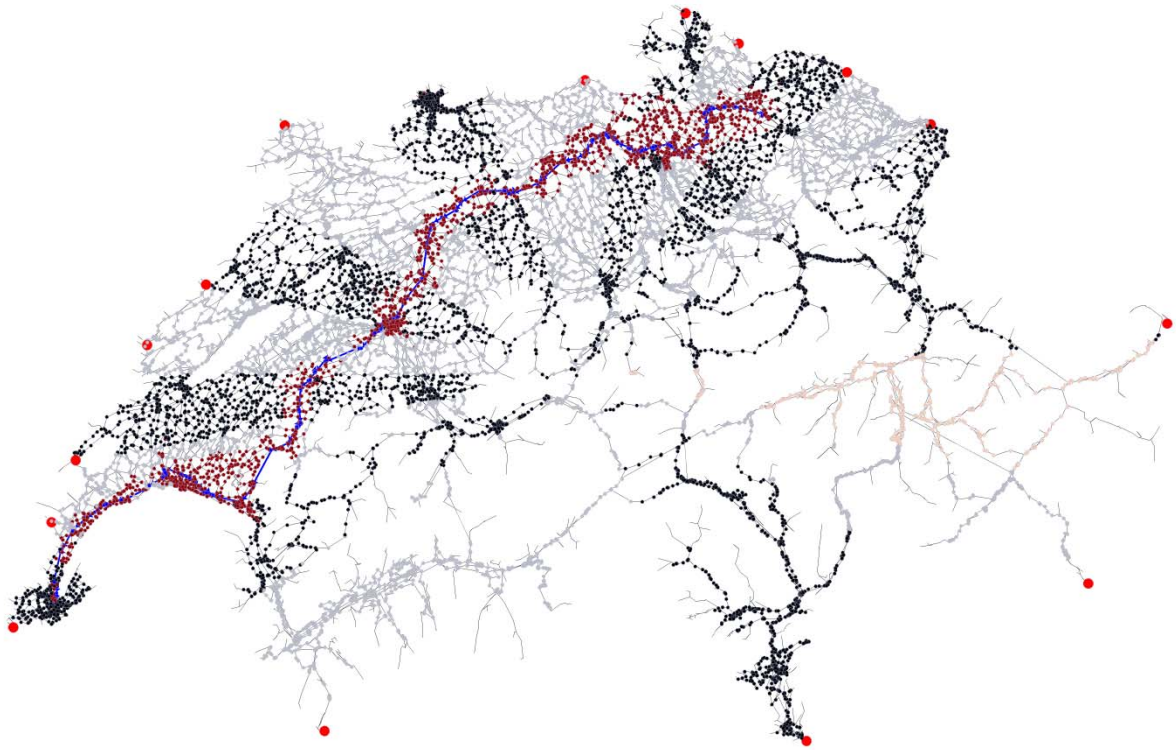


Fast shortest path computation in time-dependent traffic networks

Nicolas Lefebvre, IVT, ETH Zurich
Michael Balmer, IVT, ETH Zurich

Conference paper STRC 2007 **August 2007**

STRC | Swiss Transport Research Conference
Exchanging ideas for transport

12 - 14 September 2007, Monte Verità

Conference paper STRC 2007

Fast shortest path computation in time-dependent traffic networks

Nicolas Lefebvre
IVT
ETH Zurich

Telephone: +41-44-633 49 97
Fax: +41-44-633 10 57
Email: lefebvre@ivt.baug.ethz.ch

Michael Balmer
IVT
ETH Zurich

Telephone: +41-44-633 27 80
Fax: +41-44-633 10 57
Email: balmer@ivt.baug.ethz.ch

August 2007

Abstract

In agent based traffic simulations which use systematic relaxation to reach a steady state of the scenario, the performance of the routing algorithm used for finding a path from a start node to an end node in the network is crucial for the overall performance. For example, a systematic relaxation process for a large scale scenario with about 7.5 million inhabitants (roughly the population of Switzerland) performing approximately three trips per day on average requires about 2.25 million route calculations, assuming that 10% of the trips are adapted per iteration. Expecting about 100 iterations to reach a stable state, 225 million routes have to be delivered in total.

This paper focuses on routing algorithms and acceleration methods for point-to-point shortest path computations in directed graphs that are time-dependent, i.e. link weights vary during time. The work is done using MATSim-T (Multi-Agent Traffic Simulation Toolkit) which used for large-scale agent-based traffic simulations. The algorithms under investigation are both variations of Dijkstra's algorithm and the A*-algorithm. Extensive performance tests are conducted on different traffic networks of Switzerland. The fastest algorithm is the A* algorithm with an enhanced heuristic estimate: While it is up to 400 times faster than Dijkstra's original algorithm on short routes, the speed up compared to Dijkstra diminishes with the length of the route to be calculated.

Keywords

Shortest path problem; time-dependent networks; multi-agent traffic simulation, STRC 2007

Preferred citation style

Lefebvre, N. and M. Balmer (2007) Fast shortest path computation in time-dependent traffic networks, Paper presented at *the 6th Swiss Transport Research Conference*, Ascona, September 2007.

1 Introduction

This report focuses on the problem of finding a shortest path between two nodes, the *point-to-point shortest path problem (P2P)* in a directed time-dependent graph with positive edge weight.

Probably the most famous and most frequently used P2P algorithm is the algorithm of Dijkstra (1959). Beginning at the starting node of the route, it visits all its neighbors and ranks each of them based on the minimal sum of the cost of the visited edges connecting it to the start node. In each subsequent step, the algorithm *relaxes* the node with minimal cost (highest rank), i.e. visits all its neighbors and ranks them (eventually again). Dijkstra's algorithm stops when the end node of the route is relaxed.

All routing algorithms presented in this paper are implemented within MATSim-T (Balmer, 2007), an iterative, agent-based micro-simulation of traffic systems. It mainly consists of a microscopic traffic flow simulation on one side and different modules adapting travel demand to generalized travel costs on the other side. They are called alternately until the system reaches its stationary state, which corresponds to user equilibrium in the case of traffic systems. In MATSim-T, travel demand is represented by individual agents that follow an activity plan. Each activity plan is assigned a score. The higher the score, the better is the plan. Convergence to the stationary state is, among other measurements, judged by the development of the score aggregated over the whole agent population.

MATSim-T operates on directed graphs. Each edge has time-dependent weight or cost. It is assumed that they are represented by non-negative floating numbers. A least-cost path between a source node s and a destination node d starting at time t is defined as the path with minimal accumulated cost over all edges at the time they are visited during the trip. According to Dijkstra's algorithm, there is no possibility of waiting intentionally at a node for a time that allows traversing an edge at less cost in order to minimize the cost of a path.

1.1 Iterations

As pointed out above, an iterative approach is used to find the stationary state in the dynamic traffic system. A traffic simulation in MATSim-T consists of the following steps:

1. Initial activity plans have to be generated as a first input to the traffic flow simulation. It contains all the assumptions about the agents' personal attributes, as well as approximations for the plan attributes, e.g. free speed travel times and respective cost as an approximation for the least-cost path computation. For each agent, a set of plans is generated and stored in the agent database.

2. One plan per agent is selected. The simulation of traffic flow then executes the selected plan per person, i.e. it "moves" agent objects along the respective path delivered by the routing algorithm through a model of the traffic network. New travel times for each trip result. Additionally, the executed plans are scored. If the stop criterion of the simulation is reached (e.g. if the overall score of all plans did not change considerably over the last few iterations), the iteration process is finished.

3. A subset of about 10% of the agents is chosen for plan modification by so-called replanning modules, which can capture one or more travel behavior attributes. After that, return to step 2. So about 10% of all paths have to be routed again per iteration. Here (as opposed to the routing in step 1) the network is loaded, i.e. the cost of each link is higher or equal than the free speed cost (usually depending on how much agents used the link at the given time in the previous iteration).

2 Related work

The performance of a routing algorithm is usually measured in means of the number of nodes that it visits during routing. Dijkstra's algorithm visits nodes within a circular area encompassing the start and the end node of the route. For applications in dense networks that require a lot of shortest path computations, this is often not fast enough. So, several acceleration techniques have been developed over time. They can basically be classified as follows (as suggested by Holzer *et al.*, 2005):

- **Goal-directed search**: Change the way the nodes are ranked such that nodes which are less likely to be on the shortest path are also less likely to be visited. The most popular algorithm that uses this technique is the A* algorithm (Hart *et al.*, 1972). Goldberg and Harrelson (2005), Huang and Zhan (2007) and Chabini and Lan (2002) made more recent contributions to this type of routing algorithms.

