. Once we travel long-distance between cities, a first level of hierarchy does appear (intuitively, the long-distance trains as opposed to the local trams and buses), but not on the intra-city level.

While this may be fine for a relatively small area like that of Zurich (about one thousand stations), this is a major performance problem for large municipal areas like, for example, New York (several tens of thousands of stations, with tens of millions of arrival / departure events). A Dijkstra computation even on this local network takes on the order of seconds, and hierarchical methods are of no use to speed things up there.

Worse than that, also the precomputation time suffers on such networks. In order to identify the first level of hierarchy, a method like highway hierarchy does a local search from each node, until all paths have reached the next level of the hierarchy. But for all nodes within a municipal area, this local search will have to cover the whole municipal area, which can encompass *millions of nodes.* In contrast, we know that for road networks local searches of only *a few hundred nodes* are enough to discover the next level of the hierarchy [9].

Summary: The efficiency of hierarchical approaches in terms of both precomputation and query time is proportional to the extent of the local searches necessary to find the next level of hierarchy. For road networks, a few hundred nodes per local search are typically enough. For public transportation networks, frequently whole municipal areas with millions of nodes need to be explored.

3.3 Shortcuts / Contraction

Recall the footnote in Section 2, where we talked about modeling a long, curvy piece of road as a sequence of short straight-line road segments. As mentioned there, this is done for the purpose of nice rendering. For the purpose of the shortest-path computation, we may as well replace that sequence by a single arc again, thus significantly reducing the number of nodes and arcs in the graph. This replacement is an instance of so-called *contraction*, and the new arc is called a *shortcut*. Some methods also insert shortcuts without actually removing arcs, but instead have a mechanism to consider only selected arcs at query time [11].

For a method like highway hierarchies, contraction is of good use not only on the original graph, but also on the iteratively computed subnetworks. Just think of the subnetwork of all motorways. Most junctions there are of the kind that we either enter or leave the motorway to or from a less important road. With all non-motorways removed from the graph, we will have only very few nodes with degree larger than 2, namely the actual motorway junctions.

In fact, contraction can be taken one step further by also contracting nodes of a degree larger than 2. To contract a node x, we simply look at all pairs u, v, where u is adjacent to an incoming arc and v is adjacent to an outgoing arc, and check whether there is a shortest path containing u, x, v. If yes, we insert the shortcut (u, v). This pays off, provided that we do not insert (many) more shortcuts than we remove arcs by removing x; see [9] for details.

Note that contraction and shortcuts are not so much a stand-alone method, but have instead acted as a catalyzer for a variety of multi-level methods, in particular: [9] [11] [12].

As far as public transportation networks are concerned, consider again the tram and bus network of Zurich from Figure 2. Most stations are "junctions", where more than one line meets, and if one takes the possibility of walking between stations into account (see Section 2.2), the average number of lines to which one can transfer at a given station increases further. This is especially true in cities with many different transportation agencies and therefore many stations in the vicinity of each other. But contraction and / or the introduction of shortcuts is only effective for nodes of low degree.

We have already found in Section 3.2, that the difficult searches are the local ones, where local can mean a whole municipal area. Unfortunately, it is exactly in these area, on the lowest level of the network, that the node degree is too high for contraction to be effective.

Summary: Contraction / Shortcuts don't help us speeding up local searches on the lowest level of the hierarchy, due to the high node degree there.

3.4 Goal Direction

The simplest form of goal direction is to augment Dijkstra's algorithm by a heuristic that for each node in the graph estimates the cost to the given target. Nodes are then retrieved from the priority queue by the *sum* of their tentative cost and the value of the heuristic function. This variant of Dijkstra's algorithm is known under the name *A* (A-star) algorithm*, and was first described in 1968 [13].

The performance gain of A* depends on the quality of the heuristic. It is a three-line proof (very similar to the correctness proof for Dijkstra's algorithm) that A* is correct, whenever the heuristic function *underestimates* the actual cost of the respective node to the target. If the heuristic cost is always zero, we are back to Dijkstra's algorithm. If the heuristic function magically knows the exact cost to the target, A* will be perfect in that it settles only the nodes on the (or rather: a) shortest path.

One simple, non-magical heuristic is to underestimate the cost to the target by the geographic straight-line distance to the target divided by the maximum speed of a vehicle anywhere in the network. This heuristic always underestimates the true cost, sometimes by not much (when the shortest path to the target is geographically relatively straight and uses mainly motorways), sometimes a lot (when the shortest path to the target is long-winding and uses mainly slow roads). Overall, this heuristic gives a notable but not very dramatic improvement in query processing time by a factor of about 2 to 3, for both road and public transportation networks. A more powerful heuristic, based on precomputed distances to so-called *landmarks*, has been presented in [14].