- **Bounding boxes**: Find out whether certain nodes can at all be on a shortest path and if not, do not consider them. See Gutman (2004), Wagner and Willhalm (2003), Fu *et al.* (2004), Ertl (1998) and Lauther (2004) for further reading.

- **Multi-level approach**: Add shortcuts to the network where possible to bypass several edges at a time during routing. A sub-class of this is *hierarchical routing*: Build a hierarchy of networks with the first layer being the full resolution network; each following network contains less nodes and links and is therefore less complex for routing purposes, but also less accurate. Start routing in the most abstract layer and switch to the higher resolution layer if the accuracy of the current layer does not suffice to do further routing to the end node. Examples of such algorithms can be found in Chou *et al.* (1998), Sanders and Schultes (2005) and Köhler *et al.* (2005).

- **Bi-directed search**: Start routing at the end and at the start node at the same time. End if a node coming from one end of the route gets visited that already has been visited by the algorithm coming from the other end. Several of the above citations use bi-directional routing in order to further improve the running time.

Surveys comparing different routing algorithms can be found in Fu *et al.* (2006) and Jacob *et al.* (1999).

3 Problem setting

All of the above mentioned approaches exploit particular properties of the underlying network on which the routing has to be performed and a lot of algorithms in literature are designed for a specific problem setting. Consequentially, most of these approaches are not directly applicable to our setting. The fact that the routing is performed on a network with time-dependent link cost makes the use of most of the four above mentioned acceleration techniques for routing algorithms difficult:

- When the weights of the links do not change over time, some nodes can usually be omitted during routing based on information that was obtained by a preprocessing of the network. Gutman (2004) for example uses the fact that if the minimal remaining cost *c* of a node to the target node can be gathered, and the minimal cost of the path is not higher than *c*, one can prune the current node. However, in time-dependent networks, it is difficult to do such considerations, as the cost to traverse a link can be arbitrarily high. In other words, if all other links are congested, any link can be on the least-cost path to the target node. Exceptions are nodes in dead ends which can indeed be eliminated directly, as long these nodes are not in the same dead end of the start or the end node.

- Replacing several links by a shortcut immensely raises the amount of additional memory needed, as these shortcuts might be valid for different time bins: Links that are cost-minimal in one time bin may not be in other time bins, so a different shortcut for each possible time bin is needed, increasing the amount of stored shortcuts dramatically. The same holds for hierarchical routing: In time-dependent networks, a hierarchy for every possible time bin has to be stored.

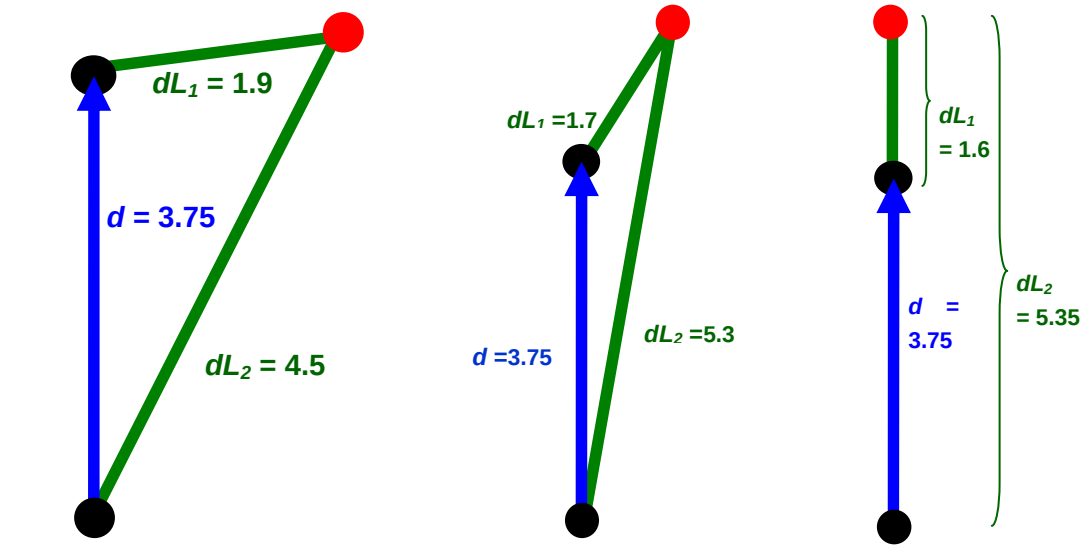
- Bidirectional routing is not possible either, since arrival times of the routes are not known before forward time-dependent route computation.

The only utilized network characteristic within MATSim-T is the fact that there is a lower bound on the possible cost of a link over all time bins, which often corresponds to the cost when no other vehicle is traversing the link. If cost is equal to travel time, this lower bound corresponds to the link length divided by the free speed travel time on that particular link. This property can be used to modify the node ranking in order to favor nodes that are more likely to lie on a least-cost path to the target node than other nodes.

Hart *et al.* (1972) described such a modified Dijkstra's algorithm called A* for the first time. It uses a heuristic estimate function *h(x)* for a node *x*. The new ranking of the nodes is then *h(n)* for each node *n* plus the cost from the start node to *n*. To guarantee that A* always finds the shortest path, *h(x)* must for each node return a value that is lower than the actual minimal cost to the target node.

The most common practice when dealing with networks is to use the Euclidean distance from the current node to the target node as heuristic estimate. This is the so-called Sedgewick-Vitter heuristic (Sedgewick and Vitter, 1986). A slight modification is to take advantage of the knowledge of the lower cost bound of the links: First, the maximum *m* of the link length divided by the corresponding lower bound link cost over all links is calculated. The heuristic estimate then is the Euclidean distance of the current node to the end node, divided by *m*.

Figure 1 Distance estimation using landmarks



5.5 – 1.9 = 2.6 < 3.75 5.3 – 1.7 = 3.6 < 3.75 5.35 – 1.6 = 3.75

A variant that was published by Goldberg and Harrelson (2005) uses so-called *landmarks* to estimate the remaining travel cost to the target node. As a preprocessing step, some nodes are elected to act as landmarks during routing. Then the minimal possible cost of a path from and to each landmark and each node in the network is calculated using the lower cost bound of each link. Using the triangle inequality theorem, stating that the measure of a given side is always greater than the difference between the two other sides, the least-cost paths from and to each landmark can be used to estimate the travel cost between two nodes.

Figure 1 depicts how this approach works: Suppose one wants to estimate the distance *d* between the two black nodes but only knows the distance dL_1 and dL_2 of the black nodes to the red landmark. In order to get a lower bound of *d*, one calculates the difference of dL_1 and dL_2. It can be seen from the pictures that the landmark should preferably lie behind one of the black nodes to get a good estimation of *d*.

Figure 2 Routing using landmarks

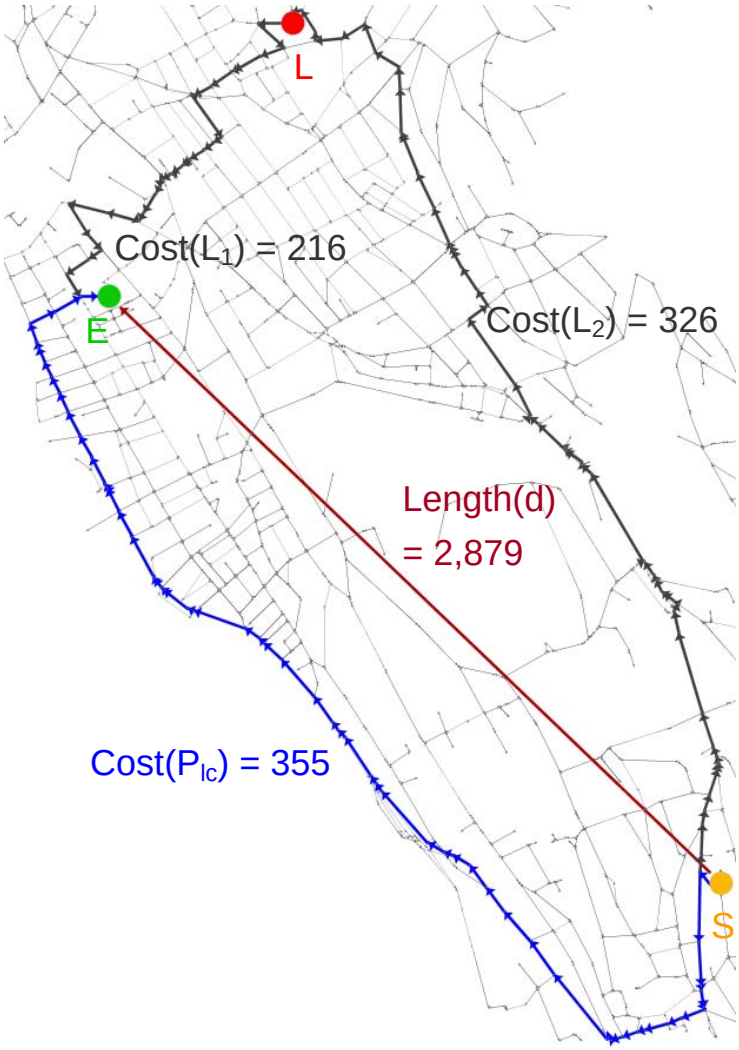


Figure 2 shows the situation in a real road network. The task is to find the least-cost route from the yellow start node *S* to the green end node *E*. The red node *L* is a landmark. The blue least-cost path P_{lc} from *S* to *E* has cost 355. The least-cost path L_2 from *S* to *L* has cost 326 and the one from *L* to *E* (L_1) has cost 216 and thus the estimate of the cost of the path from *S* to *E* is 326-216 = 110. The Euclidean distance *d* from *S* to *E* is 2,879 while the overall minimal cost per length unit in the network is 34.7. So, by using the Euclidean distance one gets

an estimate of 2,879/34.7 = 83 which is lower than the estimate of the one obtained by using landmarks. It is assumed that the higher the better the estimate is, as long as it is lower than the real cost and it is reasonable, i.e. it tends to be larger for nodes that are further away from a target node than other nodes.

As one has to store the cost of the least-cost path to all landmarks in each node of the network, the memory overhead $\tilde{O}_L$ of the Landmarks A* algorithm is (assuming that the size of a floating point number is 8 bytes)

$$\tilde{O}_L = [8 \text{ bytes}] * [\text{number of nodes in the network}] * [\text{number of landmarks}].$$

4 Experimental results

4.1 Experimental setup

The experiments were done on four networks representing Switzerland's road network, each at a different resolution. An overview is given in Table 1. The Detour Index (DI) of a network is computed as

$$DI = DD/TD.$$

DD is the sum of the linear distances of the start and end node of all links and TD is the sum of the lengths (transport distances) of all links. Note that no link's length should be shorter than the linear distance of its start and end node, otherwise the A* routing algorithm using the Sedgewick-Vitter heuristic discussed above does not necessarily deliver least-cost paths because it may overestimate the remaining travel cost.

Table 1 Networks comparison

Name	# nodes	# links	Avg. node degree	Detour Index	Avg. link length (m)	Avg. link free speed (m/s)	# dead-end nodes
Net1	7,377	20,252	5.49	1	1,816.82	14.20	1,025
Net2	14,803	39,372	5.32	0.90	1,218.43	19.40	1,776
Net3	408,636	882,120	4.32	0.97	96.68	11.00	127,904
Net4	471,879	1,053,259	4.47	0.95	108.49	9.29	142,238

The algorithms under investigation are:

- Basic Dijkstra (Dijk): Dijkstra's original algorithm: First, the visited flags of all nodes in the network are reset. Then, Dijkstra's algorithm is used to find the least-cost path from the start to the end node.

- Iteration-ID Dijkstra (It-ID Dijk): Instead of initializing all nodes before the routing, a loop variable is used to keep track of the number of routing requests, providing an identifier of the current routing operation. Using this routing identifier, one can then

determine whether a node was already visited during the current routing operation and the nodes do not have to be reset before each routing.

- Iteration-ID Dijkstra with dead-end pruning (Prune Dijk): As a preprocessing step, all nodes that lie in dead ends are detected. During routing, they can be ignored (as long as the current node does not lie in the same dead end as the start or end node). A node lies in a dead end if it is connected to the network by one node only (the "dead end entry node"). If the dead end entry node is removed from the network the nodes in the respective dead end get disconnected from the network. The memory overhead $\tilde{O}_D$ of this algorithm is

  $$\tilde{O}_D = [\text{number of nodes in dead ends}] * [X].$$

  X is the size of an object reference (which is 4 bytes in our case) and stands for the pointer of the node in the dead end to the dead end entry node of the respective dead end.

- Euclidean A* (Euc A*): The A* algorithm using the Euclidean distance divided by *m* (the maximum of the link length divided by the corresponding lower bound link cost over all links) as estimate of the remaining travel cost. All A* algorithms are implemented to prune dead ends.

- Landmarks A* (Landm A*): The A* algorithm using 16 landmarks in order to estimate the remaining travel cost. The memory needs (including the overhead for pruning dead ends) in a 32-bit environment are 0.93 MB on Net1, 1.86 MB on Net2, 51.44 MB on Net3 and 59.40 MB on Net4.

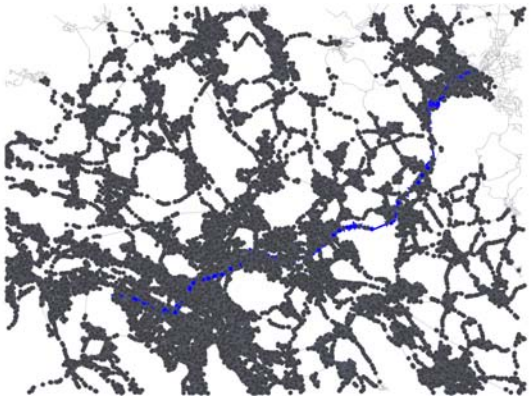
- Overdo A* (Ov A* X, where X denotes the *overdo factor*): The Euclidean A* algorithm whose remaining travel cost estimate is multiplied by an overdo factor > 1 such that this version of A* does not always find the least-cost path (rather it tends to find distance-minimal paths). The higher the overdo factor, the fewer nodes are visited during routing, but the more the resulting path differs from the least-cost one.

The showcase example of a cut of network Net3 (showing the city of Zurich) in Figure 3 and Table 2 gives a first impression on how these algorithms perform. The least-cost path found is marked in blue in the following pictures and the visited nodes are highlighted in dark. For Landmarks A*, the landmarks are marked in red.

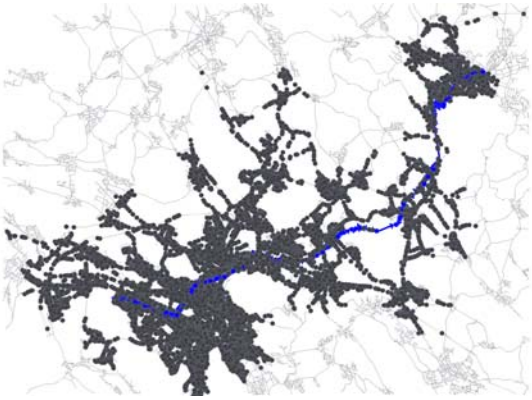
Dijkstra's algorithm visits (or expands) almost all of the 26,483 nodes in the network. By ignoring nodes in dead ends during routing, the amount of visited nodes can already be lowered by a fourth. Basic A* visits less than half as much nodes as Dijkstra and Landmarks A* visits a tenth of the nodes visited by Dijkstra.

Finally, Overdo A* with an overdo factor of 2 (thus estimating the remaining travel distance as two times the Euclidean distance divided by *m*) visits 0.046 as much nodes as Dijkstra by delivering a path that is very close to the least-cost one.

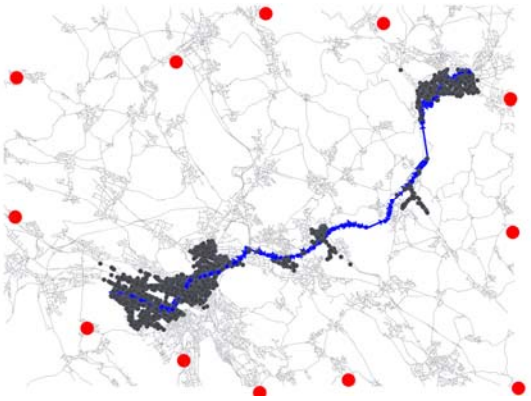
Figure 3 Amount of visited nodes for different routing algorithms



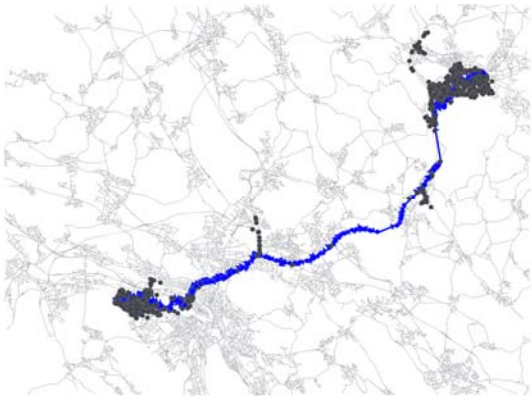
Basic/Iteration-ID Dijkstra



Euclidean A*



Landmarks A* (using 12 landmarks)



Overdo A* (overdo factor 2)

Table 2 Amount of visited nodes for different routing algorithms

Algorithm	# visited nodes	route cost (seconds)
Dijk/It-ID Dijk	24,567	1,257
Prune Dijk	18,172	1,257
Euc A*	11,613	1,257
Landm A*	2,436	1,257
Ov A* 2	1,121	1,270
Ov A* 3	892	1,677

4.2 Performance comparison

10,000 pairs of random start and end nodes in each network are selected by ensuring that the distance of the start and end node lies within a specified range X ± 1km. The values for X are 1km, 5km, 25km, 50km, 100km, 150km and 200km. The distance of the start and end node is called from-to distance, and X is called *aimed from-to distance*. Experiments with X = 250 km were also performed, but because the network is not much larger than X, only a few from-to nodes at the boundary of the network match the range criterion and get selected, thus producing artifacts within the results. With 200 km, these artifacts can already be noted (see below), but they are not as strong as with 250 km.