The most powerful form of goal direction is provided by so-called *arc flags* [15] [16]. Here the graph is partitioned into k regions, and for each arc k bits are precomputed, where the ith bit is 1 if and only if that arc is on a shortest path to a node within region i. At query time we can then simply ignore all arcs outside the region containing the target where the bit for that region is set to 0. In an extreme case, where each node forms a region on its own, the arc flags for the target node would then show us the shortest path without any detour.

These arc flags / bits can be computed by running Dijkstra's algorithm separately from each node, in the reversed graph. This, however, is equivalent to a quadratic-cost all-pairs shortest path computation. It is easy to see, that it is enough to consider *only nodes on the boundary* of each region. In a perfect grid graph with n nodes, partitioned into k parts (by $\sqrt{k}-1$ horizontal and $\sqrt{k}-1$ vertical cuts), the number of boundary nodes would be on the order of $\sqrt{n \cdot k}$, which still gives an order $n^{3/2}$ cost for the precomputation, even for small k. In real graphs, the cost tends more towards n^2.

A conceptually simple trick to reduce the precomputation cost to almost linear is to work with a multi-level partitioning of the graph. In the precomputation, the backwards Dijkstra computation from a boundary node of a cell in the partitioning can then stop, as soon as all nodes in the containing cell from the next level are settled.

It is here that we meet another fundamental difference between road networks and public transportation networks. Namely, for road networks we can indeed settle all nodes in a geographically bounded region with cost roughly proportional to the number of nodes in that region; see, for example, [6].

In public transportation networks, however, we have a fundamental and very annoying problem, which we will explain by an example. Consider a node in

Zurich and assume that we want to settle all nodes in Zurich and the surrounding villages. Even though the geographic extent of that region is relatively small, there will be several nodes in that region which can be reached only at a very high cost. The reason is simply *bad connectivity*: we might be just too late for the last bus of the day and have to wait overnight for the first bus of the next day, thus getting an optimal connection taking 15 hours. But in 15 hours, we can get to the airport, take a plane to New York and explore half of the city there ... and Dijkstra's algorithm will just do that.

As extreme as it may sound, this phenomenon is actually the rule and not the exception. We consider it a major open problem to come up with an algorithm for local searches in public transportation networks with cost proportional to the number of nodes to be settled.

Summary: Goal direction is potentially very effective but has very high precomputation costs. For public transportation networks, this cost is quadratic due to the lack of efficient algorithms for local searches on such networks.[4]

3.5 Distance Tables

An extreme precomputation would be to compute a table with distances between all pairs of nodes in the given graph. Query times would then be instantaneous (recall the bottom line of Section 2.4 that once we can compute the cost fast, we can also compute the actual paths fast), but the precomputation complexity would be quadratic in both time and space.

Distance tables for a *subset* of the nodes have been used as a "turbo" in various approaches in the past. We here briefly describe *transit node routing*, which works solely with distance tables and is the fastest method for routing in road networks (with at the same time reasonable preprocessing) to date [17] [18].

The *transit nodes* are a subset of nodes with the following "magical" properties: (1) the set is small, on the order of $\sqrt{n}$, where n is the total number of nodes in the network; (2) all shortest paths that cover a certain minimal geographic distance D have at least one transit node on them; (3) the number of transit nodes hit first on shortest paths that start from a fixed node is small; we call these few transit nodes the *access nodes* of a node.

Given such a set of transit nodes, we precompute for each node, the distances to all of its access nodes, and the distance between each pair of transit nodes. For a given query, let x and y be the number of access nodes of the source and target, respectively. To answer the query, we then need to look up a mere $x + y + x \cdot y$ of the precomputed distances and try out all $x \cdot y$ combinations of access node near the source and access node near the target. On road networks the astonishingly low number of 5 access nodes, on average, can be achieved, leading to extremely fast query times on the order of a few *microseconds*. The precomputation can be done in a number of ways, one of which is similar to the precomputation for highway hierarchies and with a comparable complexity [18].

[4] The precomputation from [2], although extremely well-tuned, is quadratic, too.

It is important to understand that due to property (2) above, this only works when the source and target are geographically at least a distance of D apart. (Obviously the *short* shortest paths cannot all be hit by a small set of common transit nodes, too.) But when D is small, this is not a problem, since for queries below this threshold any conventional method is good enough; see [18] for details.

Experiments show that also in public transportation networks we can find a good set of transit nodes with properties (1) - (3) above. The number of access nodes per node is by a factor of 5 - 10 higher than in road networks, but still small enough to yield query times on the order of milliseconds.

The problem are (a) the local searches required to precompute, for each node, the distances to its access nodes, and (b) the local searches required at query time when source and target are less than the distance D apart. Both of these can, and often will, involve computing shortest paths of very large cost, and we have no efficient solution for that case for exactly the reasons described at the end of Section 3.4 (the "it can take 15 hours to the nearby village" problem).