Table 3 Average from-to distance

Aimed from-to distance	Avg. from-to distance Net1	Avg. from-to distance Net2	Avg. from-to distance Net3	Avg. from-to distance Net4
1 km	1.86 km	1.68 km	1.17 km	1.17 km
5 km	5.27 km	5.38 km	5.06 km	5.06 km
25 km	25.29 km	25.21 km	25.06 km	25.06 km
50 km	49.98 km	50.00 km	50.02 km	50.02 km
100 km	99.83 km	100.01 km	100.02 km	100.02 km
150 km	149.67 km	150.20 km	150.03 km	150.03 km
200 km	200.36 km	200.20 km	200.04 km	200.04 km

The experiments are performed in a 32-bit environment on each of the three mentioned networks on a server-system equipped with Dual Core AMD Opteron processor 275 with

2.2GHz. Only one core was used during our test runs. The system was equipped with 4GiB of RAM of which 2.5GiB were used.

Each run consists of two phases: The preprocessing, where network-specific data is computed according to the needs of the respective routing algorithm, and the actual routing phase, during which the 10,000 paths are calculated. To get the average computing time per path, the performance measure of the second phase of each run is divided by 10,000. In a final version of the routing algorithms, the preprocessing data will be a property of the network and should not be calculated each time a least-cost path computation is done. However, currently the network is preprocessed before each set of routing operations. The typical preprocessing computation times are shown in Table 4. They are equal for free flow networks and loaded networks. Basic and Iteration-ID Dijkstra require no preprocessing. For Net3 and routing algorithm Landmarks A*, the computing time spent for preprocessing the network corresponds to computing 800 routes with from-to-distance ~50km.

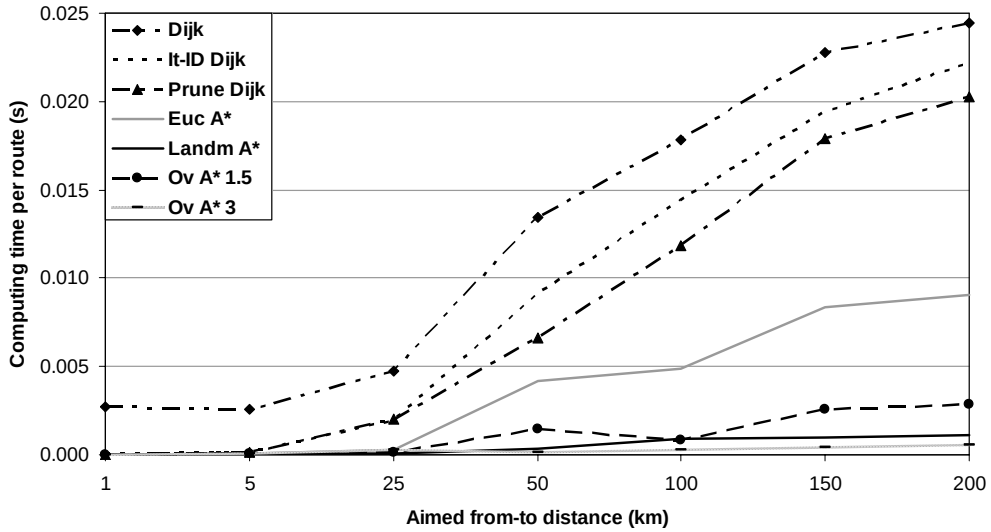
Table 4 Average preprocessing time per network

Algorithm	Net1	Net2	Net3	Net4
Dijk/It-ID Dijk	-	-	-	-
Prune Dijk/Euc A*/Ov A*	0.13 s	0.22 s	4.20 s	5.00 s
Landm A*	1.20 s	2.00 s	72.00 s	160.00 s

4.2.1 Free flow

Figures 4 to 7 show the routing computing time on each of the above traffic networks in the static case, i.e. each link has free flow capacity when traversing it.

Figure 4 CPU running time for routing on free-flow network Net1



Up to an average distance of the start and the end node of 150 km, the Landmarks A* algorithm is able to compete with the Overdo A* with an overdo factor of 3, whose resulting paths are about 20% more expensive (for from-to distances of 150 km) than the least-cost path. While Landmarks A* for very short paths (from-to distance of 1 km) is 500 times faster than Basic Dijkstra on the two larger networks (Net3 and Net4) and 100 times faster on the two smaller networks (Net1 and Net2), the speed up for the longest paths (from-to distance of 200 km) is only 21 on Net1, 4 on Net2, 2.7 on Net3 and 4.5 on Net4.

Figure 5 CPU running time for routing on free-flow network Net2

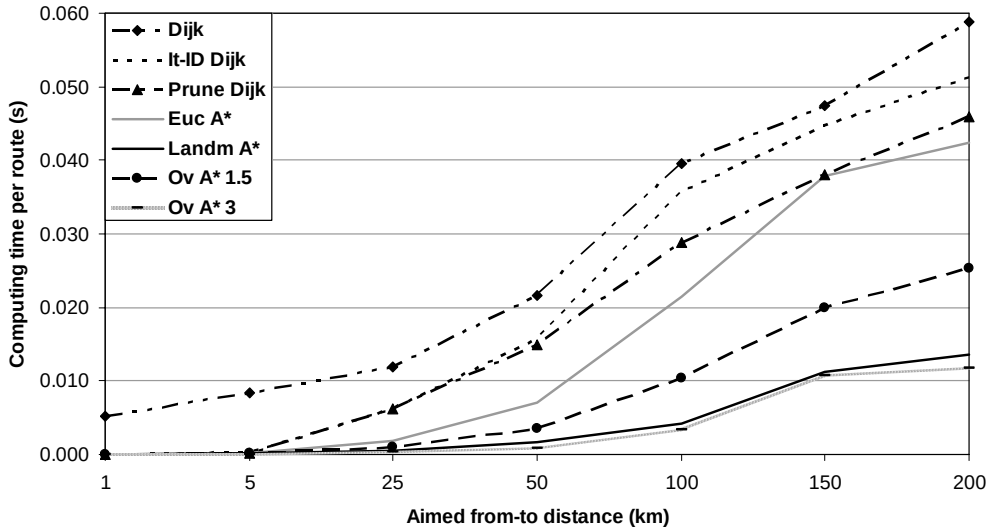
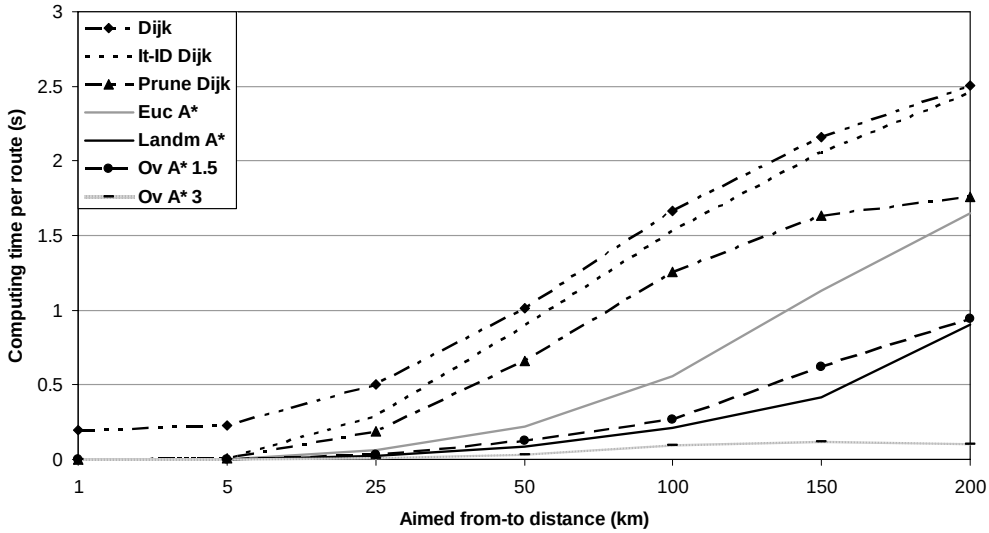


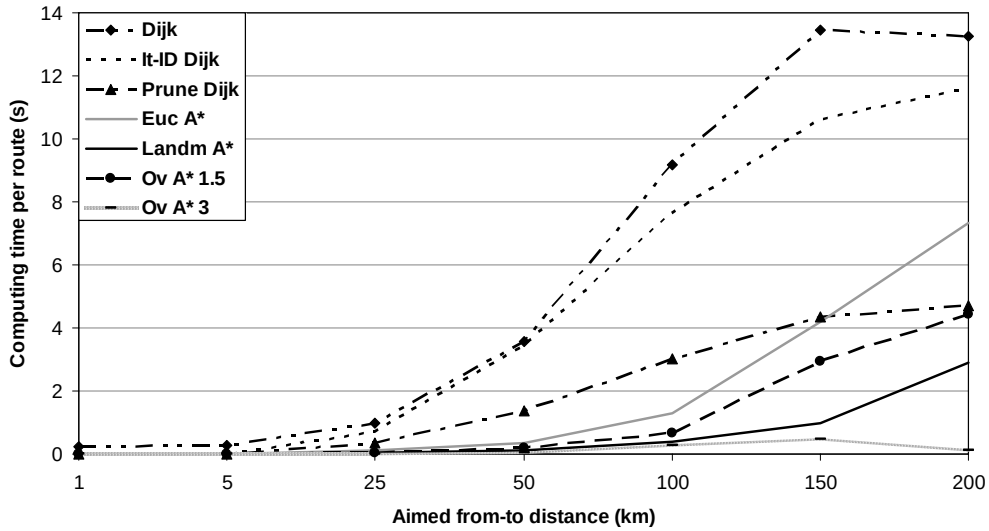
Figure 6 CPU running time for routing on free-flow network Net3



This is probably due to the fact that the costly initialization phase of the Basic Dijkstra algorithm (the visited flags of all nodes in the network are reset) is independent of the path to be calculated and therefore makes an essentially larger impact on the computing time when cal-

culating short paths. On long paths, this initialization has less impact. Congruously, one can see the same behavior of the speed up between Basic Dijkstra and Iteration-ID Dijkstra (on paths with from-to distance of 1 km the speed up is around 60 while it drops down to 1.1 on paths with from-to distance of 200 km).

Figure 7 CPU running time for routing on free-flow network Net4

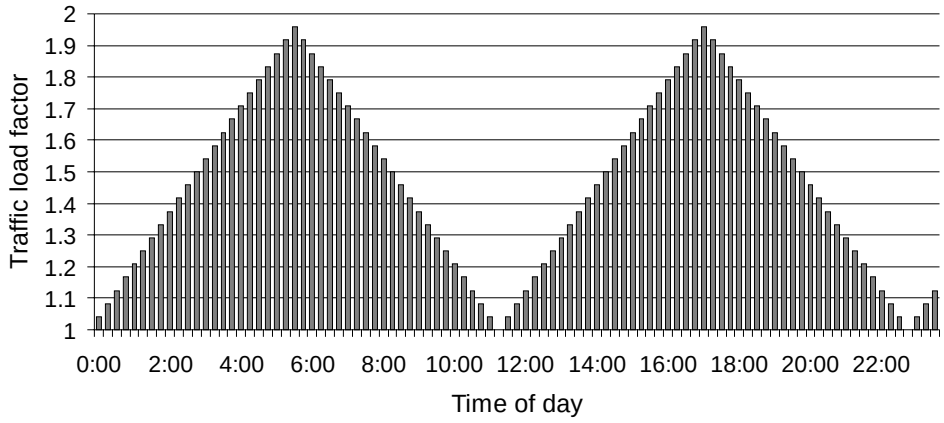


The computing time of Euc A* for from-to distances of 200 km being larger than the one of Prune Dijk is assumed to be an artifact as discussed above.

4.2.2 Loaded

In the second setup, a so-called traffic load factor is used, reflecting how congested a link is. It depends on the time the link is being traversed in order to simulate the travel cost in a real loaded traffic network (see Figure 8) similar to the one within an iteration of a traffic flow simulation done in MATSim-T. There exist two peak times during a day where the travel cost is twice as high as in the free-flow case. The travel cost of a link at a given time is calculated as follows: First, the index of the time slot the given travel time is in is calculated, where a time slot encompasses 15 minutes. Then this time slot index is augmented by a link-specific fixed random “travel time delta”, having values 0, 1, 2 or 4. Finally, the free-flow travel cost of the link is multiplied by the traffic load factor within the calculated time slot and a link-specific fixed random “amplification factor” which ranges between 1 and 1.1.

Figure 8 Traffic load factor per time slot



In a typical real loaded network, it is uncommon that each link shows the time-dependent behavior of Figure 8, but only the quite few ones which are in the areas of high population density. Our artificial traffic load imitation thus represents a highly and evenly congested network. The computation time for routing in loaded networks is shown in Figures 9 to 12.