Summary: A good set of transit nodes can be found for both road and public transportation networks. However, in public transportation networks, we do not have efficient algorithms for the local searches required to precompute the distances to the access nodes or at query time when source and target are close together.

4 Conclusions

We gave an overview of the main techniques to speed up shortest path computation on transportation networks compared to Dijkstra's algorithm. We specifically looked at: bidirectional search, hierarchies of subnetworks, goal direction, contraction and shortcuts, and distance tables. We found that all of these approaches work well (and some extremely well) for road networks, but none of them gave convincing results for public transportation networks so far. We identified two key open problems which so far have obviated fast routing on very large public transportation networks:

Open Problem 1: (Speed-up despite lack of hierarchy) How to achieve, with reasonable precomputation cost, a significant speed-up over Dijkstra's algorithm in large municipal areas with hardly any hierarchy, for example, in large bus-only networks?

Open Problem 2: (Efficient local searches) How to compute shortest paths to all nodes in a local (for example, geographic) neighborhood efficiently, in the face of (albeit few) shortest paths within that neighborhood of large cost?

Acknowledgments

Thanks to Peter Sanders for a brief, but very fruitful exchange on the right selection and categorization of the tricks of the trade in Section 3.

References

1. Dijkstra, E.: A note on two problems in connexion with graphs. Numerische Mathematik 1, 269–271 (1959)
2. Delling, D.: Time-dependent SHARC-routing. In: Halperin, D., Mehlhorn, K. (eds.) ESA 2008. LNCS, vol. 5193, pp. 332–343. Springer, Heidelberg (2008)
3. Batz, G.V., Delling, D., Sanders, P., Vetter, C.: Time-dependent contraction hierarchies. In: 11th Workshop on Algorithm Engineering and Experiments (ALENEX 2009), pp. 97–105 (2009)
4. Pyrga, E., Schulz, F., Wagner, D., Zaroliagis, C.D.: Efficient models for timetable information in public transportation systems. ACM Journal of Experimental Algorithmics 12 (2007)
5. Müller-Hannemann, M., Schnee, M.: Finding all attractive train connections by multi-criteria pareto search. In: 4th Workshop on Algorithmic Methods for Railway Optimization (ATMOS 2004), pp. 246–263 (2004)
6. Schultes, D., Sanders, P.: Dynamic highway-node routing. In: Demetrescu, C. (ed.) WEA 2007. LNCS, vol. 4525, pp. 66–79. Springer, Heidelberg (2007)
7. Müller-Hannemann, M., Schulz, F., Wagner, D., Zaroliagis, C.D.: Timetable information: Models and algorithms. In: 4th Workshop on Algorithmic Methods for Railway Optimization (ATMOS 2004), pp. 67–90 (2004)
8. 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)
9. Sanders, P., Schultes, D.: Engineering highway hierarchies. In: Azar, Y., Erlebach, T. (eds.) ESA 2006. LNCS, vol. 4168, pp. 804–816. Springer, Heidelberg (2006)
10. Bauer, R., Delling, D., Wagner, D.: Experimental study on speed-up techniques for timetable information systems. In: 7th Workshop on Algorithmic Methods for Railway Optimization (ATMOS 2007) (2007)
11. Geisberger, R., Sanders, P., Schultes, D., Delling, D.: Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In: McGeoch, C.C. (ed.) WEA 2008. LNCS, vol. 5038, pp. 319–333. Springer, Heidelberg (2008)
12. Bauer, R., Delling, D.: SHARC: Fast and robust unidirectional routing. In: 10th Workshop on Algorithm Engineering and Experiments (ALENEX 2008), pp. 13–26 (2008)
13. Hart, P., Nilsson, N., Raphael, B.: A formal basis for the heuristic determination of minimum cost paths. IEEE Transactions on Systems Science and Cybernetics 4(2), 100–107 (1968)
14. Goldberg, A., Harrelson, C.: Computing the shortest path: A^* search meets graph theory. In: 16th Symposium on Discrete Algorithms (SODA 2005), pp. 156–165 (2005)
15. Lauther, U.: An extremely fast, exact algorithm for finding shortest paths in static networks with geographical background. Münster GI-Tage (2004)
16. Köhler, E., Möhring, R.H., Schilling, H.: Acceleration of shortest path and constrained shortest path computation. In: Nikoletseas, S.E. (ed.) WEA 2005. LNCS, vol. 3503, pp. 126–138. Springer, Heidelberg (2005)
17. Bast, H., Funke, S., Matijevic, D.: Ultrafast shortest-path queries via transit nodes. In: DIMACS Implementation Challenge Shortest Paths (2006); An updated version of the paper appears in the upcoming book
18. Bast, H., Funke, S., Matijevic, D., Sanders, P., Schultes, D.: In transit to constant time shortest-path queries in road networks. In: 9th Workshop on Algorithm Engineering and Experiments (ALENEX 2007) (2007)