Figure 9 CPU running time for routing on loaded network Net1

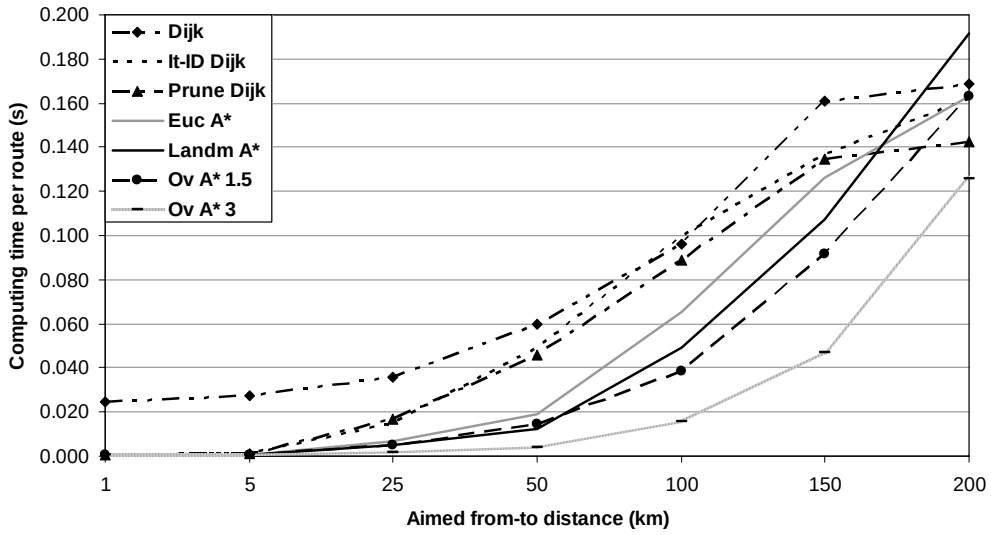


Figure 10 CPU running time for routing on loaded network Net2

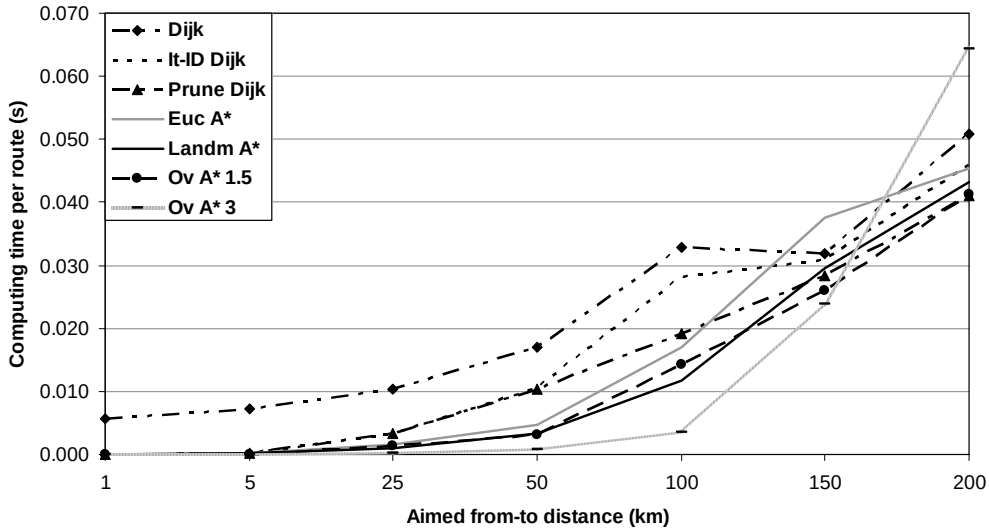


Figure 11 CPU running time for routing on loaded network Net3

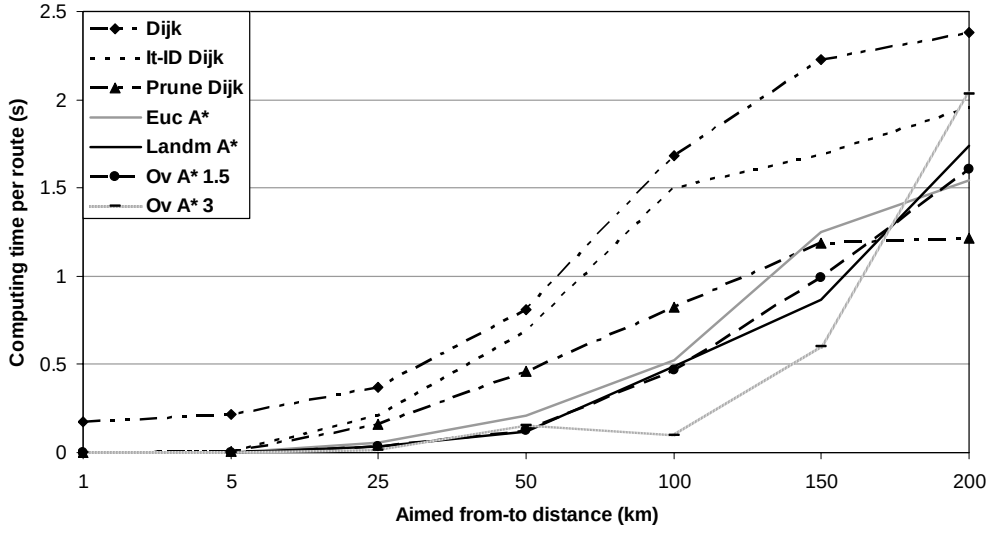
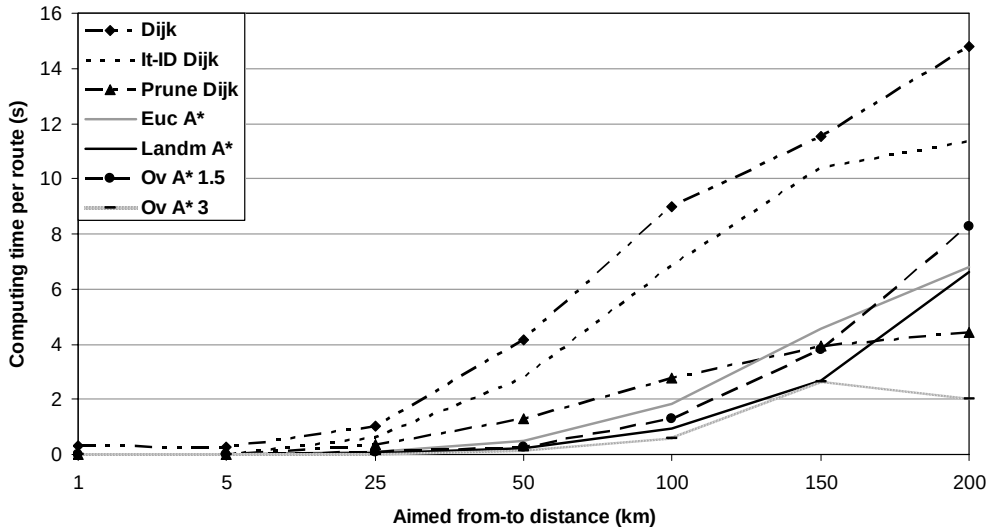


Figure 12 CPU running time for routing on loaded network Net4



To note is the seemingly chaotic behavior of the results for large from-to distances. Again it is assumed that these results are artifacts.

The speed up of Landmarks A* to Basic Dijkstra in the loaded case is not as high as in a free-flow network, especially for long paths (from-to distance 200 km): 10 on network Net1, 1.2 on Net2, 1.4 on Net3 and 2.2 on Net4. For the shortest paths, it is the same as in the free-flow case above. Probably this happens because Landmarks A*'s remaining travel cost estimate is based on the free speed of the links and is thus not as accurate in the loaded case as in the free flow case. The consequence of a worse estimate of the remaining travel cost is that more nodes get visited during routing, augmenting the computing time.

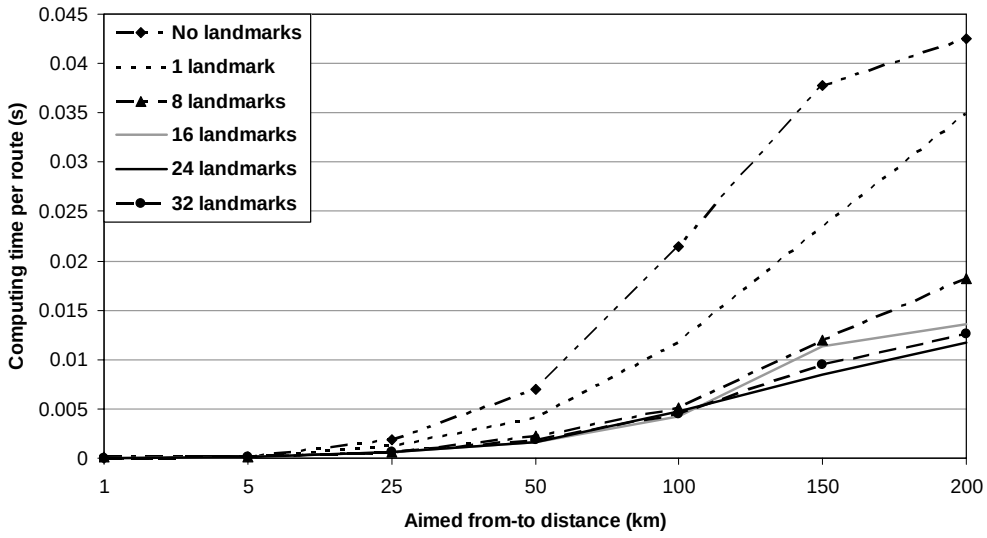
Interesting is the behavior of Euclidean A* with an overdo factor of 1.5, considering that in our case, the travel cost is equal to the travel time: About half of the 10,000 computed paths are slightly less costly than the ones found by Basic Dijkstra (and the other routing algorithms except Overdo A* (with a factor higher than 1.5)), in the other cases they are a little bit costlier. This happens basically because in the scenario above, it is possible to arrive earlier at the destination of a link by starting later. In other works, a link does not ensure the FIFO (first-in-first-out) property which states that a car that starts traversing a link will not reach the end of the link later than any subsequent car at the start. Whether or not a link ensures the FIFO property depends on the steepness of the traffic load function in Figure 8, i.e. the change of the traffic load from one time slot to the adjacent one. Thus, a car starting at the end of one time slot (say, at time 0:14:59) with higher traffic load will possibly reach the end of the edge

later than a car starting at time 0:15:00 (hitting the next time slot) if the change of the load factor between the two time slots is high enough (depending on the length of the link of course).

4.3 Landmarks

Figure 13 shows the routing computing time of the Landmarks A* algorithm on Net2 related to the number of landmarks used in the free flow case. For comparison purposes, the computing time of the Euclidean A* (named *No landmarks* below) is also shown. With one landmark, Landmarks A* is already faster than Euclidean A*. From 1 to 20 landmarks, the performance rises steadily with each additional landmark used. But for 20 to 24 landmarks, the performance decreases slowly, so it seems that from 24 landmarks on, the overhead for each additional landmark (periodically during routing, all landmarks are checked to see whether there are landmarks that yield a better cost estimate and should be added to the set of the active landmarks used for travel cost estimation) outweighs the enhancement it contributes to the estimate of the remaining travel cost.

Figure 13 Impact of number of landmarks on computing time (free flow Net2)

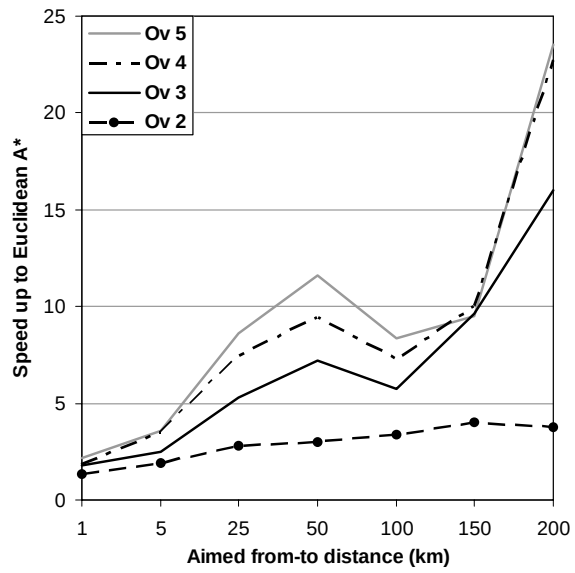


On the larger network Net3, this effect is not as explicit; it may be that the best number of landmarks (in terms of computing performance, not in terms of memory usage of course) is above 24.

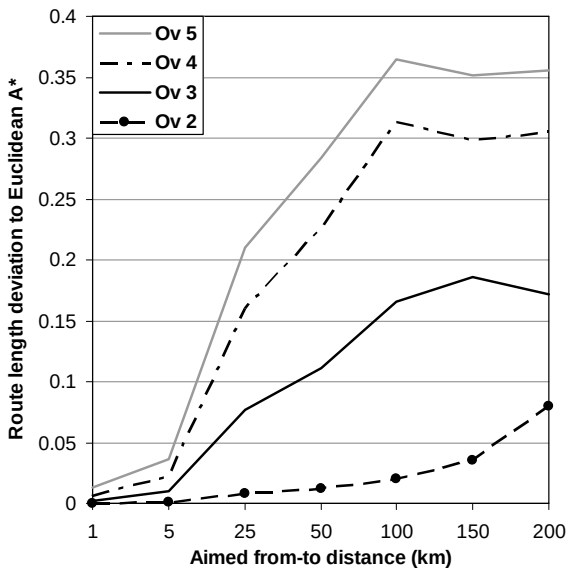
4.4 Overdo

Figure 16 shows the behavior of Overdo A* for overdo factors 1.5 to 5 compared to Euclidean A* (i.e. overdo factor 1) on free flow Net3. The speed up in computing time compared to Euclidean A* increases with the overdo factor, but so does the cost of the resulting path (again compared to the cost of the path delivered by Euclidean A*). So, at about 100 km from-to distance, an average path computed by Overdo A* 5 is about 1.36 times more expensive (or if travel cost equals travel time: takes 1.36 times longer to travel) than an average path computed by Euclidean A*. In absolute terms this is a difference in travel time of 3112 seconds (the savings in computing time are 1 seconds per path). For paths of aimed from-to distance 200 km, this difference is 5718 seconds (while the computation takes 1.6 seconds less per path).

Figure 14 Overdo A* compared to Euclidean A* (free flow Net3)



Speed up comparison



Route length comparison

4.5 Real world example

This section's subject is a real world example of agents commuting from their homes located in the surroundings of the city of Zurich to their work locations within or near Zurich, resulting in a total of 10,000 trips. Figure 15 shows the histogram of the linear distance of the home and work locations. The average distance is 5.07 km while the median distance is 3.54 km

because most of the work-location pairs gather around a distance of 1 to 4 km. 120 paths start and end at the same node (from-to distance is 0). This is typical for real world scenarios with agents that work at the same place they live, thus producing 0-length trips.

Figure 15 Real world scenario paths from-to linear distances

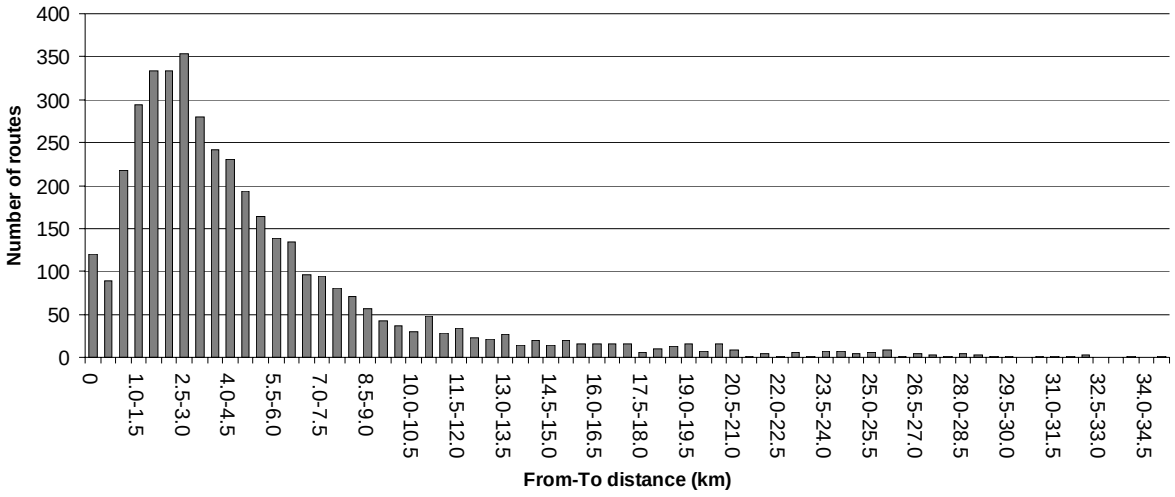
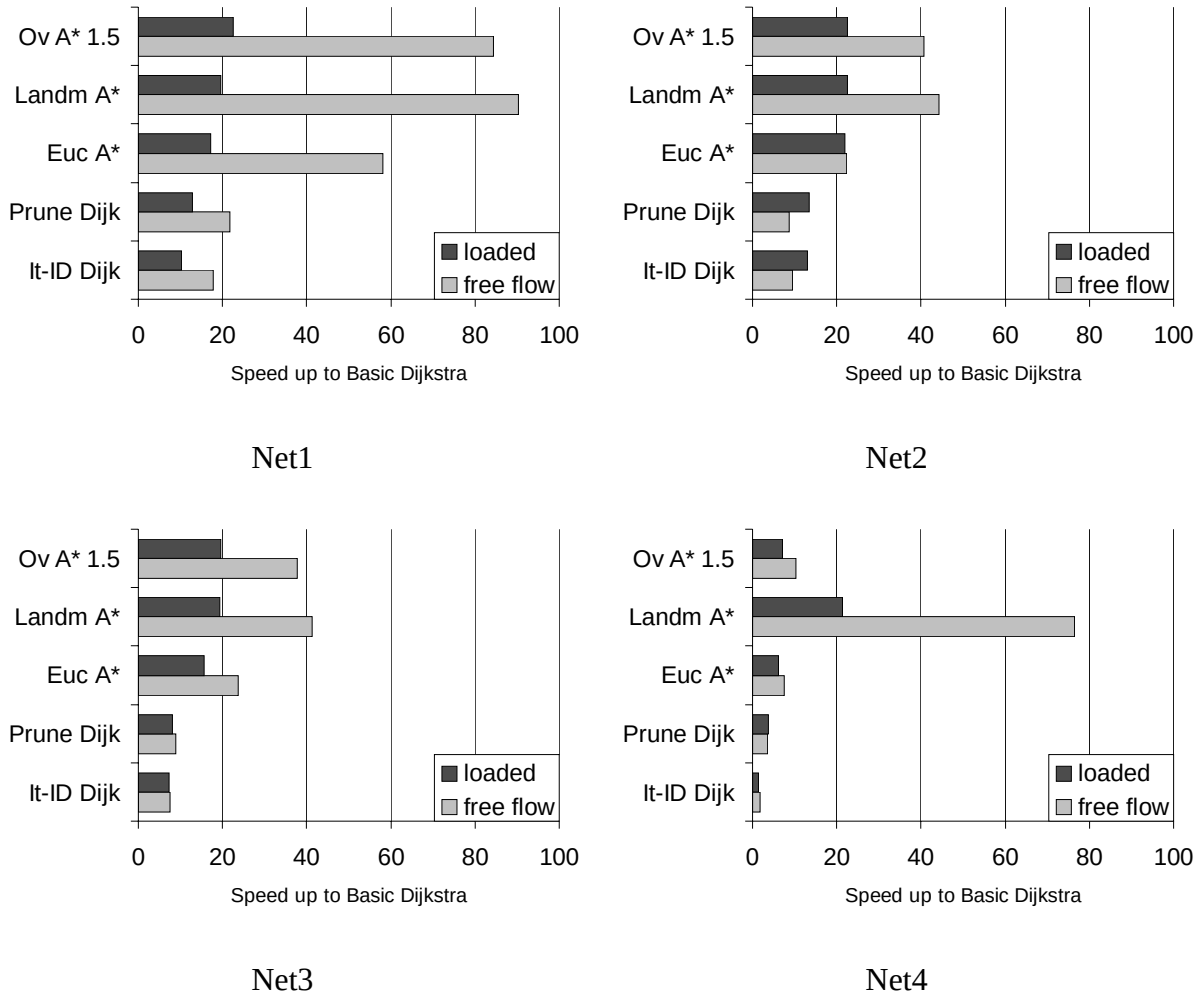


Figure 16 compares the speed up of the routing algorithms to the Basic Dijkstra algorithm (Overdo A* 3 is not listed here, because it produces suboptimal paths in both the loaded and the free flow case). On all networks, the speed up of Landmarks A* and Overdo A* 1.5 when the networks are loaded is about half the speed up as when the agents can travel with free speed on all links. This has probably to do with the noise that is introduced by loading the network which corrupts the estimate of the remaining travel cost (or travel time, in our example) of the A* routing algorithms. The estimate of Euclidean A* as well as Landmarks A* is based on the free flow travel time (if it was not, there would be the risk of overestimating the resulting travel time, thus producing suboptimal paths). And the worse the estimate of the remaining travel time is the more nodes are visited during routing, which results in a loss of computing performance.

Figure 16 Speed up to Basic Dijkstra for routing of a real world example



5 Discussion

The following improvements make the newly implemented Landmarks A* routing algorithm perform 20 times faster than the original Basic Dijkstra on a loaded network (i.e. when the cost of a link is time-dependent and on average 1.5 times higher than the least cost of that link): Using a loop variable that discriminates the current routing request from previous requests instead of initializing all nodes before a routing, ignoring nodes in dead ends during routing and using a heuristic estimate based on landmarks in order to direct the routing algorithm towards the goal and thus (in the majority of cases) examining less nodes during routing. In all our test settings, the Landmarks A* is the fastest exact routing algorithm (i.e. it guarantees to deliver least-cost paths in a free flow network). The memory needs O_L (in a 32-bit environment) are

$$O_L = [4 \text{ bytes}] * [\text{number of nodes in the network}] * [\text{number of landmarks}] + [8 \text{ bytes}] * [\text{number of nodes in the network}].$$

This data can be stored along the network to avoid having to pre-process the network each time a routing is performed. The pre-processing depends on the number of nodes and links in the network and in our case takes around 160 seconds on a network with roughly half a million nodes and a million links while the memory usage for this network is about 62 megabytes with 16 landmarks. If the cost of each link at any given time is minimal (i.e. equal to the free flow cost), Landmarks A* performs roughly double as fast as in our artificial loaded case because the heuristic function it uses to estimate the remaining travel cost from a given node to the target node is based on the free flow travel cost of each link. Using a better heuristic estimate function based on the cost of each link in each time bin seems highly complex and is an unsolved problem yet.

The Overdo A* algorithm with an overdo factor equal or above 2 performs faster than Landmarks A*, but its drawback is that it does not necessarily deliver least-cost paths. While the computing time decreases with increasing overdo factor, the cost of the resulting paths increases as well (while their length decreases). However, it should be investigated how Overdo A* performs in a system that needs several iterations to reach a stable state: While the computing time of one iteration and/or the number of needed iterations go down (i.e. the simulation ends earlier), the final state of the simulation is expected to be worse (in terms of the scores of the plans of the agents) than a final state found with an “exact” algorithm like

Landmarks A*. If memory usage matters, Overdo A* with an overdo factor of 1.5 could be considered when dealing with loaded networks, as it is computationally almost as fast as Landmarks A* and delivers paths that should be similar to the least-cost ones.

6 Outlook

The impact of Overdo A* 1.5 as opposed to Landmarks A* on the overall computation time and on the final overall score of the agents' plans of a simulation as stated above should be investigated.

The current implementation of the detection of dead ends is not very elaborate. For example, cycles in dead ends may prevent a dead end to get detected. The same holds for dead ends that are connected by multiple nodes to the rest of the network. By improving the dead end detection algorithm and thus having more nodes in dead ends which can eventually be omitted during routing, it may be possible to reduce the routing computation time. It is also conceivable of creating a hierarchy of dead end classifications, going from small dead end clusters like the ones in residential areas to whole valleys that are only connected by a few links to the rest of the net such that they can be ignored during routing. This approach seems particularly promising for road networks on mountainous topologies like in Switzerland.

Another approach would be to use information gathered by a routing operation for a subsequent one like in Demetrescu (2001) and Misra and Oommen (2006) and thus sharing the routing information among iterations of the traffic simulation. The practicability of such algorithms depends on how much links change their cost between iterations. The more links do, the worse is the performance of these algorithms. So in a first step, the difference of the link cost between iterations would have to be determined. Another crucial property of these algorithms is their possibly high memory requirements, so it is not clear whether networks with half a million nodes and a million links can be routed with those algorithms.

An option adopted by Dean (1999) is to use continuous-time dependent link cost instead of discrete-time dependent link cost. This potentially further reduces the computing time.

7 References

Balmer, M. (2007) Travel demand modeling for multi-agent traffic simulations: Algorithms and systems, Ph.D. Thesis, ETH Zurich, Zurich.

Chabini, I. and S. Lan (2002) Adaptations of the A* algorithm for the computation of fastest paths in deterministic discrete-time dynamic networks, *IEEE Transactions on Intelligent Transportation Systems*, **3** (1) 60–74.

Chou, Y.-L., H. E. Romeijn and R. L. Smith (1998) Approximating shortest paths in large-scale networks with an application to intelligent transportation systems, *INFORMS Journal on Computing*, **10** (2) 163–179.

Dean, B. C. (1999) Continuous-time dynamic shortest path algorithms, MSc. Thesis, Massachusetts Institute of Technology, Cambridge.

Demetrescu, C. (2001) Fully dynamic algorithms for path problems on directed graphs, Ph.D. Thesis, University of Rome La Sapienzia, Rome.

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

Ertl, G. (1998) Shortest path calculation in large road networks, *OR Spectrum*, **20** (1) 15–20.

Fu, L., D. Sun and L. R. Rilett (2006) Heuristic shortest path algorithms for transportation applications: State of the art, *Journal of Computers and Operations Research*, **33** (11) 3324–3343.

Fu, M., J. Li and Z. Deng (2004) A practical route planning algorithm for vehicle navigation system, paper presented at the *5th World Congress on Intelligent Control and Automation (WCICA)*, Hangzhou, Jun. 2004.

Goldberg, A. V. and C. Harrelson (2005) Computing the shortest path: A* search meets graph theory, paper presented at the *16th annual ACM-SIAM symposium on Discrete algorithms (SODA)*, Vancouver, Jan. 2005, `http://www.siam.org/meetings/DA05/`.

Gutman, R. (2004) Reach-based routing: A new approach to shortest path algorithms optimized for road networks, paper presented at *the 6th Workshop on Algorithm Engineering and Experiments (ALENEX)*, New Orleans, Jan. 2004, `http://www.siam.org/meetings/alenex04/`.

Hart, P. E., N. J. Nilsson and B. Raphael (1972) A formal basis for the heuristic determination of minimum cost paths, *ACM SIGART Bulletin*, **37**, 28–29.

Holzer, M., F. Schulz, D. Wagner and T. Willhalm (2005) Combining speed-up techniques for shortest-path computations, *ACM Journal of Experimental Algorithmics (JEA)*, **10** (2) 1–18.

Huang, B., and F. B. Zhan (2007) A shortest path algorithm with novel heuristics for dynamic transportation networks, *International Journal of Geographical Information Science*, **21** (6) 625–644.

Jacob, R., M. Marathe and K. Nagel (1999) A computational study of routing algorithms for realistic transportation networks, *ACM Journal of Experimental Algorithmics (JEA)*, **4** (6) 197–218.

Köhler, E., R. H. Möhring and H. Schilling (2005) Acceleration of shortest path and constrained shortest path computation, in S. Nikoletseas (ed.), *Lecture Notes in Computer Science*, **3503**, 126–138, Springer.

Lauther, U. (2004) An extremely fast, exact algorithm for finding shortest paths in static networks with geographical background, in M. Raubal, A. Sliwinski and W. Kuhn (eds.) *Geoinformation und Mobilität - von der Forschung zur praktischen Anwendung*, **22**, 219–230, IfGI prints, Institute for Geoinformatics, Münster.

Misra, S. and B. J. Oommen (2006) An efficient dynamic algorithm for maintaining all-pairs shortest paths in stochastic networks, *IEEE Transactions on Computers*, **55** (6) 686–702.

Sanders, P. and D. Schultes (2005) Highway hierarchies hasten exact shortest path queries, in G. S. Brodal and S. Leonardi (eds.) *Lecture Notes in Computer Science*, **3669**, 568–579, Springer.

Sedgewick, R. and J. S. Vitter (1986) Shortest paths in Euclidean graphs, *Algorithmica*, **1** (1) 31–48.

Wagner, D. and T. Willhalm (2003) Geometric speed-up techniques for finding shortest paths in large sparse graphs, in G. di Battista and U. Zwick (eds.) *Lecture Notes in Computer Science*, **2832**, 776–787, Springer, Budapes